# COL106 : 2021-22 (Semester I)
# Project

Sarang Chaudhari Venkata Koppula Anuj Rai Daman Deep Singh
Harish Yadav

November 1, 2021

## Notations

For any natural number $n$, $[n]$ denotes the set $\{1, 2, \ldots, n\}$.

**Important:**

- Deadline: 06 November, 11:59pm
- Maximum score: 120

## 1 Introduction

In this document, we present our cryptocurrency called DSCoin. Recall, in Lab Module 5, you built a blockchain. Here, we will extend those ideas to have a (nearly complete) cryptocurrency. First, we introduce some basic rules/conventions/terminology:

- Every *coin* is a six digit unique number.
- Every *transaction* has the following information:
  - the coin being transferred
  - the source (that is, the person spending this coin)
  - the destination (that is, the person receiving this coin)
  - some information to indicate when the source received this coin from someone (this will be described in more detail later).

  For simplicity, we assume every transaction consists of exactly one coin.
- A *transaction-block* consists of a set of transactions. Let tr-count denote the number of transactions per block. [1] The transaction-block will also have additional attributes, which will be discussed below.
- A *blockchain* is an authenticated linked list of transaction-blocks.[2]
- *Pending transactions* and *transaction-queue*: All the transactions in the transaction-block are processed transactions. Additionally, we have a transaction-queue which contains pending transactions. Every new transaction is first added to the transaction-queue, and later moved to a transaction-block (and thus added to the blockchain).

[1] For ease of debugging, this will be a small number for our project. Real world blockchains have $> 1000$ transactions per block.

[2] Strictly speaking, this will not be a linked list; we will discuss this later.

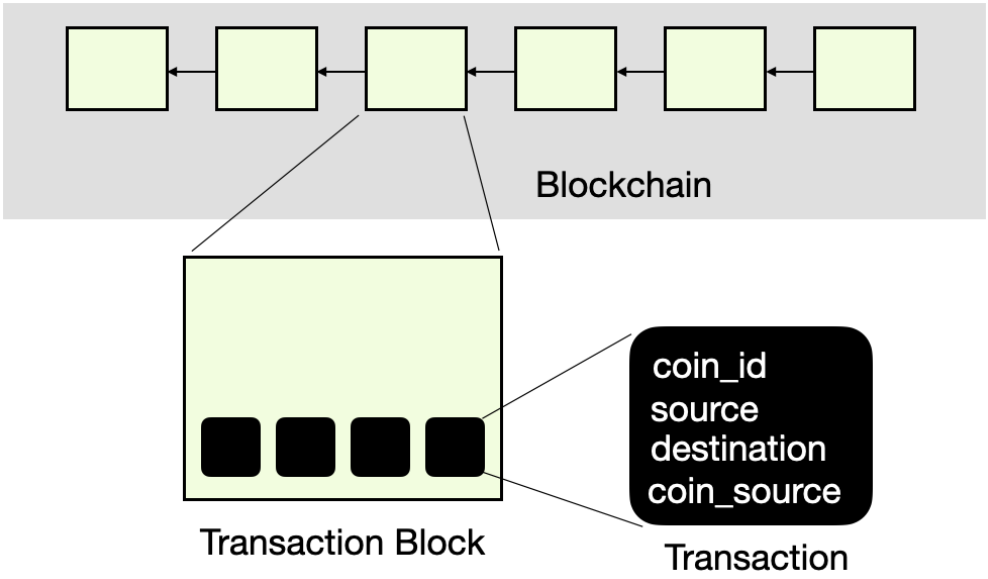


Figure 1: The blockchain is an authenticated list of transaction blocks. Each transaction block consists of transactions (together with additional attributes). Each transaction contains the coin-id, the source of the transaction (that is, the buyer), the destination (the seller) and coin-source (that is, the previous transaction block where the source received this coin).

# 2 DSCoin: Our Cryptocurrency

In our cryptocurrency, any participant can either be a

- *buyer* - someone who wishes to buy an item, and therefore send a coin,
- *seller* - someone who wishes to sell an item, and therefore wishes to receive a coin from the buyer
- *miner* - someone who verifies and approves the transactions

First, we will describe a cryptocurrency `DSCoin_Honest` where the miners behave honestly. In the next section, we will see how to handle malicious miners.

## 2.1 `DSCoin_Honest`: A cryptocoin for honest users

Initially, all participants receive a certain number of coins from the *moderator* (think of the IITD admin – this is the only role of the moderator). All these are added in (possibly separate) transaction-blocks to the blockchain and there are no pending transactions.

