Branch :- Computer Sci. & Engg.
Subject :-Block Chain Fundamentals Lab manual **Class :- Final Year**
Teacher Manual **Sem :- VII**

PRACTICAL NO 5

AIM: Create a simple BlockChain

S/W REQUIRED: Phython

Block Chain

A block chain is a distributed database or ledger that is shared among the nodes of a computer network. As a database, a block chain stores information electronically in digital format. Block chains are best known for their crucial role in cryptocurrency systems, such as Bitcoin, for maintaining a secure and decentralized record of transactions. The innovation with a block chain is that it guarantees the fidelity and security of a record of data and generates trust without the need for a trusted third party.

Implementation:

```
from hashlib import sha256
import time

class block:
        def __init__ (self, timestamp, data, previousHash = ' '):
                self.timestamp = timestamp
                self.data = data
                self.previousHash = previousHash
                self.hash = self.calculateHash()

        def calculateHash(self):
                return sha256((str(self.timestamp) + str(self.data) +
str(self.previousHash)).encode()).hexdigest()


class blockchain:
        def __init__(self):
                self.chain = [self.createGenesis()]

        def createGenesis(self):
                return block(time.ctime(), "genesisBlock", "00000")

        def mineBlock(self, data):
                node = block(time.ctime(), data, self.chain[-1].hash)
                # mining a new block to the blockchain
                self.chain.append(node)

        def printBlockchain(self):
                for i in range(len(self.chain)):
                        print("\n-----Block ", i ,"---------\n timestamp = "\
                        , self.chain[i].timestamp,"\n data = ", \
```

```
                self.chain[i].data, "\n previousHash = ",\
                self.chain[i].previousHash,"\n hash = ", \
                  self.chain[i].hash)



CEVcoin = blockchain()

data = input()

# sending data to get mined
print("\n\n ----> Mining New Block -->")
CEVcoin.mineBlock(data)

print("\n\n ----> New Block mined successfully --> ")

CEVcoin.printBlockchain()
```

Output:

```
----> New Block mined successfully -->

----Block  0 ---------
timestamp =  Thu Aug 18 13:42:34 2022
data =  genesisBlock
previousHash =  00000
hash =  d35695bf15937b88e66f362bff9ba94ea329cce7a1b2b690127e38fd18b766d9

----Block  1 ---------
timestamp =  Thu Aug 18 13:44:22 2022
data =
previousHash =  d35695bf15937b88e66f362bff9ba94ea329cce7a1b2b690127e38fd18b766d9
hash =  546e3f89d05247109797e377309fedf1b5e89c2917a769e07eff8fe544a949cd
S C:\Users\PC-1059\hello>
S C:\Users\PC-1059\hello>
```

CONCLUSION: Thus we have created a block chain .

Sipna College of Engineering & Technology, Amravati.
Department of Computer Science & Engineering
Session 2022-2023

Branch :- Computer Sci. & Engg. **Class :- Final Year**
Subject :-Block Chain Fundamentals Lab manual **Sem :- VII**
Teacher Manual

PRACTICAL NO 6

AIM: Create Your Own Cryptocurrency Using Python

S/W REQUIRED: Phython

Cryptocurrency

In computer science, a cryptocurrency, crypto-currency, or crypto is a digital currency that does not rely on any central authority to uphold or maintain it. Instead, transaction and ownership data is stored in a digital ledger using distributed ledger technology, typically a blockchain.

A Cryptocurrency is a system that meets six conditions:

1. The system does not require a central authority; its state is maintained through distributed consensus.
2. The system keeps an overview of cryptocurrency units and their ownership.
3. The system defines whether new cryptocurrency units can be created. If new cryptocurrency units can be created, the system defines the circumstances of their origin and how to determine the ownership of these new units.
4. Ownership of cryptocurrency units can be proved exclusively cryptographically.
5. The system allows transactions to be performed in which ownership of the cryptographic units is changed. A transaction statement can only be issued by an entity proving the current ownership of these units.
6. If two different instructions for changing the ownership of the same cryptographic units are simultaneously entered, the system performs at most one of them.

