

Simulação de uma Blockchain

Trabalho de Estrutura de Dados

Fatec São Caetano do Sul



Pedro Coelho, Thiago Ulloa,
Matheus Macedo e Kawai
Soares



Descrição

Passo 1

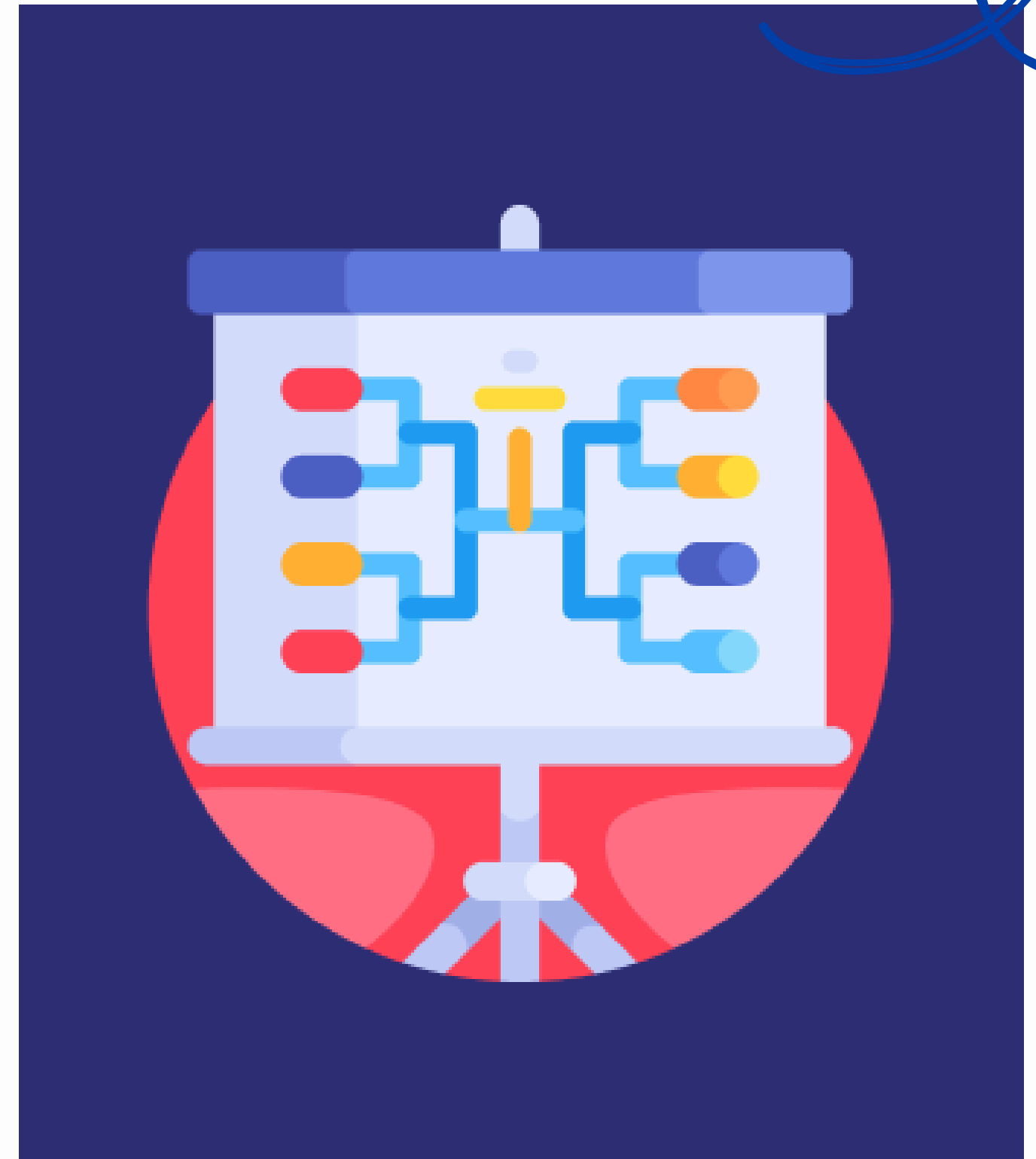
- Entender o caso de negócio

Passo 2

- Explorar a relação com Estrutura de Dados

Passo 3

- Implementação

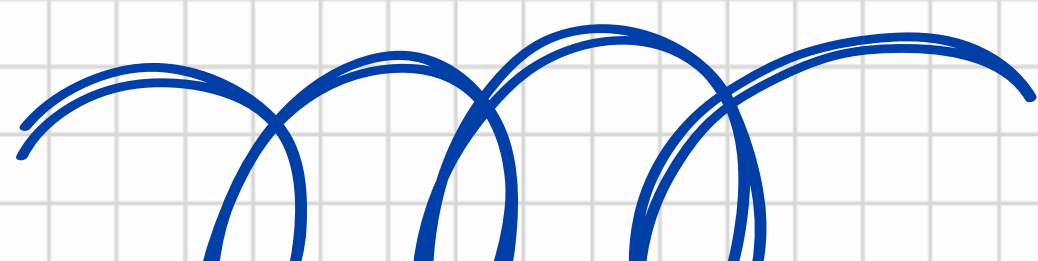




Problema



A blockchain busca resolver questões de segurança e confiança ao armazenar dados de forma descentralizada. Em um sistema tradicional, os dados podem ser manipulados ou perdidos, mas com uma blockchain:

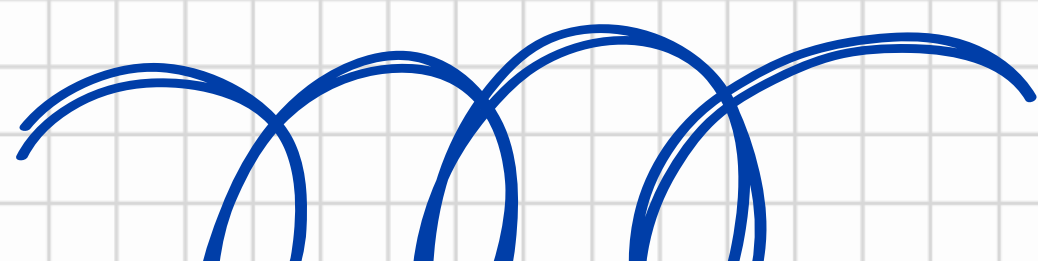
- **Segurança:** Os dados são protegidos por criptografia.
 - **Imutabilidade:** Após inseridos, os dados não podem ser modificados sem invalidar os blocos subsequentes.
 - **Descentralização:** Os dados podem ser verificados sem depender de uma entidade central.
- 



Objetivo



O objetivo é criar um sistema de blockchain funcional que realiza as seguintes tarefas:

1. Mineração de blocos, onde um hash válido deve ser encontrado.
 2. Validação da blockchain para garantir a integridade dos blocos.
 3. Armazenamento de transações em blocos, com um processo de verificação baseado em "prova de trabalho".
- 

