/\* \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Kumaresh Vijayakumar / Date 10-09-2022

2. Java version used for the execution of addition program :

java version "18.0.2.1" 2022-08-18

3. Here is commands which are used in compiling exact blockchain.java:

javac -cp "gson-2.8.2.jar" Blockchain.java

4. Here is command which are used to execute the exact blockchain.java:

java -cp ".;gson-2.8.2.jar" Blockchain 0 (Windows user)

5. Content :

a. initially Process 2 sends the start message to process 0 and 1. This is the place where blocks start to verify and puzzle solving is initiated.

b. sleep command is used in the program which is used to hold the process at each and every time for the verification of the blocks and to coordinate all the three processes.

c. Once each and every process starts to get the initiate signal form the process 2 the further process started,

- multicasting is done in public keys.

- creating a dummy block which is random and the dummy block will be the first block when the process is initiated.

- Then begin reading of the input files takes place.

- Each and every process in the blockchain has a duplicate of the ledger which they only access and to contribute later the validating block.

- If no other block is found inside it will be added to it as the newly verified block even if it is not created as new but it is changed or altered.

- By submitting the verified block to each and other process and including it into the block chain, I have constructed the complete blockchain.

7. Credits:

Thanks to Prof. Clark Elliott for providing the utility code and sample output. It was very much helpful in building the blockchain assignment.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\*/

BLOCKCHAIN Code:

import com.google.gson.Gson; // here we get the gson libraries to build the object of gson

import com.google.gson.GsonBuilder;

import com.google.gson.reflect.TypeToken;

import java.io.\*; // here we get all the I|O library

import java.lang.reflect.Type; // importing the libraries for network server sockets

import java.net.ServerSocket;

import java.net.Socket; // the Java socket libraries are imported

import java.nio.charset.StandardCharsets; // adding the Java charset libraries for UTF-8

import java.security.\*; // the Java security libraries are imported

import java.security.spec.X509EncodedKeySpec; // importing the java security library to provide public key encoding

import java.util.\*; // adding the Java tool libraries

import java.util.concurrent.BlockingQueue; // the Java blocking queue libraries are imported

import java.util.concurrent.PriorityBlockingQueue; // the Java priority blocking queue libraries are imported

public class Blockchain // here we introduce main class for blockchain

{

public static void main(String[] args) // here we introduce blockchain class's main() method

{

int process\_ID;

if (args.length < 1) {

process\_ID = 0; // If no argument is supplied, the default value is taken to be 0.

}

switch (args[0]) //We start the 3 separate Blockchain processes based on the parameter supplied in the console, which can be process 0, 1, or 2

{

//assigning the value "0" to process\_ID for further processing after determining whether the argument pass is 0

case "0":

process\_ID = 0;

break;

// assigning the value "1" to process\_ID for further processing after determining whether the argument pass is 1

case "1":

process\_ID = 1;

break;

// assigning the value "2" to process\_ID for further processing after determining whether the argument pass is 2

case "2":

process\_ID = 2;

break;

// If nothing is passed then the default value for the processID is value "0"

default:

process\_ID = 0;

break;

}

Block\_Task bcTtd = new Block\_Task(process\_ID); // sending the process ID to the blockChainTaskToDo class, which handles all blockchain-related tasks.

// By separating the blockchain work from the primary functionality, this will help maintain the code's cleanliness

}

}

class Block\_Record implements Serializable // here we are generating block record class

{

private String IDofBlock; // during the read-in of a data record from a file, a unique block ID is established by declaring a block id variable

private String RegisteredIDofBlock; // simply defining the signed block id variable

private String timeStamp; // simply defining the variable timestamp

private String B\_Number; // simply defining the variable block number in the name of B\_number

private String F\_Name; // simply defining the variable for patients first name

private String L\_Name; // simply defining the variable for patients last name

private String DOB; // simply defining the variable for patients date of birth

private String ssn\_Number; // simply defining the variable of patients social security number

private String Med\_Diagnosis; // simply defining the varibale of patients diagnosis details of the patient

private String Med\_Treatment; // simply defining the variable of the patients treatment details

private String Med\_Prex; // simply defining the variable of the patients medical prescription

private String Hash\_Creator; // simply defining the variable for the variable hash creator

private String Hash\_Signed\_Creator; // simply defining the variable for hash signed creator

private String Value\_of\_Previous\_Hash; // simply defining the variable for value of previous hash

private String Value\_of\_Winning\_Hash; // simply defining the variable for value of winning hash

private String Value\_of\_WinningSigned\_Hash; // simply defining the variable for value of winning registered|signed hash

private String ValueofRandomSeed; // simply defining the variable for value of the random seed given in the block chain

private String Verify\_process\_ID; // simply defining the variable for verify process id which helps in verifying the blocks which are given as the input

private String Creating\_Process; // simply defining the variable creating process which initiates the process during the block chain process

private UUID uuid; // simply defining the variable uuid which is used to identify an unique object or entity

// accessors for such variable declaration above are shown below

public String getIDofBlock() // for the ID of block variable's getter method

{

return IDofBlock;

}

public void setIDofBlock(String IDofBlock) // creating the ID of block variable's setter method

{

this.IDofBlock = IDofBlock;

}

public String getRegisteredIDofBlock() // for registered ID of block variable's getter method

{

return RegisteredIDofBlock;

}

public void setRegisteredIDofBlock(String RegisteredIDofBlock) // creating the registered ID of block variable's setter method

{

this.RegisteredIDofBlock = RegisteredIDofBlock;

}

public String getTimeStamp() // for timestamp variable's getter method

{

return timeStamp;

}

public void setTimeStamp(String timeStamp) // creating the timestamp variable's setter method

{

this.timeStamp = timeStamp;

}

public String getB\_Number() // for B\_Number variable's getter method

{

return B\_Number;

}

public void setB\_Number(String B\_Number) // creating the B\_Number variable's setter method

{

this.B\_Number = B\_Number;

}

public String getF\_Name() // for first name as F\_Name variable's getter method

{

return F\_Name;

}

public void setF\_Name(String F\_Name) // creating the F\_name variable's setter method

{

this.F\_Name = F\_Name;

}

public String getL\_Name() // for last name as L\_Name variable's getter method

{

return L\_Name;

}

public void setL\_Name(String L\_Name) // creating the L\_name variable's setter method

{

this.L\_Name = L\_Name;

}

public String getDOB() // for date of birth DOB variable's getter method

{

return DOB;

}

public void setDOB(String DOB) // creating the DOB variable's setter method

{

this.DOB = DOB;

}

public String getssn\_Number() // for ssn\_Number variable's getter method

{

return ssn\_Number;

}

public void setssn\_Number(String ssn\_Number) // creating the ssn\_Number variable's setter method

{

this.ssn\_Number = ssn\_Number;

}

public String getMed\_Diagnosis() // for medical diagnosis of patient Med\_Diagnosis variable's getter method

