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. 3331506/20101 Лашицкий В.А.

Преподаватель Ананьевский М. С.

Санкт-Петербург
2025

Оглавление

1. Введение

Данная работа посвящена изучению и практическому применению алгоритму под названием дерево Меркла. Этот алгоритм позволяет получить один хеш для множества фрагментов данных. Метод используют для определения целостности файлов и верификации информации.

Актуальность и значимость:

Централизованная система предоставляет данные из одного источника, на который полагаются все пользователи. Последний гарантирует корректность полученной информации.

Блокчейн является распределенной базой данных. Информация в ней хранится на множестве независимых узлов (нод). Нода не может принять сообщения от других участников без их проверки. Узлу необходимо определить, содержит ли блок корректные транзакции.

Для снижения вычислительных затрат можно использовать деревья Меркла. Они позволяют уменьшить объем загружаемых данных и оптимизировать их проверку благодаря хешированию.

Область применения:

Метод используется в сетях биткоина, Ethereum и других криптовалютах. С его помощью получают строку данных, которая верифицирует группу транзакций. Алгоритм также применяется в файловых системах и базах данных. С помощью деревьев Меркла информацию проверяют на наличие ошибок и проводят синхронизацию.

2. Дерево Меркла

2.1 Теоретические сведения

Построение дерева Меркла выполняется снизу вверх. Значения в листовых вершинах получают хешированием фрагментов данных. Узлы следующего уровня содержат хеш от суммы двух дочерних. Для объединения данных применяется конкатенация. Операция повторяется для узлов следующих уровней до получения одного хеша. Если число элементов нечетное, один из них дублируется или переносится на следующий уровень в неизменном виде.

При построении дерева получают единый хеш, который называется корнем Меркла (mercle root). Последний представляет все фрагменты данных. Таким образом, дерево Меркла является однонаправленной хеш-функцией.

Алгоритм позволяет построить бинарную структуру, в которой узловые значения формируются из двух строк. Последнее свойство

предоставляет возможность верифицировать большой объем данных без пересчета хешей для всех фрагментов. Вычислительные затраты при определении подлинности одного элемента в этом случае намного ниже. Для проверки корректности массива и его целостности корневой хеш необходимо сравнить с эталонным значением. Фрагментами могут являться данные о транзакциях или части файлов.

2.2 Построение

Заполнение значений в узлах дерева идёт снизу вверх. Сперва к каждому блоку данных применяется хеширование $\mathrm{Hash}_{00} = \mathrm{hash}(L_1)$, $\mathrm{Hash}_{01} = \mathrm{hash}(L_2)$ и так далее. Полученные значения записываются в листья дерева. Блоки, находящиеся уровнем выше, заполняются как хеши от суммы своих детей, $\mathrm{Hash}_0 = \mathrm{hash}(\mathrm{Hash}_{00} + \mathrm{Hash}_{01})$, где $+$ обычно означает конкатенацию. Эта операция повторяется, пока не будет получено верхнее значение — $\mathrm{TopHash}$

В биткойне в качестве хеш-функции используется двойное SHA-256, то есть $\mathrm{hash}(x) = \mathrm{SHA256}(\mathrm{SHA256}(x))$. Впрочем, хеш-функция может быть любой, например Tiger Tree Hash (TTH), используемый в файлообменных P2P-сетях, является деревом Меркла с хеш-функцией Tiger .

Если количество блоков на каком-то уровне дерева оказывается нечётным, то один блок дублируется или переносится без изменений на следующий уровень, как это происходит при вычислении Tiger Tree Hash .

