Исследование свойств деревьев Меркла

1. Деревья Меркла

1.1 Бинарное дерево Меркла

Псевдокод:

class BinaryMerkleTree:

properties:

data_blocks: list of blocks (numbers or another values for hashing)

blocks_hashes: list of levels (lists). Each level is list of hash values of nodes

merkle_root: root hash value of the Merkle Tree

hash_func: private property-reference to hash function from hashlib

methods:

get_merkle_root(): → merkle_root

hash(obj): → hex digest representation of the result of hashing: hash(hash_func)

if obj is None:

bytes_obj = utf-8 byte representation of ‘ ’ (empty line)

else:

bytes_obj = utf-8 byte representation of string(obj)

set_hash_func(hash_func): method for setting hash function

pair_hash(pair): → hash_func(concatenation(pair))

pair: pair (list with two values) of hashes

get_level_hashes(data_hashes): → list of [pair_hash(pair) for pair in data_hashes]

data_hashes: list of hashes from the one level

data_hashes = list of pairs of hashes [[hash_val1, hash_val2], [hash_val3], …, [hash_val{N-1}, hash_val{N}]]

find_hashes():

data_hashes = [hash_func(data) for data in data_blocks]

if len(data_hashes) % 2 == 1 (if length of data_hashes is odd):

duplicate last element (last hash value) in the end of the list

blocks_hashes.append(data_hashes) (add data_hashes to the end of blocks_hashes list)

while len(data_hashes) != 1:

if len(data_hashes) is odd:

duplicate last element of the data_hashes

data_hashes = get_level_hashes(data_hashes)

blocks_hashes.append(data_hashes)

merkle_root = the last element of the blocks_hashes

bin_tree_hashes_list():
    hashes_list = [] (empty list)
    for block in reversed blocks_hashes:
        concatenate block to the hashes_list
    set **hashes_list** property
    hashes_list: 1-d list of hashes from top of the tree to the leafs

add_block(block):
    append new block to the end of the data_blocks

update_tree(): (not optimal way: updating all tree)
    delete all elements from the blocks_hashes list
    find_hashes()
    bin_tree_hashes_list()

update_tree_optim(): (optimal updating way: update only new blocks subtree)
    new_blocks_num = number of new blocks (len(data_blocks) - len(blocks_hashes[0]))
    new_data_blocks = slice of the last blocks from data_blocks list
    for i in 0, …, number of levels in the tree:
        if i == 0:
            new_blocks_hashes = list of hashes of blocks from new_data_blocks
        else:
            new_blocks_hashes = get_level_hashes(new_blocks_hashes)
        concatenate level of i index from blocks_hashes with new_blocks_hashes

get_blocks_hashes(): → blocks_hashes

get_hashes_list(): → hashes_list

tree_height(): → len(blocks_hashes)

generate_proof(arg): → proof list
        if arg is int or float:
            arg_hash = hash(arg)
        else if arg is string: #(if arg is hash value)
            arg_hash = arg

        proof = []
        if arg_hash not in blocks_hashes[0]: (not in the list of leafs)
            raise ValueError('Block isn't list of leafs')
        index = index of arg_hash in the list of leafs (blocks_hashes[0])

```
        for level in blocks_hashes:
            if index % 2 == 1: (if index of the current node is odd, node is right)
                sibling_hash_index = index - 1
                is_left = False (node flag)
            else:
                sibling_hash_index = index + 1
                is_left = True

            if sibling_hash_index < len(level):
                sibling_hash = level[sibling_hash_index]
                proof append (sibling_hash, is_left)

            index = index // 2
        return proof
```

1.2 Разрежённое дерево Меркла

class SparseMerkleTree:

properties:

key_len: length of the key (the power of 2, height of the tree)

hash_func: hash function

default_leaf_value: default value of the unfilled nodes (empty string)

blocks_hashes: empty list, future list of levels with hashes

default_hash_values: list of default hash values on each level of the tree (before any element is added)

values: list of values (by default all values is None), that will be added (2 ** key_len values).

methods:

hash(obj): → hash value of the object (implementation is the same as in Binary Merkle Tree class)

get_default_hash_values(): → default_hash_values list

```
    default_hash_values = []
    for i in 0, …, key_len:
        if i == 0:
            def_value = hash(default_leaf_value)
        else:
            def_value = hash(concatenation of default_hash_values[i - 1])
        default_hash_values append def_value
    return default_hash_values
```

initial_tree(): **initialization of the unfilled tree**

```
    level_nodes_num = 2 ** key_len
    for i in 0, …, key_len:
```

blocks_hashes append default_hash_values[i] * level_nodes_num
level_nodes_num = level_nodes_num // 2

add_value(index, value):
if index is int and index in [0, 2 ** key_len - 1]:
values[index] = value
update(index, value)

generate_proof() (the same as in the Binary Merkle Tree)