Implementation:

```
import hashlib
import time


class Block:

  def __init__(self, index, proof_no, prev_hash, data, timestamp=None):
    self.index = index
    self.proof_no = proof_no
    self.prev_hash = prev_hash
    self.data = data
    self.timestamp = timestamp or time.time()

  @property
  def calculate_hash(self):
    block_of_string = "{}{}{}{}{}{}".format(self.index, self.proof_no,
                                            self.prev_hash, self.data,
                                            self.timestamp)
```

```
        return hashlib.sha256(block_of_string.encode()).hexdigest()

    def __repr__(self):
        return "{} - {} - {} - {} - {}".format(self.index, self.proof_no,
                                               self.prev_hash, self.data,
                                               self.timestamp)


class BlockChain:

    def __init__(self):
        self.chain = []
        self.current_data = []
        self.nodes = set()
        self.construct_genesis()

    def construct_genesis(self):
        self.construct_block(proof_no=0, prev_hash=0)

    def construct_block(self, proof_no, prev_hash):
        block = Block(
            index=len(self.chain),
            proof_no=proof_no,
            prev_hash=prev_hash,
            data=self.current_data)
        self.current_data = []

        self.chain.append(block)
        return block

    @staticmethod
    def check_validity(block, prev_block):
        if prev_block.index + 1 != block.index:
            return False

        elif prev_block.calculate_hash != block.prev_hash:
            return False

        elif not BlockChain.verifying_proof(block.proof_no,
                                            prev_block.proof_no):
            return False

        elif block.timestamp <= prev_block.timestamp:
            return False

        return True

    def new_data(self, sender, recipient, quantity):
        self.current_data.append({
            'sender': sender,
            'recipient': recipient,
            'quantity': quantity
```

```
    })
    return True

  @staticmethod
  def proof_of_work(last_proof):
    '''this simple algorithm identifies a number f' such that hash(ff') contain 4 leading zeroes
    f is the previous f'
    f' is the new proof
    '''
    proof_no = 0
    while BlockChain.verifying_proof(proof_no, last_proof) is False:
      proof_no += 1

    return proof_no

  @staticmethod
  def verifying_proof(last_proof, proof):
    #verifying the proof: does hash(last_proof, proof) contain 4 leading zeroes?

    guess = f'{last_proof}{proof}'.encode()
    guess_hash = hashlib.sha256(guess).hexdigest()
    return guess_hash[:4] == "0000"

  @property
  def latest_block(self):
    return self.chain[-1]

  def block_mining(self, details_miner):

    self.new_data(
      sender="Sipna COET",  #it implies that this node has created a new block
      receiver=details_miner,
      quantity=
      1,  #creating a new block (or identifying the proof number) is awarded with 1
    )

    last_block = self.latest_block

    last_proof_no = last_block.proof_no
    proof_no = self.proof_of_work(last_proof_no)

    last_hash = last_block.calculate_hash
    block = self.construct_block(proof_no, last_hash)

    return vars(block)

  def create_node(self, address):
    self.nodes.add(address)
    return True

  @staticmethod
  def obtain_block_object(block_data):
    #obtains block object from the block data
```

```
        return Block(
            block_data['index'],
            block_data['proof_no'],
            block_data['prev_hash'],
            block_data['data'],
            timestamp=block_data['timestamp'])

blockchain = BlockChain()

print("***Mining fccCoin about to start***")
print(blockchain.chain)

last_block = blockchain.latest_block
last_proof_no = last_block.proof_no
proof_no = blockchain.proof_of_work(last_proof_no)

blockchain.new_data(
    sender="Sipna COET",  #it implies that this node has created a new block
    recipient="SIPNA CSE Department",  #let's send Sipna CSE some coins!
    quantity=
    1,  #creating a new block (or identifying the proof number) is awarded with 1
)

last_hash = last_block.calculate_hash
block = blockchain.construct_block(proof_no, last_hash)

print("***Mining fccCoin has been successful***")
print(blockchain.chain)
```