After this initial setup, suppose person A wishes to buy an item from person B, and therefore wishes to send one coin to person B (we will refer to person A as the *buyer* and person B as the *seller*). The buyer adds a transaction to the transaction-queue; this transaction must contain the coin-id he/she wishes to spend, the buyer's identification (say IITD entry number), and the seller's identification (again IITD entry number). Additionally, the transaction also *points to* a transaction-block in the blockchain where person A received this coin; we will call this the `coinsrc_block` of this coin. Therefore, any transaction is a 4-tuple : (coin-id, buyer-id, seller-id, `coinsrc_block`). This 4-tuple is added to the queue of pending transactions. Once there sufficiently many transactions in the transaction-queue, a *miner* removes them from this queue, and *mines* a transaction-block (we discuss how this is done in Section 2.1.2). After creating the transaction-block, the miner adds this block to the blockchain.[3] At this point, the buyer (person A) can check that his/her

[3]Note that anyone can be a miner, and therefore anyone can add a transaction-block to the blockchain. This should raise a red-flag: what if a malicious party is a miner, and adds invalid blocks to the blockchain? We will address this in Section 2.2.

transaction is included in the latest transaction block. If so, the buyer sends a *proof of payment* to the seller, and the seller can check this proof. To summarize, here are the steps involved in a transaction from a buyer to a seller:

1. Buyer adds transaction to the transaction-queue.
2. Miner collects many such transactions, mines a transaction-block and adds the block to the blockchain.
3. Once the block is added to the blockchain, the buyer checks that his/her transaction is present in the last block. If so, the buyer computes a 'proof of membership' of this transaction in one of the transaction block, and sends it to the seller.
4. Finally, the seller verifies this 'proof of membership'.

Below, we explain each of these steps in some detail.

#### 2.1.1 Initializing a Coin-Send

Suppose buyer wants to send a coin to a seller. Every buyer/seller has a UID. The buyer creates a new transaction t, where the `coinID`, `Source`, `Destination` and `coinsrc_block` are set appropriately. After this, the buyer adds this transaction to the transaction queue. Additionally, the buyer also maintains his/her own queue of pending transactions (called `in_process_trans`), and the buyer adds the transaction to this queue.

#### 2.1.2 Mining a Transaction Block: Honest setting

First, we discuss the structure of a transaction block, and then discuss how it is mined. A transaction-block consists of the following attributes:

- tr-count: the number of transactions in the block. We will assume this is a power of 2.
- `trarray`: an array of transactions.
- `trsummary`: a 64-character summary of the entire transaction-array. This is computed using a Merkle tree on the `trarray`. [4]
- `Tree`: the Merkle tree on `trarray`.
- `nonce`: a 10-digit string that is used to compute the `dgst`.
- `dgst`: a 64-character string, obtained by computing the CRF on previous digest, transaction-summary and a 10-digit string called nonce, separated by #. The nonce must be such that the first four characters of `dgst` are all zeroes.

The job of the miner is to create a transaction-block consisting of tr-count *valid* transactions (that is, none of the transaction should be a *double spending*). More formally, we say that a transaction $t$ is invalid if any of the following holds true:

- the `coinsrc_block` of $t$ does not contain any transaction $t'$ such that $t'$.`coinID` = $t$.`coinID` and $t$.`Source` = $t'$.`Destination`.
- the above check passes, but this coin has been spent in one of the future transaction-blocks, or is present multiple times in the transaction queue.

The miner collects tr-count - 1 number of valid transactions from the `TransactionQueue`. The miner also gets a *reward* for mining this block. This reward, in our cryptocurrency, is one coin. The `coinID` of this coin will be the smallest six-digit coin-id ($\geq 100000$) that is available.[5] The miner creates a transaction with this `coinID`, sets the `Source` to be `null`, `Destination` is the miner, and `coinsrc_block` is set to `null`. This new coin belongs to the miner, and can be spent by the miner for future transactions. Therefore, the miner has tr-count number of transactions, which are included in the transaction block as follows:

[4] Strictly speaking, for each transaction $t$ in `trarray`, we obtain a string by concatenating the attributes of $t$. The Merkle tree is computed on these strings.

[5] Ideally, this should be a random six digit number. For ease of testing, we opted for a deterministic approach - we will maintain the smallest six-digit number that is not used yet (stored as an attribute of the blockchain), and use that as the coin-id. The miner will also increment this attribute of the blockchain after mining the block.

1. adds the transactions to the transaction array `trarray`.
2. computes Merkle tree on the transaction array. Every node in the Merkle tree has a string attribute called `val`. For a leaf node corresponding to transaction, the `val` is simply CRF applied on a concatenation of the coin-id, source id, destination id and `dgst` of the `coinsrc_block` corresponding to this transaction (separated by #).[6] For any intermediate node, it is computed by applying the CRF on the concatenation of left child's `val` and right child's `val` (separated by #).
3. finds a 10-digit string `nonce` such that CRF applied on the previous block's digest,[7] the Merkle tree's root's `val` (i.e., also the current block's `trsummary`) and `nonce` (separated by #) outputs a string with first four characters being 0. This output is the new block's digest. Such a `nonce` can be found by sequentially searching over the space of all 10-digit strings.

