História e Motivação

Ralph Merkle introduziu as Árvores de Merkle em seu artigo "A Digital Signature Based on a Conventional Encryption Function" em 1979. O objetivo era permitir o uso eficiente de assinaturas digitais de uso único, como as assinaturas de Lamport, que, embora seguras, exigiam uma chave diferente para cada mensagem. Ao organizar essas assinaturas em uma estrutura hierárquica de hash, Merkle permitiu que uma única raiz de hash representasse múltiplas assinaturas, otimizando o armazenamento e a verificação .(Wikipédia)

Estrutura e Funcionamento

Uma Árvore de Merkle é uma árvore binária onde:

- **Folhas**: Contêm os hashes dos blocos de dados individuais.
- **Nós internos**: Cada nó é o hash da concatenação dos hashes de seus dois filhos.
- **Raiz de Merkle**: O hash no topo da árvore que representa todo o conjunto de dados. (Investopedia)

Essa estrutura permite verificar a integridade de qualquer bloco de dados com eficiência, pois apenas um caminho da folha à raiz precisa ser recalculado e comparado.(Investopedia)

Aplicações Práticas

Blockchain

Em blockchains como o Bitcoin, as Árvores de Merkle são usadas para:(Investopedia)

- **Verificação de transações**: Permitem que nós verifiquem se uma transação está incluída em um bloco sem precisar baixar todo o bloco.
- **Eficiência**: Reduzem a quantidade de dados que precisam ser transmitidos e armazenados, economizando largura de banda e espaço .(Webopedia)

Sistemas de Arquivos e Redes P2P

Sistemas como o IPFS e o BitTorrent utilizam Árvores de Merkle para:(GeeksforGeeks)

- **Verificação de integridade**: Asseguram que os arquivos compartilhados não foram corrompidos ou alterados durante a transferência.
- **Eficiência na distribuição**: Permitem que partes de arquivos sejam verificadas independentemente, facilitando o download paralelo e a reconstrução de arquivos grandes .

Vantagens

- **Verificação eficiente**: A verificação de um dado específico requer apenas o cálculo de um número logarítmico de hashes em relação ao número total de folhas.

- **Integridade dos dados**: Qualquer alteração em um bloco de dados altera o hash correspondente, propagando-se até a raiz, o que facilita a detecção de modificações.
- **Economia de espaço**: A raiz de Merkle fornece uma representação compacta de todo o conjunto de dados, economizando espaço de armazenamento e largura de banda .(Wikipedia, Investopedia, academy.wirexapp.com)

Desvantagens

- **Sobrecarga computacional**: A construção e a atualização da árvore podem ser computacionalmente intensivas, especialmente para grandes volumes de dados.
- **Desafios com dados dinâmicos**: A adição ou remoção de dados pode exigir a reconstrução parcial ou total da árvore, o que pode ser ineficiente .
- **Dependência de funções de hash seguras**: A segurança da estrutura depende da resistência a colisões das funções de hash utilizadas. Vulnerabilidades nessas funções podem comprometer a integridade da árvore .(GeeksforGeeks, WallStreetMojo)

Aula Avançada de Ciência da Computação: Árvores de Merkle

---

1. Introdução

As Árvores de Merkle, ou *Merkle Trees*, são estruturas de dados fundamentais em sistemas distribuídos, especialmente em blockchains, onde garantem a integridade e a eficiência na verificação de grandes volumes de dados. Concebidas por Ralph Merkle em 1979, essas árvores utilizam funções de hash para criar uma representação compacta e segura de conjuntos de dados.

---

2. Histórico e Motivação

Ralph Merkle introduziu as Árvores de Merkle em seu artigo "A Digital Signature Based on a Conventional Encryption Function" em 1979. O objetivo era permitir o uso eficiente de assinaturas digitais de uso único, como as assinaturas de Lamport, que, embora seguras, exigiam uma chave diferente para cada mensagem. Ao organizar essas assinaturas em uma estrutura hierárquica de hash, Merkle permitiu que uma única raiz de hash representasse múltiplas assinaturas, otimizando o armazenamento e a verificação.

---

3. Estrutura e Funcionamento

Uma Árvore de Merkle é uma árvore binária onde:

- **Folhas**: Contêm os hashes dos blocos de dados individuais.
- **Nós internos**: Cada nó é o hash da concatenação dos hashes de seus dois filhos.
- **Raiz de Merkle**: O hash no topo da árvore que representa todo o conjunto de dados.

Essa estrutura permite verificar a integridade de qualquer bloco de dados com eficiência, pois apenas um caminho da folha à raiz precisa ser recalculado e comparado.

---

4. Aplicações Práticas

4.1 Blockchain

Em blockchains como o Bitcoin, as Árvores de Merkle são usadas para:

- **Verificação de transações**: Permitem que nós verifiquem se uma transação está incluída em um bloco sem precisar baixar todo o bloco.
- **Eficiência**: Reduzem a quantidade de dados que precisam ser transmitidos e armazenados, economizando largura de banda e espaço.

