Blockchain Demo - Mining in Centralized Network using Proof of Work

By Ng Yiu Wai, January 2019

Contents

1. Introduction

Bitcoin, the most well-known blockchain network, is described as

> *"A purely peer-to-peer version of electronic cash would allow online payments to be sent directly from one party to another without going through a financial institution. Digital signatures provide part of the solution, but the main benefits are lost if a trusted third party is still required to prevent double-spending. We propose a solution to the double-spending problem using a peer-to-peer network.* ***The network timestamps transactions by hashing them into an ongoing chain of hash-based proof-of-work, forming a record that cannot be changed without redoing the proof-of-work. The longest chain not only serves as proof of the sequence of events witnessed, but proof that it came from the largest pool of CPU power****. As long as a majority of CPU power is controlled by nodes that are not cooperating to attack the network, they'll generate the longest chain and outpace attackers. The network itself requires minimal structure. Messages are broadcast on a best effort basis, and nodes can leave and rejoin the network at will, accepting the longest proof-of-work chain as proof of what happened while they were gone."* [1]

In this project, I am going to build a proof-of-work blockchain network which only retains below two characteristics of Bitcoin.

i) Blockchain is a chain of blocks connected by hash-based proof-of-work.
ii) Miner who has highest CPU power could add new blocks to the blockchain.

Below is a comparison between Bitcoin and this project.

Bitcoin	**This project**
1. Transactions data, which are verified using digital signature, are packed into blocks.	1. Arbitrary strings data are packed into blocks.
2. Data are packed into blocks using merkle tree root hash.	2. Data are packed into blocks using merkle tree root hash.
3. Each block could be represented by a byte stream under a pre-defined protocol.	3. Each block could be represented by a byte stream under a pre-defined protocol.
4. Blocks are chained using hash-based proof-of-work. It is nearly impossible to change data that are already added into the blockchain.	4. Blocks are chained using hash-based proof-of-work. It is nearly impossible to change data that are already added into the blockchain.
5. New blocks are broadcasted on a best effort basis in a decentralized network. Usually the Miner who has highest CPU power could add new blocks to the blockchain.	5. New blocks are broadcasted in a centralized network. The Miner who has highest CPU power could add new blocks to the blockchain.

Step 1 & 2	Packing data into blocks	is discussed in Section 3	"*System Design – Data & Merkle Tree Root*"
Step 3 & 4	Mining a block	is discussed in Section 4	"*System Design – Block & Proof-of-Work*"
Step 5	Broadcasting	is discussed in Section 5	"*System Design – Blockchain in Centralized Network*"

The project is written in GO language. Each port of localhost represents one node. The other characteristics of bitcoin, e.g. digital signatures, peer-to-peer network, broadcasting on a best effort basis, blockchain forks and etc, are not discussed in this project.

2. Overview of System

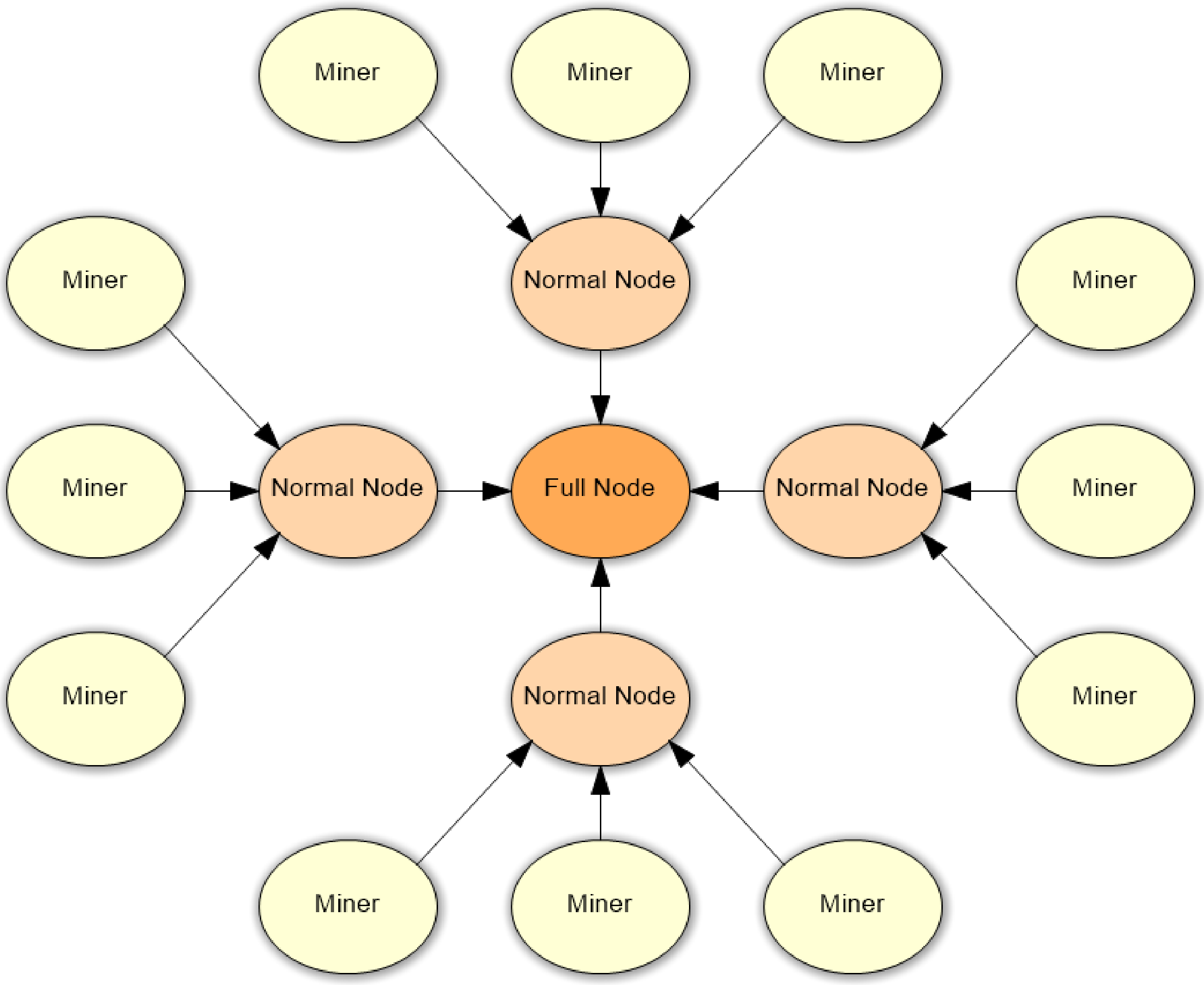
The blockchain network in this project is a centralized multi-layer server-client network. Three types of application are involved.

- Miner
- Normal Node (as a server)
- Full Node (as a server)

End User could input arbitrary strings through command line interface provided by Miner. The strings data will be saved in the blockchain if the Miner mines successfully. “Successfully” means

1) Miner create a new block faster that other Miners within the network
2) Then the new block is sent to Full Node, called “broadcasting”
3) Once the block reaches the Full Node, Full Node will check is the block valid. If yes, the new block will be saved blockchain.

Usually the Miner with highest CPU power could successfully mined.



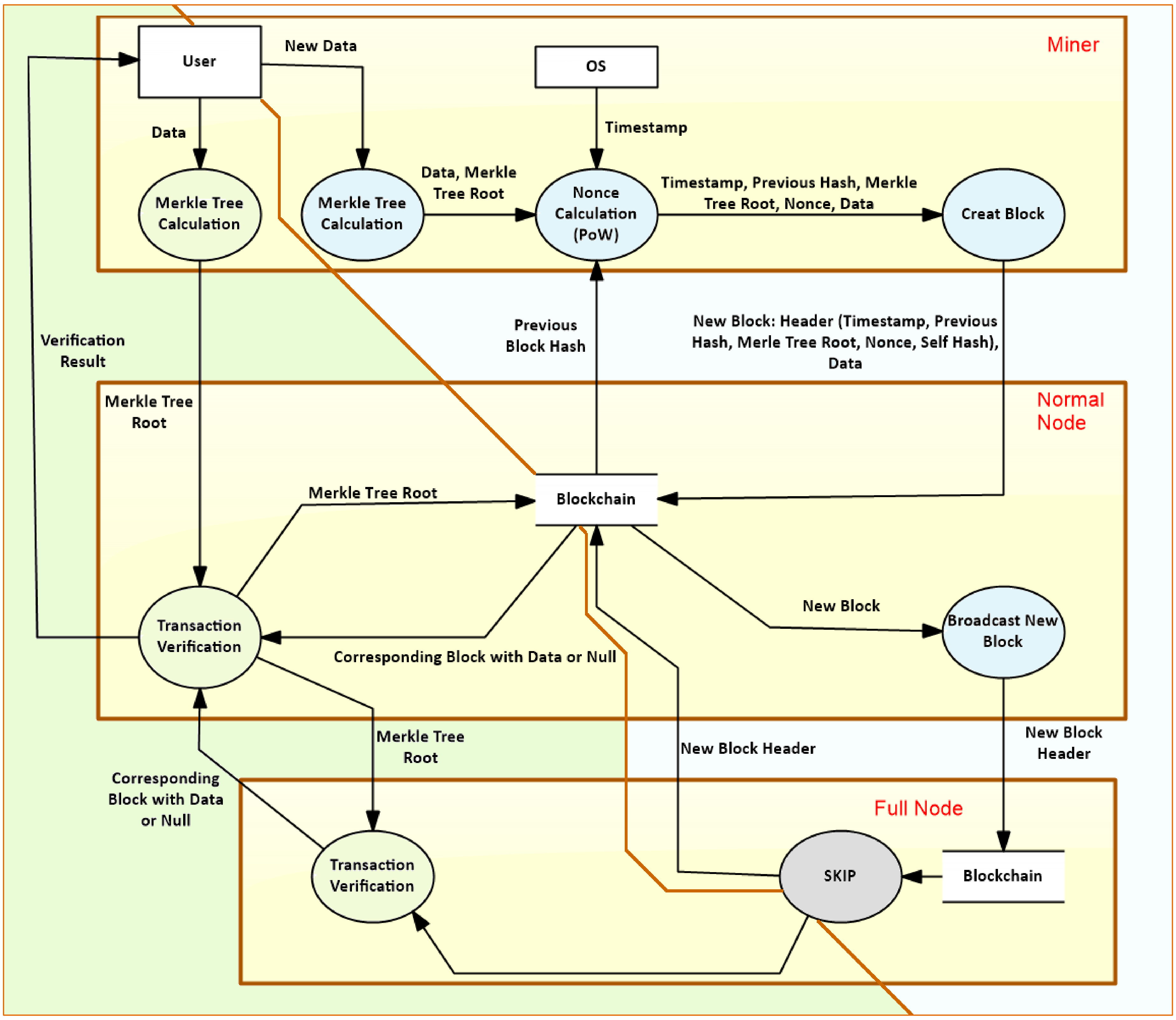
Miner, who
1. receive data from End User
2. receive previous block hash from Normal Node
3. perform hash-based proof-of-work to pack the data to a new block
4. return the new block to Normal Node and try broadcasting

Normal Node, who
1. download the latest blockchain from Full Node
2. receive and verify block from Miner
3. try broadcasting new blocks to Full Node
4. update the blockchain locally saved if Full Node admit the new blocks in blockchain

Full Node, who
1. receive and verify blocks from Normal Nodes’ broadcasting
2. add new blocks to blockchain
3. provide the latest blockchain to every Normal Nodes in the network

In next page, I will discuss about the interaction between Miner, Normal Node and Full Node from the perspective of data flow.

Below is a simplified data flow diagram that shows how the system works.



Blue: *Mining and Adding Blocks to Blockchain*
Green: *Searching Data in Blockchain*

Miner provides a user interface to End User, and performs proof-of-work to create new blocks. It does not store any data on disk.

Normal Node is responsible for verifying blocks from Miner, and broadcasting new blocks to Full Node. It also performs some searching functions. A partial blockchain is stored on disk for each Normal Node. This local blockchains only contains (i) all blocks' header and (ii) data of blocks mined by its underlying Miners.

Full Node is responsible for verifying blocks from Normal Node, and broadcasting the new block to every Nodes in the network. It also performs some searching functions if data is not found in Normal Node. A full blockchain is stored on disk for Full Node. This full blockchains contains (i) all blocks' header and (ii) all block data.

This project is written in GO, and the functions of each modules are:

Name	**Description**
sysUI.go	Provide a user interface and some network functions.
sysInputConvertor.go	Convert the user's input from strings to byte stream.
nodeMiner.go	Perform network functions and data flow management for Miner.
nodeServer.go	Perform network functions and data flow management for Normal Node/ Full Node.
calMerkleTree.go	Calculate Merkle Tree.
calBlock.go	Create a new block by using proof-of-work.
blockchainSyn.go	Validate the blockchain & keep blockchain in Normal Node/ Full Node consistent. i.e. synchronization.
blockchainDB.go	Manage how the blockchain is stored on disk.

3. System Design – Data & Merkle Tree Root

How Merkle Tree Root is applied in Blockchain?

In Bitcoin, a block is composed of an 80-bytes header and variable-size data. [2]

Block Header	version	*4 bytes*	*uint32*
	previous block header hash	*32 bytes*	*char[32]*
	merkle root hash	*32 bytes*	*char[32]*
	time	*4 bytes*	*uint32*
	nBits	*4 bytes*	*uint32*
	nonce	*4 bytes*	*uint32*
Data	Transaction 1	*vary*	
	Transaction 2	*vary*	
	…		

Firstly, a hash is generated from Transactions using SHA256 and merkle tree. This hash, as known as **merkle tree root**, is a component of block header. Then the block header is chained into blockchain by another hash function (will discuss in Section 4). By the nature of SHA256, it is nearly impossible to change transactions that are already added into the blockchain.

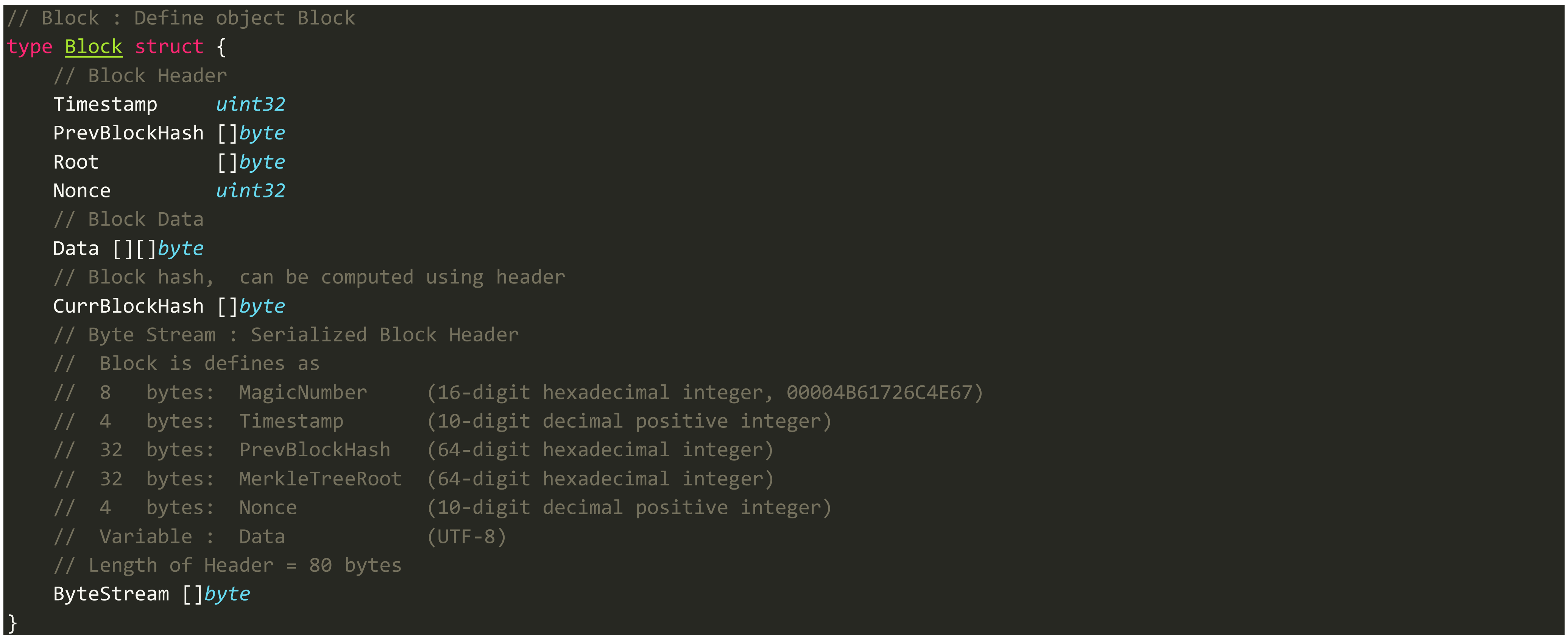
Details of merkle tree root is discussed in next page.