1.3 Индексированное дерево Меркла (Indexed Merkle Tree)

class Leaf: (simple implementation of the leaf)
properties:
value: value, which should be stored
nextidx: next index of the leaf with greater value
nextval: next greater value with nextidx index
methods:
get_concat_data(): → concatenation of value, nextidx, nextval converted to the string type

class IndexedMerkleTree:
properties:
tree_height: height of the tree
leafs_num: number of the leafs (nodes from the first level of the tree)
leafs: list of all leafs
hash_func: hash function
blocks_hashes: list of all levels with hashes values
values: [0], list of values, which are stored in leafs
add_value_index: 1, index of the element, which will be added next
max_val_index: 0, index of the leaf with maximum value
methods:
hash(obj) (the same as for BinaryMerkleTree)

initial_leafs(): → list of the initialized leafs
[Leaf(0, 0, 0) for i in 0, ..., leafs_num - 1]

initial_tree(): #initialization of the all tree with hash-values
for i in 0,...,tree_height + 1:
if i == 0:
hashes = [hash(leaf.get_concat_data()) for leaf in leafs]
#list of hashes of leafs data

```
                else:
                        prev_hashes = [list of paired hashes from the previous
level]
                        hashes = [hash(pair[0] + pair[1]) for pair in prev_hashes]

                        blocks_hashes.append(hashes)
```

max_smaller_val(value): (maximum value which is less than value)

```
        smaller_vals = []
        for val in values:
                if val <= value:
                        smaller_vals.append(val)
        return max(smaller_vals)
```

add_value(value):

```
        if len(values) < leafs_num:
                max_smaller_value = max_smaller_val(value)
                max_smaller_leaf_idx = values.index(max_smaller_val)
                max_smaller_leaf = leafs[max_smaller_leaf_idx]
                if value > values[max_val_indx]:
                        nextidx, nextval = 0,0
                        max_val_indx = len(values)
                else:
                        nextidx = max_smaller_leaf.nextidx
                        nextval = max_smaller_leaf.nextval

                leafs[max_smaller_leaf_idx].nextidx = len(values)
                leafs[max_smaller_leaf_idx].nextval = value

                leaf = Leaf(value, nextidx, nextval) #creating leaf instance
                values.append(value)
                leafs[add_value_index] = leaf
                add_value_index += 1
        else:
                raise exception IndexError('All the leafs on the tree are filled!')
```

update(): the same as initial tree except the last line

blocks_hashes[i] = hashes (the last line)

generate_proof(): (the same as before)

verify_inc_proof(proof, root_hash, target_data, hash_func = sha256)

proof: list of tuples from generate_proof method

root_hash: known root hash of the Merkle Tree
target_data: the original data that needs to be verified
hash_func: hash function to use (sha256 by default)
target_hash = hash_func(target_data converted to the string and utf-8 encoded)

```
for sibling_hash, is_left in proof:
    if is_left:
        target_hash = hash_func((target_hash + sibling_hash))
    else:
        target_hash = hash_func((sibling_hash + target_hash))

return target_hash == root_hash
```

2. Бенчмарки
2.1 Сложность добавления нового элемента в дерево.
Сложность добавления нового элемента в дерево состоит из сложности добавления хэш-значения элемента в список листьев и сложности пересчета хэш-значений вершин дерева (обновления состояния дерева) от листа до корня.

Сложность добавления элемента в список листьев (в конец списка): **O(1)**.

Сложность записи элемента в список листьев по индексу (используется в Sparse Merkle Tree): **O(1)**.

Сложность пересчета хэш-значений после добавления элемента в дерево:
2.1.1 Binary Merkle Tree.
В самой простой реализации список листьев очищается и происходит перерасчёт всех хэш-значений в дереве заново. На уровне листьев получается N хэшей (N – количество листьев), на уровне выше – N/2 хэшей, ещё на уровне выше – N/4 хэшей и т.д. до 1, если просуммировать количество операций то получается линейная сложность **O(N)**.
Изначально предполагается, что в списке листьев содержится четное количество элементов. После добавления одного элемента чётность меняется и поэтому хэш-значение последнего (нового добавленного) элемента дублируется. В результате этого, на каждом из уровней выше чётность количества вершин будет меняться и поэтому последние также дублируются. В конечном итоге это приводит к тому, что на уровне старого корня возникает ещё одна вершина и таким образом высота дерева увеличивается на 1.
При оптимизированной реализации необходимо найти хэш-значения только для ново добавленных блоков данных и их вершин-родителей (т.е. только для поддерева, листьями которого являются новые блоки). Получается, что на одном уровне происходит один расчет хэша + его дублирование итого одна операция на

одном уровне дерева. Поскольку кол-во уровней в дереве равно log(N), где N – количество листьев, то получаем оценку сложности равную **O(log(N))**, основание логарифма можно считать равным 2, т.к. дерево бинарное.
Итоговая сложность добавления элемента в бинарное дерево Меркла O(1) + O(log(N)) = **O(log(N))**, в оптимизированном случае.
2.1.2 Sparse Merkle Tree.
Аналогичная оценка сложности верна для Sparse Merkle Tree: дерево является бинарным и поэтому количество листьев равно $N = 2^h$, где h – высота дерева.
Отличительной особенностью Sparse Merkle Tree является то, что этот вид дерева может хранить только фиксированное число элементов (т.е. высота задается предварительно и пустые листья заполняются хэш-значениями по умолчанию). Получается, что дублирования элементов не будет, т.к. четность количества вершин на одном уровне поменяться не может. При записи элемента в список по фиксированному индексу требуется пересчитать хэш-значения, только самого элемента и его родителей по очереди до корня дерева. Поэтому получается оценка сложности **O(log(N))**.

