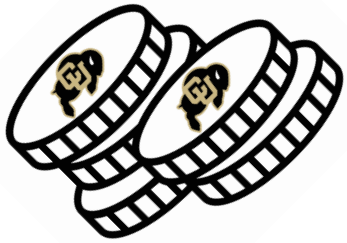
CSCI 2270: Data Structures - Project Specifications

`Buff-CUoin` (Design and Implementation of a Toy Cryptocurrency)



Due Tuesday, April 26, 11:59 PM

We have elected to put our money and faith in a mathematical framework that is free of politics and human error.

– **Tyler Winkelvoss, Founder of Gemini Cryptoexchange**

1 A Brief History of Currency.

{`From Bartering and Banknotes to Blockchains and Bitcoins`}

While the notion of currency is a relatively modern (less than 5000 years old) invention in the scale of human history, even prehistoric humans needed to exchange goods and services. Prior to currency, it was done in the form of *bartering* where two consenting parties exchanged goods (or services) for other fair valued goods (or services). As a simple example of bartering, consider a farmer paying a carpenter with produce in return of carpenter's service of building a fence around his field. While it served well in simpler times, bartering system hampered economical growth for several reasons.

The key problem with bartering is that its success is predicated upon the so-called *double coincidence of wants*: in order to make any exchange possible, two wants must exists, i.e. one party wants to buy something that the other party wants to sell, and vice-versa. Such coincidences are rare to come across; after all how many times a farmer may wish to build a new fence? Bartering also limits the growth of economies as it does not allow parties to enter into future contracts: a farmer with a bountiful, yet perishable crop may not be able to translate his fortunes into round-the-year wealth.

Currency (Banknotes, coins, gold) solves aforementioned problems by being a medium of exchange as well as a storage of value, as long as all of the involved parties put faith and trust that a given currency has a value. The traditional notions of currency served humans well over the last 5000 years: these currencies were typically issued by the governments or well-trusted banking bodies (guaranteed by gold standard), and it not only allowed for a seamless transfer of goods and values, but also paved way to accumulating wealth.

However, the advent of the Internet brought the need of digital currency to be able to exchange goods and services over the Internet. The key problem is that of *the problem of double spending*: how can a farmer selling his produce online to a carpenter be sure that the same money is not simultaneously sent to someone else? There are two solutions to this problem: the first is the introduction of *Trusted Third Parties* (like Visa and Mastercard) to mediate and verify such transaction. These entities often keep a *ledger* of all transactions and have internal protocols to eliminate the problem of double spending. The second solution, which is the topic of this project, is having a decentralized ledger (owned by no single institution), guaranteed by the power of computational complexity.

Cryptocurrencies, such a Bitcoin and Ethereum, maintain such a decentralized ledger of transaction in a data-structure known as *blockchain*. Simply put, a blockchain is a linked-list of transaction records where every node contains a hash of its previous node. Moreover, inserting a new node in the list is intentionally made a computationally challenging task for the modern computers to solve, thus making modification to the ledger intractable with the current computational technology.

> In this project, you will implement a toy version of one such currency that we dub `Buff-CUoin` as a blockchain. To allow ourselves to focus on data-structural issues, and ignore the distributed programming issues, we will implement blockchain as a list that resides only during the execution of the program, and is interacted with a simple text-based interface.
> The basic functionality must include: 1) adding transactions, 2) mining blocks (to be specified later), 3) checking the validity of the blockchain, 4) checking the balance of a given user, and 5) printing the blockchain.
> You can implement advanced functionality including 1) reading and writing a blockchain from/to a file, 2) comparing two blockchains to decide which one has a longer history, 3) the implementation of a cryptographic hash function different than the one provided by us; and 4) any other feature that elevates the functionality of the blockchain by bringing it closer to a cryptocurrency.

We hope that you have as much fun working on this project as we had designing it. Good luck!

2 Project Specification

Your *blockchain* system must support the following functionality:

- ***Blockchain.*** A blockchain is to be implemented as a list of nodes called *Blocks* where each block consists of a list of transactions, hashcode of the previous block, a timestamp, and a parameter called *nonce*.

- **Transaction.** A *transaction* is a simple structure containing three fields: sender (a `string`), receiver (a `string`), and amount (`int`).
- **List of Pending Transactions.** A blockchain must have a member called *pending* that is a list of pending transactions.
- **Crypto Mining.** Periodically, all of the pending transactions are finalized by a process called *mining*. The process of mining is simply collecting all of the pending transactions, packaging them in a block and inserting that block to the blockchain. In order to make a blockchain secure, the insertion of a new block into the blockchain must be a computationally complex task. Your blockchain must have a private integer-valued member called *difficulty* such that a block can only be inserted into the chain if the nonce of the block is such that the hash-code of the contents of the block along-with the nonce results in a hash-code with the first n characters (where $n =$ difficulty) are 0's. This process is called the *mining* of a block.
- **Rewading Miners with Buff-CUoins.** For every successful mining of a block, a corresponding user (represented simply as a string) is paid a fixed amount of reward that we call Buff-cuoins or BFCs.
- **Balance Checking.** Between the mining of blocks, the blockchain must receive transactions containing three fields (sender, receivers, amount) documenting that sender has paid the receiver number of coins equal to the given amount. These transactions will be stored within the blockchain as a list of pending transactions. It is the job of your algorithm to traverse the blockchain to see if the user corresponding to the sender string has enough balance to be able to pay to the receiver. This can be done by traversing the blockchain, block-by-block, and traversing transactions within each block in the sequence, and keeping the information about the sender's history: you need to add all the coins received by her, and subtract any coins spent.
- **Validity Checking.** You need to write a procedure to check validity of a blockchain by traversing the blockchain sequentially, computing the hashcode of every block and checking if the next block has that hashcode in the field (previousHash).