{

return Med\_Diagnosis;

}

public void setMed\_Diagnosis(String Med\_Diagnosis) // creating the Med\_Diagnosis variable's setter method

{

this.Med\_Diagnosis = Med\_Diagnosis;

}

public String getMed\_Treatment() // for medical treatment of the patient Med\_Treatment variable's getter method

{

return Med\_Treatment;

}

public void setMed\_Treatment(String Med\_Treatment) // creating the Med\_Treatment variable's setter method

{

this.Med\_Treatment = Med\_Treatment;

}

public String getMed\_Prex() // for medical prescription of the patient Med\_Prex variable's getter method

{

return Med\_Prex;

}

public void setMed\_Prex(String Med\_Prex) // creating the Med\_Prex variable's setter method

{

this.Med\_Prex = Med\_Prex;

}

public String getHash\_Creator() // for Hash\_Creator variable's getter method

{

return Hash\_Creator;

}

public void setHash\_Creator(String Hash\_Creator) // creating Hash\_Creator variable's setter method

{

this.Hash\_Creator = Hash\_Creator;

}

public String getHash\_Signed\_Creator() // for hash\_signed\_creator variable's getter method

{

return Hash\_Signed\_Creator;

}

public void setHash\_Signed\_Creator(String Hash\_Signed\_Creator) // creating hash\_signed\_creator variable's setter method

{

this.Hash\_Signed\_Creator = Hash\_Signed\_Creator;

}

public String getValue\_of\_Previous\_Hash() // for value\_of\_previous\_hash variable's getter method

{

return Value\_of\_Previous\_Hash;

}

public void setValue\_of\_Previous\_Hash(String Value\_of\_Previous\_Hash) // creating value\_of\_previous\_hash variable's setter method

{

this.Value\_of\_Previous\_Hash = Value\_of\_Previous\_Hash;

}

public String getValue\_of\_Winning\_Hash() // for value\_of\_winning\_hash variable's getter method

{

return Value\_of\_Winning\_Hash;

}

public void setValue\_of\_Winning\_Hash(String Value\_of\_Winning\_Hash) // creating value\_of\_winning\_hash variable's setter method

{

this.Value\_of\_Winning\_Hash = Value\_of\_Winning\_Hash;

}

public String getValue\_of\_WinningSigned\_Hash() // for value\_of\_winningsigned\_hash variable's getter method

{

return Value\_of\_WinningSigned\_Hash;

}

public void setValue\_of\_WinningSigned\_Hash(String Value\_of\_WinningSigned\_Hash) // creating value\_of\_winningsigned\_hash variable's setter method

{

this.Value\_of\_WinningSigned\_Hash = Value\_of\_WinningSigned\_Hash;

}

public String getValueofRandomSeed() // for value of random seed variable's getter method

{

return ValueofRandomSeed;

}

public void setValueofRandomSeed(String ValueofRandomSeed) // creating value of random seed variable's setter method

{

this.ValueofRandomSeed = ValueofRandomSeed;

}

public String getVerify\_process\_ID() // for verify process we use verify\_process\_ID its variable's getter method is mentioned here

{

return Verify\_process\_ID;

}

public void setVerify\_process\_ID(String Verify\_process\_ID) // creating verify process id variable's setter method this variable is used for the blocks to be verifyyied

{

this.Verify\_process\_ID = Verify\_process\_ID;

}

public String getCreating\_Process() // for creating process we use creating\_process its variable's getter method is mentioned here

{

return Creating\_Process;

}

public void setCreating\_Process(String Creating\_Process) // creating the setter method for the Creating\_process

{

this.Creating\_Process = Creating\_Process;

}

public UUID getUuid() // for uuid variable's getter method id mentioned here

{

return uuid;

}

public void setUuid(UUID uuid) // creating the setter method for the UUID variable

{

this.uuid = uuid;

}

public String toString()

{

return "Block\_Record{" +

"IDofBlock='" + IDofBlock + '\'' +

", RegisteredIDofBlock='" + RegisteredIDofBlock + '\'' +

", timeStamp='" + timeStamp + '\'' +

", B\_Number='" + B\_Number + '\'' +

", F\_Name='" + F\_Name + '\'' +

", L\_Name='" + L\_Name + '\'' +

", DOB='" + DOB + '\'' +

", ssn\_Number='" + ssn\_Number + '\'' +

", Med\_Diagnosis='" + Med\_Diagnosis + '\'' +

", Med\_Treatment='" + Med\_Treatment + '\'' +

", Med\_Prex='" + Med\_Prex + '\'' +

", Hash\_Creator='" + Hash\_Creator + '\'' +

", Hash\_Signed\_Creator='" + Hash\_Signed\_Creator + '\'' +

", Value\_of\_Previous\_Hash='" + Value\_of\_Previous\_Hash + '\'' +

", Value\_of\_Winning\_Hash='" + Value\_of\_Winning\_Hash + '\'' +

", Value\_of\_WinningSigned\_Hash='" + Value\_of\_WinningSigned\_Hash + '\'' +

", ValueofRandomSeed='" + ValueofRandomSeed + '\'' +

", Verify\_process\_ID='" + Verify\_process\_ID + '\'' +

", Creating\_Process='" + Creating\_Process + '\'' +

", uuid=" + uuid +

'}';

}

}

class Block\_Task // here is the new class creation in the name of 'Block\_Task'

{

public static int process\_ID; // maintains the current process id that will be fetched from the Blockchain class in a locally defined process ID variable

public static int Total\_Process = 3; // maximum number of concurrent processes. Can manage any number of peers simply altering this number

public static String Name = "localhost"; // declaring and setting the server name variable with in method

public static boolean BeginofProcessFlag = false; // a flag variable when modified to 'true', indicates all processes can begin running. Set to 'false'at the beginning.

public static boolean Flag\_pk = false; // defining a public key flag variable and creating it

public static int Count\_pk = 0; // defining and initiating a variable for the public key counter

public static KeyPair PairKeys; // used to save the processes key pair

public static PublicKey[] publicKeyList = new PublicKey[Total\_Process]; // for three processes, it will store three public keys in the array, which is utilized to save our processes public keys

public static final PriorityBlockingQueue<Block\_Record> blockQueue = new PriorityBlockingQueue<>(50, new Block\_Comparator()); // defining the queue for all the unverified blocks and all the process will be assigned a block by itself to solve the puzzel

public static LinkedList<Block\_Record> brList = new LinkedList<>(); // this is used to save all the unverified blocks;this is for the initial process of execution

public static LinkedList<Block\_Record> bcLedger = new LinkedList<>(); // this is used to save all the verified blocks which is stored in the ledger which is created after execution

private static final int iFName = 0; // these are the indexes of the token which is considered as the input files of data; here index of first name is to be 0

private static final int iLName = 1; // here index of the last name is to be 1

private static final int iDob = 2; // here index of the dob is to be 2

private static final int iSsnNum = 3; // here index of ssn is to be 3

private static final int iMed\_Diagnosis = 4; // here index of patients medical diagnosis is to be 4

private static final int iMed\_Treatment = 5; // here index of the patients medical treatment is to be 5

private static final int iMed\_Prex = 6; // here index of the patients medical prescription is to be 6

public Block\_Task(int process\_ID) // this is the constructor class that matches the present local variables and processid later begins and makes all the ports to the specific process finally executes run method and initiate the process

