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 Корхут М. А.

Преподаватель Ананьевский М. С.

Санкт-Петербург
2025

Оглавление

Введение

Дерево Меркла (Merkle Tree) — это иерархическая структура данных, обеспечивающая эффективную проверку целостности информации в распределённых системах. Его основная идея заключается в построении криптографических хеш-сумм для блоков данных, которые объединяются в древовидную структуру, где каждый родительский узел представляет собой хеш своих дочерних узлов. Это позволяет быстро обнаруживать изменения в данных, даже если их объём крайне велик. Реализация дерева Меркла на языке C представляет собой важную задачу, учитывая широкое применение этого языка в системах, требующих высокой производительности и низкоуровневого контроля, таких как блокчейн-платформы, распределённые базы данных и защищённые сетевые протоколы.

Предмет работы — разработка и реализация дерева Меркла на языке C. Это включает проектирование структуры данных, алгоритмов построения дерева, вычисления хешей, а также методов проверки наличия и целостности элементов.

Постановка задачи заключается в создании:

1. Эффективной структуры данных для хранения узлов дерева.
2. Функций для добавления данных, построения и обновления дерева.
3. Механизма верификации элементов без необходимости полного пересчёта дерева.
4. Самодокументирующегося кода, соответствующего стандартам оформления.

Значимость работы обусловлена растущим спросом на надёжные методы проверки данных в условиях увеличения объёмов информации и кибератак. Деревья Меркла лежат в основе многих современных технологий, включая блокчейн (например, Bitcoin, Ethereum), системы контроля версий (Git) и протоколы распределённого хранения (IPFS). Реализация на языке C обеспечивает переносимость, высокую

производительность и возможность интеграции в ресурсоограниченные среды.

Область применения охватывает:

1. Криптографические системы и блокчейн-технологии.
2. Распределённые и одноранговые сети.
3. Механизмы резервного копирования и восстановления данных.
4. Верификацию обновлений ПО в embedded-системах.

В данной работе представлена реализация, соответствующая требованиям к надёжности и читаемости кода, что делает её пригодной как для учебных целей, так и для интеграции в реальные проекты.

Теоретические сведения

1. Определение и назначение

Дерево Меркла (Merkle Tree) — это бинарная древовидная структура данных, используемая для эффективной проверки целостности информации.

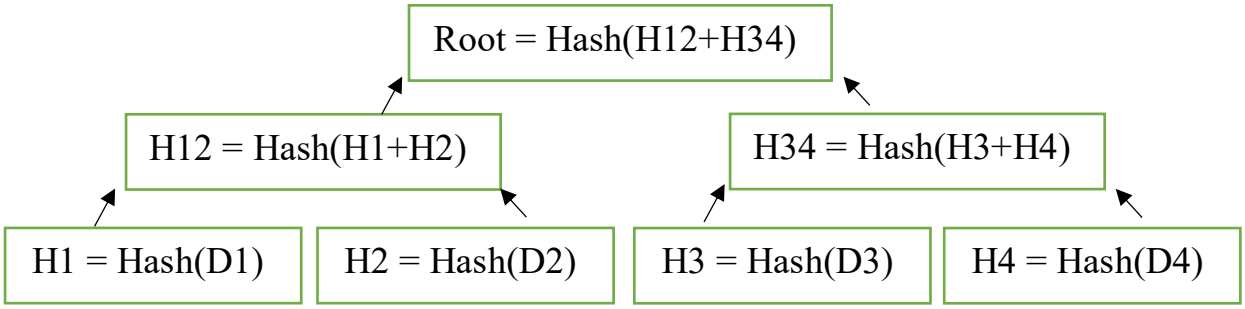
Основная задача: обеспечить криптографическое подтверждение того, что данные не были изменены.

2. Структура дерева

Дерево строится по следующим правилам:

- **Листья (узлы 0-го уровня)**: содержат хеш-суммы исходных блоков данных (например, транзакций).
- **Внутренние узлы (узлы n-го уровня)**: формируются путём конкатенации и хеширования значений двух дочерних узлов.
- **Корень (Merkle Root)**: хеш верхнего узла, представляющий сводку всех данных.

Пример структуры для 4 блоков:



Где D1, D2, D3, D4 — исходные данные, Hash — хеш-функция.