4.2 Sistemas de Arquivos e Redes P2P

Sistemas como o IPFS e o BitTorrent utilizam Árvores de Merkle para:

- **Verificação de integridade**: Asseguram que os arquivos compartilhados não foram corrompidos ou alterados durante a transferência.
- **Eficiência na distribuição**: Permitem que partes de arquivos sejam verificadas independentemente, facilitando o download paralelo e a reconstrução de arquivos grandes.

---

5. Vantagens

- **Verificação eficiente**: A verificação de um dado específico requer apenas o cálculo de um número logarítmico de hashes em relação ao número total de folhas.
- **Integridade dos dados**: Qualquer alteração em um bloco de dados altera o hash correspondente, propagando-se até a raiz, o que facilita a detecção de modificações.
- **Economia de espaço**: A raiz de Merkle fornece uma representação compacta de todo o conjunto de dados, economizando espaço de armazenamento e largura de banda.

---

6. Desvantagens

- **Sobrecarga computacional**: A construção e a atualização da árvore podem ser computacionalmente intensivas, especialmente para grandes volumes de dados.
- **Desafios com dados dinâmicos**: A adição ou remoção de dados pode exigir a reconstrução parcial ou total da árvore, o que pode ser ineficiente.
- **Dependência de funções de hash seguras**: A segurança da estrutura depende da resistência a colisões das funções de hash utilizadas. Vulnerabilidades nessas funções podem comprometer a integridade da árvore.

A seguir estão as implementações completas da estrutura de **Árvore de Merkle** em **Python** e **C**, adequadas para fins educacionais. As versões demonstram a criação da árvore e a obtenção da *Merkle Root*.

✅ Implementação em Python

Esta implementação usa `hashlib` para calcular hashes SHA-256 e constrói a árvore a partir de blocos de dados.

```
import hashlib
from typing import List

def sha256(data: str) -> str:
    return hashlib.sha256(data.encode('utf-8')).hexdigest()

class MerkleTree:
    def __init__(self, data_blocks: List[str]):
        self.leaves = [sha256(data) for data in data_blocks]
        self.root = self.build_tree(self.leaves)

    def build_tree(self, nodes: List[str]) -> str:
        if len(nodes) == 1:
            return nodes[0]
        new_level = []
        for i in range(0, len(nodes), 2):
            left = nodes[i]
            right = nodes[i+1] if i + 1 < len(nodes) else left  #
duplicar se ímpar
            combined = sha256(left + right)
            new_level.append(combined)
        return self.build_tree(new_level)

    def get_root(self) -> str:
        return self.root

# Exemplo de uso
if __name__ == "__main__":
    data = ["block1", "block2", "block3", "block4"]
    tree = MerkleTree(data)
    print("Merkle Root:", tree.get_root())
```

✅ Implementação em C

Abaixo está uma versão simplificada usando OpenSSL para cálculo SHA-256.

> Requisitos:
>
> - Compilar com: `gcc merkle.c -o merkle -lcrypto`

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
```

```c
#include <openssl/sha.h>

#define HASH_LENGTH 65  // SHA-256 hash em hex com \0

void sha256_string(const char *str, char output[HASH_LENGTH]) {
    unsigned char hash[SHA256_DIGEST_LENGTH];
    SHA256((const unsigned char *)str, strlen(str), hash);
    for (int i = 0; i < SHA256_DIGEST_LENGTH; i++) {
        sprintf(output + (i * 2), "%02x", hash[i]);
    }
    output[64] = '\0';
}

char* concat_hash(const char *a, const char *b) {
    char *concat = malloc(strlen(a) + strlen(b) + 1);
    strcpy(concat, a);
    strcat(concat, b);
    return concat;
}

char* build_merkle_tree(char **hashes, int n) {
    if (n == 1) return strdup(hashes[0]);

    int next_level = (n + 1) / 2;
    char **parent_hashes = malloc(next_level * sizeof(char*));

    for (int i = 0; i < n; i += 2) {
        char *left = hashes[i];
        char *right = (i + 1 < n) ? hashes[i + 1] : hashes[i]; //
duplicar último se necessário
        char *combined = concat_hash(left, right);
        parent_hashes[i / 2] = malloc(HASH_LENGTH);
        sha256_string(combined, parent_hashes[i / 2]);
        free(combined);
    }

    char *result = build_merkle_tree(parent_hashes, next_level);

    for (int i = 0; i < next_level; i++) {
        free(parent_hashes[i]);
    }
    free(parent_hashes);
    return result;
}

int main() {
    const char *data[] = {"block1", "block2", "block3", "block4"};
    int n = sizeof(data) / sizeof(data[0]);
    char *hashes[n];

    for (int i = 0; i < n; i++) {
        hashes[i] = malloc(HASH_LENGTH);
        sha256_string(data[i], hashes[i]);
```

