

note

147 views

Project: cs152coin, Part 1. Due Monday Feb 26 at 11:59pm

Introduction

For this year's project, you'll be working with c152coin, a made-up cryptocurrency that aims to simulate certain aspects of real cryptocurrencies like Bitcoin. There is good news and bad news. The bad news is that unlike real cryptocurrencies, cs152coin can't make you rich. The good news is that unlike some other cryptocurrencies, cs152coin isn't a Ponzi scheme — at least not yet.

Like other cryptocurrencies, at the core of cs152coin is a single *ledger* of all of the transactions (i.e., money transfers) that have ever taken place since the currency was created. The ledger is public, and so all transactions, whether they were (hypothetically) used to buy a car or a coffee are available for anyone to inspect. Transactions are denominated in cs152coins, a new currency, rather than traditional currencies like dollars. As with other cryptocurrencies, if cs152coin were a real thing, there would only two ways of obtaining cs152coins: (1) buying cs152coins from someone who already has them in exchange for goods or other currencies (i.e., being on the receiving end of a transaction), or (2) mining, a way of obtaining newly-created cs152coins (see below).

More specifically, cs152coin consists of four main components: *addresses*, *transactions*, *blocks*, and the *blockchain*. An address is like a bank account that has a unique identifier like an account number (e.g., 321a34b5fb3e8ca5f53e87b6a95aed1) and a balance (the amount of cs152coins currently associated with the address). A single user can have as many addresses as they want, each with a separate ID and balance, and they can create new addresses with a starting balance of 0 on demand. Like a bank account it's possible to deposit cs152coins into an address and withdraw from it.

A transaction represents a transfer of cs152coins from 1–5 input addresses to a single output address. The transaction specifies the amount to withdraw from each input address and the amount to deposit into the output address. The amount of the deposit should equal the sum of the withdrawals. Every input address must already exist and have a nonzero balance before the transaction. The output address could already exist, but it could also be brand new. It might have been created by a user for the express purpose of receiving the funds from the transaction. Each transaction also has a unique ID.

Transactions are, in turn, bundled into blocks, which contain a list of transactions that have all taken place in a short amount of time. Blocks are created by *miners* who gather up recent transactions that they've heard about, bundle them together, and add them to the ledger. To provide an incentive for people to create blocks, miners receive a reward for their service in the form of newly-created cs152coins. Every block begins with a single deposit of the block's mining reward to an address controlled by the miner. Like other cryptocurrencies, the mining reward is determined by a fixed formula: in the case of cs152coin, it begins at 50 cs152coins for the first block and drops by half every 1000 blocks. Thus, the process of creating blocks is called mining.

All of the blocks ever created are organized into a list ordered by creation time called the *blockchain*. The blockchain represents the public ledger and stores all of the transactions ever performed.

cs152coin only simulates the structure of the blockchain. Real cryptocurrencies like Bitcoin are more complex because, among other reasons, they're *decentralized*. Rather than storing the blockchain on a single server somewhere or even spread across multiple servers in a data center of single entity like Google, in Bitcoin, every miner stores their own copy of the whole blockchain — every transaction ever. Every miner can collect new transactions and add new blocks to the blockchain. Miners gossip among themselves about newly-created transactions and blocks over the Internet via peer-to-peer network connections. This is called Bitcoin's *network protocol*.

The fact that every miner maintains their own copy of the blockchain, the fact every miner can create new blocks, and the fact that anyone can be a miner just by downloading and running the Bitcoin software creates a problem: how do users agree on what the authentic blockchain is? Bitcoin's solution is its *consensus protocol*, a set of rules that every miner running the Bitcoin software follows to decide which transactions and blocks are valid. These rules stipulate that the first miner to submit a new block that succeeds the most recent block gets their block accepted and gets to collect the block's mining reward. The results also stipulate that a block is only considered valid if it contains a solution to a *computational puzzle* that takes significant computing resources to solve. This means that miners with greater computing resources at their disposal are likely to find solutions to the puzzles sooner than other miners and, therefore, are more likely to be the first to submit valid next blocks. Thus, the more computing resources a miner has, the more mining rewards they can expect to reap.

This introduction only scratches the surface. For more information, we recommend *Bitcoin and Cryptocurrency Technologies* by Narayanan et al. (<http://bitcoinbook.cs.princeton.edu/>). A free preprint is available [here](#).

Project, Part 1: Analyzing the Blockchain

A cryptocurrency's network and consensus protocols involve multiple users around the Internet communicating and competing with each other, and so they're difficult to simulate for a course project. As a result, this project will assume that the blockchain already exists, and your task will be to analyze it in various ways.