In this project, a block is composed of an 80-bytes header and variable-size data.

Block Header	MagicNumber	*8 bytes*	*16-digit hex integer*
	Timestamp	*4 bytes*	*uint32*
	PrevBlockHash	*32 bytes*	*64-digit hex integer*
	MerkleTreeRoot	*32 bytes*	*64-digit hex integer*
	Nonce	*4 bytes*	*uint32*
Data	Data 1	*vary*	
	Data 2	*vary*	
	…		

MagicNumber in this project is a meaningless string to keep the header length as 80-bytes, which is the same as Bitcoin.

Please refer to source code *calBlock.go* line 11 to line 32 for how a block is defined.



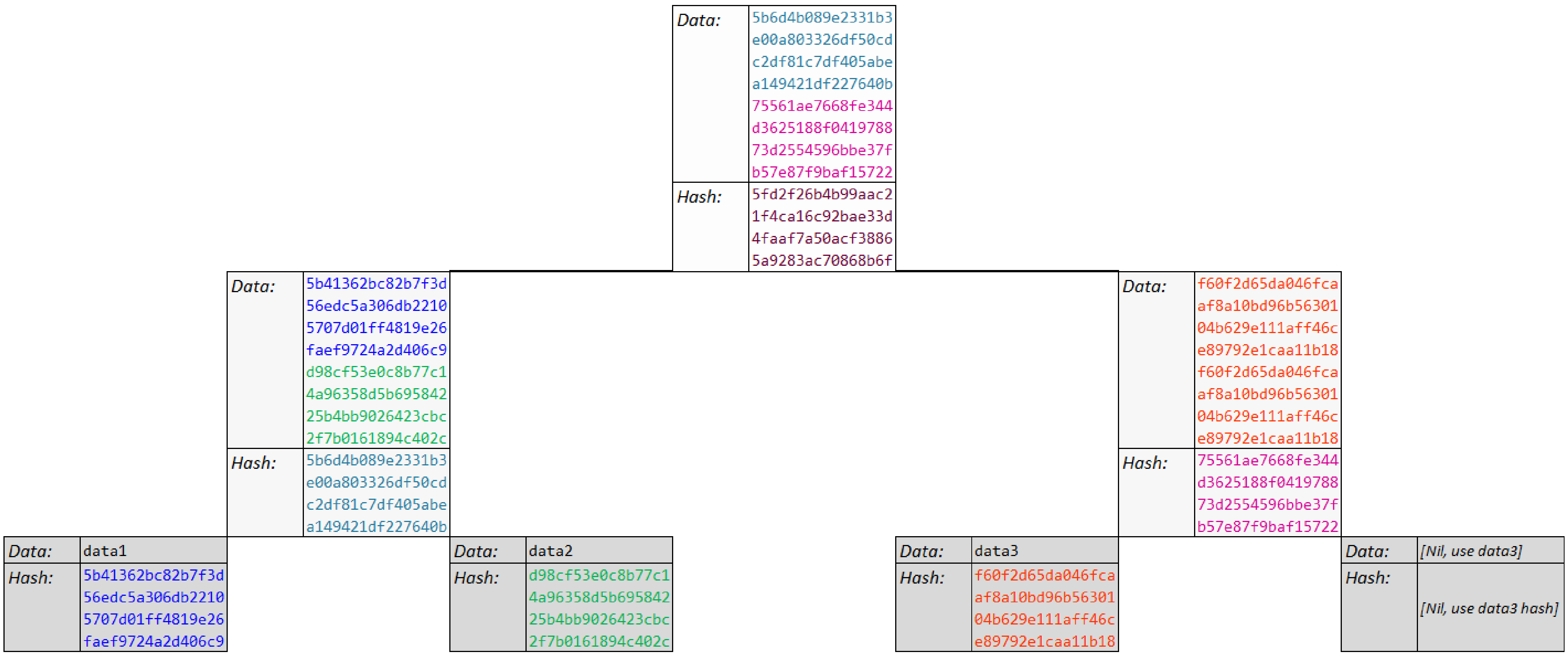
```go
// Block : Define object Block
type Block struct {
    // Block Header
    Timestamp     uint32
    PrevBlockHash []byte
    Root          []byte
    Nonce         uint32
    // Block Data
    Data [][]byte
    // Block hash,  can be computed using header
    CurrBlockHash []byte
    // Byte Stream : Serialized Block Header
    //  Block is defines as
    //  8   bytes:  MagicNumber     (16-digit hexadecimal integer, 00004B61726C4E67)
    //  4   bytes:  Timestamp       (10-digit decimal positive integer)
    //  32  bytes:  PrevBlockHash   (64-digit hexadecimal integer)
    //  32  bytes:  MerkleTreeRoot  (64-digit hexadecimal integer)
    //  4   bytes:  Nonce           (10-digit decimal positive integer)
    //  Variable :  Data            (UTF-8)
    // Length of Header = 80 bytes
    ByteStream []byte
}
```

The two additional attributes of a block, *CurrBlockHash* and *ByteStream,* could be calculated using other components of a block header. They are included in the object *block* for easier programming only. They are is not essential in building a blockchain network.

How to transform Data to Merkle Tree Root?

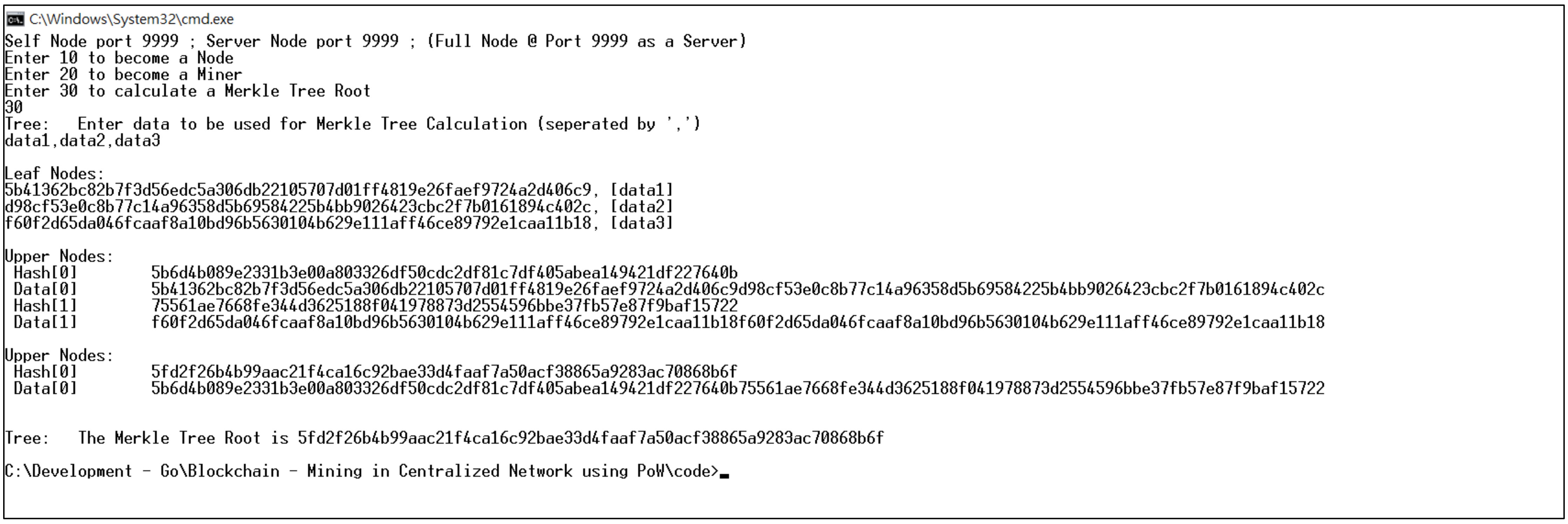
A hash function is any function that can be used to map data of arbitrary size to data of a fixed size. SHA256 is used in Bitcoin and this project.

Merkle tree is a method to generate a hash for an array of arbitrary size data. Then the resulting hash, as known as merkle tree root, is packed into a block. Below is an illustrative example of merkle tree root generation.



Step 1: Prepare leaf nodes by calculating the SHA256 hash of each data.
Step 2: Use the combination of two leaf nodes hash as upper level node data. Calculate the hash of upper level node data.
Step 3: Repeat Step 2 until there is only one upper level node remains. It is the root node.

In this example, the merkle tree root is `5fd2f26b4b99aac21f4ca16c92bae33d4faaf7a50acf38865a9283ac70868b6f`.
And below is the execution result of this project.



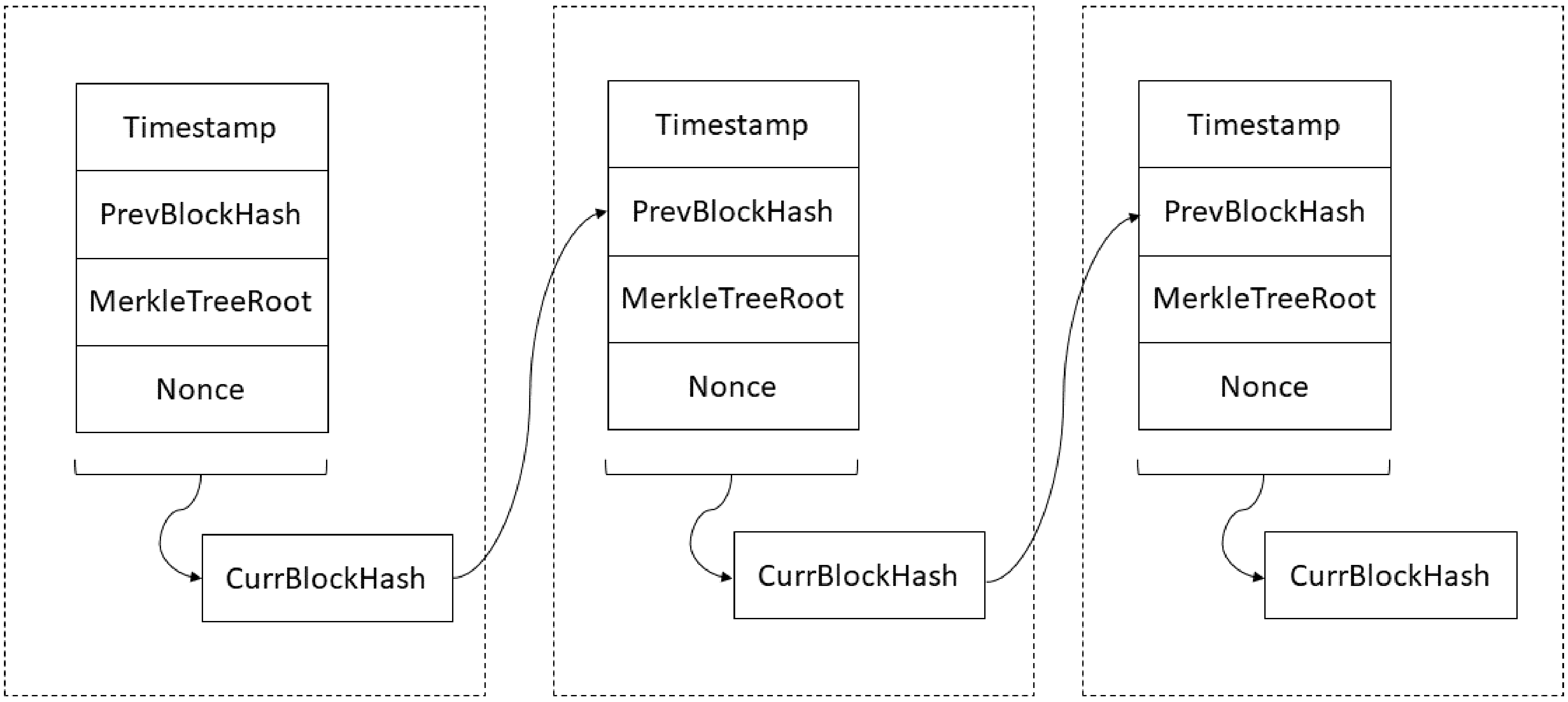
```
Self Node port 9999 ; Server Node port 9999 ; (Full Node @ Port 9999 as a Server)
Enter 10 to become a Node
Enter 20 to become a Miner
Enter 30 to calculate a Merkle Tree Root
30
Tree:   Enter data to be used for Merkle Tree Calculation (seperated by ',')
data1,data2,data3

Leaf Nodes:
5b41362bc82b7f3d56edc5a306db22105707d01ff4819e26faef9724a2d406c9, [data1]
d98cf53e0c8b77c14a96358d5b69584225b4bb9026423cbc2f7b0161894c402c, [data2]
f60f2d65da046fcaaf8a10bd96b5630104b629e111aff46ce89792e1caa11b18, [data3]

Upper Nodes:
 Hash[0]        5b6d4b089e2331b3e00a803326df50cdc2df81c7df405abea149421df227640b
 Data[0]        5b41362bc82b7f3d56edc5a306db22105707d01ff4819e26faef9724a2d406c9d98cf53e0c8b77c14a96358d5b69584225b4bb9026423cbc2f7b0161894c402c
 Hash[1]        75561ae7668fe344d3625188f041978873d2554596bbe37fb57e87f9baf15722
 Data[1]        f60f2d65da046fcaaf8a10bd96b5630104b629e111aff46ce89792e1caa11b18f60f2d65da046fcaaf8a10bd96b5630104b629e111aff46ce89792e1caa11b18

Upper Nodes:
 Hash[0]        5fd2f26b4b99aac21f4ca16c92bae33d4faaf7a50acf38865a9283ac70868b6f
 Data[0]        5b6d4b089e2331b3e00a803326df50cdc2df81c7df405abea149421df227640b75561ae7668fe344d3625188f041978873d2554596bbe37fb57e87f9baf15722


Tree:   The Merkle Tree Root is 5fd2f26b4b99aac21f4ca16c92bae33d4faaf7a50acf38865a9283ac70868b6f

C:\Development - Go\Blockchain - Mining in Centralized Network using PoW\code>_
```

Please refer to source code *calMerkleTree.go* for details. You may also refer to https://bitcoin.org/en/developer-reference#merkle-trees for more information about merkle tree algorithm.

4. System Design – Block & Proof-of-Work

How blocks are chained?

Blocks are chained by hash-based proof-of-work. *PrevBlockHash* in block header represents a hash calculated using previous block header. Each block contains the hash of previous block header, and all blocks form a blockchain.



Timestamp is the Unix epoch time.
PrevBlockHash is calculated using previous block header. In this project, *PrevBlockHash* of Genesis Block is set to be 64-digits zero.
MerkleTreeRoot is calculated using data from End User.

The remaining component, *Nonce*, is computed using proof-of-work algorithm.

