Computing Structures – Fall 2023
Project 3
Due: Nov 10th 2023 at 11:59 PM

Objective:
The goal of this project is to create a single BlockChain using the given class structures with linked lists.

Description:
Blockchain is a revolutionary technology that has gained widespread attention in recent years. It is a decentralized digital ledger that enables secure and transparent transactions without the need for intermediaries such as banks or governments. The underlying technology is based on a distributed network that is replicated across a network of computers/nodes. This makes blockchain ideal for applications that require transparency, security, and trust in a decentralized environment, such as cryptocurrency, supply chain management, and voting systems. The potential uses of blockchain are vast and have the potential to revolutionize various industries by creating a more secure, efficient, and transparent way of doing business.

In this project, we have three classes that you can see in the boilerplate code. The **transaction class** will hold information about one transaction. An array of transaction class objects will be present in the **block class**. This block class has a block number, a hash value, the current number of transactions and the maximum number of transactions that the block can hold. This maximum number of transactions per block is given in the input.
The **blockchain class** has a list (linked list) of blocks along with the number of blocks. Object of this blockchain class is what we declare in the main function.

Two concepts about Blockchain need to be defined:

1. **Proof-of-Work (PoW)**: Proof-of-Work is a mechanism used to ensure that computational effort is expended before adding a block to the blockchain. This is typically achieved by requiring the block producer (often called the "miner") to find a value that, when hashed, produces a result that meets certain criteria. In our simple implementation, "proof-of-work" is that the hash value of a block, when multiplied by a nonce, must be divisible by **a certain number** without a remainder. This number is defined as MINING_DIFFICULTY in the boilerplate code and will not be changed during implementation.
2. **Hashing**: Each block should have a unique hash based on its content and the hash of the previous block. In our simplified implementation, we will just be determining the hash of the block based solely on it's contents. **The hash of the block will be determined by taking the ASCII value sum of all characters in a block, modulo a large prime number** (provided in the boilerplate code). This will give us a unique integer for each unique block content. The prime number is defined as LARGE_PRIME in the boilerplate code and will not be changed during implementation.

3. **Nonce**: Nonce stands for “Number used once”. It is a value that can be altered during the mining process to produce a hash value that meets a certain condition. The data in a block (like transactions) doesn't change during the mining process. To achieve a hash that meets the required criteria, miners must alter something. This is where the nonce comes in. Miners increment or change the nonce and then recompute the block's hash. They continue doing this until they find a nonce that produces a valid hash. In our implementation, nonce is an int variable which will be initialized with value “1” at the beginning of mining, and be incremented until the condition (defined in the previous definition) is met.

You will read the transaction from the input and insert into the data structure (blockchain). Once a block gets full of transactions, you will mine it using the hash value generated for it. The value of MINING_DIFFICULTY and LARGE_PRIME to be used to find the hash has been provided in the boilerplate code.

Input explanation:

The input file given is in the following format:

```
10              <— number of transactions per block
15              <— total number of transactions

10001 111 222 50 0042:01012021
10002 113 111 80 0142:01012021
10003 114 111 20 0245:01022021
10004 222 111 10 0143:01022021
10005 222 113 10 0144:01022021
10006 212 113 70 0244:01022021
10007 212 113 40 0245:01022021
10008 212 113 70 0334:01022021
10009 212 113 70 0445:01032021
10010 212 113 70 0555:01032021
10011 111 222 50 0142:01032021
10012 113 111 80 0242:01042021
10013 114 111 20 0345:01042021
10014 222 111 10 1143:01042021
10015 222 113 10 2144:01042021
```

The input file starts with the number of transactions per block and then the total number of transactions given in the input file. This is followed by the list of transactions.

Each transaction has the transaction ID, followed by the fromID, toID, amount to be transferred, and finally the time stamp. All these are integers except the last one that is a string and will stay as a string throughout.

Class Definitions:

```
class transaction
{
    public:
    int tID; // transaction id
    int fromID; // source ID
    int toID; // target ID
    int tAmount; // how much is being transfered
    string timeStamp; // time of the transaction read from the input file

    transaction(); // TODO: default constructor
    transaction(int temptID, int tempfromID, int temptoID, int temptAmount, 
string temptimeStamp); // non default constructor - default 100 for values
    void displayTransaction();
};
```

The **transaction** class is the base level class in our structure of user defined classes. Used to store information regarding the individual transactions.

```
class block
{
    public:
    int nonce;
    int hashValue = 0; //Default value of hash for a block. It will be changed 
once the block is mined.
    int blockNumber; // block number of the current block
    int currentNumTransactions; // how many transactions are currently in the 
block
    int maxNumTransactions; // how many maximum transactions can be in the block
    vector<transaction> bTransactions; // vector of transaction objects

    block(); // default constructor
    block(int bNumber, int maxTransactions); // non default constructor
    // insert method to insert a new transaction
    void insertTransaction(transaction newT);
    void computeHash(); //function for determining the hash of the block. Please 
refer to project description.
    void mineBlock();
    void displayBlock();
};
```

The **block** class contains a vector of transactions, in addition to fields like nonce and hashValue which are used in determining Proof-of-Work and hash of the block. The block will only be mined when it is full with transactions.

```
class blockChain
{
    public:
    int currentNumBlocks; // maintain the size of the list/block chain list
    list<block> bChain; // blockchain as a linked list

    blockChain(); // default constructor
    blockChain(int tPerB); // non default constructor takes as argument the
number of transactions allowed per block.
    // insert method to insert new transaction into the block chain - add new
block if existing block is full
    // while inserting new block into list, insert front
    void insertTransaction(transaction newT);
    void displayBlockChain();
};
```

The **blockchain** class is the highest-level container of data for this project. It contains a linked list of blocks.

All fields in all the classes are public, so you don't need to define getter/setter functions in each of these classes.

Output:

The program should output which block the transaction is being inserted into, the nonce number of the mining of the block (when the block gets full of transactions), and it's hash value when the blocks are printed. An output file is provided, corresponding to the input file.

Submission:

Submission will be through *GradeScope*. Your program will be autograded. You may submit how many ever times to check if your program passes the test cases provided. One test case will be released at the beginning and later other test cases will be released while grading. For the autograder to work, the program you upload must be named as ***projec3.cpp***.

Your final submission must contain your source file named project3.cpp. You access GradeScope using the tab on the left in our course page on canvas.

Constraints:

1. None of the projects is a group project. Consulting on programming projects with anyone is strictly forbidden and plagiarism charges will be imposed on students who do not follow this.
2. Please review academic integrity policies in the modules.

How to ask for help:

1. Please attend the help sessions. You can even stay there while you work.
2. Please always have a lookout for announcements on canvas. This class has video lectures that were recorded in the past, so the current information on the course will be mainly through announcements.
3. Email the TA with your ***precise*** questions. Please attach a snapshot of the error and your source file to the email if the question is regarding error(s) in compilation or runtime.