3 ADT Specifications

In this section, we give the core required functionality of the blockchain including simple ADT of a transaction, block, blockchain, and the crypto-mining algorithm. You may create a more sophisticated blockchain, but it should subsume the following suggested functionality. **If you choose to add any such additional functionality, create a branch in your repository and add it there. Your main branch should only contain the core functionality.**

3.1 Transaction

```
class Transaction {
private:
    string sender; // address of the sender
    string receiver; // address of the receiver
    int amount; // number of coins transferred
public:
    Transaction(string _sender, string _receiver, int amount);
    /* getter functions */
    string getSender();
    string getReceiver();
    int getAmount();
    /* converting the transaction into a string */
    string toString();
};
```

3.2 Block

```
class Block {
  private:
    long long nonce; // An arbitrary number crucial in mining
    vector<Transaction> transactions; //vector of transactions
    string previousHash; // Hash of previous block
    time_t timestamp; // Time when created via: time(nullptr)
    string hash; // Hash of this block
  public:
    Block(vector<Transaction> _trans, time_t _time, string _prevHash);
    // setting the hashcode of this block
    void setPreviousHash(string hash);
    // Calculating the has of this block using SHA library
    string calculateHash();
    //Mine this block according to the given difficulty
    void mineBlock(unsigned int nDifficulty);
    // Convert the data to a string
    string toString();
};
```

3.3 Blockchain

```
class Blockchain {
  private:
      unsigned int difficulty; // difficulty level
      vector<Block> chain; // blockchain list
      vector<Transaction> pending; // list of pending transactions
      int miningReward; // #coins of reward for mining a block

  public:
     Blockchain(); //Constructor
     // Adding a transaction to the list of pending transactions
     void addTransaction(string src, string dst, int coins);
     //Checking validity of a transaction
     bool isChainValid();
     // mine a block containing the list of pending transaction and award
 minerAddress #coin=miningReward
     bool minePendingTransactions(string minerAddress);
     //Compute the balance of a given address
     int getBalanceOfAddress(string address);
     // A human-friendly printing of the blockchain
     void prettyPrint();

  private:
      // Return the last block on the chain
      Block getLatestBlock();
  };
```

3.4 Cryptographic Algorithm

Just like bitcoin, your blockchain can use the secure hash algorithm (SHA). For more information see https://en.wikipedia.org/wiki/Secure_Hash_Algorithms. You need not implement your own version of the SHA algorithm; instead you are allowed to use the code from http://www.zedwood.com/article/cpp-sha256-function. Simply copy

the files sha256.cpp, sha256.h and LICENSE.txt to your main directory. From any file you wish to invoke the hash function, you must include the file `sha256.h` and then can use the function `std::string sha256(std::string input)` to compute the hash value for the given input.

3.5 Function Specification for the class Blockchain

3.5.1 addTransaction

Given an input (src, dst, coins), traverse the blockchain to compute the balance associated with the address "src". You can use the function "getBalanceOfAddress" for this purpose. If the balance of the "src" address it is greater than the number of "coins" transferred, add this transaction to the list of pending transactions. If not, print an error message.

3.5.2 isChainValid

Traverse the blockchain from the second block, and verify whether hashcode of previous block matches the hashcode stored in the variable "previousHash". Moreover, check if the first #difficulty characters in all of the hashCodes (except the first block) are 0's. If any of these checks fail, return false.

3.5.3 minePendingTransactions

This function must implement the following steps:

1. Create a new Block with the list of transactions equal to the list of pending transactions, timestamp equal to the current time (using `time(nullptr)` instruction), and the field previousHash equal to the has of the latest block in the chain.
2. Invoke the mineBlock function of the class Block, and pass the difficulty level.
3. After successful return, clear the list of pending transactions.
4. Push the new block to the chain.
5. To reward the minerAddress, add a new transaction to the (now empty) list of pending transactions with src="BFC", dst=minerAddress, and amount=miningReward.

3.5.4 getBalanceOfAddress

Traverse the blockchain block-by-block, for each block traverse its list of transactions, and compute the balance of the user with the given string as the address. This can be done by adding all of the coins received by this address and from that subtracting the coins paid by this address. Make sure that the balance has never gone negative during the history of transactions.