1. **Основные свойства**:
    1. **Детерминированность**: Одинаковые данные → одинаковый Merkle Root.
    2. **Чувствительность**: Любое изменение данных меняет хеши на пути к корню.
    3. **Эффективность проверки**: Подтверждение целостности элемента требует O(log$N$) шагов (Merkle Proof).
    4. **Высокая стоимость обновления дерева**: Необходимость хранения структуры.

Реализация дерева Меркла

1. Структура узла дерева (MerkleNode)

Узел дерева представлен структурой MerkleNode, содержащей:

1. hash: Результат хеширования данных в формате HEX (например, a3f4...e1b2).
2. left и right: Ссылки на дочерние узлы. Для листьев эти указатели равны NULL.

2. Ключевые функции

2.1. Вычисление хеша (compute_hash)

Назначение: Генерирует SHA-256 хеш для входных данных.

Особенности:

1. Использует библиотеку OpenSSL.
2. Результат сохраняется в виде HEX-строки (например, 9f86d081...).
3. Буфер output имеет фиксированный размер SHA256_HEX_LENGTH (65 байт).

2.2. Создание листа (create_leaf)

Назначение: Инициализирует листовой узел.

Алгоритм:

1. Выделяет память под узел.
2. Вычисляет хеш переданных данных.
3. Устанавливает указатели left и right в NULL.

Обработка ошибок: Возвращает NULL при неудачном выделении памяти.

2.3. Построение родительского узла (build_parent)

Назначение: Создает внутренний узел, объединяя хеши потомков.

Особенности:

1. Если правый потомок отсутствует (нечётное число узлов), дублирует левый хеш.

2. Формирует комбинированную строку из хешей потомков и вычисляет новый хеш.

2.4. Рекурсивное построение слоёв (build_merkle_layer)

Назначение: Строит дерево снизу вверх.

Алгоритм:

1. На каждом уровне узлы объединяются попарно.
2. При нечётном количестве последний узел дублируется.
3. Процесс повторяется до получения корневого узла.

Обработка ошибок: При ошибке выделения памяти освобождает временные ресурсы.

2.5. Публичный интерфейс (build_merkle_tree)

Назначение: Запускает построение дерева из массива данных.

Алгоритм:

1. Создает массив листьев.
2. Вызывает build_merkle_layer для генерации корня.

Обработка ошибок: При сбое создания листьев освобождает ранее выделенную память.

2.6. Уничтожение дерева (destroy_merkle_tree)

Назначение: Рекурсивно освобождает память, занятую деревом.

Использует пост-обход (левый → правый → корень).

3. Пример использования

В функции main продемонстрировано:

1. Создание дерева из данных: {"Alice", "Bob", "Charlie", "Diana"}.
2. Вывод корневого хеша.
3. Корректное освобождение памяти.

Заключение

В ходе выполнения курсовой работы была исследована структура данных «дерево Меркла» и реализована её версия на языке программирования C. Работа подтвердила, что дерево Меркла является эффективным инструментом для обеспечения целостности данных в распределённых системах. Разработанный код соответствует требованиям надежности, читаемости и может служить основой для более сложных криптографических решений.

Список литературы

1. Merkle R. C. A Digital Signature Based on a Conventional Encryption Function // Advances in Cryptology — CRYPTO ’87. — Springer, 1988. — P. 369–378.

2. Cormen T. H., Leiserson C. E., Rivest R. L., Stein C. Introduction to Algorithms. — 3rd ed. — MIT Press, 2009. — 1292 p.

3. Knuth D. E. The Art of Computer Programming. Vol. 3: Sorting and Searching. — 2nd ed. — Addison-Wesley, 1998. — 800 p.

