Тестове завдання у Distributed Lab

Голубєв Кирило Дмитрович

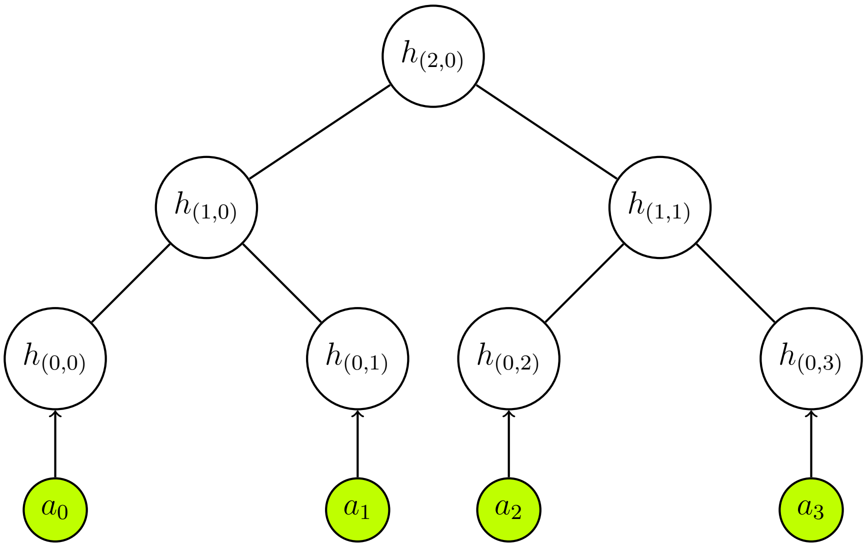
Вересень 2024

Зміст

1 Бінарне дерево Меркла.

Бінарне дерево Меркла – це граф, кожна вершина якого має по два нащадка. У кожній вершині, яка зветься вузлом, містяться свої дані, які отримуються шляхом хешування суми хешів двох вузлів-нащадків. У БДМ подається набір даних, які називають листям, а позначимо їх $a_1, a_2, \ldots, a_m$.

Так як БДМ будується знизу вверх, представимо його як список списків, що буде продемонстровано нижче. Позначимо вузли як $h_{(n,k)}$, де n – номер рядка, де знаходиться вузол (нумерація знизу вверх), а k – його порядковий номер в цьому рядку. Індексацію почнемо з 0, як це робиться в програмуванні. Тепер, коли розібралися з позначеннями, перейдемо до представлення БДМ. Якщо маємо БДМ виду:



то в программі це буде мати вигляд:

$$\Big[[h_{(0,0)}, h_{(0,1)}, h_{(0,2)}, h_{(0,3)}],\ [h_{(1,0)}, h_{(1,1)}],\ [h_{(2,0)}]\Big] \text{ – БДМ}$$

Знаходити батьків і нащадків теж можна. Якщо ми маємо вузол $h_{(n,k)}$ і нам треба знайти його нащадків, то вони мають вигляд $h_{(n-1,2k)}$, $h_{(n-1,2k+1)}$. Якщо ми маємо вузол $h_{(n,k)}$ і вимагається знайти його батька, то це робиться так:

- якщо k – парне, то батько має вигляд $h_{(n+1,\frac{k}{2})}$
- якщо k – непарне, то батько має вигляд $h_{(n+1,\frac{k-1}{2})}$

У вигляді псевдокоду маємо такі функції:

```
def neighbour_searcher(k):
     if k парне:
          return k + 1
     else:
          return k - 1
```

```
def parent_searcher(k):
    if k парне:
        return k // 2
    else:
        return (k - 1) // 2
```

Для будування БДМ також треба знати заздалегідь, скільки рівнів воно буде мати. Кількість рівнів дорівнює $\lceil \log_2(n) \rceil$ (округлення догори), де n – кількість початкових даних.

Почнемо будування БДМ з функції хешування, адже вона потрібна на кожному кроці. Функція бере на вхід елемент зі списку, перетворює його на рядок і хешує.

```
def hash(x):
     # Це знадобиться для Sparse Merkle Tree
     if x in (None,
              'dc937b59892604f5a86ac96936cd7ff09e25f18ae6b758e8014a24c7fa039e91',
              2 * 'dc937b59892604f5a86ac96936cd7ff09e25f18ae6b758e8014a24c7fa039e91'):
          return 'dc937b59892604f5a86ac96936cd7ff09e25f18ae6b758e8014a24c7fa039e91'

     bytes_obj = перетворення x на рядок
     hash_obj = хешування x
     return hash_obj
```

Далі треба задати функцію, що прийме як аргумент список початкових даних і збудує дерево. Вона перевіряє довжину списку з початковими даними на парність і робить його парним за необхідністю, хешує початкові дані і будує дерево:

```
def binary_tree_builder(x):
    tree, Level, temp = [], [], x[:]

    if довжина temp непарна:
        temp приймає свій останній елемент
    tree приймає список хешованих початкових елементів

    for i in range(1, math.ceil(math.log2(len(temp))) + 1):
        for j in range(0, len(BMT[i - 1]), 2):
            До Level додається hash(tree[i - 1][j] + tree[i - 1][j + 1])
        if довжина L непарна and i не дорівнює math.ceil(math.log2(len(temp))):
            До L додається його останній елемент
        tree приймає Level
        Level обнуляється
    return tree
```

Функцію будування БДМ з розширеним списком початкових даних було задано наступним чином: список початкових даних складається зі списком додаткових даних і передається у функцію будування БДМ:

```
def extra_binary_tree_builder(data, extra):
     temp = data + extra
     return tree_builder(temp)
```

1.1 Складність додавання нового елемента в БДМ.

