**Blockchain Technology**

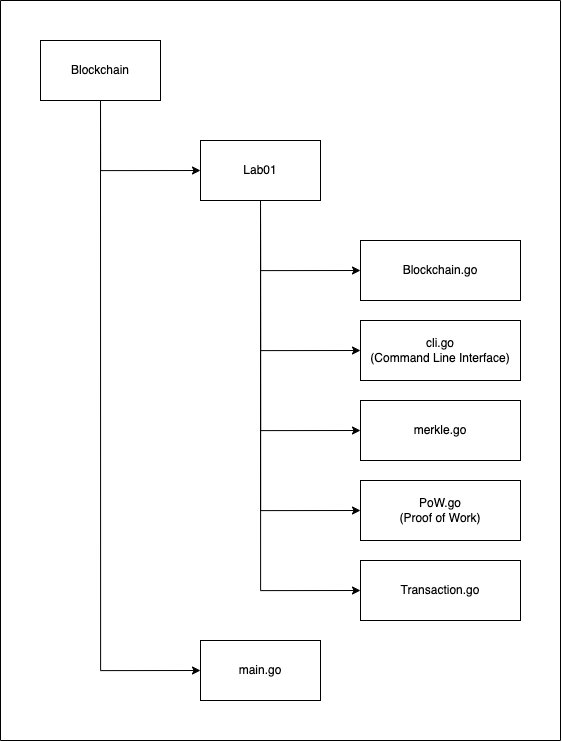
Lab 01: Simplified Blockchain Implementation and Verification

* 2TT Team Information
  + 22C11025 – Hoàng Thủy Trúc
  + 22C11046 – Nguyễn Trần Vĩnh Thuyên

# Project Structure

The project structure includes:

* Blockchain directory: The project directory.
* Lab01 directory: The directory contains source code.
* main.go: The main program file to run the project.



# Data Structure

## Blockchain

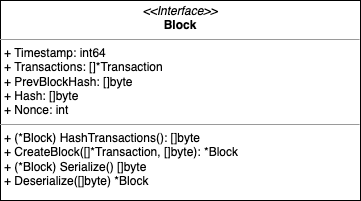
**The block has the following structure:**

Fields:

* Timestamp: the current timestamp when the block is created.
* Transactions: the transaction information
* PrevBlockHash: the hash of the previous block
* Hash: the hash of the block
* Nonce: an random integer to make sure the validity of hashes.

Functions:

* HashTransactions(): hash all data fields in a block and initialize a new merkle tree.
* CreateBlock([]\*Transaction, []byte): create a new block.
* Serialize(): serialize a block to array of bytes.
* Deserialize(): deserialize array of bytes to a block.



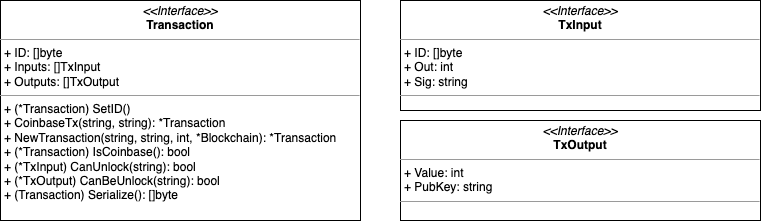
**The transaction has the following structure:**

Fields:

* ID: the transaction’s ID
* Inputs: the transaction inputs, which refer to the sources of funds for the transaction.
* Outputs: the transaction outputs, which represent the destination and amount of funds transferred.

Functions:

* SetID(): set unique transaction identifier via hashing the transaction data.
* CoinbaseTx(string, string): create the first transaction in the blockchain.
* NewTransaction(string, string, int, \*Blockchain): create new transaction for the blockchain.
* IsCoinbase(): check whether a block is the first transaction or not
* CanUnlock(string): check whether a given address is valid or not
* CanBeUnlock(string): check whether a given address is valid or not
* Serialize(): serialize a transaction to array of bytes.



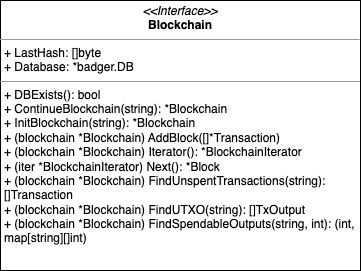
**The blockchain has the following structure:**

Fields:

* LastHash []byte: the last hash of the blockchain.
* Database \*badger.DB: a database to store blocks locally.

Functions:

* DBExist(): check whether a database exists or not
* ContinueBlockchain(string): return the existing blockchain.
* InitBlockchain(string): create a blockchain with an address.
* AddBlock([]\*Transaction): add a block to a blockchain.
* Iterator(): return a blockchain iterator
* Next(): return a next block in a blockchain.
* FindUnspentTransactions(string): finds all unspent transactions of an address and calculate balance of the address.
* FindSpentTransaction(string): finds all spent transactions of an address and calculate balance of the address.
* FindUTXO(string): find a transaction for an address.