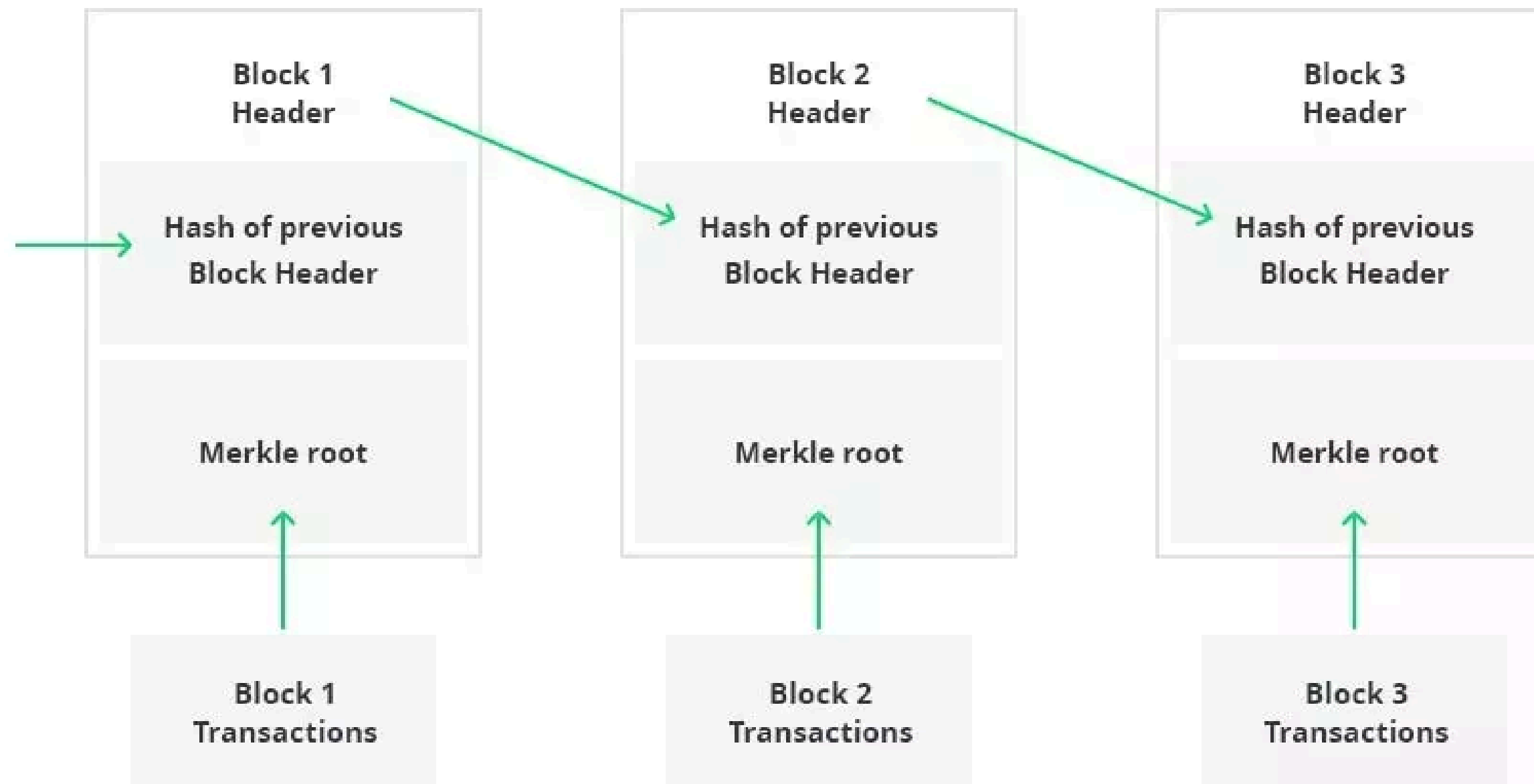
O que é uma blockchain?

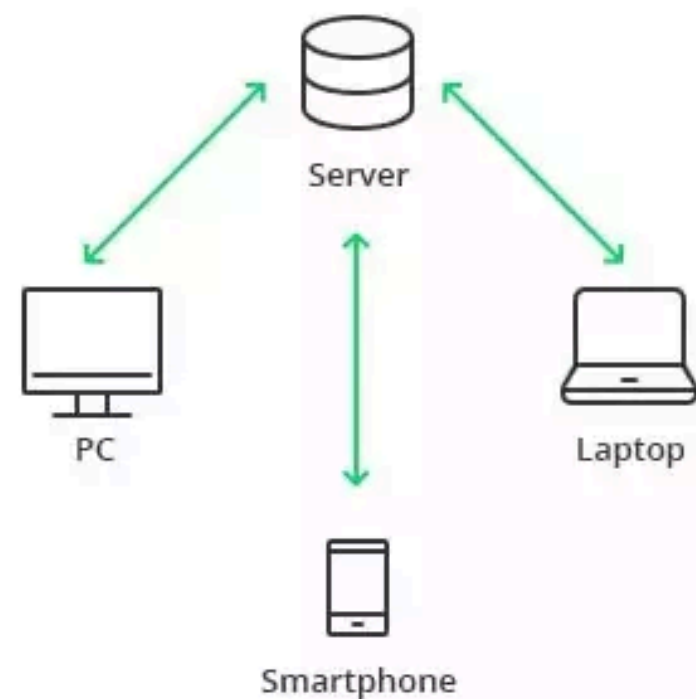
Block + Chain

- Dados armazenados em **lotes sequenciais** ou "blocos"
- Se você faz uma transação, os **dados** da transação precisam ser adicionados a um bloco para que ela seja bem-sucedida.
- Refere-se ao fato de cada bloco **referenciar**, criptograficamente, seu bloco-pai
- Os dados de um bloco **não** podem ser alterados sem mudar **todos** os blocos subsequentes, o que exigiria o consenso de toda a rede ou a invalidará a blockchain.