At this point, the miner has computed all the attributes for the new transaction block. The miner simply adds this transaction block to the blockchain.

#### 2.1.3 Finalizing a Coin-Send

Once a transaction $t$ is added to a transaction block in the blockchain, the buyer can convince a seller that he/she has sent a coin to the seller. Suppose the transaction is included in a block $b_0$, and there are $r$ blocks $b_1, b_2, \ldots, b_r$ in the chain after $b_0$ (that is, $b_r$ is the last block in the blockchain, $b_{r-1}$ the previous block, and so on). The buyer first computes a sibling-coupled path to the root of $b_0$'s Merkle tree. The buyer sends the following information to the seller:

- the sibling-coupled-path-to-root corresponding to transaction $t$
- the `dgst` of $b_0$.`previous`.
- the `dgst`, `nonce` and `trsummary` for each $b_0$, $b_1, \ldots, b_k$.

We will refer to this entire tuple as the *proof of transaction*. The buyer also removes this transaction from his/her own `in_process_trans` queue.

#### 2.1.4 Verification of Transaction by the Seller

Suppose a seller is the destination for a transaction $t$ in block $b_0$, and suppose there are $k$ blocks $b_1, \ldots, b_k$ in the blockchain after $b_0$. The seller receives a sibling-coupled-path-to-root, the `dgst` of $b_0$.`previous`, the $(\texttt{dgst}_i, \texttt{nonce}_i, \texttt{trsummary}_i)$ pairs for each of the blocks $b_i$, $i \in [0, k]$. The seller checks the following:

- first checks that he/she is indeed the destination for the transaction $t$.
- next checks the sibling-coupled-path-to-root, and checks that the final value in the sibling-coupled-path-to-root is equal to $\texttt{trsummary}_0$.
- checks that each of the $\texttt{dgst}_i$ strings are valid (for $i \in [0, k]$) – the seller checks that the first four characters of $\texttt{dgst}_i$ are 0, and that $\texttt{dgst}_i$ is correctly computed using $\texttt{dgst}_{i-1}$, $\texttt{nonce}_i$ and $\texttt{trsummary}_i$.[8]
- finally checks that $\texttt{dgst}_k$ matches the `dgst` of the last block in the blockchain.

### 2.2 `DSCoin_Malicious`: Handling Malicious Miners

The solution described in the previous section works in the setting where all miners are honest. [9] However, one of the prime advantages of a cryptocurrency is the decentralized aspect, and in this setting, we cannot assume that the miners are honest. Indeed, it is possible that a buyer (source) adds an invalid transaction to the `pendingTransactions` queue, and then the same buyer mines a block that includes the invalid

[6] If any of these values are `null`, then you should use the string "*Genesis*".
[7] If this is the first block, then you should use the start string instead.
[8] For $i = 0$, $\texttt{dgst}_{i-1}$ is the `dgst` of $b_0$.`previous`.
[9] For instance, if the miners were appointed by the moderator, and being a miner is a *Position of Responsibility*, then we can assume that all miners are honest.

transaction. Or maybe the malicious buyer has a miner friend who includes the invalid transaction in the transaction block.

Interestingly, malicious miners are handled using a clever mix of incentive-engineering and data structures (and no additional cryptography involved here). First, the blockchain is generalized – instead of having a linked list, we have a tree-like structure, where we store all *leaf blocks* of the tree, and for each block, we have a pointer to the `previous` block as before. For instance, in Figure 2, we have three leaf blocks – blocks numbered 9, 11, 12. Note that anyone can identify these maliciously mined blocks – one only needs to check the validity of each transaction in the block, check if the Merkle tree is computed correctly and finally check that the digest is computed correctly. We define a transaction block to be *valid* if the `dgst` is correct, all transactions in the block are valid and the Merkle tree is computed properly. You can assume that the first few blocks (created by the moderator) are valid blocks. The *longest valid chain* is defined to be the longest sequence of blocks, starting with the first block, consisting of only valid blocks. If a honest miner is mining a new block, then the miner finds the longest valid chain, and attaches the new transaction to this valid chain. For instance, in Figure 2, the longest valid chain is the sequence $(1, 2, 3, 4, 6)$, and therefore, the new block should have 6 as its previous block. If 6 was also an invalid block, then the longest valid chain terminates at 4. Finally, if 5 was a valid block, then the longest valid chain would be terminating at 12. [10]