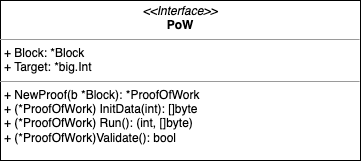
**The proof of work has the following structure:**

Fields:

* Block: a block is in the validity process.
* Target: an generated integer represents a set of predefined requirements.

Functions:

* NewProof(\*Block): create a new proof of work for a block.
* InitData(int): combine all data fields of a block.
* Run(): execute the proof of work.
* Validate(): check whether proof of work is valid or not.



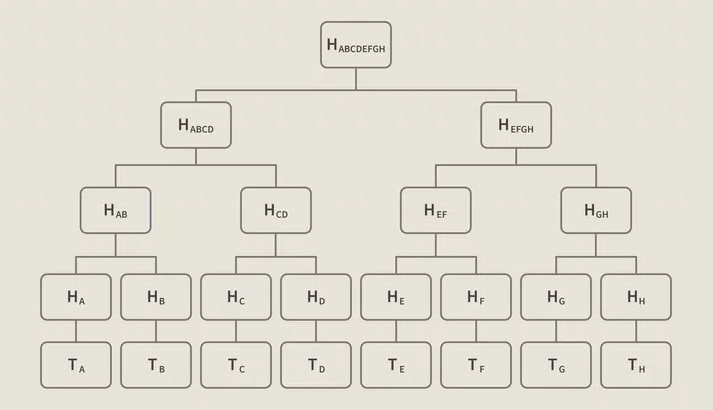
## Merkle Tree

* 1. Principle:

Merkle Trees, conceptualized by Ralph Merkle, combined the concept of hash lists into robust tree structures. Each leaf node houses a cryptographic hash of a data block, while non-leaf nodes hold hashes of their children. Though typically binary (two children per node), Merkle trees can be customized as n-nary trees.

Their efficiency in distributed systems stems from leveraging hashes instead of entire files. Hashes act as compact fingerprints, making verification significantly faster. This unique advantage fuels their application in diverse areas like Tor for anonymized routing, Bitcoin for secure transaction verification, and Git for code version integrity.

So, how it work, let's look at the picture below:



The leaf nodes HA, HB, … , HH holds the hash of the data blocks T by some hash function (example: sha256). HAB is the parent node of HA and HB which contain hash(HA, HB). Continue like this until there is only one last node left, we get the Merkle Root HABCDEFGH

In distributed systems, ensuring data consistency across multiple locations is paramount. Verifying every file byte-by-byte, however, is resource-intensive and impractical. The Merkle Tree is useful because it allows users to verify a specific block without downloading the whole blockchain.

Instead of sending entire files, Merkle trees leverage cryptographic hashes, unique fingerprints of the data. Computer A transmits a file's hash to B. This hash is then used to query whether it exists in the Merkle tree or not and with this hash, the Merkle Root Hash has been changed or not. There are many ways to query, from this hash, we can create a Merkle Path by querying back from the leaf node to the root node. The good thing here is that we only need to check related nodes, not all nodes in the tree. This greatly reduces computational costs.

If they match, verification is complete. If not, a targeted approach kicks in:

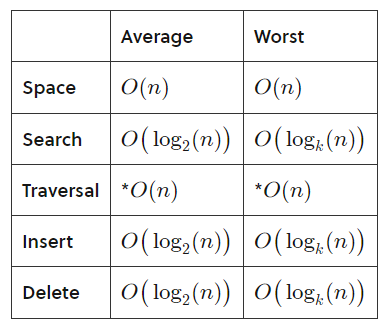
1. B requests subtrees based on the mismatch location.
2. A generates and sends the requested subtrees.
3. B repeats steps 1-2 until pinpointing the corrupted data block(s).

This iterative process minimizes network traffic by only transferring relevant data portions. Furthermore, Merkle trees empower verification from untrusted sources – a key benefit in peer-to-peer systems like Tor.

By obtaining the root hash from a trusted source, users can progressively validate downloaded data chunks from potentially untrusted peers. Matching hashes indicate valid data, while mismatches trigger rejection and retrieval from alternative sources.

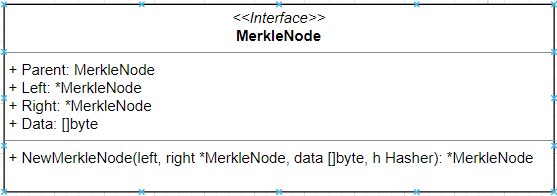
Merkle trees, therefore, offer a powerful combination of efficiency and robust verification, making them invaluable tools in both distributed and peer-to-peer networks.