The files you need to get started are here ([cs152coin_part1.zip](#)). The zip file contains skeleton .c and .h files, a Makefile, and two files containing blockchains (blockchain.txt and blockchain-short.txt) that you'll analyze. Your work will involve filling in the .c and .h files as appropriate by implementing empty functions and adding functions and fields to structs as appropriate. The header files contain extensive documentation about how each function is supposed to work.

Part 1A: Parsing the Blockchain

Your first task is to read in a blockchain file containing an entire history of transactions, parse it, and create an in-memory data structure corresponding to it. You can provide a blockchain file as input to your program as follows:

```
$ ./cs152coin < blockchain.txt
```

The format of blockchain files is as follows:

```
BEGIN_BLOCK
  DEPOSIT <miner address> <mining reward>

  BEGIN_TRANSACTION <transaction id>
    BEGIN_INPUTS
      WITHDRAWAL <address> <withdrawal amount>
      WITHDRAWAL <address> <withdrawal amount>
    ...
    END_INPUTS
    DEPOSIT <address> <deposit amount>
  END_TRANSACTION

  BEGIN_TRANSACTION <transaction id>
```

```
...
END_BLOCK

BEGIN_BLOCK
...
```

Each distinct element in the file, called a *token*, is separated by whitespace (some combination of spaces, tabs, and newlines). The amount of whitespace is irrelevant: indentation and blank lines are ignored.

cs152coin.h, the project's main header file, defines structs for deposits and withdrawals, transactions, blocks, and the blockchain. Parsing the input file involves reading it in token by token and instantiating the appropriate structs corresponding to the contents of the file. cs152coin.h declares a series of functions that you will have to implement in cs152coin.c for creating and freeing each type of struct. For example, parse_trans is supposed to read the tokens corresponding to a single transaction from the file, allocate a new trans_t on the heap, fill it in with the appropriate data, and then return a pointer to it. parse_trans should return NULL if it could not successfully parse the transaction (e.g., if it encounters unexpected tokens). These structs are nested: the blockchain contains multiple blocks, each of which contains multiple transactions, each of which contains a deposit and potentially multiple withdrawals. To simplify the handling of this complex structure, you should implement functions that parse larger structs using the functions that parse the smaller structs that they contain (e.g., parse_block should probably call parse_trans more than once).

To ease the process of interpreting the input file, we're providing you with a parser library (parser.h) that handles the task of chopping up the input into tokens so that you don't have to read each individual character of the file yourself. It provides three functions: read_string, which interprets the current token as a string and fills in a pointer to it; read_double, which interprets the current token as a double and fills in a pointer to it; and next_token, which advances to the next token in the file. See parser.h for details on each function. To give you an example of how to use the parser library, we've already implemented parse_dep_wd in cs152coin.c.

The end result of parsing should be a pointer to a single blockchain_t which in turn points to a list of block_t, each of which points to a list of trans_t, each of which contains multiple pointers to dep_wd_t. **Make sure that your parser works (i.e., successfully translates the input file into structs) before moving on to the subsequent parts of the assignment. Try parsing blockchain-short.txt first. It's short enough that it should be fairly easy to see that your parser has successfully interpreted all of the input.**

Part 1B: Computing Basic Statistics

Once you've successfully parsed the file, your tasks in parts 1B–1D will involve analyzing the blockchain by implementing the compute_stats function in cs152coin.c. compute_stats should iterate through the blockchain and collect the necessary information for parts 1B–1D along the way. **To complete parts 1B–1D, you should only need to make a single pass over the entire blockchain (once for the entire program, not once for each part).**

Part 1B is a warmup: as you're iterating through the blockchain in compute_stats, keep track of the total number of blocks, the total number of transactions, and the average value of each transaction (i.e., the amount deposited with the recipient). Store these values in the appropriate fields of the blockchain_t.

Part 1C: Tracking Address Balances

Every address has a balance associated with it — the number of cs152coins (or a fraction thereof) that it contains. As you may have noticed, however, each address's balance isn't explicitly stored anywhere. The blockchain only keeps track of transactions, which are transfers of cs152coins from addresses to other addresses. As a result, at any given time, it's not immediately clear how many cs152coins each address contains.

Your task in this part is to fix this by keeping track of the balance associated with each address. As you iterate through each transaction on the blockchain in compute_stats, you'll need to somehow update the balance associated with each address so that by the time you've processed the whole blockchain, you'll know each address's ending balance. cs152coin.c is written so that it's possible to query addresses' ending balances on the command line as follows:

```
$ ./cs152coin 321a34b5fb3e8ca5f53e87b6a95aed1 < blockchain-short.txt
...
Balance for address 321a34b5fb3e8ca5f53e87b6a95aed1: 5.815894
```

To do this, you'll need a data structure that allows you to efficiently look up an address and both query and update its current balance. Think carefully about which data structure is the right one for the job. The functions that the data structure should implement are declared in addr_bal.h. You'll need to decide what kind of data structure to use and implement it by adding fields to the addr_bal_t struct, implementing the functions in addr_bal.c, and potentially adding any other functions and struct declarations that you might need.