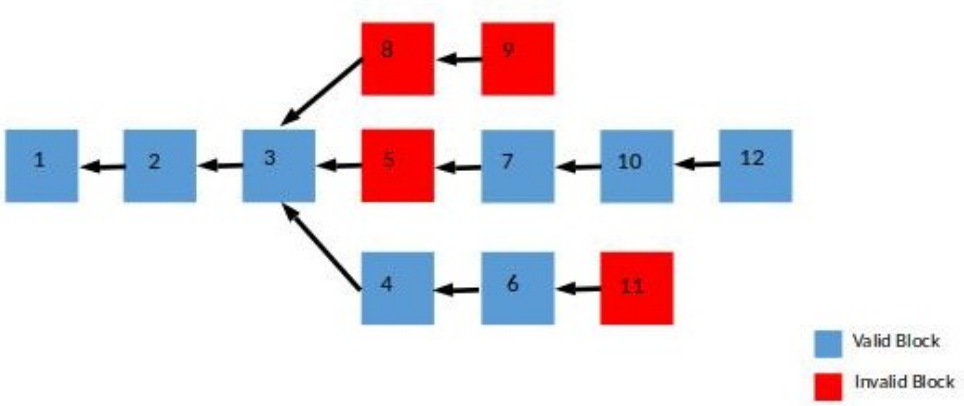


Figure 2: The malicious blockchain has a tree-like structure. Each block has a pointer to the previous block, and we store all leaf-blocks. In this figure, there are three leaf blocks, and four invalid blocks (shown in red).

### 2.2.1 How Does the Above Deter Malicious Mining

If the majority of miners are honest, then eventually, the longest chain in the blockchain will consist of only valid transactions, and malicious blocks are thus *discarded*. You can assume that the cryptocurrency software automatically checks that the digest is correctly computed, and has an appropriate number of zeroes. But the other validity check – ensuring that there is no double spending – is done by the fellow miners.

**The role of the nonce, and how it ensures that there are no invalid transactions in the blockchain:** In this assignment, we want the first four characters of the `dgst` to be zeroes, but in real-world cryptocurrencies, many more initial characters need to be zeroes.[11] Therefore, computing the nonce is the most expensive operation in the computation of a new block, and requires a lot of computational resources. The reward coin given to the miner is the incentive for the miner to behave honestly. If a miner has invested so much computational resources in computing the correct `nonce` and `dgst`, then any reasonable miner will

[10] For the test cases in this assignment, you can assume that there will be a unique longest valid chain.

[11] In cryptocurrencies, one does not need to output the smallest nonce; any nonce is allowed. However, finding *any* nonce seems to be as hard as finding the smallest nonce.

also ensure that the transactions in the transaction block are valid (otherwise, if the miner's block is invalid, then all the computing effort goes waste).

## 2.3 Conclusion

The cryptocurrency described above is close to real-world cryptocurrencies (such as Bitcoin). Here are some differences:

- Here, we assumed every person has a UID (the kerberosID). In Bitcoin, every person chooses a signing/verification key, and the verification key is the person's UID. Whenever someone mines a block, he/she also computes a signature on the block. [12]
- Here, we assumed that every transaction consists of exactly one coin spending, and every miner receives one coin as reward. In real life, a transaction allows person A to send any number of coins to person B. Moreover, person A can also include a 'reward' for the miner. This incentivises the miner to include person A's transaction in the transaction block. In this scenario, how should you implement the pending transactions queue? And how would you modify the mining procedure?
- Bitcoin uses a cool stack-based scripting language. Every transaction includes a small program in this programming language. You can read more about it here.

Finally, if you are interested in learning more about cryptocurrencies, check out this excellent book.

[12] Earlier, we intended to include this in the project, and hence discussed signatures in LM1. However, given that the project has enough components already, we skipped this part.

# 3 Assignment Questions

You are given the following classes. For each class, you must implement the missing methods. You are allowed to add additional attributes if needed.

## 3.1 Transaction

`public class Transaction:` This class has the following attributes:

- `public String coinID`: a 6-digit long string to uniquely identify a cryptocoin. You can assume this string consists of only digits, and these digits form a number $\geq 100000$
- `public Members Source`: the source member of the transaction (i.e. the buyer)
- `public Members Destination`: the destination member of the transaction (i.e. the seller)
- `public TransactionBlock coinsrc_block`: the `TransactionBlock` where the buyer received the coin in this transaction

## 3.2 Transaction Queue

`public class TransactionQueue:` This class has the following attributes:

- `public Transaction firstTransaction`: a reference to the first transaction pushed into the queue
- `public Transaction lastTransaction`: a reference to the latest transaction pushed into the queue
- `public int numTransactions`: number of transactions stored in the queue

This class has the following methods to be implemented:

- `public void AddTransactions` (`Transaction` transaction): Enqueue a new transaction to the queue and increase `numTransactions` by one.
- `public Transaction RemoveTransaction` () `throws EmptyQueueException`: Dequeue the first transaction added to the queue, return it after decreasing `numTransactions` by one. If the queue is empty, throw `EmptyQueueException` exception.
- `public int size`(): Returns the number of transactions stored in the queue

**Exercise 1. (10 points)**

Implement `AddTransactions`(`Transaction` $t$), `RemoveTransaction`() and `size`().

You must implement these methods in `TransactionQueue.java`.