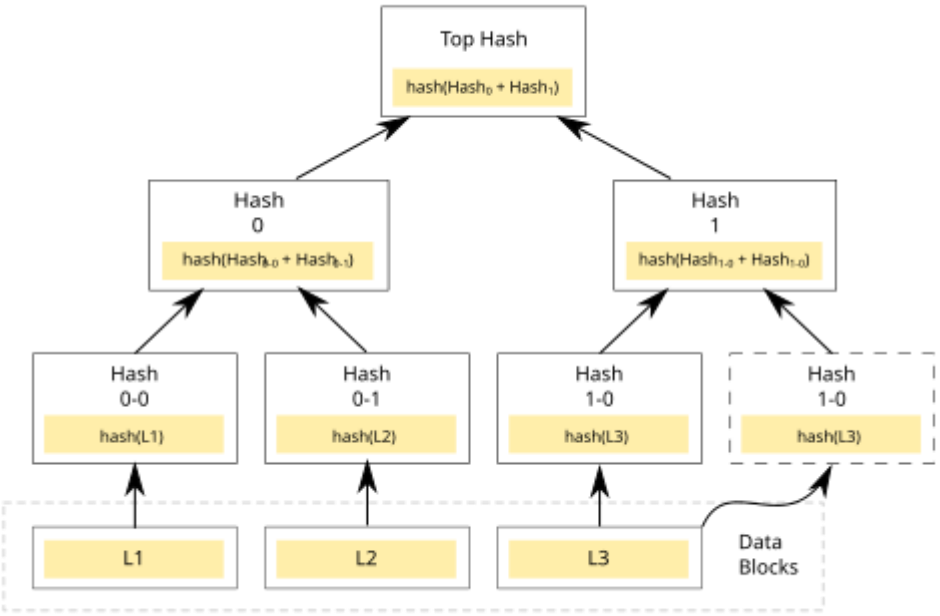

Рисунок 1 – Пример дерева Меркла

2.3 Эффективность

Хеш-деревья имеют преимущество перед хеш-цепочками или хеш-функциями. При использовании хеш-деревьев гораздо менее затратным является доказательство принадлежности определённого блока данных к множеству. Поскольку различными блоками часто являются независимые данные, такие как транзакции или части файлов, то нас интересует возможность проверить только один блок, не пересчитывая хеши для остальных узлов дерева. Пусть интересующий нас блок — это L_3 (см.рис.2). Тогда доказательством его существования и валидности будут корневой хеш, а также верхние хеши других веток

($\text{auth}_{L_4} = \text{Hash}_{11}$ и $\text{auth}_{L_2} = \text{Hash}_0$). В данном случае проверка не пройдена, если $\text{TopHash} \neq \text{hash}\left(\text{Hash}_0 + \text{hash}(\text{hash}(L_3) + \text{Hash}_{11}\right)$.

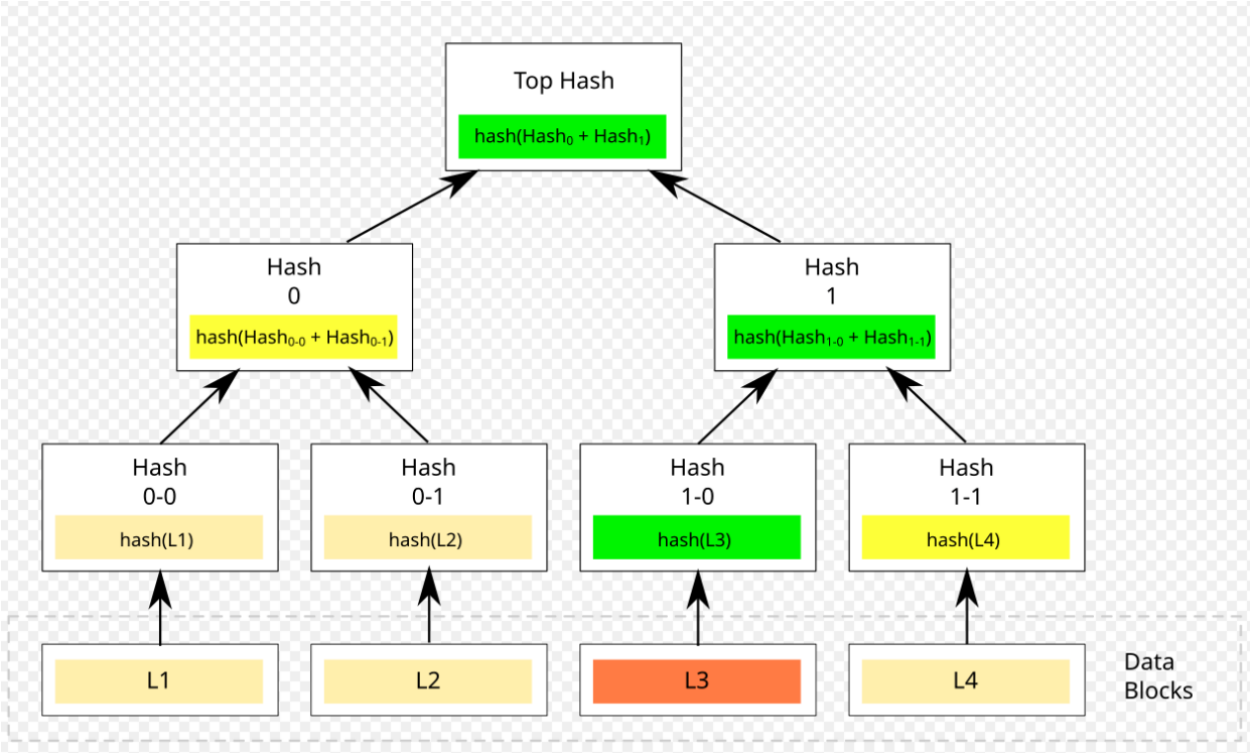

Рисунок 2 - Доказательство существования в хеш-дереве

В общем случае можно записать

$$\text{signature}(L) = (L \mid \text{auth}_{L_1}, \ldots, \text{auth}_{L_{K-1}}),$$