Part 1D: Detecting Double Spending

One of the risks of cryptocurrencies is that participants can misbehave. One of the most prominent examples of this is *double spending*, where a user tries to spend the same cs152coins more than once, thereby spending more money than they actually have. In cs152coin, double spending occurs when a transaction attempts to withdraw more cs152coins from an address than it actually contains at that point.

In this part, your task is to detect double spending. As you iterate through the blockchain in compute_stats, you'll need to check for double spending, and each time you find it, make a note of the transaction where it occurred and the address involved. You'll need to store the (transaction id, address) pairs in another data structure whose functions are declared in dbl_spends.h. Once again, think carefully about which data structure you need, and implement it in dbl_spends.h and dbl_spends.c. The data structure must implement a function dbl_spends_show that will be called after you analyze the blockchain and that must print all of the (transaction id, address) pairs corresponding to double spends to the supplied file. **The pairs must be printed in the order they appear in the blockchain.**

Submission Instructions

Fill in all of the functions marked "FILL ME IN" in cs152coin.c, addr_bal.c, and dbl_spends.c. As mentioned above, you'll probably also have to implement other functions, add fields to the addr_bal_t and dbl_spends_t structs, and define other structs in order to complete the assignment. **You should not need to modify any of the non-empty functions in cs152coin.c or the Makefile.**

Submit all of the .c and .h files and the Makefile using Subversion

Sidebar: Other Important Differences from Bitcoin

As we mentioned above, cs152coin is a vast simplification of cryptocurrencies like Bitcoin. In addition to lacking network and consensus protocols, there are at least three other important differences between cs152coin and Bitcoin. First, transactions in cs152coin only support transfers between a set of input addresses and single recipient. In Bitcoin, not only can transactions have multiple outputs, but they can support operations other than simple transfers. In fact, Bitcoin transactions are actually little programs implemented in a special programming language, and thus can do many different things. Other cryptocurrencies like Ethereum have even more expressive programming languages for their transactions, turning the blockchain into a *consensus computer* whose programs are simultaneously executed by every participant in the Ethereum network. The way the programs execute is determined by the consensus of Ethereum's participants. These programs are often used to implement *smart contracts*, which are contracts whose terms are executed automatically.

The second difference is that cs152coin doesn't provide any mechanism for a user to prove that they own a particular address. Anyone can create a transaction that transfers the cs152coins in one address to another address whether or not they own the source address. That obviously wouldn't work for a real cryptocurrency because it would let anyone steal money from anyone else's addresses. Real cryptocurrencies address this problem using [public-key cryptography](#). Every address is associated with a cryptographic private key that

only the address owner knows. A transaction that transfers Bitcoins out of an address must be **digitally signed** using the private key of the address's owner to be considered valid. Digital signatures are validated using the public key corresponding to the user's private key (in fact, Bitcoin addresses are actually shorthands for the public keys themselves).


Third, deposits and withdrawals from addresses work a bit differently in Bitcoin. The inputs to Bitcoin transactions aren't addresses and amounts to withdraw, they are the outputs of previous transactions. Furthermore, if a Bitcoin transaction "spends" the output of a previous transaction, it must spend the entire amount of that output. If that entire amount isn't needed and there is "change" left over, the change goes to an additional output of the transaction. That output is either the same address as the input address or another address controlled by the same user as the input address. The advantage of this approach is that it makes detecting double spending easier: rather than having to track the balance in every address, you just have to check that every output of every transaction is only used as the input to a subsequent transaction exactly once.


project1 assignment_itself

Updated 11 months ago by Adam Shaw and Ariel Feldman


followup discussions for lingering questions and comments


☒ Resolved ☐ Unresolved

 **Philip Adams** 1 year ago Would it be possible to publish the correct output of `dbl_spends_show` for either `blockchain.txt` or `blockchain-short.txt` for testing purposes? Computing it by hand is pretty time-intensive and error-prone.


 **Ariel Feldman** 11 months ago Computing the correct output for `blockchain-short.txt` yourself should be feasible. Make sure your code works on it before trying to process `blockchain.txt`.


☒ Resolved ☐ Unresolved


 **Rebecca Chen (rebeccachen)** 11 months ago Is this due Sunday at 11:59 pm or Monday at 11:59 pm?

 **Ariel Feldman** 11 months ago Monday.

☒ Resolved ☐ Unresolved

 **Avinash Rao** 11 months ago Should we submit the project in a folder called `cs152coin` or `project1`?

 **Gwendolyn Gilbert-Snyder** 11 months ago Based on previous assignments, `project1`.

 **Valerie Han** 11 months ago See [@412](#)