In this project, a blockchain an array of blocks as below. Please refer to source code *blockchainSyn.go* line 11 to line 16.

```
//Blockchain : Define object Blockchain
type Blockchain struct {
    UserID string
    Blocks []*Block
}
```

The method ValidateChain() in *blockchainSyn.go* line 217 to line 241 verify this “chain” characteristic of a blockchain.

```
// ValidateChain :  Check if the whole chain is valid. i.e. all CurrBlockHash & PrevBlockHash match.
func (bc *Blockchain) ValidateChain() bool {

    validFlag := true
    if len(bc.Blocks) == 0 {
        validFlag = false
    } else {

        // Check Genesis Block first:   Only Check CurrBlockHash is Valid
        if bc.Blocks[0].ValidateBlock() == false {
            validFlag = false
        }
        // Then check other Blocks :    Check CurrBlockHash is Valid & PrevBlockHash Matches
        if len(bc.Blocks) > 1 {
            for i := 1; i < len(bc.Blocks); i++ {
                if bc.Blocks[i].ValidateBlock() == false {
                    validFlag = false
                }
                if string(bc.Blocks[i-1].CurrBlockHash) != string(bc.Blocks[i].PrevBlockHash) {
                    validFlag = false
                }
            }
        }
    }
    return validFlag
}
```

How to compute Nonce using proof-of-work?

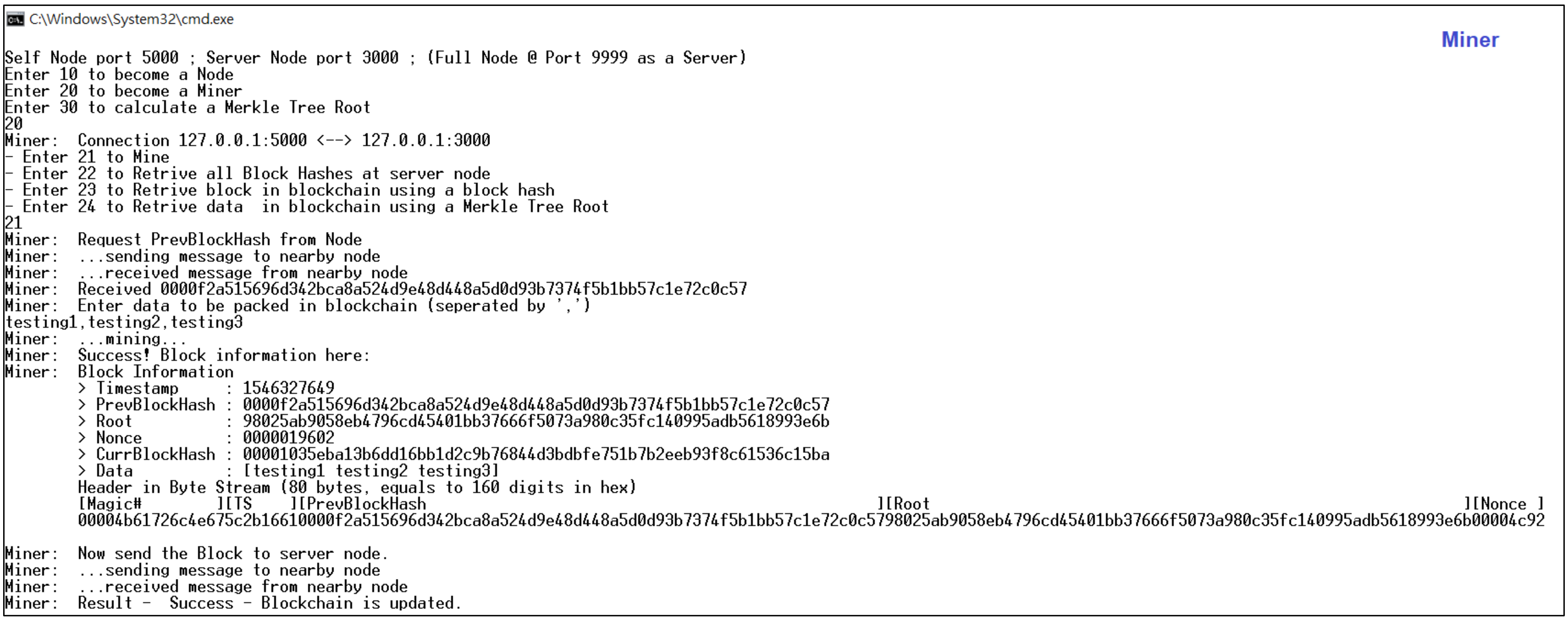
To avoid double spending, Bitcoin use the hash-based proof-of-work to increase the difficulties of adding new blocks into blockchain. Miners are required to compute a *Nonce* by try-and-error to create a valid *CurrBlockHash*, where a valid *CurrBlockHash* means

1) the hash is a SHA256 hash of block header (i.e. *Timestamp, PrevBlockHash, MerkleTreeRoot, and Nonce*)
2) the hash has the first n-bits equals to zero (i.e. threshold).

The *Nonce* computing process, i.e. proof-of-work, is time-consuming, which makes hacker nearly impossible to create a fake blockchain.

In this project, we set the threshold to be "the first 4-digit of hex integer is zero". It is different from Bitcoin but easier for demonstration. Below is an example for explaining proof-of-work.

Miner added a new block to blockchain successfully. The screenshot of result is:



```
Self Node port 5000 ; Server Node port 3000 ; (Full Node @ Port 9999 as a Server)
Enter 10 to become a Node
Enter 20 to become a Miner
Enter 30 to calculate a Merkle Tree Root
20
Miner:  Connection 127.0.0.1:5000 <--> 127.0.0.1:3000
- Enter 21 to Mine
- Enter 22 to Retrive all Block Hashes at server node
- Enter 23 to Retrive block in blockchain using a block hash
- Enter 24 to Retrive data  in blockchain using a Merkle Tree Root
21
Miner:  Request PrevBlockHash from Node
Miner:  ...sending message to nearby node
Miner:  ...received message from nearby node
Miner:  Received 0000f2a515696d342bca8a524d9e48d448a5d0d93b7374f5b1bb57c1e72c0c57
Miner:  Enter data to be packed in blockchain (seperated by ',')
testing1,testing2,testing3
Miner:  ...mining...
Miner:  Success! Block information here:
Miner:  Block Information
        > Timestamp     : 1546327649
        > PrevBlockHash : 0000f2a515696d342bca8a524d9e48d448a5d0d93b7374f5b1bb57c1e72c0c57
        > Root          : 98025ab9058eb4796cd45401bb37666f5073a980c35fc140995adb5618993e6b
        > Nonce         : 0000019602
        > CurrBlockHash : 00001035eba13b6dd16bb1d2c9b76844d3bdbfe751b7b2eeb93f8c61536c15ba
        > Data          : [testing1 testing2 testing3]
        Header in Byte Stream (80 bytes, equals to 160 digits in hex)
        [Magic#         ][TS     ][PrevBlockHash                                                 ][Root                                                          ][Nonce ]
        00004b61726c4e675c2b16610000f2a515696d342bca8a524d9e48d448a5d0d93b7374f5b1bb57c1e72c0c5798025ab9058eb4796cd45401bb37666f5073a980c35fc140995adb5618993e6b00004c92

Miner:  Now send the Block to server node.
Miner:  ...sending message to nearby node
Miner:  ...received message from nearby node
Miner:  Result -  Success - Blockchain is updated.
```

Take the **block with *Timestamp* = 1546327649** as example.

The End User would like to save strings *[testing1, testing2, testing3]* into the blockchain. At first, the End User told Miner that he/ she would like to add a new block into blockchain.

Then the Miner

1) Ask for the pervious block hash from Normal Node.
   The result is `**0000**f2a515696d342bca8a524d9e48d448a5d0d93b7374f5b1bb57c1e72c0c57`.
2) Ask for the strings from End User.
3) Calculate the merkle tree root of the strings.
   The result is `98025ab9058eb4796cd45401bb37666f5073a980c35fc140995adb5618993e6b`.
4) Start creating a new block. From the screenshots, we can see that the new block's *Nonce* = `0000019602`.
   It means that the Miner found a valid *Nonce* after 19,602 trials.
5) Send the new block to Normal Node to broadcast it among the network. Normal Node replies Miner the result.
   The result is "`Success!!`"
6) The block header of this new block, in hex integer, is:

Magic Number	00004b61726c4e67
Timestamp	5c2b1661
PrevBlockHash	0000f2a515696d342bca8a524d9e48d448a5d0d93b7374f5b1bb57c1e72c0c57
MerkleTreeRoot	98025ab9058eb4796cd45401bb37666f5073a980c35fc140995adb5618993e6b
Nonce	00004c92

The hash of this header, *CurrBlockHash*, is **0000**1035eba13b6dd16bb1d2c9b76844d3bdbfe751b7b2eeb93f8c61536c15ba. The first 4-digits are zero which means it is a valid block.

Proof-of-work is a time-consuming algorithm to perform. In Bitcoin, it takes approximately 10 minutes to get a valid *Nonce*. Only the Miner who first obtain a valid *CurrBlockHash*, i.e. the one who has the highest CPU power, could add a block into the blockchain. This "First Come First Serve" characteristic will be further discussed in next section.

5. System Design – Blockchain in Centralized Network

In Bitcoin, blockchain exists in a peer-to-peer decentralized network. Miner should add new blocks to "the most-difficult-to-recreate chain" [2] to ensure that only one chain exists.

In this project, a centralized network is adopted and there is always one chain. All Miner will add new blocks to the blockchain held by Full Node. Miner who sends a new block to Full Node first could successfully add a new block into the blockchain.

In this section I will show how data are exchanged among nodes, and how Nodes perform "Adding Blocks" and "Searching Data". In order to perform these functions, nodes send messages to nearby nodes to request desired information using TCP together with different message header. Three important message headers, "GetBlock", "GetData" and "Inv", is discussed here.

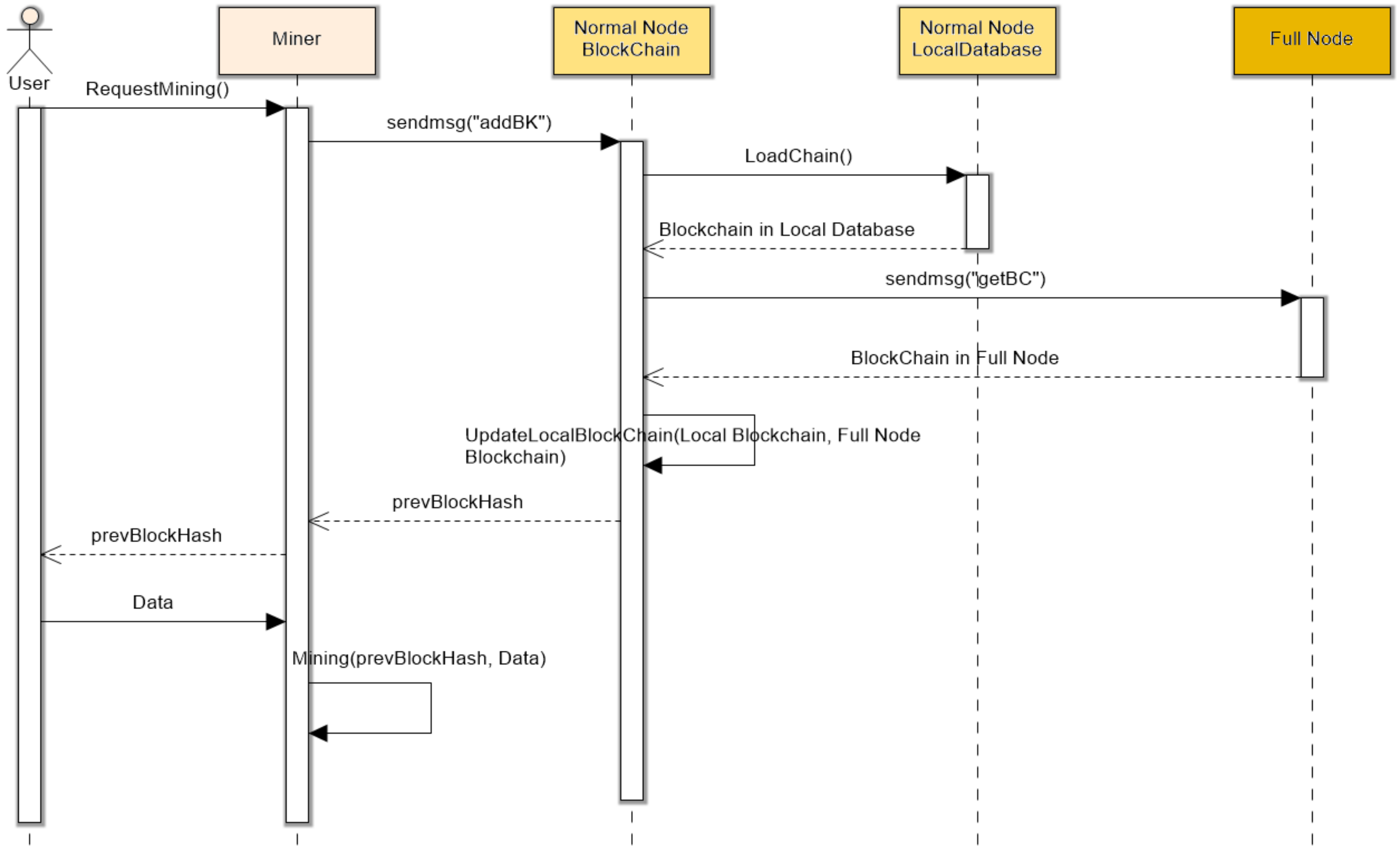
"*inv*":	the *Inv* message (inventory message) transmits one or more inventories of objects known to the transmitting peer.
"*GetBlock*"	the *GetBlocks* message requests an *inv* message that provides block header hashes starting from a particular point in the blockchain.
"*GetData*"	The *GetData* message requests one or more data objects from another node. The objects are requested by an inventory, which the requesting node typically received previously by way of an *Inv* message.

The set of message header is used in Bitcoin. [3]

Synchronization of Blockchain

Only one blockchain exists in this centralized network. Synchronization of blockchain could be done by ensuring that:
- Only Full Node contains the most updated blockchain.
- Only blocks admitted by Full Node could be added to the blockchain.
- The first block reaches Full Node will be admitted.
- Normal Node should update its local blockchain database from Full Node before performing any actions.