{

Block\_Task.process\_ID = process\_ID;

new Ports().setPorts(process\_ID); // assigning the ports for each processID after creating an instance of the portsclass

Block\_run(); // introducing the new method 'Block\_run()' which can be proceeded to the next continuing process

}

public void Block\_run() // begining of the 'Block\_run()' method

{

System.out.println("Blockchain of Kumaresh Vijayakumar is initiating...\n"); // this is the output message which will be in the output|console page in the begining of the blokchain process

System.out.println("There will be a sleep time of 30 Secs...\n");

System.out.println("Used Text File is " + String.format("BlockInput%d.txt", process\_ID)); // this is the output message which will be in the console page which mentions the user that which input is used

new Thread(new InitiateMainServer()).start(); // here new thread has been created so that main server will be started for execution this will change the beginofprocessflag to true from false

new Thread(new PK\_Server()).start(); // here new thread has been created which helps to get all the public keys which are needed to all the process

new Thread(new UnverifiedBlock(blockQueue)).start(); // here new thread has been created to get all the blocks which are unverified at the inital state for every process

new Thread(new UpdatedBCSever()).start(); // here new thread has been created to get all the update blocks from every process and later it is to be printed in the created ledger file

try { // here sleep process is initiated for 3 seconds as every process have to be active before proceeding further

Thread.sleep(3000);

} catch (Exception exception)

{

exception.printStackTrace(); // handling the exception for the try-catch

}

if (process\_ID == 2) // here is a condition that if the obtained process id is 2 then all the process will be initiated and later convey's the signal to initiate process

{

Initiateallprocess(); // introducing start all process

}

try { // here pair keys is created and new method in the name of 'generateKeyPair()'is introduced;this is done as we need to send the random seed values

PairKeys = generateKeyPair(999);

} catch (Exception exception)

{

exception.printStackTrace(); // handling exception for try-catch

}

while (!BeginofProcessFlag) // here if the beginofprocessflag is not true as mentioned above need to wait for every process start and so calls the method sleep

{

Sleep\_calls();

}

System.out.println("Initialising..."); // this is the output message in the console|user side when starting the process

PulicKeys\_Mcast(); // introducing new method where every process share the generated public keys between them

while (!Flag\_pk) // if flag\_pk if not true, it will automatically calls the sleeps method and wait for the turn

{

Sleep\_calls();

}

if (process\_ID == 0) // when proocess\_ID equlas 0, introduce the random fake entry which is blocks

{

GeneisBlock();

}

Input\_readfile(); // introducing new method in the name of 'Input\_readfile()' here every process makes its own inputs

M\_Cast\_toProcess(); // calling the M\_Cast\_toProcess()function to send unverified blocks to all processes in order to complete the task and verify the block

try { // calling sleep so every process finishes the respective process and multicasting un-verified block before going to compete

Thread.sleep(3000);

} catch (InterruptedException exception)

{

exception.printStackTrace(); // handling exception for try-catch

}

new Thread( new PuzzleWork(blockQueue)).start(); // this thread is to processes which are solving the puzzlework

try {

Thread.sleep(31000); //

} catch (Exception exception) // handling exception for try-catch

{

exception.printStackTrace();

}

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"); // this is the display message in the console side of the user who gives the input file and start the blockchain process

System.out.println("Ledger of Blockchain in the name of 'BlockchainLedger' is created and the format will be JSON");

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

}

// sleep calls method is used to hold the process as every process comes together for the execution process

public static void Sleep\_calls()

{

try {

Thread.sleep(1000);

} catch (Exception exception)

{

exception.printStackTrace();

}

}

// this method checks the duplicate records which are in the generated ledger;If there is an duplicate blocks of record

public static boolean Duplicate(Block\_Record Block\_RecordIn)

{

// here we are establishing a local variable to store the block record that the calling function provided in as a parameter

Block\_Record checkRec = Block\_RecordIn;

//Reading our blockchain ledger's data

for (Block\_Record Block\_Record : bcLedger)

{

// here we are utilizing the block id field, compares our provided blockRec with each record that already exists in the ledger. If a match is found, true is returned

if (checkRec.getIDofBlock().equals(Block\_Record.getIDofBlock()))

return true;

}

// if-not it is returned as false

return false;

}

// a process for producing a pair of keys, a public key and a private key. It is contained in the utility code that Professor Elliott provided.

public static KeyPair generateKeyPair(long randomSeed) throws Exception

{

//using the Java KeyPairGenerator class by building a generateKey object. Basically, creating a 1024 bit key pair using the Digital Signature (RSA) algorithm involves calling the getInstance() function on it and giving the encryption algorithm "RSA"

KeyPairGenerator generateKey = KeyPairGenerator.getInstance("RSA");

// making a SecureRandom object and reinitializing it with a new random seed value

SecureRandom RNG = SecureRandom.getInstance("SHA1PRNG", "SUN");

RNG.setSeed(randomSeed);

// setting the generate key instance's initial key size

generateKey.initialize(1024, RNG);

// returns to the caller method after using generateKeyPair() to create the key pair

return (generateKey.generateKeyPair());

}

// P2 uses this method to signal all other processes to "initiate." They flip the BeginofProcessFlag after the signal is received and proceed with the remaining operation or job

public boolean Initiateallprocess()

{

// defining initiate socket variable to start all the process

Socket initiateSocket;

// defining toServer variable which is to print the process

PrintStream toServer;

try {

// here we are counting the total number of processes in a loop

for (int i = 0; i < Total\_Process; i++)

{

// here we are setting up the initiateSocket with the localhost address and the initiateServer port base and process number

initiateSocket = new Socket(Name, Ports.portbaseInitiateServer + i);

// by using initiateSocket to create a new printStream object and assigning it to toServer

toServer = new PrintStream(initiateSocket.getOutputStream());

// here it is giving the commands as it is starting the process

toServer.println("Initiate");

// here it shows that printing the signal transmission info in the process2

System.out.println("Sending the initiating signal.....");

// here we are flushing the stream output

toServer.flush();

// here is the closes the initiate socket

initiateSocket.close();

}

} catch (Exception exception)

{

// handling the exception in try-catch block

exception.printStackTrace();

}

// here is the important message that it returns true; This is done to initiate all the process

return true;

}

// Obtains the public key and broadcasts it to other processes using the multicast of public keys technique. Additionally, it has a Process ID that will enable us to determine which process's public key was utilized to verify the block.

public void PulicKeys\_Mcast()