```
    }

    char *root = build_merkle_tree(hashes, n);
    printf("Merkle Root: %s\n", root);

    for (int i = 0; i < n; i++) {
        free(hashes[i]);
    }
    free(root);
    return 0;
}
```

- A versão em **Python** é ótima para prototipagem e estudo rápido.
- A versão em **C** é mais adequada para aplicações com restrições de desempenho e uso em sistemas embarcados.
- Ambas demonstram os conceitos fundamentais: hashing, emparelhamento de nós, e propagação até a raiz.

A **prova de inclusão** (ou *Merkle proof*) permite verificar que um dado bloco pertence à Árvore de Merkle **sem precisar da árvore inteira**. Essa verificação é usada, por exemplo, em **SPV nodes no Bitcoin**.

A seguir, explico o conceito e trago implementações em **Python** e **C**.

🧠 Conceito de Merkle Proof

Dado um **bloco de dados**, a prova de inclusão consiste em:

- O **hash do bloco**.
- Os **hashes irmãos** (também chamados de *siblings*) em cada nível da árvore, necessários para reconstruir o caminho até a raiz.

O verificador recebe:

- O dado a ser verificado
- A raiz da árvore
- A lista de irmãos com suas posições (esquerda ou direita)

E então, reconstrói o hash até a raiz para confirmar se bate.

✅ Implementação em Python (com Merkle Proof)

Classe extendida com `get_proof` e `verify_proof`

```
import hashlib
from typing import List, Tuple
```

```python
def sha256(data: str) -> str:
    return hashlib.sha256(data.encode('utf-8')).hexdigest()

class MerkleTree:
    def __init__(self, data_blocks: List[str]):
        self.data_blocks = data_blocks
        self.levels = []
        self.build_tree()

    def build_tree(self):
        current_level = [sha256(data) for data in self.data_blocks]
        self.levels.append(current_level)

        while len(current_level) > 1:
            next_level = []
            for i in range(0, len(current_level), 2):
                left = current_level[i]
                right = current_level[i+1] if i+1 < len(current_level)
else left
                combined = sha256(left + right)
                next_level.append(combined)
            self.levels.append(next_level)
            current_level = next_level

    def get_root(self) -> str:
        return self.levels[-1][0]

    def get_proof(self, index: int) -> List[Tuple[str, str]]:
        proof = []
        for level in self.levels[:-1]:
            is_right = index % 2
            sibling_index = index - 1 if is_right else index + 1
            if sibling_index >= len(level):
                sibling_hash = level[index]  # Duplicado se não houver
par
            else:
                sibling_hash = level[sibling_index]
            proof.append((sibling_hash, "left" if is_right else
"right"))
            index = index // 2
        return proof

    @staticmethod
    def verify_proof(data: str, proof: List[Tuple[str, str]], root: str)
-> bool:
        current_hash = sha256(data)
        for sibling_hash, position in proof:
            if position == "left":
                current_hash = sha256(sibling_hash + current_hash)
            else:
                current_hash = sha256(current_hash + sibling_hash)
        return current_hash == root
```

```
# Exemplo de uso
if __name__ == "__main__":
    data = ["block1", "block2", "block3", "block4"]
    tree = MerkleTree(data)
    index = 2
    proof = tree.get_proof(index)
    root = tree.get_root()

    print("Merkle Root:", root)
    print("Proof:", proof)
    print("Verificado:", MerkleTree.verify_proof(data[index], proof,
root))
```

✅ Implementação em C (Explicação e Estratégia)

A implementação de `Merkle proof` em C é mais complexa por envolver alocação dinâmica de listas de pares (hash + posição) e concatenação segura. Aqui está o **planejamento**:

Etapas:

1. Criar uma estrutura para armazenar cada par (hash, posição).
2. Na construção da árvore, manter um vetor de vetores com os níveis.
3. Para uma posição `i`, gerar os irmãos de cada nível.
4. Na verificação, repetir o cálculo de hash do nó com os irmãos até a raiz.

Se quiser, posso construir esse código completo em C também — só confirme se deseja uma implementação com memória dinâmica ou algo mais simples.

✅ Conclusão

A Merkle Proof:

- **Reduz custos** de transmissão e verificação em redes distribuídas.
- **Garante integridade** dos dados com poucos hashes.
- **É usada amplamente** em blockchains, P2P, e sistemas de arquivos seguros.

Conclusão

As Árvores de Merkle são uma solução elegante para garantir a integridade e a eficiência na verificação de grandes conjuntos de dados em sistemas distribuídos. Sua aplicação em blockchains, sistemas de arquivos e redes P2P destaca sua versatilidade e importância na computação moderna. No entanto, é crucial considerar suas limitações e garantir o uso de funções de hash seguras para manter a confiabilidade da estrutura.