Output:

```
PS C:\Users\PC-1059\hello>  & 'C:\Python310\python.exe' 'c:\Users\PC-1059\.vscode\extensions\ms-
python.python-2022.14.0\pythonFiles\lib\python\debugpy\adapter/../..\debugpy\launcher' '51936' '--
'c:\Users\PC-1059\hello\cryptocurrency.py'
***Mining fccCoin about to start***
[0 - 0 - 0 - [] - 1662359736.2395442]
***Mining fccCoin has been successful***
[0 - 0 - 0 - [] - 1662359736.2395442, 1 - 88914 -
cf01d26b936e6e87a464b53979bbd9ce51e6e5fd50e6ba3b96cd6d4ae780dd80 - [{'sender': 'Sipna COET',
'recipient': 'SIPNA CSE Department', 'quantity': 1}] - 1662359736.6053078]
PS C:\Users\PC-1059\hello>
```

CONCLUSION: Thus we have studied and created our own Cryptocurrency Using Python.

Branch :- Computer Sci. & Engg.
Subject :-Block Chain Fundamentals Lab manual **Class :- Final Year**
Teacher Manual **Sem :- VII**

PRACTICAL NO 7

AIM: Design a simple Smart contract

***S/W* REQUIRED:** Remix IDE

A "smart contract" is simply a program that runs on the Ethereum blockchain. It's a collection of code (its functions) and data (its state) that resides at a specific address on the Ethereum blockchain.

Smart contracts are a type of Ethereum account. This means they have a balance and can be the target of transactions. However they're not controlled by a user, instead they are deployed to the network and run as programmed. User accounts can then interact with a smart contract by submitting transactions that execute a function defined on the smart contract. Smart contracts can define rules, like a regular contract, and automatically enforce them via the code. Smart contracts cannot be deleted by default, and interactions with them are irreversible.

Benefits of smart contracts

1. Speed, efficiency and accuracy: Once a condition is met, the contract is executed immediately. Because smart contracts are digital and automated, there's no paperwork to process and no time spent reconciling errors that often result from manually filling in documents.
2. Trust and transparency: Because there's no third party involved, and because encrypted records of transactions are shared across participants, there's no need to question whether information has been altered for personal benefit.
3. Security: Blockchain transaction records are encrypted, which makes them very hard to hack. Moreover, because each record is connected to the previous and subsequent records on a distributed ledger, hackers would have to alter the entire chain to change a single record.
4. Savings:Smart contracts remove the need for intermediaries to handle transactions and, by extension, their associated time delays and fees.

Ethereum has developer-friendly languages for writing smart contracts:

- Solidity
- Vyper

We will be using solidity

Solidity:

Solidity is one of the most popular languages used for building smart contracts on Ethereum Blockchain. It's also an object-oriented programming language. Solidity's code is encapsulated in contracts which means a contract in Solidity is a collection of code (its functions) and data (its state) that resides at a specific address on the Ethereum blockchain. A contract is a fundamental block of building an application on Ethereum.

Implementation:

```
// SPDX-License-Identifier: MIT
// compiler version must be greater than or equal to 0.8.13 and less than 0.9.0
pragma solidity ^0.8.13;

contract HelloWorld {
   string public greet = "Hello World!";
}
```

Output: Byte code generated as well as API generated

API code

```
[
        {
                "inputs": [],
                "name": "greet",
                "outputs": [
                        {
                                "internalType": "string",
                                "name": "",
                                "type": "string"
                        }
                ],
                "stateMutability": "view",
                "type": "function"
        }
]
```

CONCLUSION: Thus we have created a simple Smart contract

Sipna College of Engineering & Technology, Amravati.
Department of Computer Science & Engineering
Session 2022-2023

Branch :- Computer Sci. & Engg.
Subject :-Block Chain Fundamentals Lab manual **Class :- Final Year**
Teacher Manual **Sem :- VII**

PRACTICAL NO 8

AIM: Create a smart contract for Merkle tree

S/W REQUIRED: Remix IDE

Merkle tree is a fundamental part of blockchain technology. It is a mathematical data structure composed of hashes of different blocks of data, and which serves as a summary of all the transactions in a block. It also allows for efficient and secure verification of content in a large body of data. It also helps to verify the consistency and content of the data. Both Bitcoin and Ethereum use Merkle Trees structure. Merkle Tree is also known as Hash Tree.