About the complexity, Merkle Trees have very little overhead when compared with hash lists. For instance, binary Merkle trees operate similarly to binary search trees in that their depth is bounded by their branching factor 2. Included below is worst-case analysis for a Merkle tree with a branching factor of k.



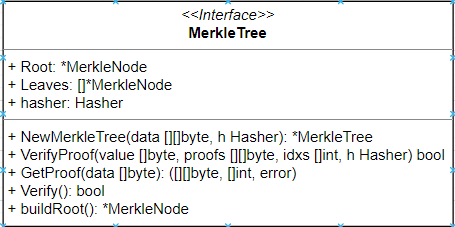
* 1. Implementation

The Merkle node has the following structure:



* Fields:
  + Parent: pointer point to parent node
  + Left: pointer point to left child node
  + Right: pointer point to right child node
  + Data: data of the node
* Functions:
  + NewMerkleNode: create and return a Merkle Node
    - left, right is the input child of the node want to create
    - data: data of node
    - h: hash object contain hash function use for this Merkle Node

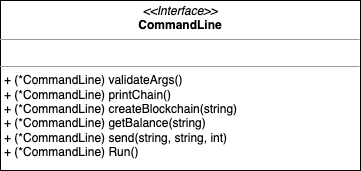
The Merkle tree has the following structure:



* Fields:
  + Root: Merkle Root Node of the tree
  + Leaves: list of leaf Node in tree
  + hasher: hash object contain hash function use for this Merkle Tree
* Functions:
  + NewMerkleTree: create new Merkle Tree
    - data: list of node data
    - h: hash object contain hash function for this tree
  + VerifyProof: verifies the integrity of the given value
    - value: data need to verify
    - proofs & idxs: Merkle Path of data
    - h: hash object contain hash function
  + GetProof: returns the Merkle path proof to verify the integrity of the given data
    - data: data need to verify
  + Verify: verify by rebuild the tree and compare new tree hash with current hash
  + buildRoot: build tree with given leaf nodes

## Command Line Interface (CLI)

The CLI has the following structure:



# System Overview

The system is a simplified blockchain. A blockchain is basically a distributes database of records. The system’s functionalities involve:

* Create a blockchain and sends genesis reward to address.
* Send amount of coins to another address.
* Get the current balance for an address.
* Prints the blocks in the blockchain.

## Consensus:

In blockchain, consensus is a mechanism for ensuring that all nodes in the network agree on the state of the blockchain. Consensus is essential to prevent network attacks.

There are many different types of consensus mechanisms used in blockchain, but some of the most common include:

* **Proof of work (PoW)**: PoW is a consensus mechanism based on solving complex mathematical problems. Nodes in the network compete to solve these problems, and the first node to solve a problem is allowed to add a new block to the blockchain.
* **Proof of stake (PoS)**: PoS is a consensus mechanism based on staking assets. Nodes in the network stake a certain amount of assets into the network, and nodes with a higher stake have a higher chance of adding a new block to the blockchain.
* **Delegated proof of stake (DPoS)**: DPoS is a variation of PoS in which nodes in the network elect a certain number of nodes to represent them. These representative nodes then become responsible for adding new blocks to the blockchain.
* **Proof of Capacity (PoC)**: The PoC mechanism heavily relies on free space available in the hard drive. This is because there are many solutions to a coin's hash problem that a trader needs to store. It is highly efficient as compared to PoW and PoC mechanisms. Coins such as Burst, Storj, SpaceMint and Chia use these mechanisms.
* **Proof of Activity (PoAc)**: This mechanism is a combination of both Proof of Work and Proof of Stake. It has been designed to combine the best features of PoW and PoS. In the beginning, the Proof-of-Activity mechanism functions like PoW. Once a new block is completed, it starts to function like a Proof-Of-Stake mechanism. Coins such as DCR (Decred) use this mechanism.
* **Proof of Authority (PoAuth)**: Different organizations and private companies created this unique mechanism. There are validators with approved accounts which authorize transactions and the creation of new blocks. These validators must disclose their true identity to get the right to validate a transaction.
* **Proof of Burn (PoB)**: aims to improve the quality of blockchain so that it can be used easily and extensively as a tool for faster and more secured transactions. After PoW and PoS, PoB is designed to prevent fraud activities on a blockchain network. Cryptocurrencies such as Bitcoin use this mechanism to offer secure transactions to traders.
* **Proof of Elapsed Time (PoET)**: Intel Corporation created this mechanism to permit blockchain to decide the person who will create the next block. It uses a lottery system to decide the next block creator. Thus, it gives a fair chance to all traders to create the next block. It is an efficient process involving utilizing lesser resources and low energy consumption.