{

// defining the MCsocket variable which is used to multicast publickeys

Socket MCSocket;

// defining toServer variable which is to print the process

PrintStream toServer;

byte[] publicKey = PairKeys.getPublic().getEncoded();

// here we are translating the digital signature's byte[] format into string format before inserting it into our block

String strPublicKey = Base64.getEncoder().encodeToString(publicKey);

// here we are showing the terminal the public key string

System.out.println("For Multicasting Public key is created" + strPublicKey);

try {

// counting the total number of processes in a loop

for (int i = 0; i < Total\_Process; i++)

{

// here we are forming a object of socket named "MCSocket" and sending the public key server port number and the server name

MCSocket = new Socket(Name, Ports.portKeyofBaseServer + i);

// here to multicast the keys, create a printStream object and assign an output stream to the socket mentioned above

toServer = new PrintStream(MCSocket.getOutputStream());

// here we are combining all the processID and string of public key

String pIDPublicKey = process\_ID + " " + strPublicKey;

// here we send all the publickeys to every server and process

toServer.println(pIDPublicKey);

// here we flush the printing process

toServer.flush();

// here we close the MCSocket

MCSocket.close();

}

} catch (Exception exception)

{

// handling the exception in try-catch block

exception.printStackTrace();

}

}

// here is the method which helps in multicasting not verified blocks with in every process; By doing this each and every process try to solve the problem and validate the bricks

public void M\_Cast\_toProcess()

{

// defining the variable for socket

Socket s;

// defining the variable for prntstream

PrintStream toServer;

// defining the variable for Temp\_block

Block\_Record temp\_Block;

// defining an iterator to iterate over each record in our unverified block linked list

Iterator<Block\_Record> iteratorVar = brList.iterator();

try {

while (iteratorVar.hasNext())

{

temp\_Block = iteratorVar.next();

// generating block records in JSON

String blockRec = Builder\_json(temp\_Block);

for (int i = 0; i < Total\_Process; i++)

{

//setting up a connection for each process on its uv block server

s = new Socket(Name, Ports.portofBaseUBServer + i);

toServer = new PrintStream(s.getOutputStream());

// here we forward the blockrecord to every process

toServer.println(blockRec);

// here flush() is used to slush the prntstream

toServer.flush();

// here the socket is closing the connection

s.close();

}

}

} catch (Exception excpt)

{

// handling the exception in try-catch block

excpt.printStackTrace();

}

}

// this method is used to form the fake block which is dummy block which acts as a basics of each and every process. By arranging the initial values for the blocks we are confirming that every block finds their data in common

public static void GeneisBlock()

{

// defining the variable of dataofSHA256

String dataofSHA256;

// making an object of type Block\_Record

Block\_Record blockRec = new Block\_Record();

// constructing a date type object that retrieves the system-formatted date to store date values

// Ex: Mon Oct 24 01:10:10 CST 2022

Date dateValue = new Date();

// defining and launching the timevalue variable - which helps in fetching live time

long timeValue = dateValue.getTime();

// here we change the timevalue in different format which is str

String strTimeValue = String.valueOf(timeValue);

// the process\_ID is added after the time value to create a timestamp

// This will assist in resolving the conflict between block records' identical timestamps

String timeStamped = strTimeValue + "." + process\_ID;

// UUIDs are generated at random and assigned to the setUUID variable.

String setUUID = UUID.randomUUID().toString();

// Initializing (setting default values for) each field in our block record

blockRec.setIDofBlock(setUUID);

blockRec.setTimeStamp(timeStamped);

blockRec.setF\_Name("George");

blockRec.setL\_Name("Bushel");

blockRec.setssn\_Number("111-00-1111");

blockRec.setDOB("1890.10.10");

blockRec.setMed\_Diagnosis("Cancer");

blockRec.setMed\_Treatment("Chemotheraphy");

blockRec.setMed\_Prex("HealthyFood");

blockRec.setValue\_of\_Previous\_Hash("1111111111");

blockRec.setB\_Number("1");

// putting our block record values in a string format that will help create a SHA256 hash value later

String Block\_Record = blockRec.getIDofBlock() +

blockRec.getF\_Name() +

blockRec.getL\_Name() +

blockRec.getssn\_Number() +

blockRec.getDOB() +

blockRec.getMed\_Diagnosis() +

blockRec.getMed\_Treatment() +

blockRec.getMed\_Prex();

// here we are introducing the message digest function in the str blocksrecord in the name of string\_builder

dataofSHA256 = String\_Builder(Block\_Record);

// here we are making the value of hash of block which is generated now as value of wining hash as fake record (dummy)

blockRec.setValue\_of\_Winning\_Hash(dataofSHA256);

// here we join the first dummy to the ledger of blockchain as index of 0

bcLedger.add(0, blockRec);

// this is the output message on the console which shows the perfect size of the created ledger

System.out.println("BlockChain Ledger size is " + bcLedger.size());

// here this allows the dummy record to be printed on the created ledger

if (process\_ID == 0)

{

// here is the output message in the console ;shows the required process is initiated

System.out.println("First Block - Not real entry");

// here it helps in the blocks to be delivered in the created ledger

BlocktoLedger(blockRec, "Block Ledger Update");

// here we are printing JSON on the preferred disk

write2JSON();

}

}

// Java object marshalling technique utilizing gson. This function takes a blockrec as input and converts it to JSON; it then returns the result in JSON format

public static String Builder\_json(Block\_Record blockRec)

{

Gson gson = new GsonBuilder().setPrettyPrinting().create();

String json = gson.toJson(blockRec);

return json;

}

// method that reads in BlockInput0.txt, BlockInput1.txt, and BlockInput2.txt, the three input files. Then, uses the token that was created for each data value to construct an unverified block. A SHA256 hash string value will be present in this block, which will help later when producing a digital signature for auditing

public static void Input\_readfile()

{

// formatting the input file according to the relevant process ID Making it dynamic so that peers of any number may be entered blocks of input

String inputFile = String.format("BlockInput%d.txt", process\_ID);

try {

// the inputData variable is read from each process's own input file

BufferedReader inputData = new BufferedReader(new FileReader(inputFile));

// making tokens to arrange the input data in accordance with the specified java variables for it

String[] DTokens;

// a string that contains all text input data

String inputStrData;

// defining blockUUID as a variable to hold the distinct blockID

String blockUUID;

try {

// if the inputData is not equals to 0, then

while ((inputStrData = inputData.readLine()) != null)

{

// making instance of date in the inputreadfile

Date dateValue = new Date();

// making instance for blockrecord

Block\_Record blockRec = new Block\_Record();

// defining the variable of the timevalue which helps is getting the live time

long timeValue = dateValue.getTime();

// helps if changing to str format from the time format

String timeStamp = String.valueOf(timeValue);

// block records with identical timestamps can be resolved by constructing a timestamp from a time value and adding the processID to it