3.6 prettyPrint

Traverse the entire chain and print the contents of every block to the standard output. You can utilize the `toString` method provided in the Transaction class.

3.7 Function Specification for the Class Block

3.7.1 CalculateHash

The function calculateHash first convert all of the variables in the class (except the varaible hash) to a string. A stringstream can be used for this purpose. You have complete freedom to decide the format of this string, except that the string must contain the information about the nonce, the string form of all transactions, the hash of the previous block, and the timestamp. You can implement this functionality using the function "toString()".

Once this string is created, you can invoke the function "sha256(input)" (defined in the files sha256.cpp and sha256.h) to compute its SHA value.

3.7.2 mineBlock

The function mineBlock is the key proof-of-work component. In this function, you need to find the value of the nonce such that the SHA value of the string resulting from combining the data of the block (the nonce, timestamp, previousHash, the string corresponding to all the transactions) has the first #difficulty characters equal to 0. The best way to accomplish this is to brute-force the value of nonce until such a value is encountered. For more information see: `https://en.wikipedia.org/wiki/Proof_of_work`.

3.8 Textual Interface

You may implement a simple text-based interface to your blockchain. A sugggested template is shown below:

```
bool quit = false;
 while (!quit) {
    int option;
    string inputLine;
    cout << "======Main Menu======" << endl;
    cout << "1. Add a transation" << endl;
    cout << "2. Verify Blockchain" << endl;
    cout << "3. Mine Pending Transactions" << endl;
    // Other options
    cout << "n. Quit" << endl;
    getline(cin, inputLine);
```

```
    option = stoi(inputLine);
    switch (option) {
    /.../
    }
}
```

4 Sample Interaction with the Blockchain

Let's consider the following evolution of the blockchain:

- The Blockchain is initialized. This process will create the special genesis block as the first block in the chain, and then will add a first dummy transaction to the blockchain. This transaction can have source="BFC", dst="BFC",amount=0. We will represent such trasnaction tuples for convenience as ("BFC","BFC",0).

```
Blockchain buffCUoin;
```

- The user "ashutosh" is an early adopter. He wishes to mine the first block. The function minePendingTransactions("ashutosh") is invoked, and as a result the user "ashutosh" receives the first miningReward. Feel free to set this value to some default. I assume that it is 10 BFC. The list of pending transaction must have the new transaction ("BFC","ashutosh", 10). Note that this transaction is still not the part of any block.

```
    buffCUoin.minePendingTransactions("ashutosh");
```

- Assume that, next user "asa" is entering the game. The function minePendingTransactions is invoked with the minerAddress "asa". After a successful mining, the list of pending transaction will be set to empty. As a result, for the user "ashutosh", his reward from the previous mining will get finalized to the block, and "asa" will receive the miningReward. A transition ("BFC","asa",10) will be added to the list of pending transactions.

```
    buffCUoin.minePendingTransactions("asa");
```

- At this point, the user “ashutosh” has 10BFC. Suppose, he gives the user “maciej” 2 BFC for some goods/services. The transaction (“ashutosh”,“maciej”,2) gets added to the list of pending transactions.

```
    buffCUoin.addTransaction("ashutosh","maciej", 2);
```

- The user “maciej”sends 5 BFC to the user “asa”. This transaction must get blocked as the user “maciej” does not have sufficient balance (computed by your algorithm getBalanceOfAddress(“maciej”)).

```
    buffCUoin.addTransaction("maciej","asa",5); //Unsuccessful
```

- The user “maciej” next decides to enter the mining game. The user “maciej” mines the pending transaction and as a result gets paid 10 BFC. It means that the list of pending transaction get converted into a block, and a new pending transaction (“BFC”,“maciej”,10) is added to the list of pending transactions.

```
    buffCUoin.minePendingTransactions("asa");
```

- The user “maciej” sends 5 BFC to the user “asa”. This time, the transaction succeeds.

```
    buffCUoin.addTransaction("maciej","asa",5);
```

- Print the blockchain at this step to see if it matches your intuition.

```
    buffCUoin.prettyPrint();
```

5 Project Submission and Grading

5.1 Deliverables

In order for your project to be graded, it must written in C++ and compile with the `g++ -std=c++17` command in a standard Mac or Linux terminal. **You can work individually or in a group of up to two students.** If you work in a group, a single submission is to

be made. The final submission should contain your source code files with the functioning class-based implementation of the `miniGit` program. **The last commit you make to your GitHub repository prior to the deadline will be used for grading.** The following files will be graded:

- `code_1/Block.cpp` - class containing block data and definitions
- `code_1/Blockchain.cpp` - class containing chain data and definitions
- `readme.md` - description of program functionality user interaction; **names of team members**

5.2 Interview Grading

In order to receive credit for your project, you will need to schedule an interview with a TA. You will be required to demonstrate the functionality of the project and explain any questions that might arise. Keep an eye out for an announcement regarding the interview sign-ups.