а проверку осуществить как $\text{TopHash} = \text{Hash}_K$, где

$$\text{Hash}_k = \begin{cases} \text{hash}(L), & \text{если } k = 1, \\ \text{hash}(\text{Hash}_{k-1} + \text{auth}_{L_{k-1}}), & \text{если } 2 \leqslant k \leqslant K. \end{cases}$$

Набор блоков $\{\text{auth}_{L_1}, \ldots, \text{auth}_{L_{K-1}}\}$ называется *аутентификационный путь* или *путь Меркла*.

Видно, что проверку выше можно выполнить за $O(K) = O(\log_2 N)$ действий, где K — это высота дерева или длина аутентификационного пути, а N — это количество блоков данных. Такая же проверка в случае хеш-цепочки имела бы сложность O(N).

Таблица ниже демонстрирует, что даже при значительном количестве транзакций в блоке о сложности вычислений можно не беспокоиться

Количество транзакций	Приблизительный размер блока	Размер пути (в хешах)	Размер пути (в байтах)
16	4 килобайта	4	128
512	128 килобайт	9	288
2048	512 килобайт	11	352
65536	16 мегабайт	16	512

2.4 Упрощённая проверка оплаты

Заголовок блока Биткойна хранит значение корня Меркла- то есть хеш всех транзакций в блоке. Это создает определённые преимущества и позволяет снизить общую нагрузку на сеть.

После накопления достаточного числа блоков появляется возможность удалять старые транзакции для экономии места. При этом заголовок блока остаётся неизменным, так как в нём хранится только корня Меркла. Блок без транзакций занимает 80 Б, или 4,2 МБ в год (при генерации блока каждые 10 минут).

Становится возможной упрощённая проверка оплаты (*simplified payment verification*). SPV-узел загружает не весь блок, а только его заголовок. Для интересующей его транзакции он запрашивает также её аутентификационный путь. Таким образом он загружает всего несколько килобайт, тогда как полный размер блока может быть больше 10 мегабайт (см. таблицу). Использование этого метода, однако, требует, чтобы пользователь доверял узлу сети, у которого будет запрашивать заголовки блоков. Один из способов избежать атаки, то есть подмены узла недобросовестной стороной, — рассылать оповещения по всей сети при обнаружении ошибки в блоке, вынуждая пользователя загружать блок целиком.

На упрощённой проверке оплаты основана работа так называемых «тонких» биткойн-клиентов.

3. Пример

```cpp
#include <iostream>
#include <vector>
#include <string>
#include <sstream>

std::string simpleHash(const std::string& data) {
    unsigned int sum = 0;
    for (char c : data) {
        sum += static_cast<unsigned int>(c);
    }
    return std::to_string(sum);
}

class MerkleNode {
public:
    std::string hash;
    MerkleNode* left;
    MerkleNode* right;

    MerkleNode(const std::string& data) : left(nullptr), right(nullptr) {
        hash = simpleHash(data);
    }

    MerkleNode(MerkleNode* left, MerkleNode* right) : left(left), right(right) {
        hash = simpleHash(left->hash + right->hash);
    }
};

MerkleNode* buildMerkleTree(std::vector<std::string> leaves) {
    if (leaves.size() % 2 != 0) {
        leaves.push_back(leaves.back());
    }

    std::vector<MerkleNode*> nodes;
    for (const auto& leaf : leaves) {
        nodes.push_back(new MerkleNode(leaf));
    }

    while (nodes.size() > 1) {
        if (nodes.size() % 2 != 0) {
            nodes.push_back(nodes.back());
        }
        std::vector<MerkleNode*> parents;
        for (size_t i = 0; i < nodes.size(); i += 2) {
            parents.push_back(new MerkleNode(nodes[i], nodes[i + 1]));
        }
        nodes = parents;
    }

    return nodes.front();
}
```

```cpp
int main() {
    std::vector<std::string> data = {
        "transaction1",
        "transaction2",
        "transaction3",
        "transaction4",
        "transaction5",
        "transaction6"
    };

    MerkleNode* root = buildMerkleTree(data);
    std::cout << "Merkle Root (simple hash): " << root->hash << std::endl;

    return 0;
}
```