In blockchain, the consensus keepers are the nodes in the network responsible for validating and agreeing on the state of the blockchain. These nodes are often referred to as validators.

For example, in the Bitcoin blockchain, the nodes that solve the PoW problem first are allowed to add a new block to the blockchain. These nodes are called miners. In the Ethereum blockchain, the nodes that stake a certain amount of ETH into the network have a higher chance of adding a new block to the blockchain. These nodes are called validators.

Consensus mechanisms are an important part of blockchain, and they help to ensure the security and integrity of the blockchain. Some of the key benefits of consensus mechanisms include:

* Security: Consensus mechanisms help to prevent network attacks by making it difficult for one node or group of nodes to control the network.
* Transparency: Consensus mechanisms are transparent, meaning that anyone can see how the blockchain is being updated.
* Efficiency: Consensus mechanisms are efficient, meaning that they can be used to add new blocks to the blockchain quickly and securely.

In this lab, we focus on **Proof of Work** to build the basic Blockchain network.

## Create a blockchain

The blockchain is implemented as **an ordered list of blocks** and uses **Badger.DB** library as a database to store blocks.

Each block comprises several key components: a **Timestamp** indicating the current timestamp when the block is created; an **Transactions** array that holds transaction information within the block; a **PrevBlockHash**, represents the hash of the preceding block; a **Hash**, signifying is the hash of the block and a **Nonce**, a randomly generated integer that represents a set of requirements to ensure the validity of hashes.

Upon initial system execution, the user is required to create a blockchain with an address. Once this process is successfully completed, the system will then allocate **a reward 100 coins** to the address.

During the blockchain creation process, the initial transaction, referred to as **"CoinbaseTx"** is generated, and subsequently, the system proceeds to construct the block. The first block incorporates the CoinbaseTx transaction and undergoes validation through the **proof-of-work mechanism**. Each block is intricately connected to the preceding one via a hash function. The hash data results from the concatenation of all fields within a block. The hash algorithm initiates a counter (nonce) at 0, creating a hash of the data plus the counter. This process iterates until the hash is scrutinized to ascertain whether it exceeds a predefined target, often characterized by a specified number of leading zero bytes. After the block is created, the system stores the block to the database via a **Badger.DB** library.

To create a blockchain, use the following command line:

| go run main.go createblockchain -address ADDRESS |
| --- |

## Send coins

This function allows users to send coins to another address. During the send process, the system identifies all transactions associated with the specified address and computes the existing balance. If the calculated balance is equal to or greater than the intended amount of coins to be sent, a new transaction is generated. Following this, the system proceeds to construct the block using the aforementioned mechanism.

To send coins, use the following command line:

| go run main.go send -from ADDRESS -to ADDRESS -amount AMOUNT |
| --- |

## Get the current balance

This function enables users to check the current balance of their address. Throughout this process, the system identifies all transactions associated with the given address and calculates the balance.

To get the current balance of an address, use the following command line:

| go run main.go getbalance -address ADDRESS |
| --- |

## Print the blockchain

This function allows users to print all blocks currently stored in the blockchain. To print the blockchain, use the following command line:

| go run main.go print |
| --- |

# Challenges

* How to use Go: This is the first time we use Golang so we often use ChatGPT to understand samples in the tutorials.
* How blockchain systems work: We still can not get the idea how to implement the blockchain after reading the assignment and the scope of blockchain. So we search some blockchain tutorials to understand how the blockchain works and modify them to align the assignment.

# Lessons

* Learn about the Go language.
* Learn about how to implement basic blockchain modules.
* Learn about how to implement Proof Of Work and what difficulty is in a PoW consensus algorithm.
* Learn about the Nonce.
* Learn about building command line interfaces with Go.
* Learn about Merkle Tree, its implementation and problems

# References

* Consensus Mechanism:
  + <https://cleartax.in/s/consensus-in-blockchain>
  + <https://www.geeksforgeeks.org/consensus-algorithms-in-blockchain/>
  + <https://www.investopedia.com/terms/c/consensus-mechanism-cryptocurrency.asp>
* Merkle Tree Codes:
  + <https://github.com/douglasmakey/mktree/tree/main>
  + <https://github.com/cbergoon/merkletree>
  + <https://steemit.com/utopian-io/@tensor/building-a-blockchain-with-go---go-modules-and-a-basic-blockchain---part-1>
* Merkle Tree Docs:
  + <https://brilliant.org/wiki/merkle-tree/>
  + <https://www.investopedia.com/terms/m/merkle-tree.asp>
  + <https://www.simplilearn.com/tutorials/blockchain-tutorial/merkle-tree-in-blockchain>