Example – Adding Block Part 1: Collect Information to Create a New Block

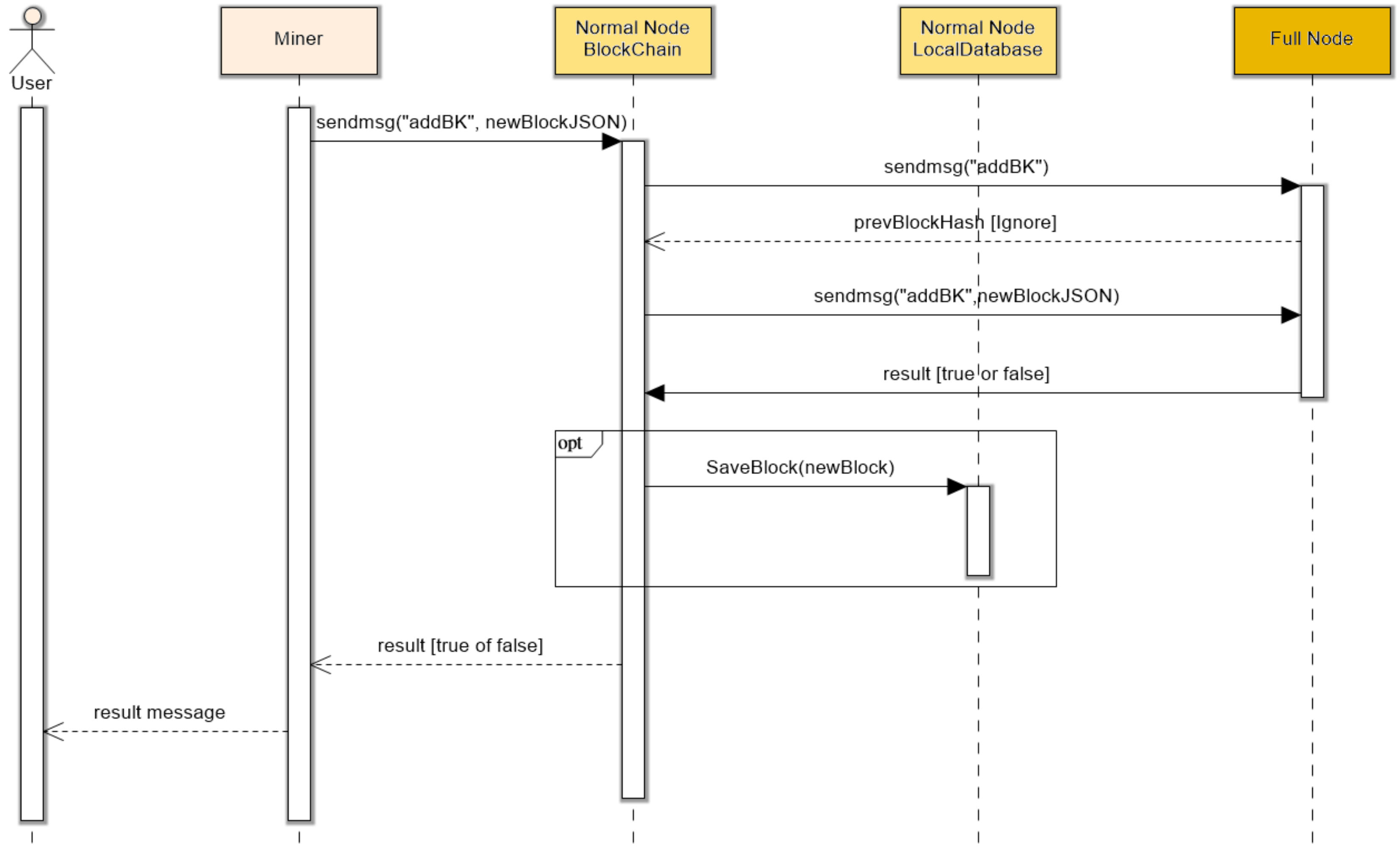


Note: "getBC" is equivalent to "GetBlock" in Bitcoin.
Only block header hashes are returned.

Explanation:

Step 1:	Miner sends a message "addBK" to Node.
Step 2:	Node reads blockchain from local database on disk.
Step 3:	Node requests blockchain from Full Node.
Step 4:	Node updates its local database on disk.
Step 5:	Node returns prevBlockHash to Miner.
Step 6:	Miner starts mining (i.e. performs merkle tree calculation & proof-of-work).

Example – Adding Block Part 2: Try to Add a New Block to Blockchain

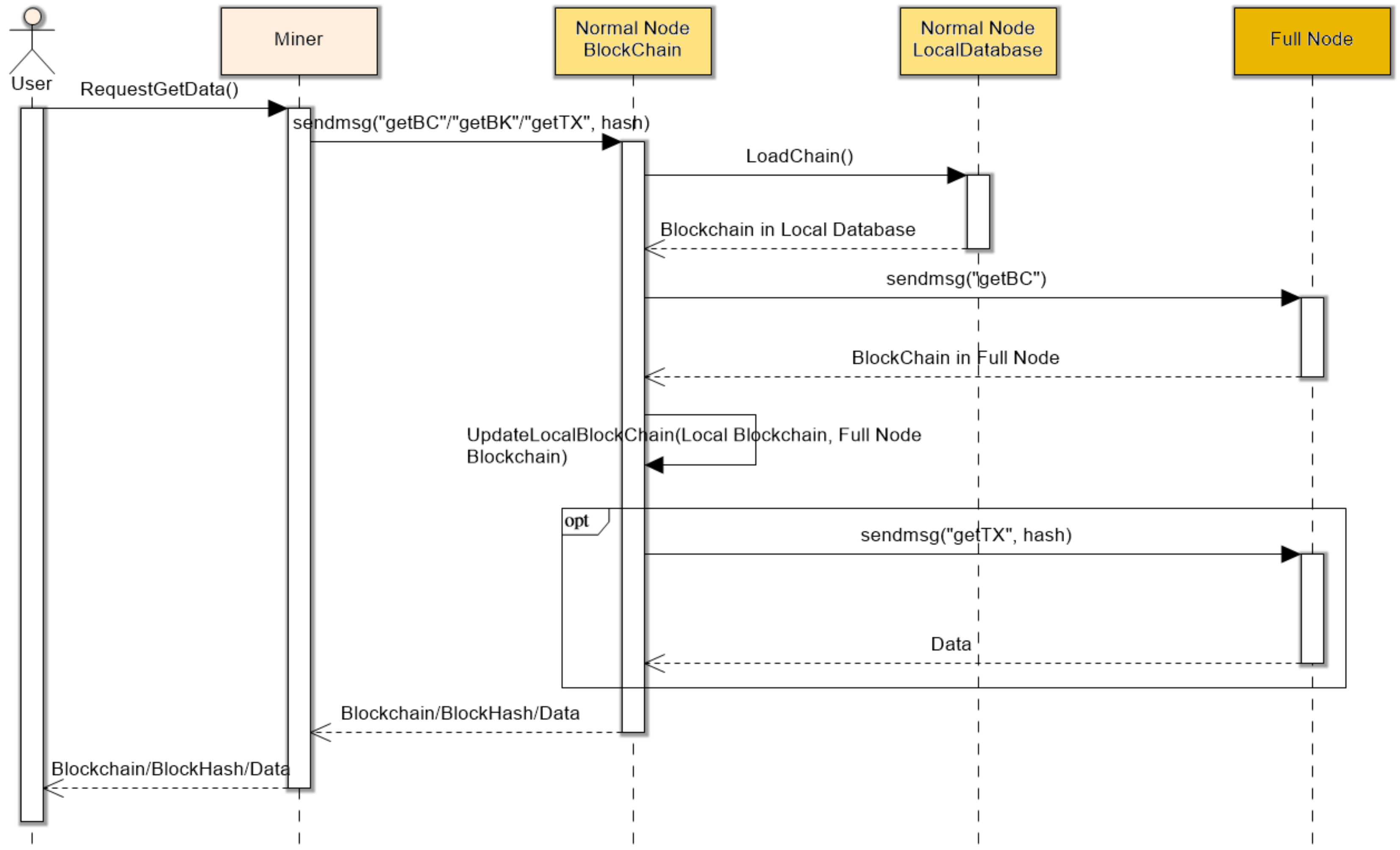


Note: Miner is expected to send "addBK" twice. Node should first reply a "PrevBlockHash", then reply a "True/ False" result.

Explanation:

Step 1: Miner sends a message "addBK" again with the newly created block (in JSON) to Node.
Step 2: Node sends messages "addBK" with the newly created block (in JSON) to Full Node.
Step 3: Full Node saves the block in blockchain if it is the first one reached. Return the result to Normal Node.
Step 4: Node saves the block in local blockchain if Full Node says the new block is admitted into blockchain.
Step 5: Miner receives a result message of "Success"/ "Fail".

Example – Searching Data in Blockchain: Block Header Hashes/ Single Block/ String Data


User
Miner
Normal Node BlockChain
Normal Node LocalDatabase
Full Node
RequestGetData()
sendmsg("getBC"/"getBK"/"getTX", hash)
LoadChain()
Blockchain in Local Database
sendmsg("getBC")
BlockChain in Full Node
UpdateLocalBlockChain(Local Blockchain, Full Node Blockchain)
opt
sendmsg("getTX", hash)
Data
Blockchain/BlockHash/Data
Blockchain/BlockHash/Data

Note: "getBK"/"getTX" is equivalent to "GetData" in Bitcoin.

Explanation:

Step 1: Miner sends a message "getBC/ getBK/ getTX" to Node.
Step 2: Node updates its local database on disk by checking with Full Node. Start searching desired data in local database.
Step 3: In case that desired data is not found, Node will request the data from Full Node.
Step 4: Node sends the requested data to Miner if it is found.

6. System Design – Data Storage

In this project, blockchain is stored in JSON on disk. It allows us to read the blockchain using text editor, which makes debugging/ demonstration easier.

Below is a typical blockchain stored by a Full Node.

```
[
    {
        "Timestamp": 1546327577,
        "PrevBlockHash": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=",
        "Root": "yBySmtuUuIFg8ZSduYoSMJb2v71wtqXYKjDTc6DMk9Q=",
        "Nonce": 6516,
        "Data": [
            "TmV3",
            "R2VuZXNpcw==",
            "QmxvY2s="
        ],
        "CurrBlockHash": "AADypRVpbTQryopSTZ5I1Eil0Nk7c3T1sbtXwecsDFc=",
        "ByteStream": "AABLYXJsTmdcKxYZAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAADIHJKa25S4gWDxlJ25ihIwlva/vXC2pdgqMNNzoMyT1AAAGXQ="
    },
    {
        "Timestamp": 1546327649,
        "PrevBlockHash": "AADypRVpbTQryopSTZ5I1Eil0Nk7c3T1sbtXwecsDFc=",
        "Root": "mAJauQWOtHls1FQBuzdmb1BzqYDDX8FAmVrbVhiZPms=",
        "Nonce": 19602,
        "Data": [
            "dGVzdGluZzE=",
            "dGVzdGluZzI=",
            "dGVzdGluZzM="
        ],
        "CurrBlockHash": "AAAQNeuhO23Ra7HSybdoRNO9v+dRt7LuuT+MYVNsFbo=",
        "ByteStream": "AABLYXJsTmdcKxZhAADypRVpbTQryopSTZ5I1Eil0Nk7c3T1sbtXwecsDFeYAlq5BY60eWzUVAG7N2ZvUHOpgMNfwUCZWttWGJk+awAATJI="
    }
]
```

Below is a typical blockchain stored by a Normal Node.

```
[
    {
        "Timestamp": 1546327577,
        "PrevBlockHash": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=",
        "Root": "yBySmtuUuIFg8ZSduYoSMJb2v71wtqXYKjDTc6DMk9Q=",
        "Nonce": 6516,
        "Data": null,
        "CurrBlockHash": "AADypRVpbTQryopSTZ5I1Eil0Nk7c3T1sbtXwecsDFc=",
        "ByteStream": null
    },
    {
        "Timestamp": 1546327649,
        "PrevBlockHash": "AADypRVpbTQryopSTZ5I1Eil0Nk7c3T1sbtXwecsDFc=",
        "Root": "mAJauQWOtHls1FQBuzdmb1BzqYDDX8FAmVrbVhiZPms=",
        "Nonce": 19602,
        "Data": [
            "dGVzdGluZzE=",
            "dGVzdGluZzI=",
            "dGVzdGluZzM="
        ],
        "CurrBlockHash": "AAAQNeuhO23Ra7HSybdoRNO9v+dRt7LuuT+MYVNsFbo=",
        "ByteStream": "AABLYXJsTmdcKxZhAADypRVpbTQryopSTZ5I1Eil0Nk7c3T1sbtXwecsDFeYAlq5BY60eWzUVAG7N2ZvUHOpgMNfwUCZWttWGJk+awAATJI="
    }
]
```

Please note that the Genesis Block is mined by Full Node when the blockchain network is initialized.
Since Normal Node stores data mined by its underlying Miner in its database only, it does not store data of the Genesis Block (which is mined by the Full Node).

Please refer to *blockchainDB.go* for details.

To conclude the system design,

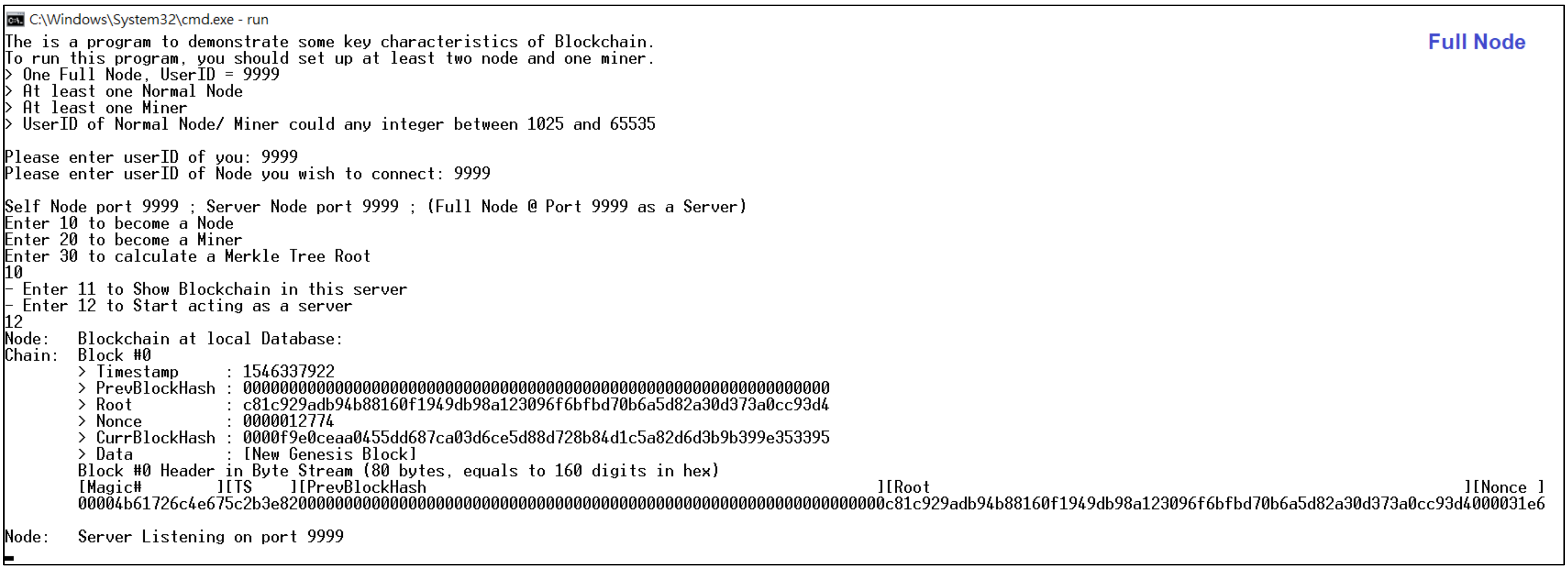
- In Section 3 "*System Design – Data & Merkle Tree Root*" & Section 4 "*System Design – Block & Proof-of-Work*", the creation of blocks and blockchain is discussed.
- In Section 5 "*System Design – Blockchain in Centralized Network*", the data exchange between Nodes is discussed.
- In Section 6 "*System Design – Data Storage"*, blockchain generated in this project and how to read it is discussed.

For more information, please read the comments in source code. Most of the algorithms are detailly explained in the comments.