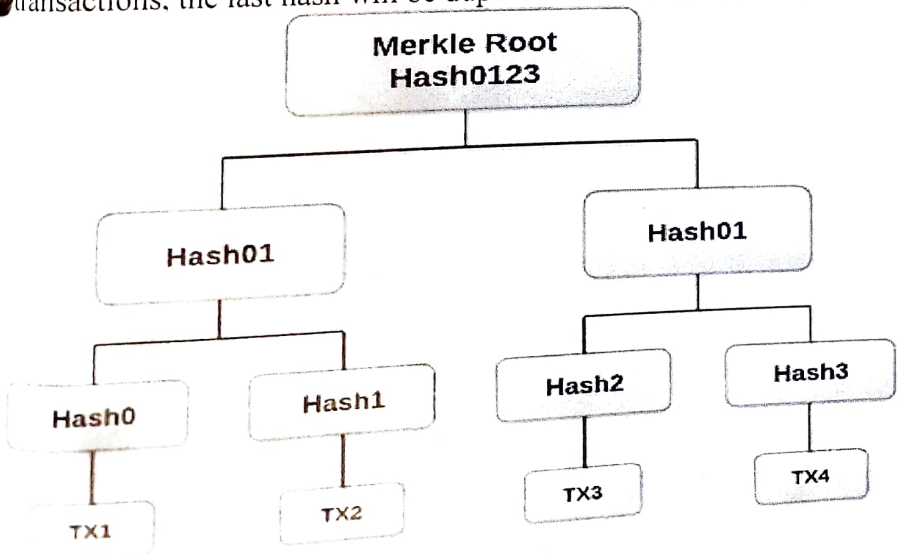
The concept of Merkle Tree is named after Ralph Merkle, who patented the idea in 1979. Fundamentally, it is a data structure tree in which every leaf node labelled with the hash of a data block, and the non-leaf node labelled with the cryptographic hash of the labels of its child nodes. The leaf nodes are the lowest node in the tree.

How does merkle tree works?

A Merkle tree stores all the transactions in a block by producing a digital fingerprint of the entire set of transactions. It allows the user to verify whether a transaction can be included in a block or not.

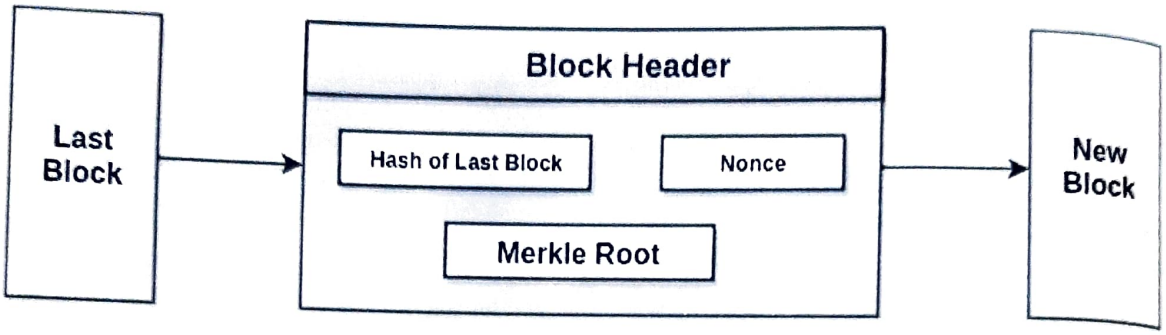
Merkle trees are created by repeatedly calculating hashing pairs of nodes until there is only one hash left. This hash is called the Merkle Root, or the Root Hash. The Merkle Trees are constructed in a bottom-up approach.

Every leaf node is a hash of transactional data, and the non-leaf node is a hash of its previous hashes. Merkle trees are in a binary tree, so it requires an even number of leaf nodes. If there is an odd number of transactions, the last hash will be duplicated once to create an even number of leaf nodes.



