ZETECH UNIVERSITY

UNIT NAME: BLOCK CHAIN TECHNOLOGY

UNIT CODE: BCE 413

LESSON 4 Block Chain Scripting & merkel tree

- Build simple Block chain system written in Python
- Use our Block chain with pre-established transactions represented as strings
- Test the immutability of our Block chain

How to Create a Block chain with Python?

We can divide Block chain into two simple terms:

- Block: A space where we store transactions
- Chain: A set of linked records

Block chain introduces an interesting set of features:

- ✓ History immutability
- ✓ Information persistency
- ✓ No errors with stored data

A lot of systems currently rely on Block chain, such as cryptocurrencies, asset transfer (NFTs), and possibly in the near future, voting.

- We're not going to use **JSON** but Python lists. This will let us simplify the process and focus on applying the key concepts of a Blockchain.

What you'll need to follow is:

- Understanding of classes and methods in Python
- Basic usage of command line

Creating the Block class

- Open your favorite code editor and create a main.py file. This will be the file we'll work with.
- Now, import hashlib, a module that lets us create one-way encrypted messages. Cryptography techniques like *hashing make Blockchain create secure transactions.*

```
# main.py file
"""
A simple Blockchain in Python
"""

import hashlib
```

This module includes most of the hashing algorithms you'll need. Just keep in mind we'll be using the **hashlib. sha256()** function.

```
class GeekCoinBlock:

    def __init__(self, previous_block_hash, transaction_list):

        self.previous_block_hash = previous_block_hash
        self.transaction_list = transaction_list

        self.block_data = f"{' - '.join(transaction_list)} - {previous_block_hash}"
        self.block_hash = hashlib.sha256(self.block_data.encode()).hexdigest()
```

GeekCoinBlock Explanation

First, we create a class named GeekCoinBlock, a wrapper for objects that will have certain characteristics (attributes) and behaviors (methods).

- Then we define the __init__ method (also named constructor), which gets invoked each time a GeekCoinBlock object gets created.

This method has three parameters:

1. **self** (the instance of each object)
2. **previous_block_hash** (a reference to the previous block)
3. **transaction_list** (a list of transactions made in the current block).

We store the previous hash and transaction list and create an instance variable block_data as a string. This doesn't happen with real cryptocurrencies, in which we

store that kind of data as another hash, but for simplicity purposes, we'll store every block of data as a string.

Finally, we create the block_hash, which other blocks will use to continue the chain. Here's where hashlib comes in handy; instead of creating a custom hash function, we can use the pre-built sha256 to make immutable blocks.

This function receives encoded strings (or bytes) as parameters. That's why we're using the block_data.encode() method. After that, we call hexdigest() to return the encoded data into hexadecimal format.

so let's play with hashlib on a Python shell.

```
In [1]: import hashlib

In [2]: message = "Python is great"

In [3]: h1 = hashlib.sha256(message.encode())

In [4]: h1
Out[4]: <sha256 ... object @ 0x7efcd55bfbf0>

In [5]: h1.hexdigest()
Out[5]: 'a40cf9cca ... 42ab97'

In [6]: h2 = hashlib.sha256(b"Python is not great")

In [7]: h2
Out[7]: <sha256 ... object @ 0x7efcd55bfc90>

In [8]: h2.hexdigest()
Out[8]: 'fefe510a6a ... 97e010c0ea34'
```

- As you can see, a slight change in the input like "Python is great" to "Python is not great" can produce a totally different hash. This has all to do with Blockchain integrity.
- If you introduce some little change into a blockchain, its hash will dramatically change. This is the reason why the saying "You can't corrupt a Blockchain" is true.