String timeStampPID = timeStamp + "." + process\_ID;

// create random UUIDs, then assign them to the setUUID variable

blockUUID = UUID.randomUUID().toString();

// here we do splitting the data of inputs to tokens and save''s all the token in this format 'String[]'

DTokens = inputStrData.split(" +");

// defining the variable of the signedBlock

String signedBlock = "";

try {

// introducing the new method named as 'DataSignature' which applies the signature digitally into our block

byte[] digitalSign1 = DataSignature(blockUUID.getBytes(), PairKeys.getPrivate());

// the signature which is in digital format is encode with the help of base64 and maped to variable of signed block

signedBlock = Base64.getEncoder().encodeToString(digitalSign1);

} catch (Exception excpt)

{

// handling exception in try-catch block

excpt.printStackTrace();

}

// Configuring read-in data and updating UUID, signedBlockID, creator process ID, and current timestamp

blockRec.setIDofBlock(blockUUID);

blockRec.setTimeStamp(timeStampPID);

blockRec.setRegisteredIDofBlock(signedBlock);

blockRec.setCreating\_Process(String.valueOf(process\_ID));

blockRec.setF\_Name(DTokens[iFName]);

blockRec.setL\_Name(DTokens[iLName]);

blockRec.setssn\_Number(DTokens[iSsnNum]);

blockRec.setDOB(DTokens[iDob]);

blockRec.setMed\_Diagnosis(DTokens[iMed\_Diagnosis]);

blockRec.setMed\_Treatment(DTokens[iMed\_Treatment]);

blockRec.setMed\_Prex(DTokens[iMed\_Prex]);

// then the blck is merged with the list of blocks which are un verified

brList.add(blockRec);

// converting our block of record values which are un verified to string format, which will help create a SHA256 hash value later

String blockRecStr = blockRec.getIDofBlock() + blockRec.getF\_Name() + blockRec.getL\_Name() +

blockRec.getssn\_Number() + blockRec.getDOB() + blockRec.getMed\_Diagnosis() +

blockRec.getMed\_Treatment() + blockRec.getMed\_Prex() + blockRec.getCreating\_Process();

// block data's hash function is generated by calling the StringBuilder method

String StringDigest\_SHA256 = String\_Builder(blockRecStr);

// defining the hash variable which is signed and verified

String hashSigned = "";

// here the end block which is unverified is signed with the help of stringdigest\_SHA256 and the key which is private

try {

byte[] digitalSign2 = DataSignature(StringDigest\_SHA256.getBytes(), PairKeys.getPrivate());

hashSigned = Base64.getEncoder().encodeToString(digitalSign2);

} catch (Exception excpt)

{

// handling the exception in try-catch block

excpt.printStackTrace();

}

// here we are trying to set the hashcreator to stringdigest\_SHA256

blockRec.setHash\_Creator(StringDigest\_SHA256);

// here we are trying to set the signedcreator into hashsigned

blockRec.setHash\_Signed\_Creator(hashSigned);

// introducing the commonly used method which helps to hold for sometime which is 'sleep\_calls'

Sleep\_calls();

}

} catch (IOException ioException)

{

// handling exception in try-catch block

ioException.printStackTrace();

}

} catch (FileNotFoundException e)

{

// handling exception in try-catch block

e.printStackTrace();

}

}

// here every block string is hashed using the SHA256 algorithm, translated to hexadecimal, then returned to the caller function

private static String String\_Builder(String blockRecStr)

{

StringBuffer String\_hexx;

String StringDigest\_SHA256 = "";

try

{

// here we create the object in the name of 'MessageDisgest' by using hashing technique in SHA256algo

MessageDigest MessageDisgest = MessageDigest.getInstance("SHA-256");

// here the update() function of the previously constructed msgDigest object is called to adjust the digest using the supplied number of bytes

MessageDisgest.update(blockRecStr.getBytes());

// calculating the hash for the revised msgDigest object (output will be in byte[] format)

byte[] byteValue = MessageDisgest.digest();

// here we are converting each and every bytes to different format which is hex

String\_hexx = new StringBuffer();

for (byte bd : byteValue) {

String\_hexx.append(Integer.toString((bd & 0xff) + 0x100, 16).substring(1));

}

StringDigest\_SHA256 = String\_hexx.toString();

} catch (NoSuchAlgorithmException exception)

{

// handling the exception in try-catch block

exception.printStackTrace();

}

// here we return the hash256str back the the funct used for calling

return StringDigest\_SHA256;

}

// When used, the method enables sending blocks to our blockchain ledger. Based on the type of operation: The caller method's passed-in bcLedgerUpdate or reVerifyBlock is used to make the choice.

//If a block is approved and ready to be added to our ledger, the bcLedgerUpdate procedure is utilized. Additionally, this will send each process an update on its new blockchain server port.

//ReVerifyBlock: If a block has to be revised, it is routed to the process's unverified block server port rather than being added to the blockchain ledger.

public static void BlocktoLedger(Block\_Record blockRec, String operation)

{

// defining the variable of the socket

Socket SbSocket;

// defining the variable of the toserver c

PrintStream toServer;

// here we use the new functinalty named switch

switch (operation)

{

case "Block Ledger Update":

try {

// here we are counting the total number of processes in a loop

for (int i = 0; i < Total\_Process; i++)

{

// here we are setting up a port socket on the upgraded blockchain server for transmitting confirmed blocks

SbSocket = new Socket(Name, Ports.portofBaseUpdateBC + i);

// in order to transmit the block to the server port, set up the printStream object

toServer = new PrintStream(SbSocket.getOutputStream());

// here we are utilizing the jsonBuilder function to marshal the record as a JSON object before delivering it to the appropriate process port

toServer.println(Builder\_json(blockRec));

// here this is the output message in the console side to the user

System.out.println("Block is being verified and sent " + blockRec.getIDofBlock());

// here we use flushing technique to flush the prntstream

toServer.flush();

// here we close the connection of the socket

SbSocket.close();

}

} catch (IOException ioException)

{

// handling the exception in try-catch block

ioException.printStackTrace();

}

break;

case "reVerifyBlock":

try

{

// here we are counting the total number of processes in a loop

for (int j = 0; j < Total\_Process; j++)

{

// here we are setting up a port socket for an unverified block server to receive blocks that need to be re-verified

SbSocket = new Socket(Name, Ports.portofBaseUBServer + j);

// to deliver the block to the server port, set up the printStream object

toServer = new PrintStream(SbSocket.getOutputStream());

// here we are utilizing the jsonBuilder function to marshal the record as a JSON object before delivering it to the appropriate process port

toServer.println(Builder\_json(blockRec));

// here this is the output message in the console side to the user

System.out.println("Block is being broadcast: " + blockRec.getIDofBlock());

// here we use flushing technique to flush the prntstream

toServer.flush();

// here we close the connection of the socket

SbSocket.close();

}

} catch (IOException ioException)

{

// handling the exception in try-catch block

ioException.printStackTrace();

}

break;

}

}

// a procedure for utilizing the private key to sign documents. This is a piece of utility code that Professor Elliott gave

public static byte[] DataSignature(byte[] bytesData, PrivateKey aPrivateKey)

throws SignatureException, InvalidKeyException, NoSuchAlgorithmException {

Signature signer = Signature.getInstance("SHA1withRSA");

signer.initSign(aPrivateKey);

signer.update(bytesData);

return (signer.sign());

}

// a technique that enables programs to check whether data has been signed using a public key or not

public static boolean verifySignature(byte[] bytesData, PublicKey publicKey, byte[] decode)

throws NoSuchAlgorithmException, InvalidKeyException, SignatureException {

Signature signer = Signature.getInstance("SHA1withRSA");

signer.initVerify(publicKey);

signer.update(bytesData);

return (signer.verify(decode));

}

// here is a technique to constructing a JSON record. This is a portion of the Prof. Elliott's utility code.