7. Demonstration

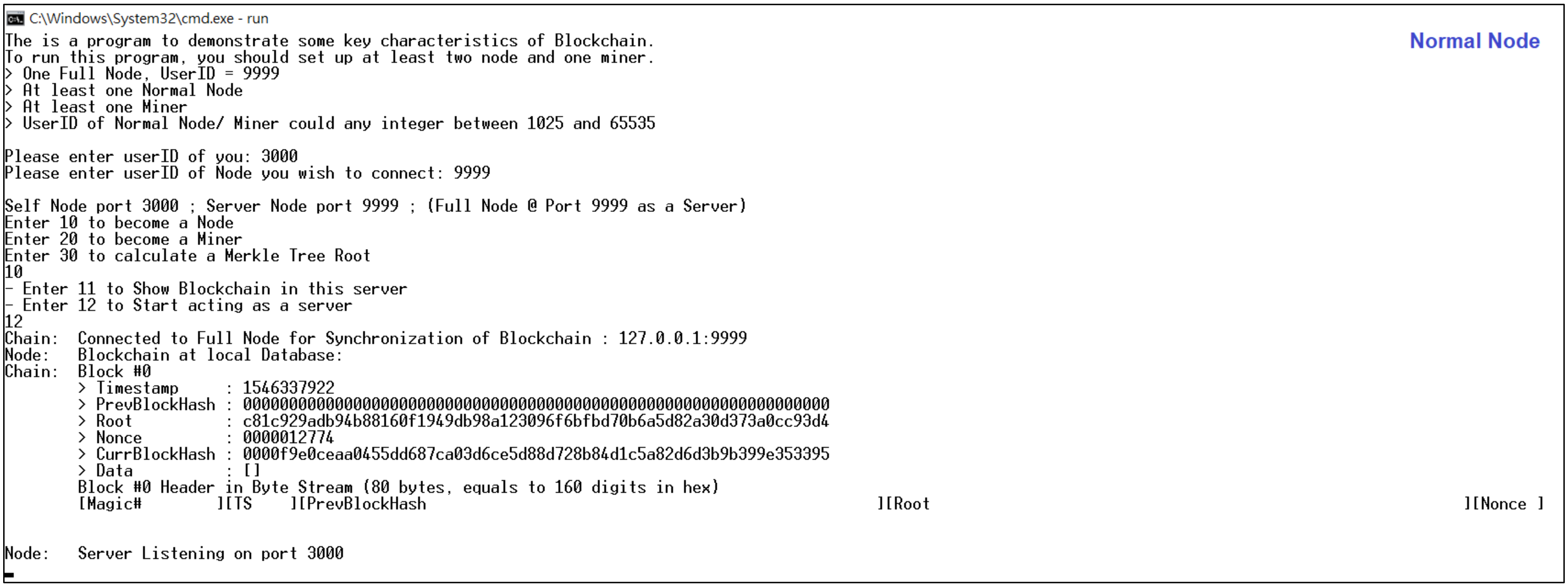
Demonstration of Mining & Adding New Block to Blockchain

Step 1: Set up the Full Node (Full Node port = 9999, connect to any because it is not used).
Full Node mines the Genesis Block if not blockchain database is found on disk.



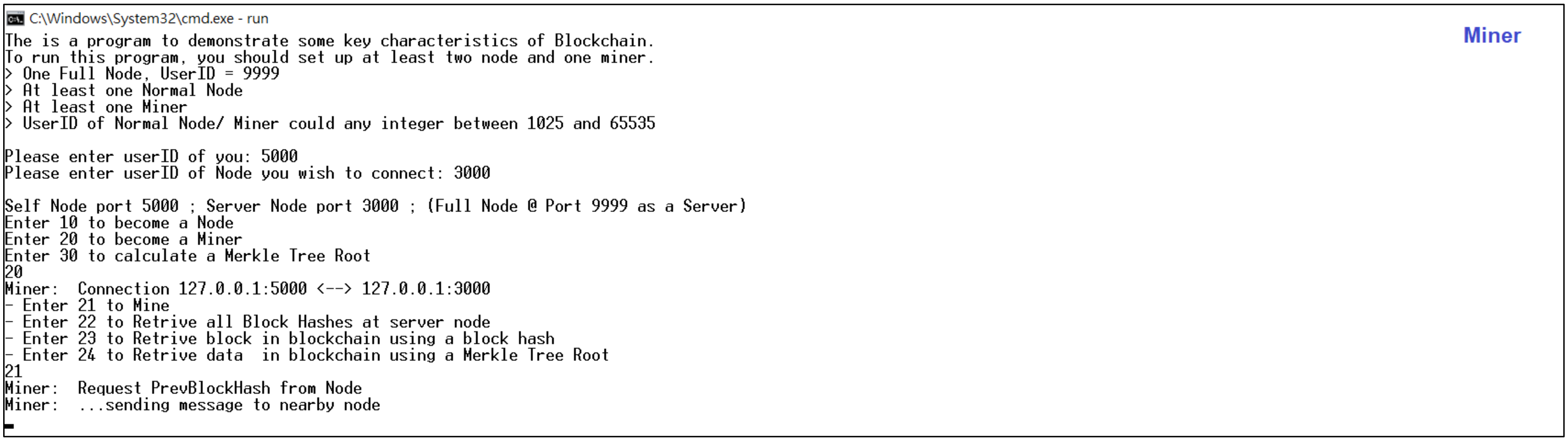
```
The is a program to demonstrate some key characteristics of Blockchain.
To run this program, you should set up at least two node and one miner.
> One Full Node, UserID = 9999
> At least one Normal Node
> At least one Miner
> UserID of Normal Node/ Miner could any integer between 1025 and 65535

Please enter userID of you: 9999
Please enter userID of Node you wish to connect: 9999

Self Node port 9999 ; Server Node port 9999 ; (Full Node @ Port 9999 as a Server)
Enter 10 to become a Node
Enter 20 to become a Miner
Enter 30 to calculate a Merkle Tree Root
10
- Enter 11 to Show Blockchain in this server
- Enter 12 to Start acting as a server
12
Node:   Blockchain at local Database:
Chain:  Block #0
        > Timestamp     : 1546337922
        > PrevBlockHash : 0000000000000000000000000000000000000000000000000000000000000000
        > Root          : c81c929adb94b88160f1949db98a123096f6bfbd70b6a5d82a30d373a0cc93d4
        > Nonce         : 0000012774
        > CurrBlockHash : 0000f9e0ceaa0455dd687ca03d6ce5d88d728b84d1c5a82d6d3b9b399e353395
        > Data          : [New Genesis Block]
        Block #0 Header in Byte Stream (80 bytes, equals to 160 digits in hex)
        [Magic#        ][TS    ][PrevBlockHash                                                 ][Root                                                          ][Nonce ]
        00004b61726c4e675c2b3e820000000000000000000000000000000000000000000000000000000000000000c81c929adb94b88160f1949db98a123096f6bfbd70b6a5d82a30d373a0cc93d4000031e6

Node:   Server Listening on port 9999
```

Step 2: Set up Normal Node (Normal Node port = 3000, connect to Full Node port = 9999).
The Normal Node updates its local blockchain once it is connected to Full Node.



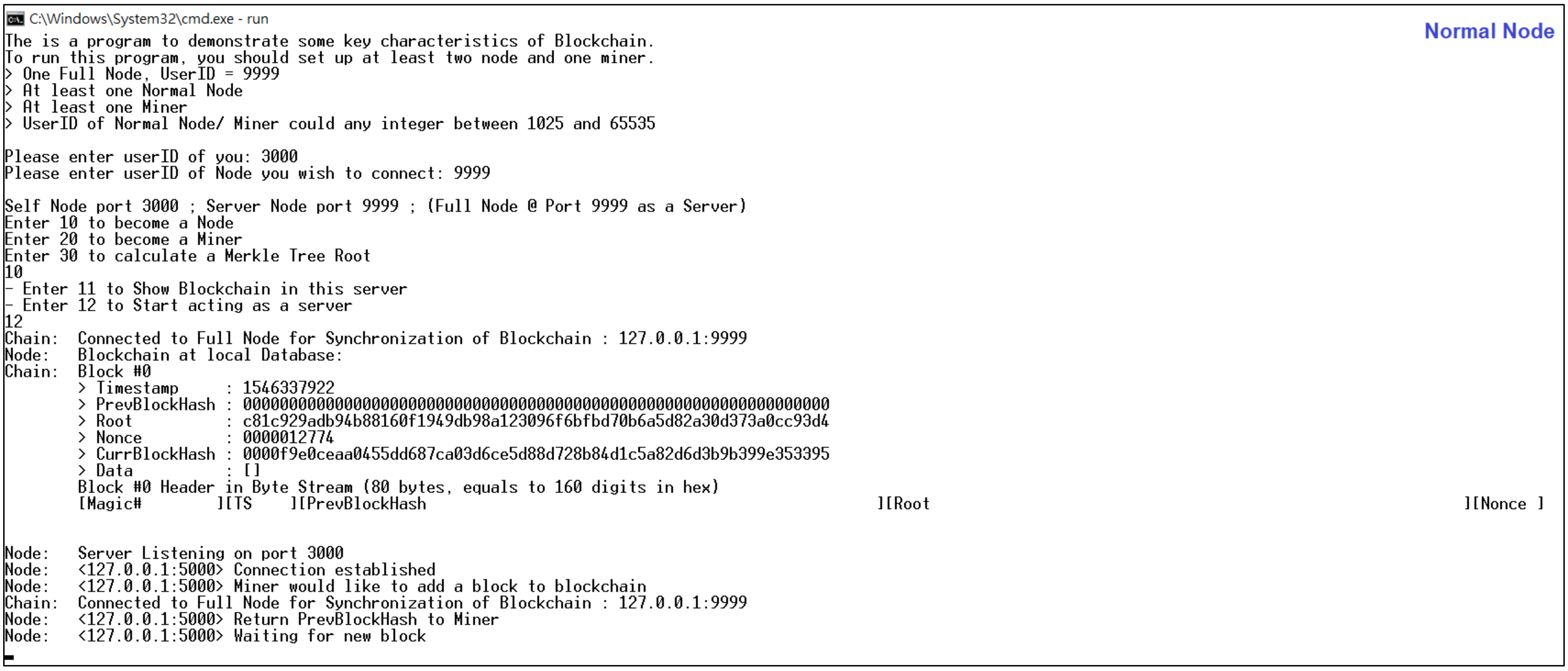
```
The is a program to demonstrate some key characteristics of Blockchain.
To run this program, you should set up at least two node and one miner.
> One Full Node, UserID = 9999
> At least one Normal Node
> At least one Miner
> UserID of Normal Node/ Miner could any integer between 1025 and 65535

Please enter userID of you: 3000
Please enter userID of Node you wish to connect: 9999

Self Node port 3000 ; Server Node port 9999 ; (Full Node @ Port 9999 as a Server)
Enter 10 to become a Node
Enter 20 to become a Miner
Enter 30 to calculate a Merkle Tree Root
10
- Enter 11 to Show Blockchain in this server
- Enter 12 to Start acting as a server
12
Chain:  Connected to Full Node for Synchronization of Blockchain : 127.0.0.1:9999
Node:   Blockchain at local Database:
Chain:  Block #0
        > Timestamp     : 1546337922
        > PrevBlockHash : 0000000000000000000000000000000000000000000000000000000000000000
        > Root          : c81c929adb94b88160f1949db98a123096f6bfbd70b6a5d82a30d373a0cc93d4
        > Nonce         : 0000012774
        > CurrBlockHash : 0000f9e0ceaa0455dd687ca03d6ce5d88d728b84d1c5a82d6d3b9b399e353395
        > Data          : []
        Block #0 Header in Byte Stream (80 bytes, equals to 160 digits in hex)
        [Magic#        ][TS    ][PrevBlockHash                                                 ][Root                                                          ][Nonce ]


Node:   Server Listening on port 3000
```

Step 3: Set up Miner. Miner should be connected to Normal Node (Port 3000).
Select the Miner's function "Mine" to add new block into blockchain.
Then Miner requests *"PrevBlockHash"* from Normal Node.



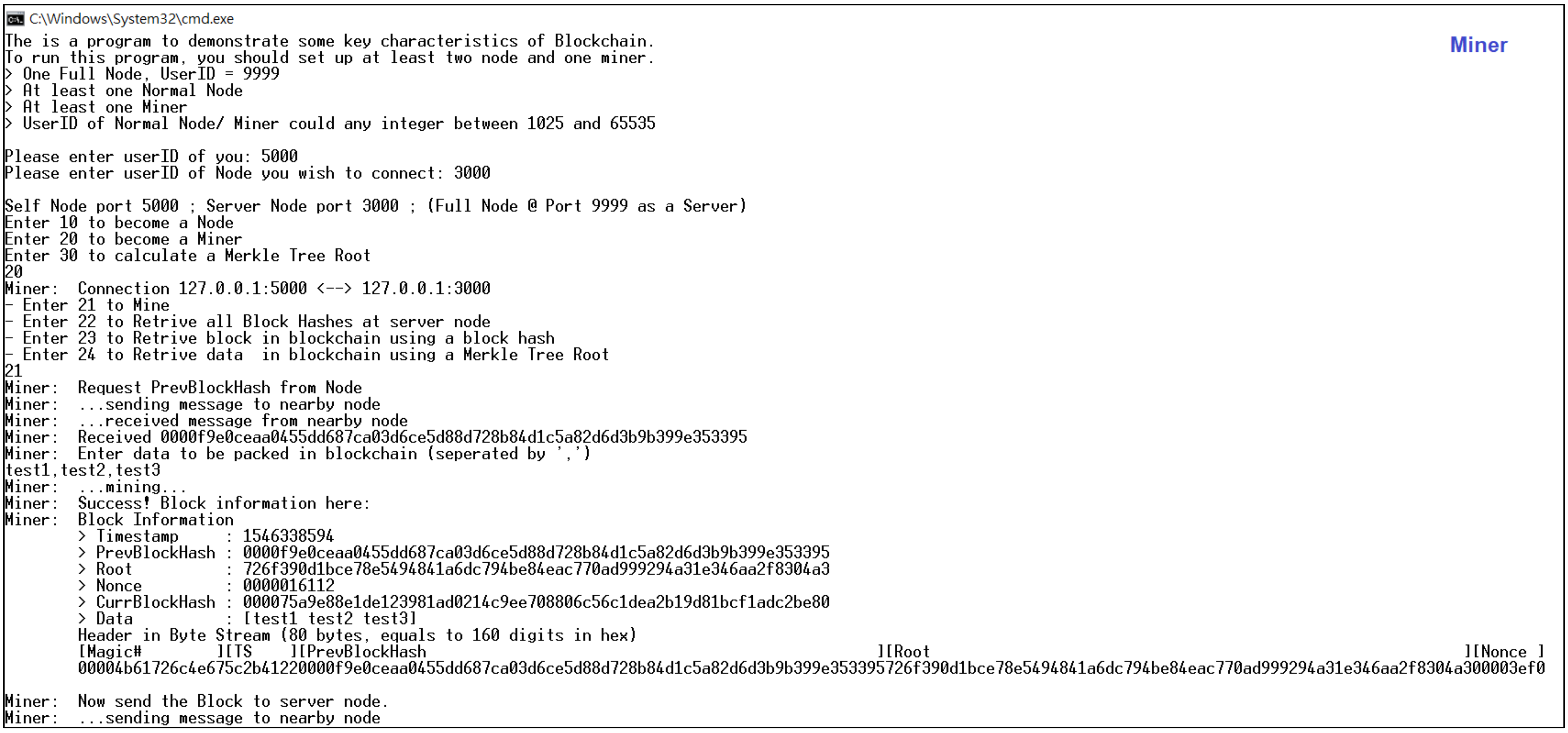
```
The is a program to demonstrate some key characteristics of Blockchain.
To run this program, you should set up at least two node and one miner.
> One Full Node, UserID = 9999
> At least one Normal Node
> At least one Miner
> UserID of Normal Node/ Miner could any integer between 1025 and 65535

Please enter userID of you: 5000
Please enter userID of Node you wish to connect: 3000

Self Node port 5000 ; Server Node port 3000 ; (Full Node @ Port 9999 as a Server)
Enter 10 to become a Node
Enter 20 to become a Miner
Enter 30 to calculate a Merkle Tree Root
20
Miner:  Connection 127.0.0.1:5000 <--> 127.0.0.1:3000
- Enter 21 to Mine
- Enter 22 to Retrive all Block Hashes at server node
- Enter 23 to Retrive block in blockchain using a block hash
- Enter 24 to Retrive data  in blockchain using a Merkle Tree Root
21
Miner:  Request PrevBlockHash from Node
Miner:  ...sending message to nearby node
```

Step 4: Normal Node receives request from Miner.
After updating the local blockchain from Full Node, it replies “*PrevBlockHash”*.