Using our Block Class

In the same file, create a couple of transactions made up of simple strings stored in variables, for example:

```
class GeekCoinBlock:
    ...

t1 = "Noah sends 5 GC to Mark"
t2 = "Mark sends 2.3 GC to James"
t3 = "James sends 4.2 GC to Alisson"
t4 = "Alisson sends 1.1 GC to Noah"
```

Of course, GC refers to GeekCoin

Now, build the first block of our Blockchain by using the GeekCoinBlock class and print its attributes. Take into account that the **previous_hash** parameter of the genesis block (first block that precedes other blocks) will always be some arbitrary string or hash, in this case, "firstblock."

```
block1 = GeekCoinBlock('firstblock', [t1, t2])

print(f"Block 1 data: {block1.block_data}")
print(f"Block 1 hash: {block1.block_hash}")
```

Building a blockchain in Python

- We divide the process of building a block chain into several steps for better understanding. These steps are as follows:

Step 1: Creating a Blockchain class

Step 2: Writing a Function to build New Blocks

Step 3: Writing Functions to create New Transactions and get the Last Block

Step 4: Writing a Function to "Hash" the Blocks

Step 5: Creating a New Blockchain and Sending some money

1. Creating a Blockchain class

We will start by importing the required libraries. In this case, we will be needing the hashlib library for the encryption, the JSON library for our blocks formatting, and the time library for the timestamp of each block. We will then be creating a class and initializing the following variables:

1. chain: *This will be an empty list to which we will add blocks. Quite literally, the 'blockchain'.*
2. pendingTransactions: *When users send the coins to each other, their transactions will locate in this array until we approve and insert them into a new block.*
3. newBlock: *This is a method that we will define soon, and we will utilize it in order to include each block in the chain.*

2. Writing a Function to construct New Blocks

Now that we have initialized an empty chain, let us begin inserting blocks into it. We will then define the JSON object with the following properties:

1. **index:** *Taking the length of the blockchain and adding 1 to it. We will use this to reference an individual block, so for instance, the genesis block has index = 1.*

2. **timestamp:** *With the help of the time() module, we will stamp the block when it's created. Users can now check when their transaction was confirmed on-chain.*

3. **transactions:** *Any transactions that have been in the 'pending' list will be displayed in the new block*.

4. **proof:** *This property comes from the miner who thinks they found a valid 'proof' or 'nonce'*.
5. **previous_hash:** *A hashed version of the most recent approved block.*

Example:

```
# Creating a new block listing key/value pair of
# block information in a JSON object.
# Reset the list of pending transactions &
# append the newest block to the chain.
    def newBlock(self, the_proof, previousHash = None):
        the_block = {
            'index': len(self.chain) + 1,
            'timestamp': time(),
            'transactions': self.pendingTransactions,
            'proof': the_proof,
            'previous_hash': previousHash or self.hash(self.chain[-1]),
        }
        self.pendingTransactions = []
        self.chain.append(the_block)

        return the_block
```

Once we add the above properties to the new block, we will include them in the chain. Initially, we empty the pending list of transactions and then add the new block to the self.chain and return it.

3.Writing Functions to Create New Transactions and Get the Last Block

- This **method** will consist of the three most significant variables - the_sender, the_recipient, and the_amount.
- **Without these variables** included in every transaction, the *users cannot spend*, *earn*, or *buy things* with the newly produced cryptocurrency.
- Remember that these transactions are over-simplified and do not reflect the things one may find in a true cryptocurrency.

    - We will include the the_transaction JSON object to the pool of pendingTransactions. These will stay in an indetermination state until a new block is mined and added to the blockchain. And for future reference, we will return the index of the block to which the new transaction is about to be added.

Example:

```
#Searching the blockchain for the most recent block.
    @property
    def lastBlock(self):

        return self.chain[-1]

# Adding a transaction with relevant info to the 'blockpool' - list of pending transactions.
    def newTransaction(self, the_sender, the_recipient, the_amount):
        the_transaction = {
            'sender': the_sender,
            'recipient': the_recipient,
            'amount': the_amount
        }
        self.pendingTransactions.append(the_transaction)
        return self.lastBlock['index'] + 1
```

Explanation:

In the above snippet of code, we defined the method as lastBlock(), which returns the most recent block added. We have then defined the function as newTransaction(), within which we have defined the **JSON** object as the_transaction and included the addresses to the sender, recipient, and amount. We added this JSON object to the pendingTransaction and returned the last block.