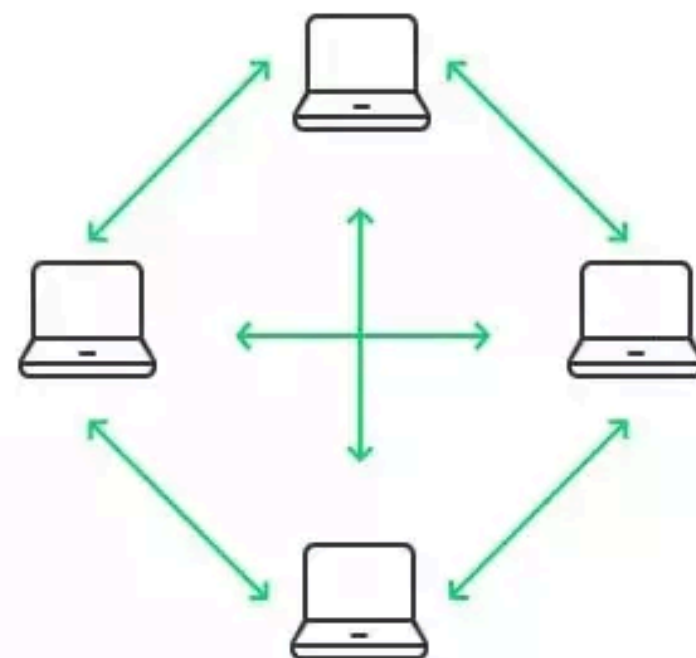
A estrutura de dados do blockchain pode ser comparada a um livro razão ou um registro de transações.

Estrutura



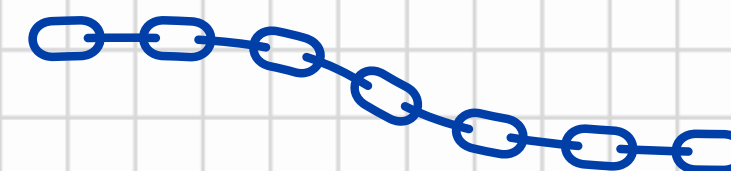


Client-server



P2P network

Arquitetura de Banco de Dados vs Blockchain



Em um banco de dados, os dados são estruturados em tabelas, enquanto em um blockchain, os dados são estruturados em pedaços (blocos) que são “amarrados” juntos (corrente).

Estrutura de Blocos

```
struct Block {  
    index: u64,  
    timestamp: u128,  
    data: String,  
    previous_hash: String,  
    hash: String,  
    nonce: u64,  
    next: Option<Box<Block>>, // Apontador para o próximo bloco  
}
```

- **index (u64)**: Representa o índice do bloco na cadeia.
- **timestamp (u128)**: Indica o momento em que o bloco foi minerado, em milissegundos.
- **data (String)**: Contém as informações no bloco, como transações.
- **previous_hash (String)**: O hash do bloco anterior, que conecta os blocos da cadeia.
- **hash (String)**: O hash atual do bloco, calculado durante o processo de mineração.
- **nonce (u64)**: Um número incrementado na mineração até que se encontre um hash válido.
- **next (Option<Box<Block>>)**: Aponta para o próximo bloco na lista ligada.

Estrutura da Blockchain

```
struct Blockchain {  
    head: Option<Box<Block>>, // Primeira referência na lista ligada  
    difficulty: usize,  
}
```

- **head: Option<Box<Block>>:** A referência ao primeiro bloco (cabeça) da lista ligada.
- **difficulty: usize:** O nível de dificuldade para a mineração, que determina quantos zeros iniciais devem estar presentes no hash.

- A estrutura **Blockchain** representa a cadeia de blocos e contém os seguintes campos:

Laços de Repetição Mineração:

```
while !Blockchain::is_valid_hash(&new_block.hash, self.difficulty) {  
  new_block.nonce += 1;  
  new_block.hash = Blockchain::calculate_hash(&new_block);  
}
```

- Durante a mineração, a função **mine_block** ajusta o valor do **nonce** repetidamente até que o hash do bloco satisfaça o critério de dificuldade.