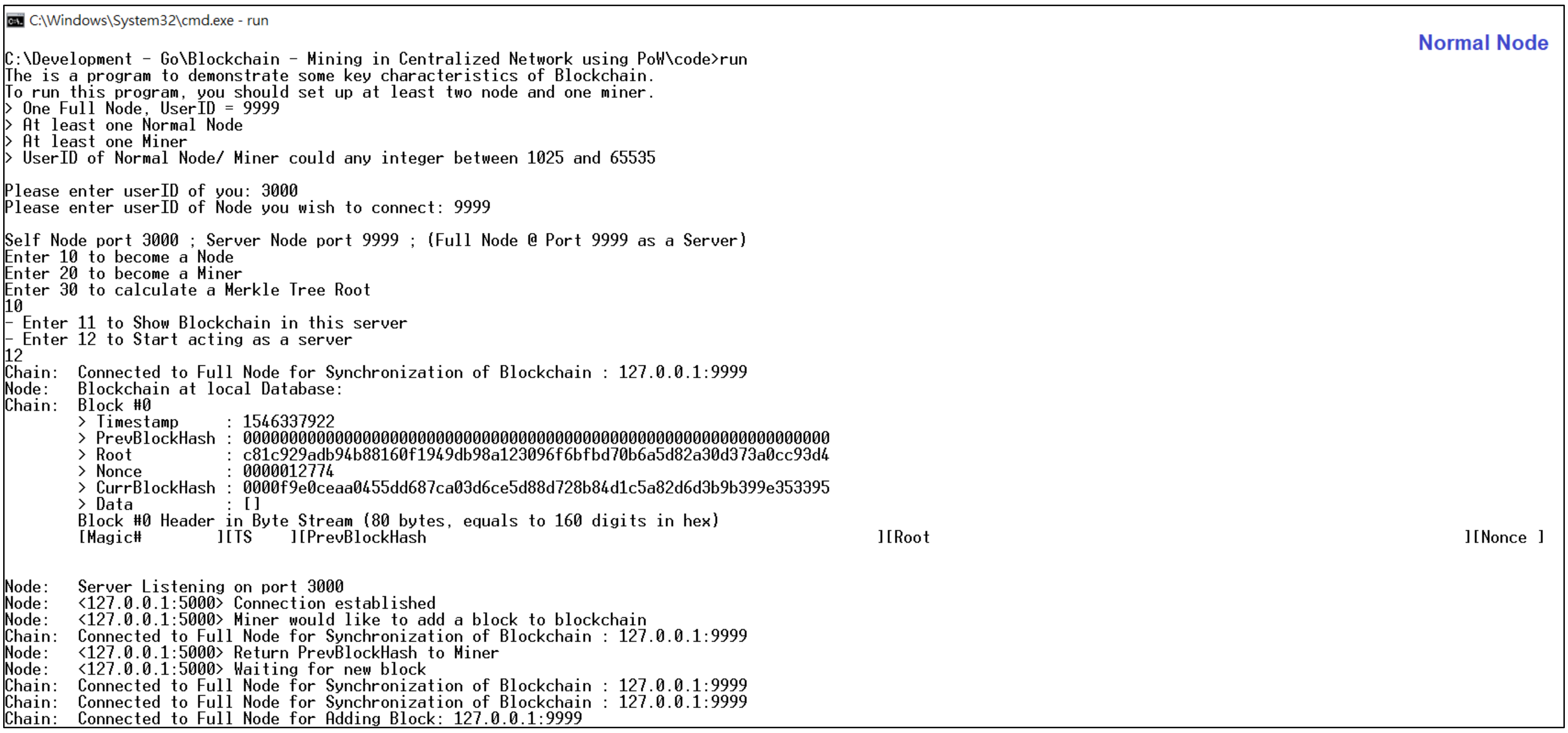
```
The is a program to demonstrate some key characteristics of Blockchain.
To run this program, you should set up at least two node and one miner.
> One Full Node, UserID = 9999
> At least one Normal Node
> At least one Miner
> UserID of Normal Node/ Miner could any integer between 1025 and 65535

Please enter userID of you: 3000
Please enter userID of Node you wish to connect: 9999

Self Node port 3000 ; Server Node port 9999 ; (Full Node @ Port 9999 as a Server)
Enter 10 to become a Node
Enter 20 to become a Miner
Enter 30 to calculate a Merkle Tree Root
10
- Enter 11 to Show Blockchain in this server
- Enter 12 to Start acting as a server
12
Chain:  Connected to Full Node for Synchronization of Blockchain : 127.0.0.1:9999
Node:   Blockchain at local Database:
Chain:  Block #0
        > Timestamp     : 1546337922
        > PrevBlockHash : 0000000000000000000000000000000000000000000000000000000000000000
        > Root          : c81c929adb94b88160f1949db98a123096f6bfbd70b6a5d82a30d373a0cc93d4
        > Nonce         : 0000012774
        > CurrBlockHash : 0000f9e0ceaa0455dd687ca03d6ce5d88d728b84d1c5a82d6d3b9b399e353395
        > Data          : []
        Block #0 Header in Byte Stream (80 bytes, equals to 160 digits in hex)
        [Magic#        ][TS     ][PrevBlockHash                                                 ][Root                                                          ][Nonce ]


Node:   Server Listening on port 3000
Node:   <127.0.0.1:5000> Connection established
Node:   <127.0.0.1:5000> Miner would like to add a block to blockchain
Chain:  Connected to Full Node for Synchronization of Blockchain : 127.0.0.1:9999
Node:   <127.0.0.1:5000> Return PrevBlockHash to Miner
Node:   <127.0.0.1:5000> Waiting for new block
```

Step 5: Miner receives “*PrevBlockHash*”. Then Miner asks End User to input strings to be packed into the blockchain.
After Miner receives input strings from End User, it starts mining by computing “*Nonce”* using proof-of-work.
Once valid “*Nonce”* is computed, it sends the new block to Normal Node.



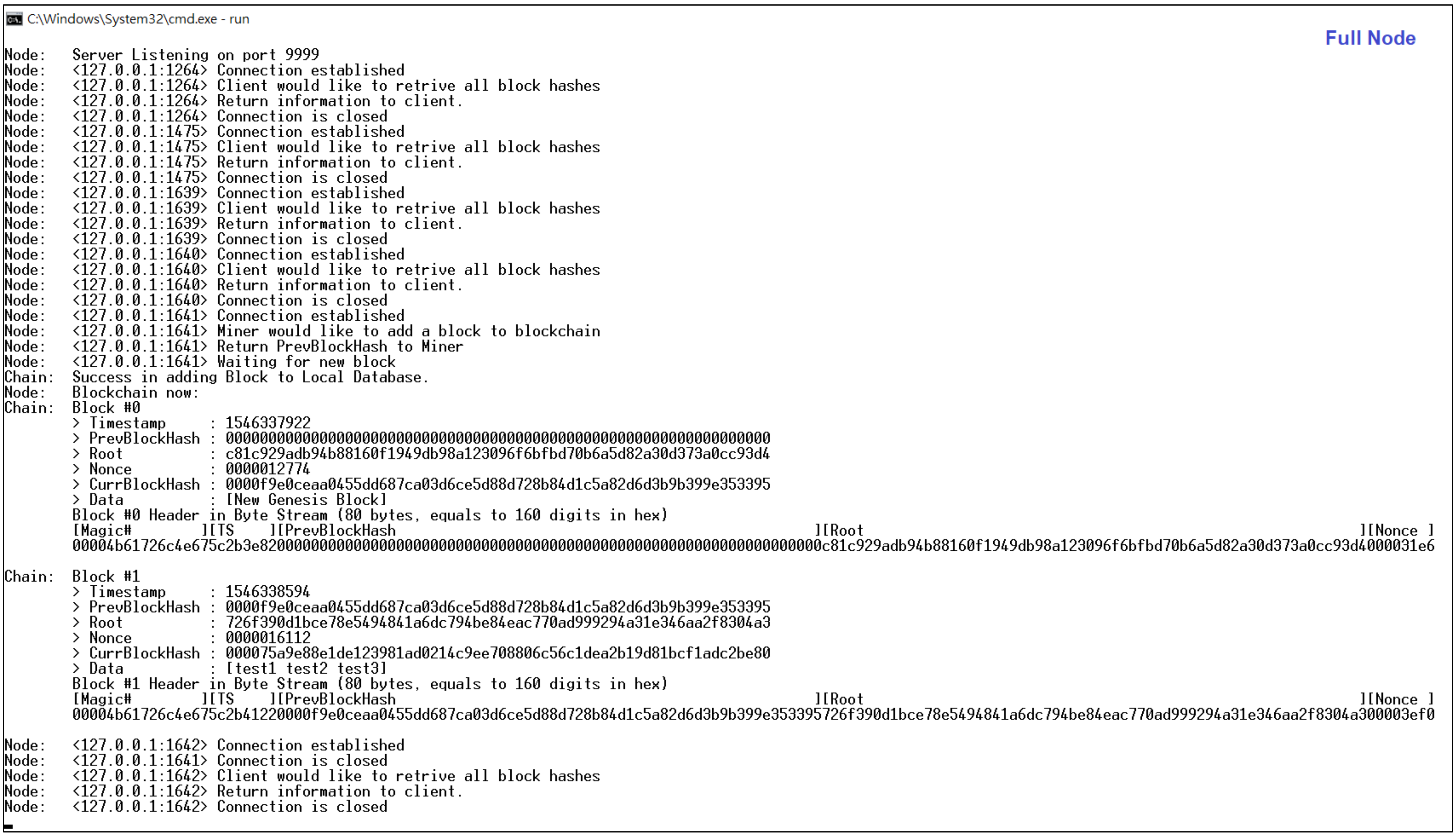
```
The is a program to demonstrate some key characteristics of Blockchain.
To run this program, you should set up at least two node and one miner.
> One Full Node, UserID = 9999
> At least one Normal Node
> At least one Miner
> UserID of Normal Node/ Miner could any integer between 1025 and 65535

Please enter userID of you: 5000
Please enter userID of Node you wish to connect: 3000

Self Node port 5000 ; Server Node port 3000 ; (Full Node @ Port 9999 as a Server)
Enter 10 to become a Node
Enter 20 to become a Miner
Enter 30 to calculate a Merkle Tree Root
20
Miner:  Connection 127.0.0.1:5000 <--> 127.0.0.1:3000
- Enter 21 to Mine
- Enter 22 to Retrive all Block Hashes at server node
- Enter 23 to Retrive block in blockchain using a block hash
- Enter 24 to Retrive data  in blockchain using a Merkle Tree Root
21
Miner:  Request PrevBlockHash from Node
Miner:  ...sending message to nearby node
Miner:  ...received message from nearby node
Miner:  Received 0000f9e0ceaa0455dd687ca03d6ce5d88d728b84d1c5a82d6d3b9b399e353395
Miner:  Enter data to be packed in blockchain (seperated by ',')
test1,test2,test3
Miner:  ...mining...
Miner:  Success! Block information here:
Miner:  Block Information
        > Timestamp     : 1546338594
        > PrevBlockHash : 0000f9e0ceaa0455dd687ca03d6ce5d88d728b84d1c5a82d6d3b9b399e353395
        > Root          : 726f390d1bce78e5494841a6dc794be84eac770ad999294a31e346aa2f8304a3
        > Nonce         : 0000016112
        > CurrBlockHash : 000075a9e88e1de123981ad0214c9ee708806c56c1dea2b19d81bcf1adc2be80
        > Data          : [test1 test2 test3]
        Header in Byte Stream (80 bytes, equals to 160 digits in hex)
        [Magic#        ][TS     ][PrevBlockHash                                                 ][Root                                                          ][Nonce ]
        00004b61726c4e675c2b41220000f9e0ceaa0455dd687ca03d6ce5d88d728b84d1c5a82d6d3b9b399e353395726f390d1bce78e5494841a6dc794be84eac770ad999294a31e346aa2f8304a300003ef0

Miner:  Now send the Block to server node.
Miner:  ...sending message to nearby node
```

Step 6: Normal Node receives new block from Miner.
Then Normal Node forwards the new block to Full Node after verification.



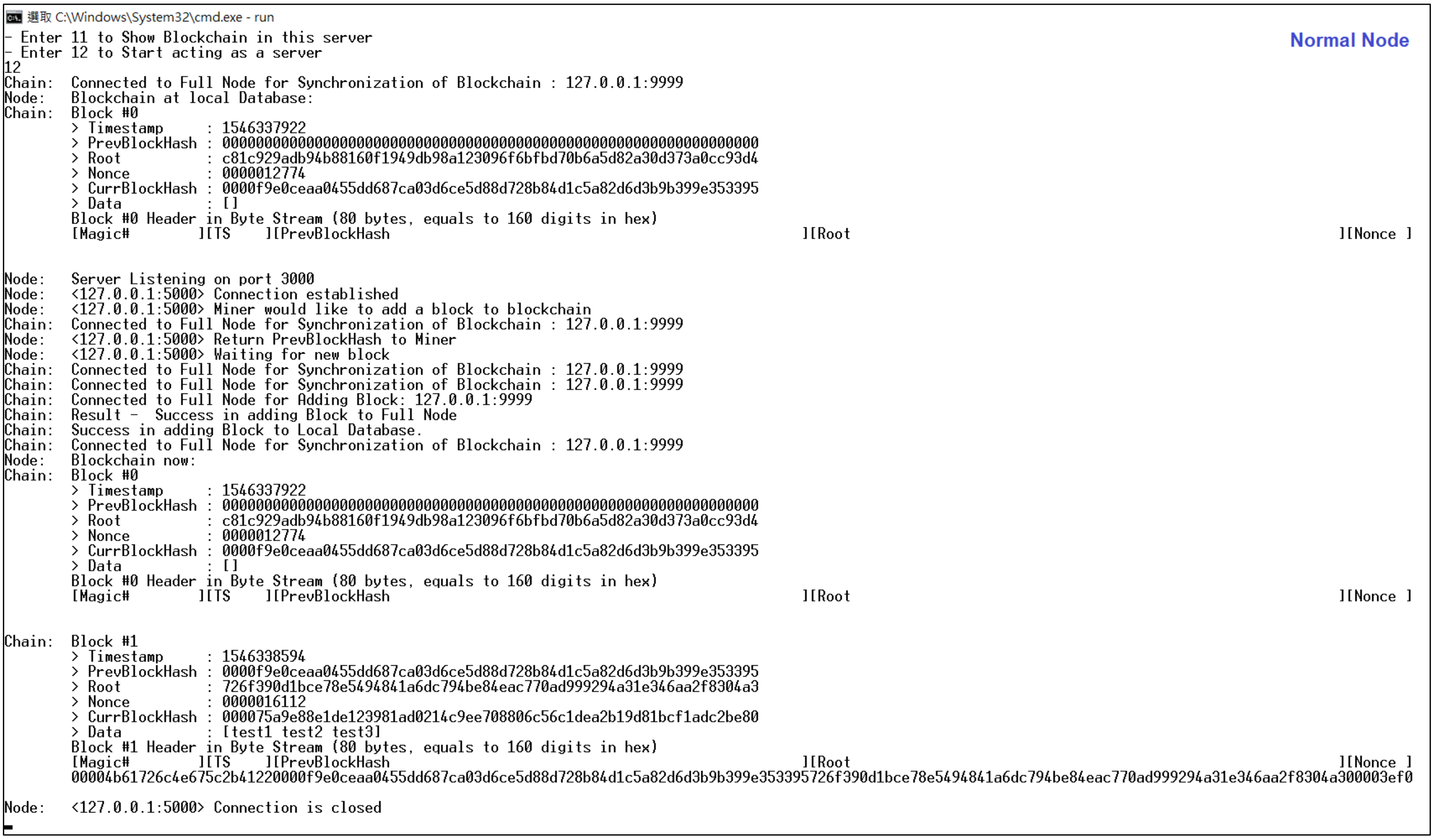
```
C:\Development - Go\Blockchain - Mining in Centralized Network using PoW\code>run
The is a program to demonstrate some key characteristics of Blockchain.
To run this program, you should set up at least two node and one miner.
> One Full Node, UserID = 9999
> At least one Normal Node
> At least one Miner
> UserID of Normal Node/ Miner could any integer between 1025 and 65535

Please enter userID of you: 3000
Please enter userID of Node you wish to connect: 9999

Self Node port 3000 ; Server Node port 9999 ; (Full Node @ Port 9999 as a Server)
Enter 10 to become a Node
Enter 20 to become a Miner
Enter 30 to calculate a Merkle Tree Root
10
- Enter 11 to Show Blockchain in this server
- Enter 12 to Start acting as a server
12
Chain:   Connected to Full Node for Synchronization of Blockchain : 127.0.0.1:9999
Node:    Blockchain at local Database:
Chain:   Block #0
         > Timestamp     : 1546337922
         > PrevBlockHash : 0000000000000000000000000000000000000000000000000000000000000000
         > Root          : c81c929adb94b88160f1949db98a123096f6bfbd70b6a5d82a30d373a0cc93d4
         > Nonce         : 0000012774
         > CurrBlockHash : 0000f9e0ceaa0455dd687ca03d6ce5d88d728b84d1c5a82d6d3b9b399e353395
         > Data          : []
         Block #0 Header in Byte Stream (80 bytes, equals to 160 digits in hex)
         [Magic#        ][TS    ][PrevBlockHash                                                 ][Root                                                          ][Nonce ]


Node:    Server Listening on port 3000
Node:    <127.0.0.1:5000> Connection established
Node:    <127.0.0.1:5000> Miner would like to add a block to blockchain
Chain:   Connected to Full Node for Synchronization of Blockchain : 127.0.0.1:9999
Node:    <127.0.0.1:5000> Return PrevBlockHash to Miner
Node:    <127.0.0.1:5000> Waiting for new block
Chain:   Connected to Full Node for Synchronization of Blockchain : 127.0.0.1:9999
Chain:   Connected to Full Node for Synchronization of Blockchain : 127.0.0.1:9999
Chain:   Connected to Full Node for Adding Block: 127.0.0.1:9999
```

Step 7: Full Node checks if the new block is the first one reach it.
i.e. the block is using correct "*PrevBlockHash*", which should be "*CurrBlockHash*" of newest node in the existing blockchain.
If it is true, the new block will be added to blockchain. Then Full Node returns the result to Normal Node.