*Grading policy:*

- -3 for missing `AddTransactions`
- -3 for missing `RemoveTransaction`
- -2 for not handling exception
- -2 for missing `size`

### 3.3 Merkle Tree and Related Classes

`public class Pair<A,B>` : Objects of this class would be used to store each element in the sequence *Sibling-Coupled Path to Root*. The attributes of the class are:

- `public` A `First`: the first element stored in the object.
- `public` B `Second`: the second element stored in the object.

`public class TreeNode`: This class has the following attributes:

- `public TreeNode parent`: pointer to the parent node
- `public TreeNode left`: pointer to the left child node
- `public TreeNode right`: pointer to the right child node
- `public String val`: the value contained in this node

`public class MerkleTree`: This class has the following attributes:

- `public TreeNode rootnode`: pointer to the root node of the Merkle Tree

This class has the following methods (already implemented):

- `public String Build` (`Transaction[] trarray`) : Accepts an array of transactions and builds a Merkle Tree using it. It returns the `rootnode` value.

For this class, you're free to add required attributes and methods from previous modules.

### 3.4 Transaction Block

`public class TransactionBlock:` This class has the following attributes:

- `public Transaction[] trarray`: an array of objects of class `Transaction`. You can assume that the size of this array is a power of 2 (and this size is greater than 1).
- `public TransactionBlock previous`: pointer to the previous `TransactionBlock` in the chain
- `public MerkleTree Tree`: A Merkle Tree built on `trarray`, where key of each leaf node = CRF(`coinID`+"#"+`Source`.UID +"#"+`Destination`.UID +"#"+`coinsrc_block.dgst`). If the `Source` or `coinsrc_block` are `null`, then use "Genesis".
- `public String trsummary`: value of the root node of the Merkle Tree
- `public String nonce`: a 10-digit string that is used to compute the `dgst`.

  How to compute `nonce`: start with the 10-digit string "1000000001" $s$ (in sequential manner), and check if the first four characters of `CRF.Fn`(`prev_dgst`+"#"+`trsummary`+"#"+ $s$) are all "0". If so, then this is the `nonce`; else repeat with the next 10-digit string. (Please note that, after "1000000009", the next string should be "1000000010", i.e. the next string should in the manner integers are incremented)
- `public String dgst`: a digest string computed by the hash of the previous digest, `trsummary` and `nonce`

This class has the following methods to be implemented:

- `TransactionBlock` (`Transaction[]` $t$ ): Constructor method of the class that takes an array of transactions as input (the size of this array is `tr-count`) and creates a block by performing the following steps:

- sets the elements of `trarray` using the input array $t$.
- sets `previous` to be `null`.
- computes the `MerkleTree Tree`
- sets `trsummary` to be the string stored at root of `Tree`.
- sets `dgst= null`

**Note**: The `previous` and `dgst` attributes are to be carefully set during insertion of the block in the block chain.

- `public boolean checkTransaction` (`Transaction` $t$ ): Checks whether the given transaction is valid by testing for *double spending* by traversing the block chain. The method takes as input a transaction $t$ and does the following:

  - checks that the `coinsrc_block` of $t$ contains the transaction $t'$ such that $t'$.`coinID` = $t$.`coinID` and $t'$.`Destination`.UID = $t$.`Source`.UID. [13]
  - checks that $t$.`coinID` has not been spent in any of the transaction blocks between $t$.`coinsrc_block` and the current transaction block.

  Basically, the idea is to start with the current transaction block, traverse all the way back to the `coinsrc_block` using the `previous` pointer until you reach $t$.`coinsrc_block`, and check that none of these intermediate blocks are an instance of *double spending*. [14]

**Exercise 2. (10 points)**

Implement `checkTransaction`(`Transaction` t).

You must implement this method in `TransactionBlock.java`.

*Grading policy:*

- -2 for not checking if there exists a $t'$ in $t$.`coinsrc_block` such that $t'$.`coinID` = $t$.`coinID` and $t'$.`Destination`.UID = $t$.`Source`.UID.
- -8 for not checking if $t$.`coinID` is contained in any of the intermediate blocks between $t$.`coinsrc_block` and the current block

**Exercise 3. (15 points)**

Implement constructor `TransactionBlock`(`Transaction`[] t). Note that the `previous` attribute is set to `null`.

You must implement this method in `TransactionBlock.java`.

*Grading policy:*

- -15 for initializing any of the attributes incorrectly. Note that if the input array $t$ is modified after calling `TransactionBlock`(), the attribute `trarray` **must not change**.

[13] If the `coinsrc_block` is null, then one should check that either this transaction was generated by the Moderator (by checking the Source.UID), or this is a 'reward transaction'. However, in our test cases for `checkTransaction`, we will not include invalid transactions where `coinsrc_block` is `null`. Therefore, if `coinsrc_block` is `null`, you can declare that the transaction is valid.