The above example is the most common and simple form of a Merkle tree, i.e., Binary Merkle Tree. There are four transactions in a block: TX1, TX2, TX3, and TX4. Here you can see, there is a top hash which is the hash of the entire tree, known as the Root Hash, or the Merkle Root. Each of these is repeatedly hashed, and stored in each leaf node, resulting in Hash 0, 1, 2, and 3. Consecutive pairs of leaf nodes are then summarized in a parent node by hashing Hash0 and Hash1, resulting in Hash01, and separately hashing Hash2 and Hash3, resulting in Hash23. The two hashes (Hash01 and Hash23) are then hashed again to produce the Root Hash or the Merkle Root.

Merkle Root is stored in the block header. The block header is the part of the bitcoin block which gets hash in the process of mining. It contains the hash of the last block, a Nonce, and the Root Hash of all the transactions in the current block in a Merkle Tree. So having the Merkle root in block header makes the transaction tamper-proof. As this Root Hash includes the hashes of all the transactions within the block, these transactions may result in saving the disk space.



The Merkle Tree maintains the integrity of the data. If any single detail of transactions or order of the transaction's changes, then these changes reflected in the hash of that transaction. This change would cascade up the Merkle Tree to the Merkle Root, changing the value of the Merkle root and thus invalidating the block. So everyone can see that Merkle tree allows for a quick and simple test of whether a specific transaction is included in the set or not.

Merkle trees have three benefits:

- It provides a means to maintain the integrity and validity of data.
- It helps in saving the memory or disk space as the proofs, computationally easy and fast.
- Their proofs and management require tiny amounts of information to be transmitted across networks.

Implementation:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.13;

contract MerkleProof {
    function verify(
        bytes32[] memory proof,
        bytes32 root,
        bytes32 leaf,
        uint index
    ) public pure returns (bool) {
        bytes32 hash = leaf;

        for (uint i = 0; i < proof.length; i++) {
            bytes32 proofElement = proof[i];
```

```
        if (index % 2 == 0) {
            hash = keccak256(abi.encodePacked(hash, proofElement));
        } else {
            hash = keccak256(abi.encodePacked(proofElement, hash));
        }

        index = index / 2;
    }

    return hash == root;
  }
}

contract TestMerkleProof is MerkleProof {
  bytes32[] public hashes;

  constructor() {
    string[4] memory transactions = [
      "alice -> bob",
      "bob -> dave",
      "carol -> alice",
      "dave -> bob"
    ];

    for (uint i = 0; i < transactions.length; i++) {
      hashes.push(keccak256(abi.encodePacked(transactions[i])));
    }

    uint n = transactions.length;
    uint offset = 0;

    while (n > 0) {
      for (uint i = 0; i < n - 1; i += 2) {
        hashes.push(
          keccak256(
            abi.encodePacked(hashes[offset + i], hashes[offset + i + 1])
          )
        );
      }
      offset += n;
      n = n / 2;
    }
  }

  function getRoot() public view returns (bytes32) {
    return hashes[hashes.length - 1];
  }
}

/* verify
3rd leaf
0xdca3326ad7e8121bf9cf9c12333e6b2271abe823ec9edfe42f813b1e768fa57b

root
```

```
    0xcc086fcc038189b4641db2cc4f1de3bb132aefbd65d510d817591550937818c7

    index
    2

    proof
    0x8da9e1c820f9dbd1589fd6585872bc1063588625729e7ab0797cfc63a00bd950
    0x995788ffc103b987ad50f5e5707fd094419eb12d9552cc423bd0cd86a3861433
    */
}
```

Output: Byte code generated as well as API generated

API code

```
[
        {
                "inputs": [
                        {
                                "internalType": "bytes32[]",
                                "name": "proof",
                                "type": "bytes32[]"
                        },
                        {
                                "internalType": "bytes32",
                                "name": "root",
                                "type": "bytes32"
                        },
                        {
                                "internalType": "bytes32",
                                "name": "leaf",
                                "type": "bytes32"
                        },
                        {
                                "internalType": "uint256",
                                "name": "index",
                                "type": "uint256"
                        }
                ],
                "name": "verify",
                "outputs": [
                        {
                                "internalType": "bool",
                                "name": "",
                                "type": "bool"
                        }
                ],
                "stateMutability": "pure",
                "type": "function"
        }
]
```