// This code is used by Process 0 to write the full blockchain ledger to disk. Every time a block is added, Process 0 writes the complete ledger.

// Our LinkedList, bcLedger, contains validated blocks that need to be added to the ledger.

public static void write2JSON()

{

// here is the method for changing the object of java tp JSON and need to be written on the respected disk

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* In Write JSON \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

Gson gson = new GsonBuilder().setPrettyPrinting().create();

String json = gson.toJson(Block\_Task.bcLedger);

// here we are establishing the name of the file which is the desired output in the client side as 'BlockchainLedger.json'

try (FileWriter writeData = new FileWriter("BlockchainLedger.json")) {

gson.toJson(Block\_Task.bcLedger, writeData);

} catch (IOException ioException)

{

// handling the exception in the try-catch block

ioException.printStackTrace();

}

}

}

// here we are creating a class in the name of ports which helps in defining and setting of ports to each and every process

class Ports

{

// this is the initiate server port numbr

public static int portbaseInitiateServer = 4600;

// this is the port numbr of public key server

public static int portKeyofBaseServer = 4710;

// this is the port numbr of unverified blockServer

public static int portofBaseUBServer = 4820;

// this is the port number of updated blockchain servver

public static int portofBaseUpdateBC = 4930;

// defining variable of the initiate server port

public static int portStartServer;

// here we define the variable of the keyserverport

public static int portKeyServer;

// here we define the variable of the unverified block server

public static int portUBServer;

// here we are declarring the variable of the blockchain port

public static int portBCServer;

// here the setPorts() function uses the process ID to set each port appropriately. format used here: portBase + processID

public void setPorts(int process\_ID)

{

// here shows the initiate server portnumb

portStartServer = portbaseInitiateServer + process\_ID;

// here all the incoming public keys from the inputs are recevd by portstartserver; this is done for every process\_ID

portKeyServer = portKeyofBaseServer + process\_ID;

// here all the unverified blocks are recvd by portUBServer; this is done for the every process\_ID

portUBServer = portofBaseUBServer + process\_ID;

// here all the updated BC are recvd by portBCServer; this is done for the every process\_ID

portBCServer = portofBaseUpdateBC + process\_ID;

}

}

// this is a piece of utility code supplied by Prof. Elliott and it is a comparator class that will

//make our priority queue organize blocks based on timestamp in it.

class Block\_Comparator implements Comparator<Block\_Record> {

@Override

public int compare(Block\_Record Block\_Record1, Block\_Record Block\_Record2)

{

String date1 = Block\_Record1.getTimeStamp();

String date2 = Block\_Record2.getTimeStamp();

if (date1.equals(date2))

{

return 0;

}

if (date1 == null)

{

return -1;

}

if (date2 == null) {

return 1;

}

return date1.compareTo(date2);

}

}

// start the BWorker class's main server class. By altering the value for the beginProcessFlag in its BWorker class, this server will assist in launching all processes.

// When given the order to do so, it will flip the flag to True and enable other operations to carry on.

class InitiateMainServer implements Runnable

{

public void run()

{

// declaring the length of the que.

int qLen = 6;

// defining the variable of the socket in run() method

Socket socket;

// this is the output message which is printed on the console and can be find in the blockchain.log

System.out.println("Main Server : " + Ports.portStartServer);

try

{

// giving the object which is the servsock with its number of the port and length of the que

ServerSocket serverSocket = new ServerSocket(Ports.portStartServer, qLen);

while (true)

{

// here we accept the request to initiate all process

socket = serverSocket.accept();

new BWorker(socket).start();

}

} catch (IOException ioException)

{

// handling exception in try-catch block

ioException.printStackTrace();

}

}

}

// here the wroker class in the name of 'BWorker' which is used to initiate main server class

class BWorker extends Thread

{

// defining the variable of the socket

Socket socket;

// here we are declaring the const. and giving the socket to define the variable of the socket in local

public BWorker(Socket socket)

{

this.socket = socket;

}

// here we are opening the run() method

public void run()

{

try

{

// transferring the data's frm the inputfile which is frm the socket

BufferedReader inputData = new BufferedReader(new InputStreamReader(socket.getInputStream()));

// Change the beginProcessFlag flag to true after input data has been received.

// assuming that this particular server would not get any more instructions.

String dataRead = inputData.readLine();

Block\_Task.BeginofProcessFlag = true;

// here we close the socket and connection is intrrupted

socket.close();

} catch (IOException ioException)

{

// handling the exception in try-catch block

ioException.printStackTrace();

}

}

}

// activates the associated worker class through the public key server class. Public keys are obtained by PKS from each and every process.

// this public key is composed of two parts: the sender process's process ID and its original public key.

// these are subsequently stored in the appropriate array index.

class PK\_Server implements Runnable {

public void run()

{

// declaring the length of the que.

int qLen = 6;

// defining the variable of the socket in run() method

Socket socket;

// this is the output message which is printed on the console and can be find in the blockchain.log

System.out.println("Initiate PK server at ports: " + Ports.portKeyServer);

try

{

// giving the object which is the servsock with its number of the port and length of the que

ServerSocket serverSocket = new ServerSocket(Ports.portKeyServer, qLen);

while (true)

{

// here we accept the request

socket = serverSocket.accept();

// here the creation of a new worker thread and the execution of the public key server worker class ()

new PKBWorker(socket).start();

}

} catch (IOException ioException)

{

// handling the exception in try-catch block

ioException.printStackTrace();

}

}

}

// here is the worker class for the class named publickeyServer

class PKBWorker extends Thread

{

// here we are defining the variable of socket named as keysocket

Socket keySocket;

// here we define the cons. and giving the localvariable to the socket

public PKBWorker(Socket socket) {

this.keySocket = socket;

}

public void run()

{

try {

// here we are giving in all the data's from the inputstream

BufferedReader inputData = new BufferedReader(new InputStreamReader(keySocket.getInputStream()));

// here we are going to split the data of input which are recvd in the array of string