[14] Ideally, if you traverse the block chain and do not find $t$.`coinsrc_block`, then you should throw an exception. Here, for the sake of simplicity, you can assume that `coinsrc_block` block will be present in all the test cases.

## 3.5 Blockchain

### 3.5.1 Blockchain_Honest

`public class BlockChain_Honest`: This class has the following attributes:

- `public int tr_count`: the number of transactions in each `TransactionBlock` of the blockchain
- `public static final String` start_string: a string representing the starting `dgst` for all `BlockChain` objects. **This string should be equal to "DSCoin".**
- `public TransactionBlock lastBlock`: represents the last block in the blockchain.

This class has the following methods to be implemented -

- `public void InsertBlock_Honest (TransactionBlock newBlock)`: This method will be used to insert `newBlock` to the end of the blockchain. The method does the following:
  - sets `newBlock.dgst`= `CRF.Fn(lastBlock.dgst`+ "#" + `newBlock.trsummary`+ "#"+ `newBlock.nonce`).
  - the `nonce` computation is to be done while computing the `newBlock.dgst` following the constraints for `dgst` of a block.
  - sets the `previous` node of `newBlock`, and `lastBlock` of the blockchain appropriately.

**Exercise 4. (10 points)**

Implement `InsertBlock_Honest(TransactionBlock newBlock)`.

You must implement these methods in `BlockChain_Honest.java`.

*Grading policy:*

- -5 for incorrect nonce.
- -5 for incorrectly setting `previous` of `newBlock`, `dgst` of `newBlock` or `lastBlock` of the blockchain.

### 3.5.2 Blockchain_Malicious

`public class BlockChain_Malicious`: This class has the following attributes:

- `public int tr_count`: the number of transactions in each `TransactionBlock` of the blockchain
- `public static final String` start_string: a string representing the starting `dgst` for all `BlockChain` objects. **This string should be equal to "DSCoin".**
- `public TransactionBlock[] lastBlocksList`: The transaction blocks in this list represent the last blocks in the blockchain. For instance, in Figure 2, this corresponds to the blocks $\mathsf{tB}_9, \mathsf{tB}_{12}$ and $\mathsf{tB}_{11}$.

This class has the following methods to be implemented -

- `public static boolean checkTransactionBlock(TransactionBlock` tB): This static method takes as input a transaction block, and checks if it is valid. A transaction block is valid if the following conditions hold:
  - the `dgst` of tB begins with "0000", and is equal to `CRF.Fn`(tB.`previous.dgst`+ "#" + tB.`trsummary`+ "#" + tB.`nonce`), [15]

[15] If tB.`previous` is `null`, then tB.`previous.dgst` should be the start_string.

- tB.`trsummary` is correctly computed using tB.`trarray`
- every transaction in tB.`trarray` is valid. [16]

if previous is `null`, then use start_string as the previous digest.

- `public TransactionBlock FindLongestValidChain()`: Returns the last transaction block of the longest chain (in blockchain) in which all the transaction blocks are valid. For this, you have to consider all the chains with their respective last blocks in the list `lastBlocksList`. Traverse each chain up to a valid block, say *tb*, before which all the blocks are valid in that chain. This chain up to *tb* is a *valid* chain Lastly, return the last block of the longest *valid* chain. For better understanding, you may refer to the Figure 2.

- `public void InsertBlock_Malicious (TransactionBlock newBlock)`: This method will be used by to insert `newBlock` to the end of the longest valid chain in the blockchain. The method does the following:

  - finds the last block of the longest valid chain in the block chain (using `FindLongestValidChain` method). Let `lastBlock` be this block.
  - computes the first 10-digit lexicographic string $s$ such that `CRF.Fn(lastBlock.dgst+` "#" + `newBlock.trsummary+` "#"+ $s$) begins with "0000", and sets `newBlock.nonce`= $s$.
  - sets `newBlock.dgst= CRF.Fn(lastBlock.dgst+` "#" + `newBlock.trsummary+` "#"+`newBlock.nonce`).
  - sets the `previous` node of `newBlock`, and `lastBlock` of the blockchain appropriately.

**Exercise 5. (10 points)**

Implement `checkTransactionBlock(TransactionBlock` tB).

You must implement this method in `BlockChain_Malicious.java`.

*Grading policy:*

- -3 if digest or nonce are incorrect.
- -5 if `trsummary` is incorrectly computed.
- -2 if any of the transactions are invalid.

**Exercise 6. (20 points)**

Implement `InsertBlock_Malicious(TransactionBlock newBlock)`.

You must implement these methods in `BlockChain_Malicious.java`.

*Grading policy:*

- -4 for not picking the longest chain, given only valid blocks
- -5 for not picking the longest chain, given that only a subset of the `lastBlocksList` blocks are invalid
- -8 for not picking the longest chain, in the general case (where we can have an invalid block anywhere)
- -3 for incorrectly setting `previous` of `newBlock`, or `lastBlocksList` of the blockchain.