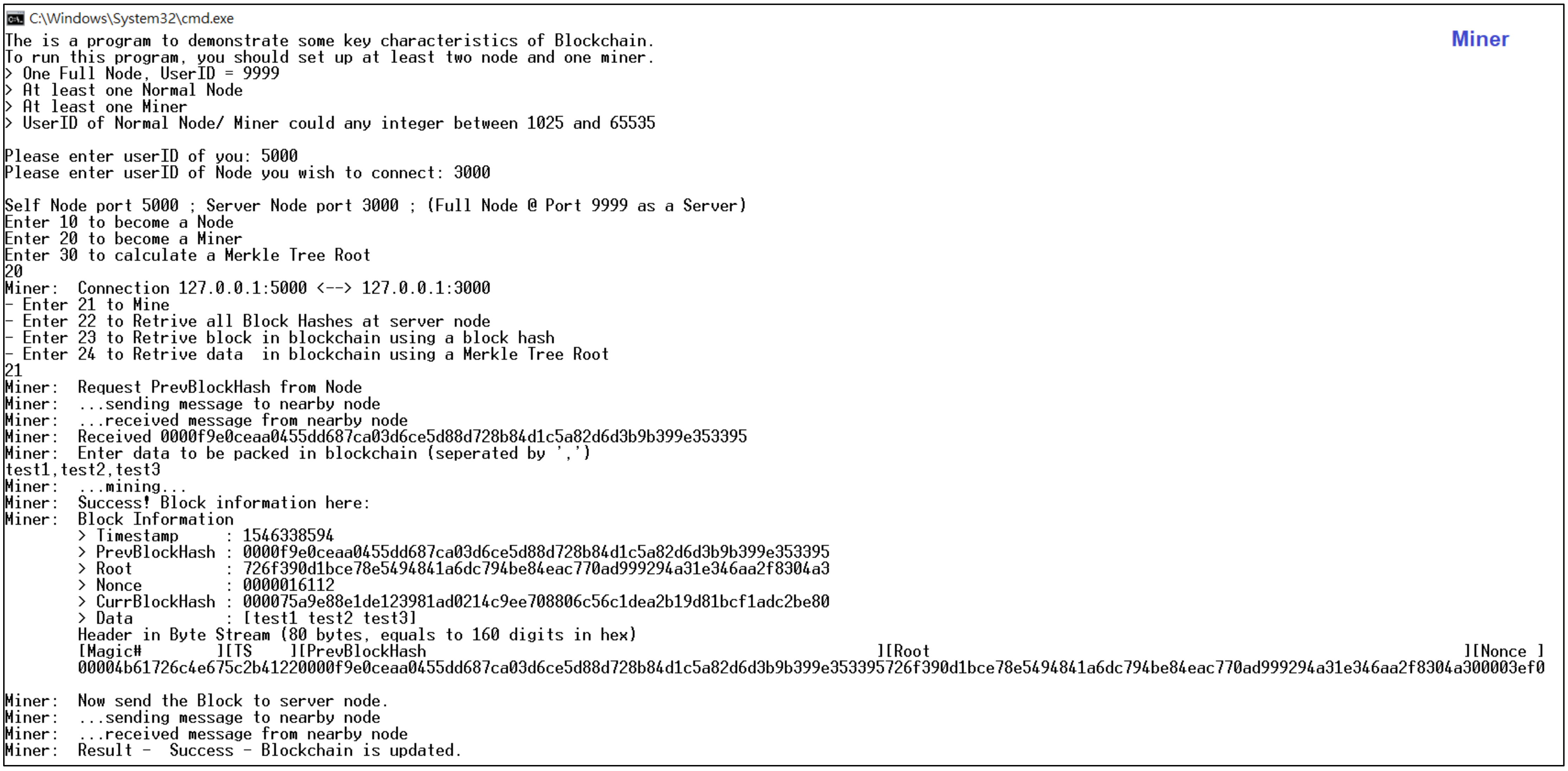
```
Node:    Server Listening on port 9999
Node:    <127.0.0.1:1264> Connection established
Node:    <127.0.0.1:1264> Client would like to retrive all block hashes
Node:    <127.0.0.1:1264> Return information to client.
Node:    <127.0.0.1:1264> Connection is closed
Node:    <127.0.0.1:1475> Connection established
Node:    <127.0.0.1:1475> Client would like to retrive all block hashes
Node:    <127.0.0.1:1475> Return information to client.
Node:    <127.0.0.1:1475> Connection is closed
Node:    <127.0.0.1:1639> Connection established
Node:    <127.0.0.1:1639> Client would like to retrive all block hashes
Node:    <127.0.0.1:1639> Return information to client.
Node:    <127.0.0.1:1639> Connection is closed
Node:    <127.0.0.1:1640> Connection established
Node:    <127.0.0.1:1640> Client would like to retrive all block hashes
Node:    <127.0.0.1:1640> Return information to client.
Node:    <127.0.0.1:1640> Connection is closed
Node:    <127.0.0.1:1641> Connection established
Node:    <127.0.0.1:1641> Miner would like to add a block to blockchain
Node:    <127.0.0.1:1641> Return PrevBlockHash to Miner
Node:    <127.0.0.1:1641> Waiting for new block
Chain:   Success in adding Block to Local Database.
Node:    Blockchain now:
Chain:   Block #0
         > Timestamp     : 1546337922
         > PrevBlockHash : 0000000000000000000000000000000000000000000000000000000000000000
         > Root          : c81c929adb94b88160f1949db98a123096f6bfbd70b6a5d82a30d373a0cc93d4
         > Nonce         : 0000012774
         > CurrBlockHash : 0000f9e0ceaa0455dd687ca03d6ce5d88d728b84d1c5a82d6d3b9b399e353395
         > Data          : [New Genesis Block]
         Block #0 Header in Byte Stream (80 bytes, equals to 160 digits in hex)
         [Magic#        ][TS    ][PrevBlockHash                                                 ][Root                                                          ][Nonce ]
         00004b61726c4e675c2b3e820000000000000000000000000000000000000000000000000000000000000000c81c929adb94b88160f1949db98a123096f6bfbd70b6a5d82a30d373a0cc93d4000031e6

Chain:   Block #1
         > Timestamp     : 1546338594
         > PrevBlockHash : 0000f9e0ceaa0455dd687ca03d6ce5d88d728b84d1c5a82d6d3b9b399e353395
         > Root          : 726f390d1bce78e5494841a6dc794be84eac770ad999294a31e346aa2f8304a3
         > Nonce         : 0000016112
         > CurrBlockHash : 000075a9e88e1de123981ad0214c9ee708806c56c1dea2b19d81bcf1adc2be80
         > Data          : [test1 test2 test3]
         Block #1 Header in Byte Stream (80 bytes, equals to 160 digits in hex)
         [Magic#        ][TS    ][PrevBlockHash                                                 ][Root                                                          ][Nonce ]
         00004b61726c4e675c2b41220000f9e0ceaa0455dd687ca03d6ce5d88d728b84d1c5a82d6d3b9b399e353395726f390d1bce78e5494841a6dc794be84eac770ad999294a31e346aa2f8304a300003ef0

Node:    <127.0.0.1:1642> Connection established
Node:    <127.0.0.1:1641> Connection is closed
Node:    <127.0.0.1:1642> Client would like to retrive all block hashes
Node:    <127.0.0.1:1642> Return information to client.
Node:    <127.0.0.1:1642> Connection is closed
```

Step 8: Normal Node is informed that the block is added to Full Node.
Normal Node updates its local database, then returns the result to Miner.



```
- Enter 11 to Show Blockchain in this server
- Enter 12 to Start acting as a server
12
Chain:  Connected to Full Node for Synchronization of Blockchain : 127.0.0.1:9999
Node:   Blockchain at local Database:
Chain:  Block #0
        > Timestamp     : 1546337922
        > PrevBlockHash : 0000000000000000000000000000000000000000000000000000000000000000
        > Root          : c81c929adb94b88160f1949db98a123096f6bfbd70b6a5d82a30d373a0cc93d4
        > Nonce         : 0000012774
        > CurrBlockHash : 0000f9e0ceaa0455dd687ca03d6ce5d88d728b84d1c5a82d6d3b9b399e353395
        > Data          : []
        Block #0 Header in Byte Stream (80 bytes, equals to 160 digits in hex)
        [Magic#        ][TS     ][PrevBlockHash                                                   ][Root                                                            ][Nonce ]


Node:   Server Listening on port 3000
Node:   <127.0.0.1:5000> Connection established
Node:   <127.0.0.1:5000> Miner would like to add a block to blockchain
Chain:  Connected to Full Node for Synchronization of Blockchain : 127.0.0.1:9999
Node:   <127.0.0.1:5000> Return PrevBlockHash to Miner
Node:   <127.0.0.1:5000> Waiting for new block
Chain:  Connected to Full Node for Synchronization of Blockchain : 127.0.0.1:9999
Chain:  Connected to Full Node for Synchronization of Blockchain : 127.0.0.1:9999
Chain:  Connected to Full Node for Adding Block: 127.0.0.1:9999
Chain:  Result -  Success in adding Block to Full Node
Chain:  Success in adding Block to Local Database.
Chain:  Connected to Full Node for Synchronization of Blockchain : 127.0.0.1:9999
Node:   Blockchain now:
Chain:  Block #0
        > Timestamp     : 1546337922
        > PrevBlockHash : 0000000000000000000000000000000000000000000000000000000000000000
        > Root          : c81c929adb94b88160f1949db98a123096f6bfbd70b6a5d82a30d373a0cc93d4
        > Nonce         : 0000012774
        > CurrBlockHash : 0000f9e0ceaa0455dd687ca03d6ce5d88d728b84d1c5a82d6d3b9b399e353395
        > Data          : []
        Block #0 Header in Byte Stream (80 bytes, equals to 160 digits in hex)
        [Magic#        ][TS     ][PrevBlockHash                                                   ][Root                                                            ][Nonce ]


Chain:  Block #1
        > Timestamp     : 1546338594
        > PrevBlockHash : 0000f9e0ceaa0455dd687ca03d6ce5d88d728b84d1c5a82d6d3b9b399e353395
        > Root          : 726f390d1bce78e5494841a6dc794be84eac770ad999294a31e346aa2f8304a3
        > Nonce         : 0000016112
        > CurrBlockHash : 000075a9e88e1de123981ad0214c9ee708806c56c1dea2b19d81bcf1adc2be80
        > Data          : [test1 test2 test3]
        Block #1 Header in Byte Stream (80 bytes, equals to 160 digits in hex)
        [Magic#        ][TS     ][PrevBlockHash                                                   ][Root                                                            ][Nonce ]
        00004b61726c4e675c2b41220000f9e0ceaa0455dd687ca03d6ce5d88d728b84d1c5a82d6d3b9b399e353395726f390d1bce78e5494841a6dc794be84eac770ad999294a31e346aa2f8304a300003ef0

Node:   <127.0.0.1:5000> Connection is closed
_
```

Step 9: Miner is informed that the block is added to Blockchain.
Miner shows message “Result – Success” on user interface.

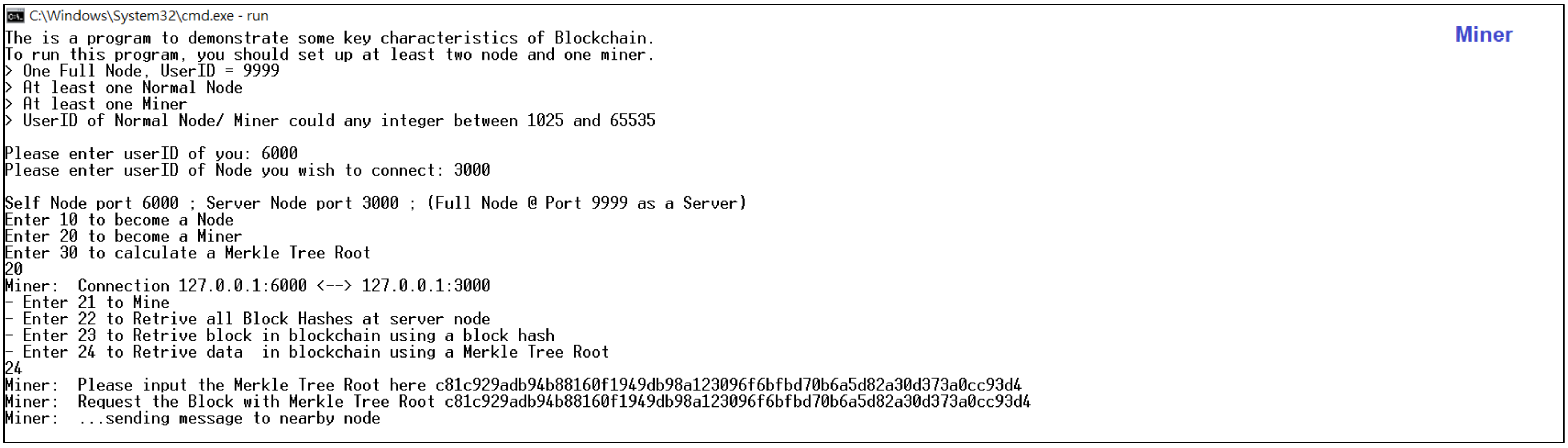


```
The is a program to demonstrate some key characteristics of Blockchain.
To run this program, you should set up at least two node and one miner.
> One Full Node, UserID = 9999
> At least one Normal Node
> At least one Miner
> UserID of Normal Node/ Miner could any integer between 1025 and 65535

Please enter userID of you: 5000
Please enter userID of Node you wish to connect: 3000

Self Node port 5000 ; Server Node port 3000 ; (Full Node @ Port 9999 as a Server)
Enter 10 to become a Node
Enter 20 to become a Miner
Enter 30 to calculate a Merkle Tree Root
20
Miner:  Connection 127.0.0.1:5000 <--> 127.0.0.1:3000
- Enter 21 to Mine
- Enter 22 to Retrive all Block Hashes at server node
- Enter 23 to Retrive block in blockchain using a block hash
- Enter 24 to Retrive data  in blockchain using a Merkle Tree Root
21
Miner:  Request PrevBlockHash from Node
Miner:  ...sending message to nearby node
Miner:  ...received message from nearby node
Miner:  Received 0000f9e0ceaa0455dd687ca03d6ce5d88d728b84d1c5a82d6d3b9b399e353395
Miner:  Enter data to be packed in blockchain (seperated by ',')
test1,test2,test3
Miner:  ...mining...
Miner:  Success! Block information here:
Miner:  Block Information
        > Timestamp     : 1546338594
        > PrevBlockHash : 0000f9e0ceaa0455dd687ca03d6ce5d88d728b84d1c5a82d6d3b9b399e353395
        > Root          : 726f390d1bce78e5494841a6dc794be84eac770ad999294a31e346aa2f8304a3
        > Nonce         : 0000016112
        > CurrBlockHash : 000075a9e88e1de123981ad0214c9ee708806c56c1dea2b19d81bcf1adc2be80
        > Data          : [test1 test2 test3]
        Header in Byte Stream (80 bytes, equals to 160 digits in hex)
        [Magic#        ][TS     ][PrevBlockHash                                                   ][Root                                                            ][Nonce ]
        00004b61726c4e675c2b41220000f9e0ceaa0455dd687ca03d6ce5d88d728b84d1c5a82d6d3b9b399e353395726f390d1bce78e5494841a6dc794be84eac770ad999294a31e346aa2f8304a300003ef0

Miner:  Now send the Block to server node.
Miner:  ...sending message to nearby node
Miner:  ...received message from nearby node
Miner:  Result -  Success - Blockchain is updated.
```