String[] dataRead = inputData.readLine().split(" ");

// as process\_ID is a integer as we are changing the str to integer form; this is done as it will be helpful for futurre index process

int process\_ID = Integer.parseInt(dataRead[0]);

// here we are doing the decode technique to the pub\_key to save it in the position of index 1 in the byte format we are changing the

// str to byte[] form

byte[] publicKeyB = Base64.getDecoder().decode(dataRead[1]);

X509EncodedKeySpec pubSpec = new X509EncodedKeySpec(publicKeyB);

KeyFactory publicKeyFact = KeyFactory.getInstance("RSA");

PublicKey RestoredKey = publicKeyFact.generatePublic(pubSpec);

// here the publickeylist is created as the public key for every process is joined

Block\_Task.publicKeyList[process\_ID] = RestoredKey;

// here the counter process is incremented from the basic

Block\_Task.Count\_pk++;

// here is the condition that if we are getting all the public keys from every process we move and set the flag value to not false

if (Block\_Task.Count\_pk == 3)

{

Block\_Task.Flag\_pk = true;

}

// this is the output message in the console side which can be found in the blockchainlog file

System.out.println("For Process ID Public Key is Obtained" + process\_ID);

// here we close the complete connection of the socket

keySocket.close();

} catch (Exception exception)

{

// here we are handling exception in the try-catch block

exception.printStackTrace();

}

}

}

// Unverified Block Server class that calls the appropriate worker class receives blocks that are read in by each process through an input file,

// or blocks that need to be re-verified because the blockchain ledger has changed.

class UnverifiedBlock implements Runnable

{

// here we define thevariable of the blockq locally of the type blockrecord

BlockingQueue<Block\_Record> blockQueue;

// here the definition of the constructor is taking place

public UnverifiedBlock(BlockingQueue<Block\_Record> blockQueue) {

this.blockQueue = blockQueue;

}

// here we are initiating the server of the unverified block for the already recvd or recving block

public void run()

{

// here we are declaring that the lenq which is the length of the coming request that is to be and entry or stored inside the queue

int qLen = 6;

// here we are defining the variable of the socket

Socket UVBSocket;

// here is the output in the console which is found on the blockchain.log

System.out.println("Unverified Block server input thread " + Ports.portUBServer);

try {

// here we create the object of the serversocket

ServerSocket UnverifiedBlock = new ServerSocket(Ports.portUBServer, qLen);

while (true)

{

// here we confirm the req and starts to receive the unverified block

UVBSocket = UnverifiedBlock.accept();

new UnverifiedBlockBWorker(UVBSocket).start();

}

} catch (IOException ioException)

{

// here we are handling the exception in try-catch block

ioException.printStackTrace();

}

}

}

// receives confirmed blocks rather than the complete blockchain ledger. Worker class for updated blockchain server class.

// then, THIS process ledger is updated with these confirmed blocks.

class UnverifiedBlockBWorker extends Thread

{

// defining the variable of the local socket

Socket uvbSocket;

// here the definition of the constructor is taking place

public UnverifiedBlockBWorker(Socket uvbSocket) {

this.uvbSocket = uvbSocket;

}

public void run()

{

try {

// here we are getting every data in from the socketinput

BufferedReader inputData = new BufferedReader(new InputStreamReader(uvbSocket.getInputStream()));

// defining the variable of the string

String inputString;

// here we are making the object if gson

Gson gson = new Gson();

// here we create the object of the buffer

StringBuffer strBuffer = new StringBuffer();

// here we are saving all the input data in json form

while ((inputString = inputData.readLine()) != null)

{

strBuffer.append(inputString);

}

// here we parse the data of json

Block\_Record brInput = gson.fromJson(strBuffer.toString(), Block\_Record.class);

// here we are checking the

System.out.println("Block Queue gets " + brInput.getIDofBlock() + "\n");

Block\_Task.blockQueue.put(brInput);

uvbSocket.close();

} catch (Exception exception)

{

exception.printStackTrace();

}

}

}

//

class UpdatedBCSever implements Runnable

{

public void run()

{

// declaring the length of the que.

int qLen = 6;

// defining the variable of the socket in run() method

Socket Block\_Socket;

// this is the output message which is printed on the console and can be find in the blockchain.log

System.out.println("Block Chian server input thread " + Ports.portBCServer);

try

{

// giving the object which is the servsock with its number of the port and length of the que.

ServerSocket servsock = new ServerSocket(Ports.portBCServer, qLen);

while (true)

{

// here we accept the request

Block\_Socket = servsock.accept();

new UBlockWorker(Block\_Socket).start();

}

} catch (IOException ioe)

{

ioe.printStackTrace();

}

}

}

//receives confirmed blocks rather than the complete blockchain ledger. Worker class for updated blockchain server class.

//then, THIS process ledger is updated with these confirmed blocks.

class UBlockWorker extends Thread

{

// defining the variable of the local socket

Socket Block\_Socket;

// here the definition of the constructor is taking place

public UBlockWorker(Socket Block\_Socket) {

this.Block\_Socket = Block\_Socket;

}

public void run()

{

try {

// here we are getting every data in from the socketinput

BufferedReader inputData = new BufferedReader(new InputStreamReader(Block\_Socket.getInputStream()));

// here we are making the object if gson

Gson gson = new Gson();

// defining the variable of the string

String brData;

// here we create the object of the buffer

StringBuffer brDataBuff = new StringBuffer();

// here we are going to store all the input data to str buffer in the form as json

while ((brData = inputData.readLine()) != null)

{

brDataBuff.append(brData);

}

// here we parse the data of json

Block\_Record Block\_RecordIn = gson.fromJson(brDataBuff.toString(), Block\_Record.class);

// verify the block in ledger if it is not found then it is adding the

// blocks to procss the ledger copies

if (!Block\_Task.Duplicate(Block\_RecordIn)) {

Block\_Task.bcLedger.add(0, Block\_RecordIn);

System.out.println("Verified Block added to BlockChain Ledger");

System.out.println("Verified Block Count in the Ledger is: " + Block\_Task.bcLedger.size());

}

// whne the id of all the process equals zero then ledger is written on the disk

if (Block\_Task.process\_ID == 0) {

Block\_Task.write2JSON();

}

// here we close the complete socket

Block\_Socket.close();

} catch (IOException ioException)

{

// here we are handling the exception

ioException.printStackTrace();

}

}

}

// this is the BlockChain work class. We are supplying the priority queue of THIS process, which contains the UVB, as input to its constructor.

// The block is taken out of the queue and put into a temporary variable so that it may be processed later.

class PuzzleWork implements Runnable

{

// declaring the variable of the que.