4: Writing a Function to "Hash" the Blocks

Now, let us add Cryptography to the program

We will define the method that accepts the new block and alter its key/value pairs into strings. We will then convert that string into Unicode, which we will pass into the SHA256 method from the hashlib library and create a hexadecimal string from its return value. We will then return the new hash.

```
# receiving one block. Turning it into a string, turning that into
# Unicode (for hashing). Hashing with SHA256 encryption,
# then translating the Unicode into a hexadecimal string.
    def hash(self, the_block):
        stringObject = json.dumps(the_block, sort_keys = True)
        blockString = stringObject.encode()

        rawHash = hashlib.sha256(blockString)
        hexHash = rawHash.hexdigest()

        return hexHash
```

Explanation:

In the above snippet of code, we have defined the hash() function and accepts one block and turned them into Strings and then into Unicode for hashing. We have then used the SHA256() function for encryption and then translated the Unicode into a Hexadecimal string.

Step 5: Creating a New Block chain and Sending some money

We will initialize an instance of the Block_chain class and perform some dummy transactions. Make sure to list them in some blocks that we include in the chain.

Example:

```
block_chain = Block_chain()
transaction1 = block_chain.newTransaction("Satoshi", "Alex", '10 BTC')
transaction2 = block_chain.newTransaction("Alex", "Satoshi", '2 BTC')
transaction3 = block_chain.newTransaction("Satoshi", "James", '10 BTC')
block_chain.newBlock(10123)

transaction4 = block_chain.newTransaction("Alex", "Lucy", '2 BTC')
transaction5 = block_chain.newTransaction("Lucy", "Justin", '1 BTC')
transaction6 = block_chain.newTransaction("Justin", "Alex", '1 BTC')
block_chain.newBlock(10384)

print("Genesis block: ", block_chain.chain)
```

Building a block chain in Python – full codes……..

```
1. # importing the required libraries
2. import hashlib
3. import json
4. from time import time
5.
6. # creating the Block_chain class
7. class Block_chain(object):
8.     def __init__(self):
9.         self.chain = []
10.        self.pendingTransactions = []
11.
12.        self.newBlock(previousHash = "The Times 03/Jan/2009 Chancellor on brink of seco
    nd bailout for banks.", the_proof = 100)
13.
14. # Creating a new block listing key/value pairs of
15. # block information in a JSON object.
16. # Reset the list of pending transactions &
17. # append the newest block to the chain.
18.     def newBlock(self, the_proof, previousHash = None):
19.         the_block = {
20.             'index': len(self.chain) + 1,
21.             'timestamp': time(),
22.             'transactions': self.pendingTransactions,
23.             'proof': the_proof,
24.             'previous_hash': previousHash or self.hash(self.chain[-1]),
25.         }
26.         self.pendingTransactions = []
27.         self.chain.append(the_block)
28.
29.         return the_block
30.
31. #Searching the blockchain for the most recent block.
```

```
    @property
    def lastBlock(self):

        return self.chain[-1]

# Adding a transaction with relevant info to the 'blockpool' - list of pending tx's.
    def newTransaction(self, the_sender, the_recipient, the_amount):
        the_transaction = {
            'sender': the_sender,
            'recipient': the_recipient,
            'amount': the_amount
        }
        self.pendingTransactions.append(the_transaction)
        return self.lastBlock['index'] + 1

# receiving one block. Turning it into a string, turning that into
# Unicode (for hashing). Hashing with SHA256 encryption,
# then translating the Unicode into a hexidecimal string.
    def hash(self, the_block):
        stringObject = json.dumps(the_block, sort_keys = True)
        blockString = stringObject.encode()

        rawHash = hashlib.sha256(blockString)
        hexHash = rawHash.hexdigest()

        return hexHash

block_chain = Block_chain()
transaction1 = block_chain.newTransaction("Satoshi", "Alex", '10 BTC')
transaction2 = block_chain.newTransaction("Alex", "Satoshi", '2 BTC')
transaction3 = block_chain.newTransaction("Satoshi", "James", '10 BTC')
block_chain.newBlock(10123)

transaction4 = block_chain.newTransaction("Alex", "Lucy", '2 BTC')
```

```
transaction5 = block_chain.newTransaction("Lucy", "Justin", '1 BTC')
transaction6 = block_chain.newTransaction("Justin", "Alex", '1 BTC')
block_chain.newBlock(10384)

print("Genesis block: ", block_chain.chain)
```