2.2 Сложность генерации доказательства включения (MP)
Сложность этой операции состоит из поиска хэш-значения среди листьев дерева и добавления в список хэш-значений братских узлов для вершин-родителей этого листа и их положения в дереве (положение в моей реализации описывается флагом is_left). Таким образом, составляется путь от листовой вершины до корня дерева. На первом уровне дерева (список листов) сложность поиска хэша **O(N)**.
Далее на каждом уровне дерева просто добавляется в список элемент-родитель, находящийся по определенному индексу. Получение элемента по индексу имеет сложность **O(1)** и мы получаем их на каждом уровне дерева. Таким образом получаем **O(1) * log(N)**. И итоговая сложность **O(N) + O(1) * log(N) = O(N)+O(log(N))=O(N)**.

2.3 Размер MP
Размер зависит MP зависит от нескольких факторов. В первую очередь он зависит от используемой хэш-функции. В моей реализации использовалась хэш-функция SHA-256, которая сопоставляет любой строке строку размера 256 бит или 32 байта. Итак, обозначим размер числа как **hash_size**. Тогда размер док-ва можно оценить как **hash_size** * **log(N)**. Также вместе с узлами хранилась информация об их положении (левый или правый потомок) в виде переменной с типом bool. В Python с помощью функции sys.getsizeof получаем, что тип данных bool весит 28 байт. Таким образом, каждая пара в доказательстве, состоящая из хэша и флага is_left суммарно весит **28 + 32 = 60 байт**, а кол-во пар **log(N)**.
Эта оценка является приблизительной, т.к. важно также учитывать размер типов данных tuple и list в Python. Поэтому суммарный вес доказательства может быть больше. Это можно проверять делая эксперименты с кодом.

2.4 Сложность верификации MP

Сложность процедуры верификации MP состоит из последовательного вычисления хэш-значений от листа до корня и в конце сравнения полученного корневого хэш-значения с истинным корневым хэшем. Оценка сложности пропорциональна количеству уровней в дереве **O(log(N))**.

2.4 Сложность генерации доказательства невключения EP

Доказательство не включения элемента в дерево Меркла состоит только из проверки того, что хэш-значение этого элемента не входит в список листов-хэшей дерева поскольку, если этого хэша нет в этом списке, то считать хэш-значения для родительских узлов уже нет никакого смысла. Поэтому необходимо просто провести поиск хэш-значения в списке. Сложность такой операции составляет **O(N),** так как происходит поэлементное последовательное сравнение строк-хэшей в списке**.** Очевидно, что в самом худшем случае хэш-значение находится в самом конце списка. Но согласно [1] если использовать в Python другую структуру данных для хранения листьев, например, словарь, то оценка может быть улучшена до **O(1)**.

2.5 Размер EP

Размер равен размеру в байтах значения булевского типа данных (28 байт).

2.6 Верификация EP

Верифицировать доказательство невключения можно, если док-во содержит только элемент, означающий булевский тип данных (False), показывающий, что элемента нет в списке листьев. Т.е. если длина списка-док-ва меньше чем, если бы элемент был в дереве (т.е. меньше чем log(N)). Такая проверка занимает одну операцию сравнения, которую можно оценить как **O(1)**.

	Бинарное дерево Меркла	Разрежённое дерево Меркла	Индексированное дерево Меркла
Добавление нового элемента	**O(log(N))**	**O(log(N))**	**O(log(N))**
Генерация доказательства включения	**O(N)**	**O(N)**	**O(N)**
Размер док-ва	**60*log(N) байт**	**60*log(N) байт**	**60*log(N) байт**
Верификация док-ва	**O(log(N))**	**O(log(N))**	**O(log(N))**
Генерация док-ва невключения	**O(N)**	**O(N)**	**O(N)**

Размер док-ва невключения	**28 байт**	**28 байт**	**28 байт**
Верификация невключения	**O(1)**	**O(1)**	**O(1)**

Ссылки на использованные источники

1. https://wiki.python.org/moin/TimeComplexity
2. https://docs.aztec.network/aztec/concepts/storage/trees/indexed_merkle_tree
3. https://blog.ziden.io/indexed-merkle-tree-a-more-optimized-solution-for-zk-proofs-c75b0d0b1786