BlockingQueue<Block\_Record> blockQ;

// this is the random string which is an alphanumb; This is the exact guess

private static final String alphaNumericStr = "ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";

// here the definition of the constructor is taking place

public PuzzleWork(PriorityBlockingQueue<Block\_Record> blockQueue) {

this.blockQ = blockQueue;

}

public void run()

{

try {

while (true)

{

// here we pull the blocks in one by one at a time

Block\_Record blockRec = Block\_Task.blockQueue.take();

// this is the str form of our rec. this can be used on the msgDigest

String blockRecStr = blockRec.getIDofBlock() + blockRec.getF\_Name() +

blockRec.getL\_Name() + blockRec.getssn\_Number() +

blockRec.getDOB() + blockRec.getMed\_Diagnosis() +

blockRec.getMed\_Treatment() + blockRec.getMed\_Prex() +

blockRec.getCreating\_Process();

// here we save all the seeds which are random

String randomStr;

// here we are arranging all the random seed together with our present live data

String concatenateStr;

// str which are created with the help of sha256

String hashStr;

boolean isHashVerified;

boolean isBlockIDVerified;

// here we verify all the blcks which are found again on the ledger which is created by executing the process

if (Block\_Task.Duplicate(blockRec) && blockRec != null)

// here duplicated blocks can be verified

{

continue;

}

// here verification of the blockID which are signed with the help of creator process PK

isBlockIDVerified = Block\_Task.verifySignature(blockRec.getIDofBlock().getBytes(),

Block\_Task.publicKeyList[Integer.parseInt(blockRec.getCreating\_Process())],

Base64.getDecoder().decode(blockRec.getRegisteredIDofBlock()));

String messageBlock = isBlockIDVerified ? "Block ID Signed" : "Block ID not Signed";

System.out.println(messageBlock);

// here verification of the process from the creator which are signed by SHA 256

isHashVerified = Block\_Task.verifySignature(blockRec.getHash\_Creator().getBytes(),

Block\_Task.publicKeyList[Integer.parseInt(blockRec.getCreating\_Process())],

Base64.getDecoder().decode(blockRec.getHash\_Signed\_Creator()));

String messageHash = isHashVerified ? "Hash Signed" : "Hash not Signed";

System.out.println(messageHash);

// here we are finding the id of the former block which are found on the ledger which is created on the execution of our program

String previousBlockID = Block\_Task.bcLedger.get(0).getIDofBlock();

// here the work number which is mentioned should be with in 0 to 65535 which can be written in hexa decimal

// which helps in the finding the puzzelwork is solved or not solved

int workNum;

// here it is a condition that is the puzzelwork is solved the blocks are added to the respective variable found

String updatedBlock = blockRecStr;

// here we add the first part whic is taken from the data mentioned or given

updatedBlock = updatedBlock + Block\_Task.bcLedger.get(0).getValue\_of\_Winning\_Hash();

// here we allow the problem or puzzel to solve when the record found is not present in the newly created ledger

if (!Block\_Task.Duplicate(blockRec))

{

try {

// here we limit the program that how many time it should hold

for (int i = 1; i < 20; i++)

{

// again gets a new random/guess variale in apha numeric format

randomStr = AlphaNum(8);

// here we add the data's third part which is a randam seed to our updated block

concatenateStr = updatedBlock + randomStr;

// here we get the hash value of our data

MessageDigest MessageDisgest = MessageDigest.getInstance("SHA-256");

byte[] bytesHash = MessageDisgest.digest(concatenateStr.getBytes(StandardCharsets.UTF\_8));

// here we are changing the value to str

hashStr = ArraytoString(bytesHash);

System.out.println("In the BlockData found the Hash Value is " + hashStr);

// range is b\w 0 to 65535 in hexa decimal.

// get the value of numb in 16 bits

workNum = Integer.parseInt(hashStr.substring(0, 4), 16);

System.out.println("The list of First 16 bits in the hex and decimal format is " + hashStr.substring(0, 4) + " and " + workNum);

if (!(workNum < 20000))

{

System.out.format("%d is not the mentioned one. So,solving once again\n\n", workNum);

}

if (workNum < 20000)

{

if (!previousBlockID.equals(Block\_Task.bcLedger.get(0).getIDofBlock())) {

System.out.println("Reading..");

Block\_Task.BlocktoLedger(blockRec, "RE-VerifyBlock");

}

else {

blockRec.setValue\_of\_Winning\_Hash(hashStr);

blockRec.setValueofRandomSeed(randomStr);

System.out.format("%d is less than the mentioned which is less than 20,000.Hence the Puzzle is solved!!\n", workNum);

System.out.println(" Final random seed String which is the Winning one is : " + randomStr);

blockRec.setValue\_of\_Previous\_Hash(Block\_Task.bcLedger.get(0).getValue\_of\_Winning\_Hash());

int B\_Number = Integer.parseInt(Block\_Task.bcLedger.get(0).getB\_Number());

B\_Number++;

blockRec.setB\_Number(String.valueOf(B\_Number));

// here we setlleup the processID for the verified data

blockRec.setVerify\_process\_ID(String.valueOf(Block\_Task.process\_ID));

// here we are using the verifier digital sign to sign the final winning hash

String signHashVerifier;

byte[] digitalSign = Block\_Task.DataSignature(hashStr.getBytes(),

Block\_Task.PairKeys.getPrivate());

signHashVerifier = Base64.getEncoder().encodeToString(digitalSign);

blockRec.setValue\_of\_WinningSigned\_Hash(signHashVerifier);

// atlast we add all the signed and verified block

Block\_Task.bcLedger.add(0, blockRec);

// this is the output message in the console which can be found on the blockchainlog file

System.out.println("All the Blocks are adding to the newly created Blochain Ledger.");

System.out.println("Verified blocks are : " + Block\_Task.bcLedger.size());

// here multicasting is done

Block\_Task.BlocktoLedger(blockRec, "Block Ledger Update");

continue;

}

break;

}

if (Block\_Task.Duplicate(blockRec))

{

// here it is checking often that the live records are being verified or not verified

break;

}

// here we are again introducing the sleep calls method

Block\_Task.Sleep\_calls();

}

} catch (Exception excpt)

{

// here we are handling the exception in try-catch block

excpt.printStackTrace();

}

}

}

} catch (Exception excpt) {

excpt.printStackTrace();

}

}

// here is the method to generate the random seed which is given at the initial start process in the alphanum. format.

// This utility code can be found in Prof.Elliott webpage

public static String AlphaNum(int count) {

StringBuilder stringBuilder = new StringBuilder();

while (count-- != 0) {

int character = (int) (Math.random() \* alphaNumericStr.length());

stringBuilder.append(alphaNumericStr.charAt(character));

}

return stringBuilder.toString();

}

// here is the method which is taking the byte argu. and converting it into the str form.This utility code can be found in Prof.Elliott webpage

public static String ArraytoString(byte[] ba2s) {

StringBuilder String\_hexx = new StringBuilder(ba2s.length \* 2);

for (byte byteA2S : ba2s) {

String\_hexx.append(String.format("%02X", byteA2S));

}

return String\_hexx.toString();

}

}