Output

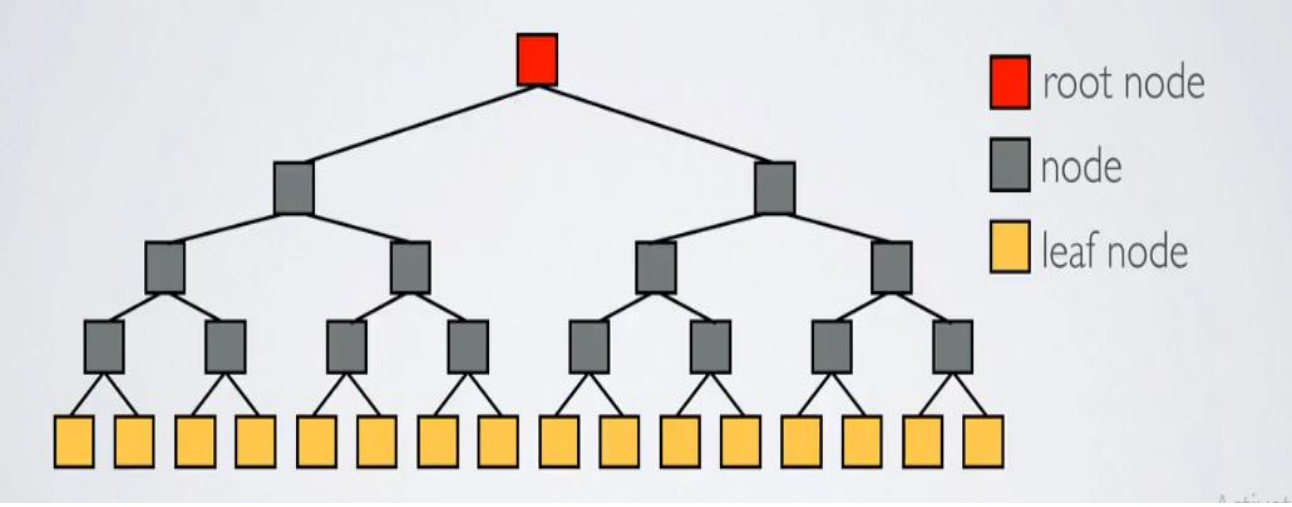
```
Genesis block:  [{'index': 1, 'timestamp': 1666143831.7977703, 'transactions': [], 'proof': 100, 'previous_hash': 'The Times 03/Jan/2009 Chancellor on brink of second bailout for banks.'}, {'index': 2, 'timestamp': 1666143831.7977812, 'transactions': [{'sender': 'Satoshi', 'recipient': 'Alex', 'amount': '10 BTC'}, {'sender': 'Alex', 'recipient': 'Satoshi', 'amount': '2 BTC'}, {'sender': 'Satoshi', 'recipient': 'James', 'amount': '10 BTC'}], 'proof': 10123, 'previous_hash': 'efe671227454b2bb8a5e16953d468098cb84a235a483ca5942d86be0f65d55b8'}, {'index': 3, 'timestamp': 1666143831.7978356, 'transactions': [{'sender': 'Alex', 'recipient': 'Lucy', 'amount': '2 BTC'}, {'sender': 'Lucy', 'recipient': 'Justin', 'amount': '1 BTC'}, {'sender': 'Justin', 'recipient': 'Alex', 'amount': '1 BTC'}], 'proof': 10384, 'previous_hash': '9059e64c427491fa88cd7c711738200ee613a7988f0145f02a8a48f4acac5f96'}]

>
```