Laços de Repetição Inserção de Blocos:

```
let mut current = &mut self.head;
while let Some(ref mut block) = current {
    if block.next.is_none() {
        // Achou o último bloco, podemos criar o novo bloco aqui
        let previous_hash = block.hash.clone();
        let mut new_block = Block {
            index: block.index + 1,
            timestamp: Blockchain::current_timestamp(),
            data,
            previous_hash,
            hash: String::new(),
            nonce: 0,
            next: None,
        };
    }
}
```

- A inserção de um novo bloco na blockchain também envolve um laço para encontrar o último bloco na lista ligada:

Validação da Blockchain com Recursividade:

```
// Valida a blockchain recursivamente
fn validate_chain_recursively(&self) -> bool {
    if let Some(ref block) = self.head {
        return Blockchain::validate_block_recursively(block);
    }
    true
}

// Valida cada bloco com o anterior usando recursão
fn validate_block_recursively(block: &Block) -> bool {
    if let Some(ref next_block) = block.next {
        if block.hash != next_block.previous_hash {
            return false;
        }
        if next_block.hash != Blockchain::calculate_hash(next_block) {
            return false;
        }
        return Blockchain::validate_block_recursively(next_block);
    }
    true
}
```

A recursividade ocorre na função `validate_block_recursively`. A função chama a si mesma para validar o próximo bloco na cadeia. Essa chamada recursiva continua até que o último bloco da cadeia seja atingido, que não possui um próximo bloco.

Validação da Blockchain com Iteração:

```
// Valida a blockchain utilizando iteração
fn validate_chain_iteratively(&self) -> bool {
    let mut current = &self.head;

    // Verifica a integridade de cada bloco em relação ao anterior
    while let Some(ref block) = current {
        if let Some(ref next_block) = block.next {
            // Verifica se o hash atual do bloco é igual ao hash anterior do próximo bloco
            if block.hash != next_block.previous_hash {
                return false;
            }
            // Verifica se o hash do próximo bloco é válido
            if next_block.hash != Blockchain::calculate_hash(next_block) {
                return false;
            }
        }
        current = &block.next;
    }
    true
}
```

A iteração no código é feita utilizando um loop while let, que percorre cada bloco da blockchain a partir do bloco inicial (chamado de head).

O laço while let é utilizado para iterar enquanto houver um bloco válido. Ele verifica se a variável current contém um bloco (Some(ref block)). Se houver um bloco, a iteração continua; caso contrário, o loop termina.

Ferramentas e linguagens Utilizadas:

- **Rust:** Linguagem de programação utilizada para implementar a simulação da blockchain. O rust oferece segurança de memória e excelente desempenho para sistemas concorrentes, como blockchains.
- **Sha2:** Biblioteca utilizada para gerar hashes criptográficos no padrão **SHA-256**, que é amplamente utilizado em blockchains reais, como o Bitcoin.
- **Sistema de Tempo (SystemTime):** Utilizado para gerar o timestamp de cada bloco, permitindo o registro de quando o bloco foi criado.
- **Cargo:** Ferramenta de build e gerenciamento de pacotes do ecossistema Rust.

Iteração vs Recursividade na Validação da Blockchain

Iteração

Vantagem:

- Direto e eficiente: O loop for percorre a cadeia de forma linear, realizando um número constante de operações por bloco.
- Menor sobrecarga: Não há chamadas de função recursivas, o que reduz a sobrecarga da pilha.

Desvantagem:

- Menos elegante: Para problemas recursivos, a solução iterativa pode ser menos intuitiva.

Recursividade

Vantagem:

- Elegante: A solução recursiva reflete a natureza hierárquica da blockchain.
- Mais concisa: A lógica pode ser mais concisa em alguns casos.

Desvantagem:

- Sobrecarga de chamadas: Cada chamada recursiva consome memória da pilha.
- Menos eficiente: A chamada de função recursiva tem um custo adicional.



Obrigado pela Atenção

**Entre em nosso
Servidor!**



Pedraõ:
11934454351