Приложение

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <openssl/sha.h>

#define SHA256_HEX_LENGTH (SHA256_DIGEST_LENGTH * 2 + 1)

// Узел дерева Меркла с хеш-суммой и ссылками на потомков
typedef struct MerkleNode {
    char hash[SHA256_HEX_LENGTH];
    struct MerkleNode* left;
    struct MerkleNode* right;
} MerkleNode;

// Прототипы внутренних функций
static void compute_hash(const char* data, char* output);
static MerkleNode* create_leaf(const char* data);
static MerkleNode* build_parent(MerkleNode* left, MerkleNode* right);
static MerkleNode* build_merkle_layer(MerkleNode** nodes, int count);

// Основной интерфейс
MerkleNode* build_merkle_tree(const char** data, int count);
void destroy_merkle_tree(MerkleNode* root);
const char* get_merkle_root(const MerkleNode* root);

// Вычисляет SHA-256 и сохраняет в виде HEX-строки
static void compute_hash(const char* data, char* output) {
    unsigned char raw[SHA256_DIGEST_LENGTH];
    SHA256((const unsigned char*)data, strlen(data), raw);

    for (int i = 0; i < SHA256_DIGEST_LENGTH; i++) {
        sprintf(output + (i * 2), "%02x", raw[i]);
    }
    output[SHA256_HEX_LENGTH - 1] = '\0';
}

// Создает конечный узел с хешем данных
static MerkleNode* create_leaf(const char* data) {
    MerkleNode* node = (MerkleNode*)malloc(sizeof(MerkleNode));
    if (!node) return NULL;

    compute_hash(data, node->hash);
    node->left = node->right = NULL;
    return node;
}

// Создает родительский узел из двух дочерних (дублирует хеш при нечетном числе)
static MerkleNode* build_parent(MerkleNode* left, MerkleNode* right) {
    MerkleNode* parent = (MerkleNode*)malloc(sizeof(MerkleNode));
    if (!parent) return NULL;

    char combined[SHA256_HEX_LENGTH * 2 + 1] = {0};
    const char* right_hash = right ? right->hash : left->hash;

    snprintf(combined, sizeof(combined), "%s%s", left->hash, right_hash);
    compute_hash(combined, parent->hash);

    parent->left = left;
    parent->right = right;
    return parent;
}

// Рекурсивно строит слои дерева до корня
static MerkleNode* build_merkle_layer(MerkleNode** nodes, int count) {
    if (count == 1) return nodes[0];
```

```
    int new_count = (count + 1) / 2;
    MerkleNode** parents = (MerkleNode**)calloc(new_count, sizeof(MerkleNode*));
    if (!parents) return NULL;

    for (int i = 0; i < new_count; i++) {
        int left_idx = 2 * i;
        int right_idx = (left_idx + 1 < count) ? left_idx + 1 : left_idx;

        parents[i] = build_parent(nodes[left_idx], nodes[right_idx]);
        if (!parents[i]) {
            for (int j = 0; j < i; j++) free(parents[j]);
            free(parents);
            return NULL;
        }
    }

    MerkleNode* root = build_merkle_layer(parents, new_count);
    free(parents);
    return root;
}

// Публичный интерфейс: строит дерево из массива строк
MerkleNode* build_merkle_tree(const char** data, int count) {
    if (count == 0) return NULL;

    MerkleNode** leaves = (MerkleNode**)calloc(count, sizeof(MerkleNode*));
    if (!leaves) return NULL;

    for (int i = 0; i < count; i++) {
        leaves[i] = create_leaf(data[i]);
        if (!leaves[i]) {
            for (int j = 0; j < i; j++) destroy_merkle_tree(leaves[j]);
            free(leaves);
            return NULL;
        }
    }

    MerkleNode* root = build_merkle_layer(leaves, count);
    free(leaves);
    return root;
}

// Рекурсивно освобождает память дерева
void destroy_merkle_tree(MerkleNode* root) {
    if (!root) return;
    destroy_merkle_tree(root->left);
    destroy_merkle_tree(root->right);
    free(root);
}

// Возвращает корневой хеш дерева
const char* get_merkle_root(const MerkleNode* root) {
    return root ? root->hash : NULL;
}

// Пример использования
int main() {
    const char* data[] = {"Alice", "Bob", "Charlie", "Diana"};
    const int count = sizeof(data)/sizeof(data[0]);

    MerkleNode* tree = build_merkle_tree(data, count);
    if (!tree) {
        fprintf(stderr, "Ошибка построения дерева\n");
        return 1;
    }
```

```
    printf("Merkle Root: %s\n", get_merkle_root(tree));
    destroy_merkle_tree(tree);
    return 0;
}
```