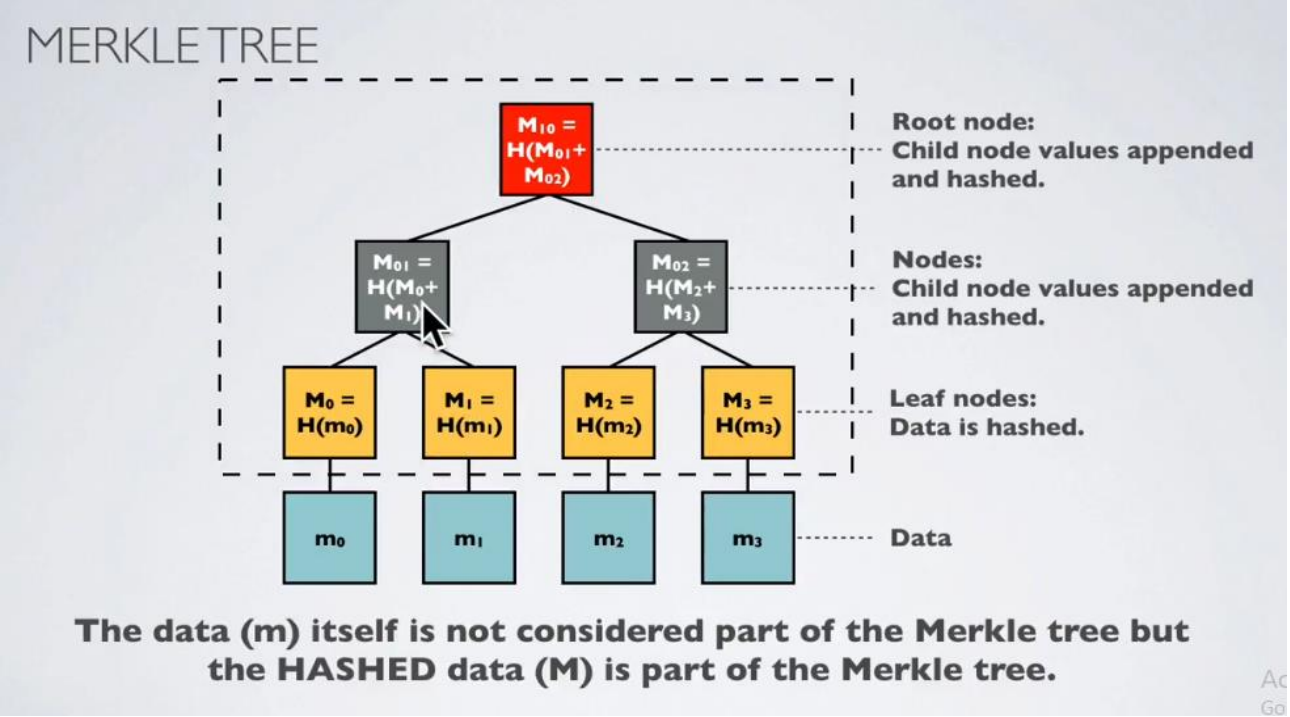
What are Merkle trees? How important are Merkle trees in Blockchains?