--- End of Mining & Adding New Block to Blockchain ---

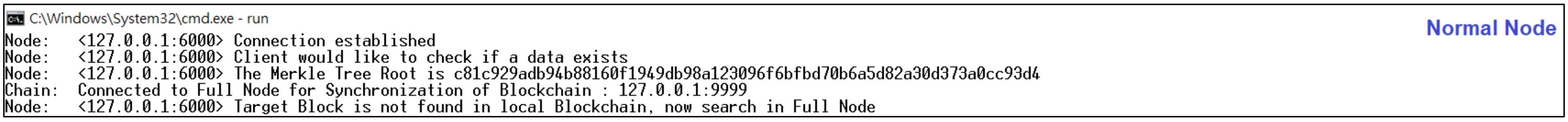
Demonstration of Searching Data

Step 1: Miner would like to search if *[New, Genesis, Block]* is stored in blockchain.
The merkle tree root is `c81c929adb94b88160f1949db98a123096f6bfbd70b6a5d82a30d373a0cc93d4`.
Then Miner sends the search request to Normal Node.



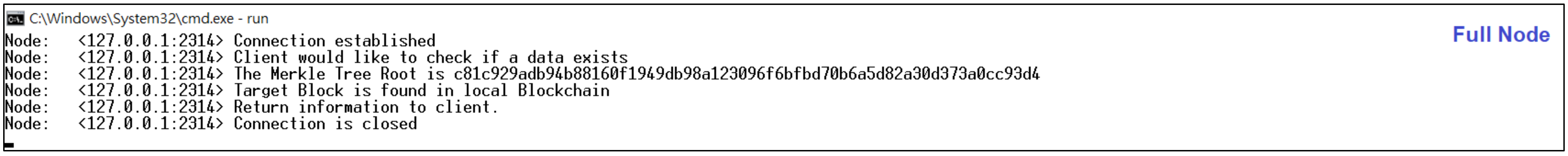
```
The is a program to demonstrate some key characteristics of Blockchain.
To run this program, you should set up at least two node and one miner.
> One Full Node, UserID = 9999
> At least one Normal Node
> At least one Miner
> UserID of Normal Node/ Miner could any integer between 1025 and 65535

Please enter userID of you: 6000
Please enter userID of Node you wish to connect: 3000

Self Node port 6000 ; Server Node port 3000 ; (Full Node @ Port 9999 as a Server)
Enter 10 to become a Node
Enter 20 to become a Miner
Enter 30 to calculate a Merkle Tree Root
20
Miner:  Connection 127.0.0.1:6000 <--> 127.0.0.1:3000
- Enter 21 to Mine
- Enter 22 to Retrive all Block Hashes at server node
- Enter 23 to Retrive block in blockchain using a block hash
- Enter 24 to Retrive data  in blockchain using a Merkle Tree Root
24
Miner:  Please input the Merkle Tree Root here c81c929adb94b88160f1949db98a123096f6bfbd70b6a5d82a30d373a0cc93d4
Miner:  Request the Block with Merkle Tree Root c81c929adb94b88160f1949db98a123096f6bfbd70b6a5d82a30d373a0cc93d4
Miner:  ...sending message to nearby node
```

Step 2: Normal Node receives the search request from Miner. It searches its local database, and data is not found.
Hence, Normal Node sends the search request to Full Node.



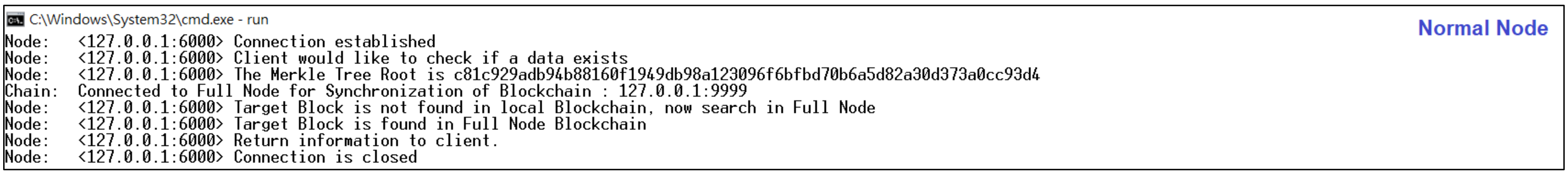
```
Node:   <127.0.0.1:6000> Connection established
Node:   <127.0.0.1:6000> Client would like to check if a data exists
Node:   <127.0.0.1:6000> The Merkle Tree Root is c81c929adb94b88160f1949db98a123096f6bfbd70b6a5d82a30d373a0cc93d4
Chain:  Connected to Full Node for Synchronization of Blockchain : 127.0.0.1:9999
Node:   <127.0.0.1:6000> Target Block is not found in local Blockchain, now search in Full Node
```

Step 3: Full Node receives the search request from Normal Node. It searches its local database, and the data is found.
Hence, Full Node replies the corresponding block information to Normal Node.



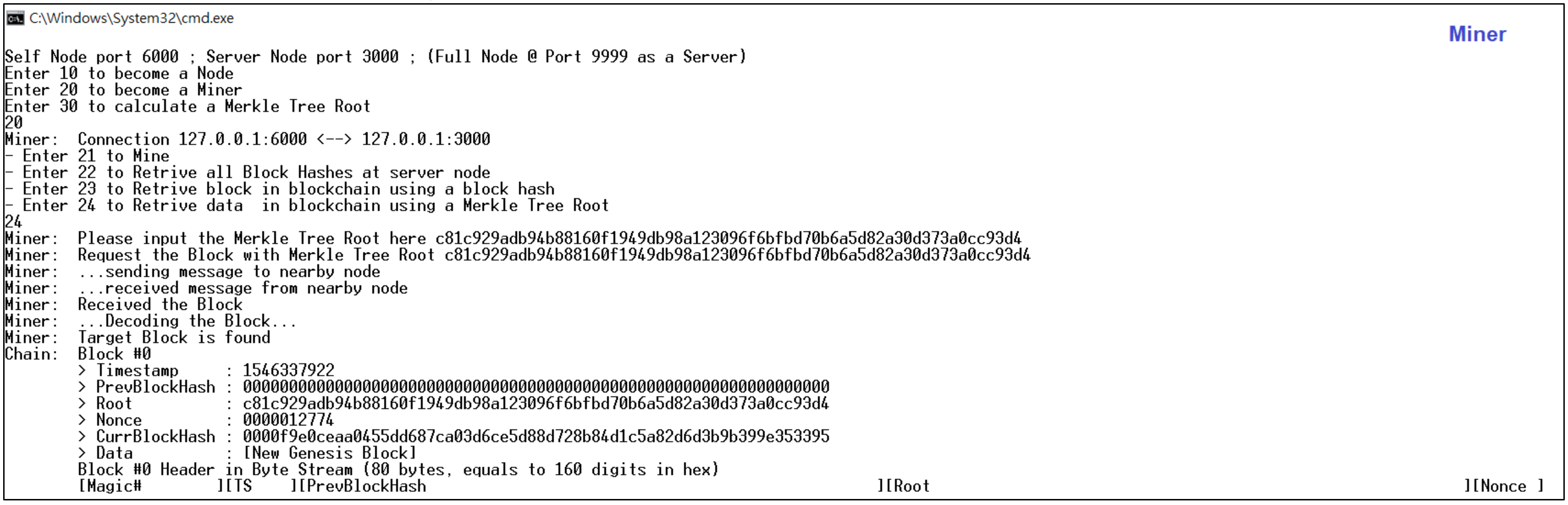
```
Node:   <127.0.0.1:2314> Connection established
Node:   <127.0.0.1:2314> Client would like to check if a data exists
Node:   <127.0.0.1:2314> The Merkle Tree Root is c81c929adb94b88160f1949db98a123096f6bfbd70b6a5d82a30d373a0cc93d4
Node:   <127.0.0.1:2314> Target Block is found in local Blockchain
Node:   <127.0.0.1:2314> Return information to client.
Node:   <127.0.0.1:2314> Connection is closed
```

Step 4: Normal Node receives block from Full Node.
Then it forwards the block to Miner.



```
Node:   <127.0.0.1:6000> Connection established
Node:   <127.0.0.1:6000> Client would like to check if a data exists
Node:   <127.0.0.1:6000> The Merkle Tree Root is c81c929adb94b88160f1949db98a123096f6bfbd70b6a5d82a30d373a0cc93d4
Chain:  Connected to Full Node for Synchronization of Blockchain : 127.0.0.1:9999
Node:   <127.0.0.1:6000> Target Block is not found in local Blockchain, now search in Full Node
Node:   <127.0.0.1:6000> Target Block is found in Full Node Blockchain
Node:   <127.0.0.1:6000> Return information to client.
Node:   <127.0.0.1:6000> Connection is closed
```

Step 5: Miner receives block from Normal Node.
Then it displays "`Target Block is found`" on user interface together with relevant block information.



```
Self Node port 6000 ; Server Node port 3000 ; (Full Node @ Port 9999 as a Server)
Enter 10 to become a Node
Enter 20 to become a Miner
Enter 30 to calculate a Merkle Tree Root
20
Miner:  Connection 127.0.0.1:6000 <--> 127.0.0.1:3000
- Enter 21 to Mine
- Enter 22 to Retrive all Block Hashes at server node
- Enter 23 to Retrive block in blockchain using a block hash
- Enter 24 to Retrive data  in blockchain using a Merkle Tree Root
24
Miner:  Please input the Merkle Tree Root here c81c929adb94b88160f1949db98a123096f6bfbd70b6a5d82a30d373a0cc93d4
Miner:  Request the Block with Merkle Tree Root c81c929adb94b88160f1949db98a123096f6bfbd70b6a5d82a30d373a0cc93d4
Miner:  ...sending message to nearby node
Miner:  ...received message from nearby node
Miner:  Received the Block
Miner:  ...Decoding the Block...
Miner:  Target Block is found
Chain:  Block #0
          > Timestamp     : 1546337922
          > PrevBlockHash : 0000000000000000000000000000000000000000000000000000000000000000
          > Root          : c81c929adb94b88160f1949db98a123096f6bfbd70b6a5d82a30d373a0cc93d4
          > Nonce         : 0000012774
          > CurrBlockHash : 0000f9e0ceaa0455dd687ca03d6ce5d88d728b84d1c5a82d6d3b9b399e353395
          > Data          : [New Genesis Block]
        Block #0 Header in Byte Stream (80 bytes, equals to 160 digits in hex)
        [Magic#        ][TS    ][PrevBlockHash                                                 ][Root                                                          ][Nonce ]
```

-- End of Searching Data --

8. Summary and Extra Information – Blockchain should be in a decentralized network

Blockchain is a distributed, open and unalterable digital ledger which is suitable for data storage in decentralized system.

Open and Unalterable

In this project, I demonstrated

- why blockchain is an unalterable ledger
- why blockchain is an open ledger

Blockchain is unalterable because merkle tree root of data is saved in the header hashes chain. Any modification of data will lead to a change of the whole chain, which make it nearly impossible to change any data without creating a new chain. Since all nodes only agree to follow the existing chain in blockchain network, this newly created chain will not be agreed by any node. Hence, blockchain is unalterable digital ledger.

Blockchain is an open ledger because every node agrees to provide information to other nodes if it receives request with valid message header. It means that the blockchain is “open” to every node. An implementation is that any node can help verifying data without connecting to a centralized server. All node holds a part of the blockchain, which could be used for verifying data. In case if data is not found in this part of blockchain held by the node, it could send request to other nodes in the network to ask for help in verifying data.

Decentralized

In this project, the blockchain works in a centralized system. It could be modified to run in a decentralized system by following:

Blockchain in this project could be modified to be a decentralized network if any node can perform mining (i.e. save data in blockchain) without the approval of a centralized server. In this project, a centralized Full Node model is adopted. Normal Node need to update its header hashes chain from Full Node before mining, and only blocks admitted by Full Node could be added to the blockchain. To build a decentralized system, we could replace the Full Node by neighbour nodes (i.e. Normal Node update its header hashes chain by asking neighbour nodes, but not a Full Node, before mining). Successfully mining a block means that majority of nodes in the network agrees to admit this new block into the blockchain.

The most important problem in a decentralized system how to achieve this “*majority of nodes agrees to admit this new block into the blockchain*”. In other words, making “consensus”. For instance, if two miners successfully create a block simultaneously, nodes in the network should mutually agree that which block should be added into the blockchain.

Bitcoin solves this consensus problem by using proof-of-work algorithm. Nodes agree to add blocks into “the most-difficult-to-recreate chain”. Hence, proof-of-work algorithm is introduced to increase the difficulty of creating a block, and subsequently creating a chain. Below is explanation of why a consensus could be made.

> *“The proof-of-work also solves the problem of determining representation in majority decision making. If the majority were based on one-IP-address-one-vote, it could be subverted by anyone able to allocate many IPs. Proof-of-work is essentially one-CPU-one-vote. The majority decision is represented by the longest chain, which has the greatest proof-of-work effort invested in it. If a majority of CPU power is controlled by honest nodes, the honest chain will grow the fastest and outpace any competing chains. To modify a past block, an attacker would have to redo the proof-of-work of the block and all blocks after it and then catch up with and surpass the work of the honest nodes.”* [1]

Please refer to https://bitcoin.org/en/developer-reference#p2p-network for further information about how blockchain works in a decentralized network of Bitcoin.

Although proof-of-work could solve the consensus problem, it is criticized because huge computational power and energy are consumed for calculation of *Nonce*. Therefore, other consensus algorithms are invented to make consensus without such a waste of computational power. Examples of alternative consensus algorithms are “proof-of-stake” in Ethereum, “delegated byzantine fault tolerance” in NEO, etc.

9. Reference

1. "Bitcoin: A Peer-to-Peer Electronic Cash System", Satoshi Nakamoto, https://bitcoin.org/bitcoin.pdf
2. “Bitcoin Developer Reference”, Bitcoin.org, https://bitcoin.org/en/developer-reference
3. “Bitcoin Developer Reference – P2P Network”, Bitcoin.org, https://bitcoin.org/en/developer-reference#p2p-network

The source codes are written using Visual Studio Code. https://code.visualstudio.com/
Most of the diagrams are drawn using Software Ideas Modeler. https://www.softwareideas.net/

This project is a rebuild of student project I worked on for The Hong Kong Polytechnic University COMP5311 (Semester 1 Year 2018-2019). Special thanks to Dr Bin Xiao and Mr Haotian Wu, who taught me the fundamentals of Bitcoin and blockchain.

--End--