[16] In our test cases, we will not check if there are some transactions that have `coinsrc_block` as `null`, but they are neither moderator-generated nor reward-transactions. Therefore, for the purpose of this project, if the transaction's `coinsrc_block` is `null`, then you can assume this transaction is valid.

## 3.6 DSCoin

### 3.6.1 DSCoin_Honest

`public class DSCoin_Honest:` This class has the following attributes:

- `public TransactionQueue pendingTransactions`: a simple queue of all pending transactions in DSCoin
- `public BlockChain_Honest` bChain: the blockchain where all transactions of DSCoin are stored
- `public Members[]` memberlist: list of all members in the DSCoin system.
- `public String latestCoinID`

### 3.6.2 DSCoin_Malicious

`public class DSCoin_Malicious:` This class has the following attributes:

- `public TransactionQueue pendingTransactions`: a simple queue of all pending transactions in DSCoin
- `public BlockChain_Malicious` bChain: the blockchain where all transactions of DSCoin are stored
- `public Members[]` memberlist: list of all members in the DSCoin system.
- `public String latestCoinID`

## 3.7 Members

`public class Members:` This class has the following attributes:

- `public String` UID: a unique string representing the member
- `public List<Pair<String,TransactionBlock>> mycoins`: a list of all coins id of the member along with the `TransactionBlock` from where the member received the coin. This list must be arranged in increasing order of coin-id.
- `public Transaction[] in_process_trans`: a list of all transactions (sending coins) that have been initiated but not finalized yet.

This class has the following methods to be implemented:

- `public void initiateCoinsend (String destUID, DSCoin_Honest DSobj)`: Picks the first (`coinID`, `coinsrc_block`) tuple from the `mycoins` list (that is, the pair with the smallest `coinID`), and removes it. Next, it creates a `Transaction` object `tobj`, adds it to its own `in_process_trans` list. Finally, it adds the transaction to the `pendingTransactions` list of `DSobj`.
- `public Pair<List<Pair<String,String>>,List<Pair<String,String>>> finalizeCoinsend(Transaction` tobj, `DSCoin_Honest` DSobj) throws `MissingTransactionException`: This method takes as input a transaction tobj, and an object DSobj of the `DSCoin_Honest` class. The method first finds the transaction block in the blockchain containing this transaction. Starting with the last block in the blockchain, check if the transaction is present in the block (this can be done by performing a sequential search in the `trarray` of this block). If transaction is not found in the last block, proceed to the previous block, and repeat. Let tB denote the transaction block containing the transaction tobj.

  First, compute the sibling-coupled-path-to-root for proving the membership of transaction tobj in the Merkle tree of transaction block tB. Recall, the sibling-coupled-path-to-root is a list of pairs of strings, where each pair stores a node's value and its sibling node's value.

Next, the method outputs another list of pairs of strings. Suppose there are $k$ transaction blocks after tB. The method returns a list having $k+2$ pairs. The first pair is (tB.`previous.dgst`, `null`). The next $k+1$ pairs are pairs of the form ($t_i$.`dgst`, $t_i$.`previous.dgst` + "#" + $t_i$.`trsummary` + "#" + $t_i$.`nonce`), where $t_0$ is the transaction block tB, and $t_i$ is the $i^{\text{th}}$ transaction block after tB.

The method finally deletes this transaction from list `in_process_trans`, returns these two lists of pairs, **and adds the sent coin to the Destination's `mycoins` list**. See example in supporting documents.

The method throws an exception "MissingTransactionException" if the concerned transaction not found in the blockchain.

- `public void MineCoin` (`DSCoin_Honest` DSobj): This method removes the first (DSobj.bChain.tr-count - 1) number of *valid* transactions from `pendingTransactions`. The miner also ensures that there are no two transactions with the same coin-id among these valid transactions (if there are two or more transactions with the same coin-id, then the miner keeps only the first one, and dequeues additional transactions from `TransactionQueue` so that there are tr-count - 1 number of valid transactions in the `trarray`).

  Next, it adds an extra transaction 'minerRewardTransaction' at the end for the miner. To create this extra transaction, the method uses the `latestCoinID` attribute in `DSCoin_Honest` class. Specifics of the 'minerRewardTransaction' are:

  - `coinID`: `newCoinID` (obtained by incrementing `latestCoinID`)
  - `Source`: `null`
  - `Destination`: miner
  - `coinsrc_block`: `null`

  Using these tr-count number of transactions, the method first creates a block tB, and then inserts it at the end of the blockchain.

  Finally, the method adds (`newCoinID`, tB) to the miner's `mycoins`, and updates `latestCoinID`.

- `public void MineCoin` (`DSCoin_Malicious` DSobj): The malicious `MineCoin` method is similar to the honest one, except that the block needs to be inserted at the appropriate position (after computing `FindLongestValidChain`).