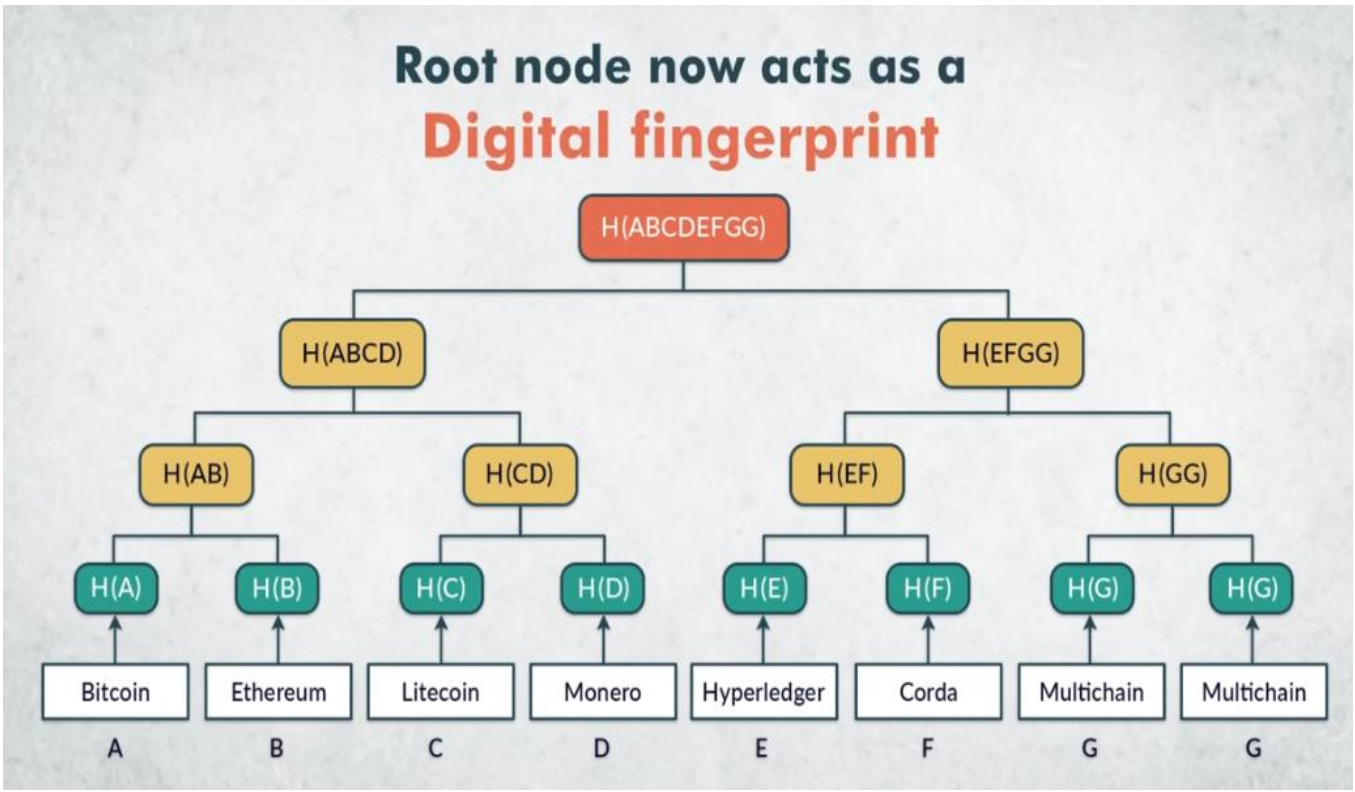
- Merkle Tree also known as 'hash tree' is a data structure in cryptography in which each leaf node is a hash of a block of data, and each non-leaf node is a hash of its child nodes.
- **The benefit of using the Merkle Tree in blockchain-** *is that instead of downloading every transaction and every block, a "light client" can only download the chain of block headers.*

