Санкт-Петербургский политехнический университет Петра Великого
Институт машиностроения, материала и транспорта
Высшая школа автоматизации и робототехники

Курсовая работа

Дисциплина: Объектно-ориентированное программирование

Тема: Дерево Меркла

Выполнил студент гр. 3331506/20102 Дубровский Г.К.

Преподаватель Ананьевский М.С.

Санкт-Петербург
2025

Оглавление

1. Введение

Данная работа посвящена изучению и практическому применению структуры данных, известной как, **дерево Меркла** (Merkle Tree). Дерево Меркла — это криптографическая структура данных, используемая для эффективной проверки целостности и согласованности больших объемов информации.

Постановка задачи:

В современных распределенных системах (например, блокчейн) возникает необходимость проверки целостности данных без загрузки всего набора информации. Наивный подход, подразумевающий проверку каждого элемента, обладает линейной сложностью O(n). Дерево Меркла позволяет решить эту задачу с логарифмической сложностью O(log n). Целью данной работы является изучение и анализ Дерева Меркла, а также его практическое применение.

Актуальность и значимость:

Деревья Меркла являются основополагающей частью блокчейна, на котором основаны криптовалюты, такие как «Биткойн» и «Эфириум». Они позволяют быстро и эффективно проверять транзакции, даже когда обрабатывают большие объёмы данных. Это делает технологию масштабируемой и безопасной. Кроме того, деревья Меркла исследуют для использования в децентрализованных системах хранения файлов, где они могут обеспечить целостность данных и позволить эффективно извлекать файлы. Если какая-либо часть данных внутри блока изменяется, хэш этого блока тоже меняется. Это позволяет обнаружить подделку. Деревья Меркла подходят для сценариев, где ограничены хранилище или пропускная способность, так как требуется только корневой хэш для проверки.

Область применения:

Дерево Меркла широко применяется в областях для проверки целостности данных, таких как:

- **В блокчейнах** деревья Меркла используются для обеспечения целостности данных транзакций, хранящихся в каждом блоке. Корневой хеш дерева Меркла включается в заголовок блока, что позволяет эффективно проверять содержание всего блока.
- **В безопасных коммуникациях** деревья Меркла помогают обнаруживать подделанные данные во время передачи. Например, при передаче большого файла по ненадёжной сети файл делится на части, для каждой из которых создаётся дерево Меркла. На стороне получателя проверяется целостность каждой части, и если какая-то часть изменена, нужно переслать только её, а не весь файл заново.
- **В цифровых подписях** деревья Меркла улучшают эффективность проверки подлинности и целостности цифровых документов за счёт сокращения времени проверки и вычислительной нагрузки. В структуре дерева хранятся открытые ключи участников, что делает процесс верификации более эффективным.
- **Обеспечение целостности данных**. Создавая дерево Меркла из блоков или секторов файла, можно обнаружить повреждения или изменения хранимых данных.
- **Восстановление данных**. Сравнивая пересчитанные хеши листовых узлов с исходными, хранящимися в корневом хеше, можно обнаружить любые несоответствия.
- **Распределённые базы данных**. Например, Apache Cassandra использует деревья Меркла для обеспечения согласованности и целостности данных на нескольких узлах.
- **Сжатие данных**. Алгоритмы сжатия данных, такие как RLE (кодирование длины серии), используют деревья Меркла для проверки целостности и подлинности данных при сжатии.

2. Дерево Меркла

2.1 Теоретические сведения

Дерево Меркла — это бинарное дерево, где: листья содержат хеши отдельных блоков данных, внутренние узлы — хеши от конкатенации хешей дочерних узлов, корень (Merkle Root) — итоговый хеш, представляющий все данные.

Заполнение значений в узлах дерева идёт снизу вверх. Сперва к каждому блоку данных применяется хеширование. Полученные значения записываются в листья дерева. Блоки, находящиеся уровнем выше, заполняются как хеши от суммы своих веток. Эта операция повторяется, пока не будет получено верхнее значение TopHash. Если количество блоков на каком-то уровне дерева оказывается нечётным, то один блок дублируется или переносится без изменений на следующий уровень.

2.2 Алгоритм построения:

2.2.1 Хеширование каждого элемента данных (листья). В основном используются криптографические хеш-функции (SHA-256), рис. 1:

```
18        MerkleNode(const std::string& data)
19            : left(nullptr), right(nullptr), hash(sha256(data)) {}
20
21        //Input node
22        MerkleNode(MerkleNode* left, MerkleNode* right)
23            : left(left), right(right) {
24            std::string combinedHash = left->hash + (right ? right->hash : left->hash);
25            hash = sha256(combinedHash);
26        }
27
28        ~MerkleNode() {
29            delete left;
30            delete right;
31        }
32
33    private:
34        // SHA-256
35        static std::string sha256(const std::string& input) {
36            unsigned char hash[SHA256_DIGEST_LENGTH];
37            SHA256_CTX sha256;
38            SHA256_Init(&sha256);
39            SHA256_Update(&sha256, input.c_str(), input.size());
40            SHA256_Final(hash, &sha256);
41
42            char outputBuffer[SHA256_DIGEST_LENGTH * 2 + 1];
43            for (int i = 0; i < SHA256_DIGEST_LENGTH; i++) {
44                sprintf(outputBuffer + (i * 2), "%02x", hash[i]);
45            }
46            outputBuffer[SHA256_DIGEST_LENGTH * 2] = '\0';
```

Рисунок 1 – хэширование данных

2.2.2 Рекурсивное объединение хешей пар узлов до получения корня, рис. 2:

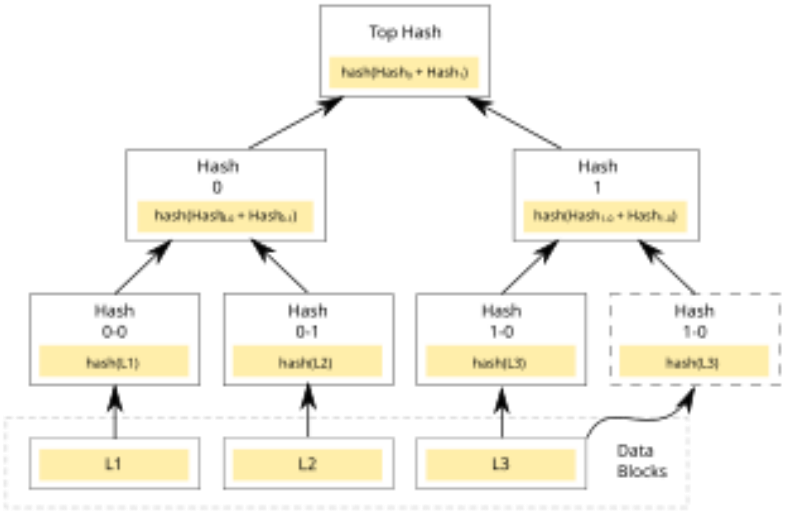

Рисунок 2 – Дерево Меркла из трех элементов

2.2.3 Для проверки наличия элемента в дереве достаточно:

Получить путь (Merkle Path) от элемента до корня, пересчитать хеши на пути и сравнить с Merkle Root.

3 Заключение

В ходе выполнения курсовой работы была изучена структура данных «Дерево Меркла», его алгоритмы построения и применения в современных компьютерных системах.Дерево Меркла обеспечивает логарифмическую сложность (O(log n)) при проверке целостности данных, что делает его значительно быстрее наивного подхода (O(n)).Позволяет компактно хранить и передавать Merkle Proof (путь верификации) вместо полного набора данных.Криптографические свойства SHA-256 гарантируют, что любое изменение данных изменяет хеши всех родительских узлов, включая корень. Широко применяется в блокчейне (Bitcoin, Ethereum) для защиты от подделки транзакций и в распределённых базах данных (IPFS, Git), кибербезопасности и аудитинге.

4 Литература

1. Меркл Р. "A Digital Signature Based on a Conventional Encryption Function", 1989.
2. Накамото С. "Bitcoin: A Peer-to-Peer Electronic Cash System", 2008.
3. Кнут Д.Э. "Искусство программирования. Том 3", 2019.

5 Приложение

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <openssl/sha.h>

using namespace std;


// Merkle tree
class MerkleNode {
```

```
public:
    MerkleNode* left;
    MerkleNode* right;
    string hash;

    //Input leaf
    MerkleNode(const std::string& data)
        : left(nullptr), right(nullptr), hash(sha256(data)) {}

    //Input node
    MerkleNode(MerkleNode* left, MerkleNode* right)
        : left(left), right(right) {
        std::string combinedHash = left->hash + (right ? right->hash : left->hash);
        hash = sha256(combinedHash);
    }

    ~MerkleNode() {
        delete left;
        delete right;
    }

private:
    // SHA-256
    static std::string sha256(const std::string& input) {
        unsigned char hash[SHA256_DIGEST_LENGTH];
        SHA256_CTX sha256;
        SHA256_Init(&sha256);
        SHA256_Update(&sha256, input.c_str(), input.size());
        SHA256_Final(hash, &sha256);

        char outputBuffer[SHA256_DIGEST_LENGTH * 2 + 1];
        for (int i = 0; i < SHA256_DIGEST_LENGTH; i++) {
            sprintf(outputBuffer + (i * 2), "%02x", hash[i]);
        }
        outputBuffer[SHA256_DIGEST_LENGTH * 2] = '\0';
        return std::string(outputBuffer);
    }
};
```

```
// Build tree
class MerkleTree {
public:
    MerkleNode* root;
    MerkleTree(const vector<string>& data) {
        if (data.empty()) {
            root = nullptr;
            return;
        }

       vector<MerkleNode*> nodes;
        for (const auto& item : data) {
            nodes.push_back(new MerkleNode(item));
        }

        root = buildTree(nodes);
    }

    ~MerkleTree() {
        delete root;
    }

    // Output Tree
    void printTree() const {
        if (!root) {
           cout << "Tree is empty" << endl;
            return;
        }

        printNode(root, 0);
    }


     string getRootHash() const {
     if (return == nullptr)
     {
         return "empty";
     }
     if (return != nullptr)
     {
```

```
        return hash;
    }
    }

private:
    MerkleNode* buildTree(vector<MerkleNode*>& nodes) {
        if (nodes.size() == 1) {
            return nodes[0];
        }
        vector<MerkleNode*> newLevel;
        for (size_t i = 0; i < nodes.size(); i += 2) {
            MerkleNode* left = nodes[i];
            MerkleNode* right = (i + 1 < nodes.size()) ? nodes[i + 1] : nullptr;

            MerkleNode* parent = new MerkleNode(left, right);
            newLevel.push_back(parent);
        }

        return buildTree(newLevel);
    }

    // Output
    void printNode(MerkleNode* node, int depth) const {
        if (!node) return;

        string indent(depth * 2, ' ');
        cout << indent << "Level " << depth << ": " << node->hash << endl;

        printNode(node->left, depth + 1);
        printNode(node->right, depth + 1);
    }
};

int main() {
    // Example of 5 elements
    vector<string> data = {
        "1 string",
        "2 string",
        "3 string",
        "4 string",
```

```
        "5 string"
    };

    MerkleTree tree(data);

    cout << "Merkle Tree Structure:" << endl;
    tree.printTree();

    cout << "\nRoot Hash: " << tree.getRootHash() << endl;

    return 0;
}
```