**Exercise 7. (15 points)**

Implement the methods `initiateCoinsend` and `finalizeCoinsend` in `Members.java`.

*Grading policy:*

- `initiateCoinsend`:
  - -1 for not removing the `coinID` from `mycoins` list.
  - -2 for not adding it to the `in_process_trans` list.
  - -2 for not adding the transaction to the `pendingTransactions` list of `DSobj`.
- `finalizeCoinsend`:
  - -2 for not removing the `Transaction` from `in_process_trans` list.
  - -4 for not returning the correct sibling coupled path to root
  - -4 for incorrect second list of pairs

**Exercise 8. (20 points)**

Implement the method `MineCoin` in `Members.java` for both the honest and malicious setting.

*Grading policy:*

- -6 for adding some invalid transaction(s) to the transaction block.
- -2 for not removing valid `transactions` from `pendingTransactions` list.
- -2 for not adding an extra miner reward transaction, or not incrementing the `latestCoinID`
- -1 for not adding a new coin to miner's `mycoins` list
- -4 for not creating a new `Block`, or incorrectly creating new block
- -5 for not adding the newly created `Block` at the correct position

## 3.8 Moderator

`public class` Moderator: This class has the following methods to be implemented:

- `public void initializeDSCoin` (`DSCoin_Honest DSobj`, `int coinCount`): Given an object of class `DSCoin_Honest` this method allots a total of `coinCount` coins to all members of `DSobj` in a round-robin fashion **sequentially** starting with the `coinID` **"100000"**, adds the newly allotted coins to the `mycoins` list of all members and sets the `latestCoinID` attribute of `DSobj` as the last allotted coin. (Please refer to the supporting files to understand the way coins are distributed to the members).

  The moderator creates a total of `coinCount` transactions for distributing each such coin, creates a total of `coinCount` / `tr-count` number of transaction blocks with each having `tr-count` [17] number of transactions, and inserts them in the block chain. The source of each transaction must be "Moderator",[18] the destination of each transaction must be appropriately set, and the `coinsrc_block` must be `null`. Finally, some attributes of `DSobj` and memberlist[i] need to be updated by the Moderator.

  **Note:** The coinIDs, thus, should go like: "100000", "100001", "100002"... so on.

- `public void initializeDSCoin` (`DSCoin_Malicious DSobj`, `int coinCount`): same as the `DSCoin_Honest` case. Note that the moderator is assumed to be honest, and therefore the `lastBlocksList` consists of only one block at the end of this initialization.

[17] Use DSobj.bchain.`tr-count` here. You can assume that this will be initialized appropriately in the driver code.
[18] For the moderator, create an object of `Members` class, and set its UID as "Moderator"

**Exercise 9. (10 points)**

Implement `initializeDSCoin(DSCoin_Honest DSobj, int coinCount)` and `initializeDSCoin(DSCoin_Malicious DSobj, int coinCount)`. Note that the malicious one will be almost similar to the honest one.

You must implement these methods in `Moderator.java`.

*Grading policy:*

- -3 if each member doesn't get `coinCount` coins in round robin fashion. If there are $k$ Members, then each member receives `coinCount`$/k$ number of coins. The coin numbers start with "100000", and all these coins must be present in the transaction blocks.
- -3 if the transaction blocks have incorrect attributes (that is, if the value/digest of any block is incorrect, or if the previous pointer is incorrect).
- -4 if the attributes of `DSobj` and members array are not appropriately updated. (For each $i$, some attributes of memberlist[i] need to be updated by the Moderator. Similarly, the moderator must update some attributes of `DSobj`.)

You must implement this method in `Moderator.java`.

# 4 Instructions and FAQs

- You can add import statements to the files, and attributes to classes as needed.
- You can assume all arrays have size at most 100.
- **Submission instructions for project:** You must create a directory whose name is your kerberos id, followed by "Project" (for example, if your entry number is "xyz120100", then the folder name should be "xyz120100Project"). The directory must contain all the files included in supporting code.

  Finally, compress this directory, and upload it on Moodle. The file name should be `kerberosidProject.zip` (that is, `xyz120100Project.zip` in the above example).


# Acknowledgements

This is version 3 of the document. Many thanks to the following students for identifying errors in the previous version(s) of supporting code/pdf : Chirag, Prateek Mishra, Aditya Agrawal, Aniruddha Deb, Adit Malhotra, Sachit Sachdeva, Chinmay Mittal, Shamoon Rashid, Vaibhav Saha, Ekansh Singh, Tushar Sethi, Tanish Gupta, Divyanshu Agarwal, Yash Pravin, Arin Kedia, Aryan Dua, Yeruva Hitesh Reddy, Nikhil Agarwal, Manthan Dalmia, Divyansh Mittal, Vansh Kachhwal.

If you find any other mistakes, please send an email to `kvenkata@iitd.ac.in`, `cs1180381@iitd.ac.in` and `csy217544@iitd.ac.in`. Thanks!