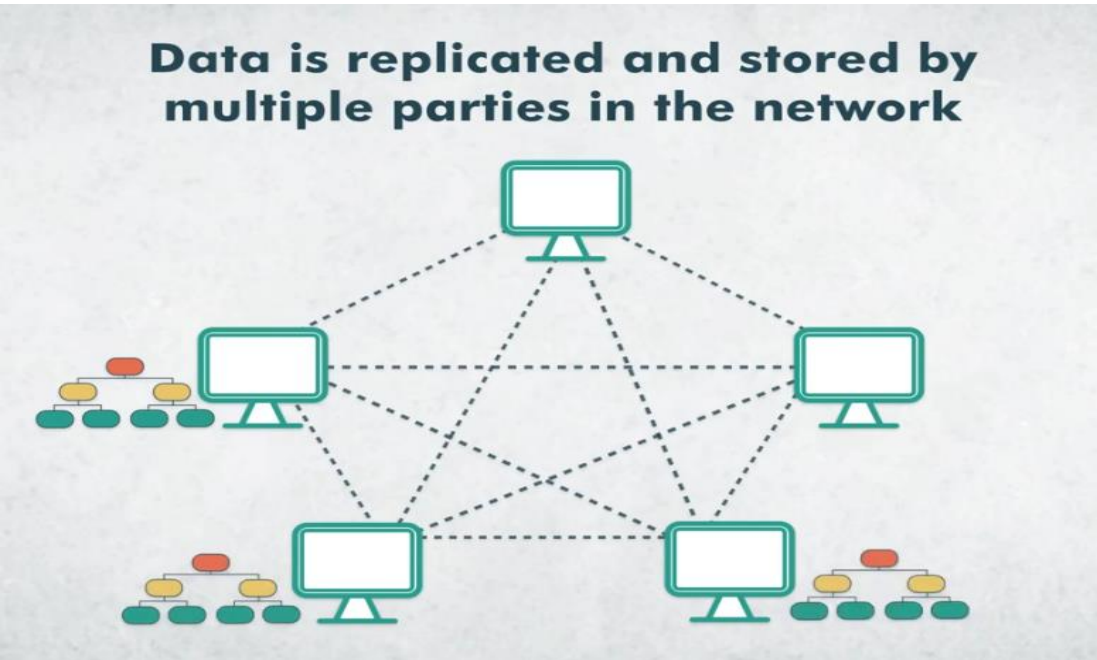
Why use merkel data structure.

1. **Validate the integrity of data:** It can be effectively used to validate the integrity of the data.
2. **Takes little disk space:** Merkle tree takes little disk space compared to other data structures.
3. **Tiny information across networks:** Merkle trees can be divided into tiny information for verification.
4. **Efficient verification:** The data structure is efficient and takes only a while to verify the integrity of the data.




MERKLE TREE
$M_{10} = H(M_{01} + M_{02})$
$M_{01} = H(M_0 + M_1)$
$M_{02} = H(M_2 + M_3)$
$M_0 = H(m_0)$
$M_1 = H(m_1)$
$M_2 = H(m_2)$
$M_3 = H(m_3)$
m_0
m_1
m_2
m_3
Root node:
Child node values appended and hashed.
Nodes:
Child node values appended and hashed.
Leaf nodes:
Data is hashed.
Data
The data (m) itself is not considered part of the Merkle tree but the HASHED data (M) is part of the Merkle tree.


Root node now acts as a
Digital fingerprint
H(ABCDEFGG)
H(ABCD)
H(EFGG)
H(AB)
H(CD)
H(EF)
H(GG)
H(A)
H(B)
H(C)
H(D)
H(E)
H(F)
H(G)
H(G)
Bitcoin
Ethereum
Litecoin
Monero
Hyperledger
Corda
Multichain
Multichain
A
B
C
D
E
F
G
G



Block chain Revision questions

1. Blockchain is a distributed database. How does it differ from traditional databases?

2. What are the properties of Blockchain?

3. What is encryption? What is its role in Blockchain?

4.What type of records can be kept in a Blockchain? Is there any restriction on same?

5. explain the components of Blockchain Ecosystem?

6.Discuss some of the widely used cryptographic algorithms?

- RSA
- Triple DES
- MD5
- Blowfish
- AES

7.How Blockchain solves the problem