Оцінимо O()-складність. Нехай n – кількість початкових елементів, k – кількість додаткових даних. Функція **extra_binary_tree_builder** повністю перебудовує дерево з новим розширеним списком початкових даних. Тому проаналізуємо складність будування БДМ зі списком із $n + k$ елементів:

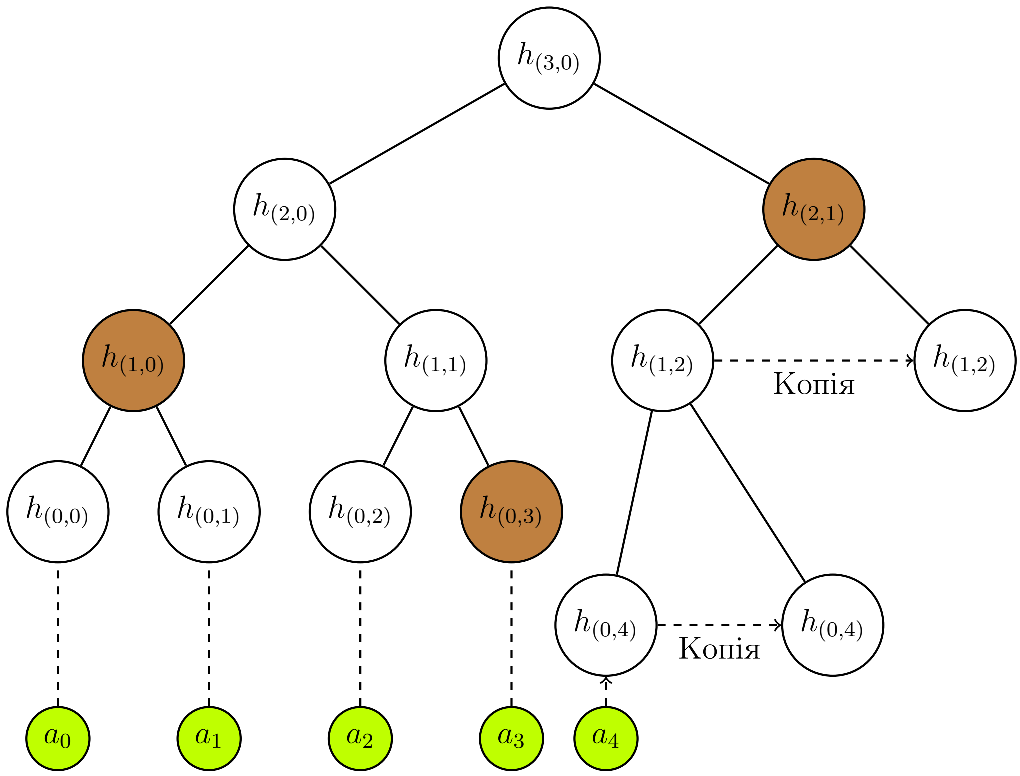
- Об'єднання списків займає $O(n + k)$, бо проходимо по всім двом спискам і формуємо новий;
- Якщо список має непарну кількість елементів, то в кінець додається останній його елемент, що має складність O(1);
- Хешування елементів на першому рівні займає $n+k$ розрахунків, на другому $\dfrac{n+k}{2}$, на третьому $\dfrac{n+k}{4}$, ...;

- Сумарно маємо $(n+k) + \dfrac{n+k}{2} + \dfrac{n+k}{4} + \ldots \approx 2 \cdot (n+k)$ – кількість операцій для будування БДМ, а тому $O(2 \cdot (n+k)) = O(n+k)$.

Об'єднуючи обидві складності, отримуємо $O(n+k) + O(n+k) = O(n+k)$.

1.2 Складність генерації доказу включення.

Сутність генерації доказу включення полягає у формуванні списку із конкретних елементів БДМ. Розглянемо дерево із п'яти початкових елементів:



Для прикладу сформуємо доведення для a_3. На першому кроці знаходимо хеш сусіднього елемента і додаємо до списку $Proof$, далі знаходимо сусіда до батька і додаємо у список і так поки не дійдемо до передостаннього рівня дерева (потрібні елементи позначені коричневим). Як результат отримуємо список:

$$Proof_{a_2} = \left[h_{(0,3)},\ h_{(1,0)},\ h_{(2,1)}\right]$$

Будемо вважати, що ВМТ – збудоване за **binary_tree_builder(leaves)** дерево. Тоді у вигляді псевдокоду це виглядає наступним чином:

```
def binary_tree_mp(x):
    proof, n, k = [], 0, 0

    for i in range(len(leaves)):
        if x == i-й елемент із leaves:
            k = i
        else:
            for i in range(1, len(BMT)):
                for j in range(len(BMT[i])):
                    if x == j-й елемент із i-го рівня BMT:
                        n, k = i, j

    proof приймає сусідній до k-го елемента на рівні n BMT
```

```
    for i in range(n + 1, len(BMT) - 1):
        proof приймає батька для елемента k з рівня n BMT
        for j in range(len(BMT[i])):
            if останній елемент proof == j-й елемент з рівня i BMT:
                k = j
    return proof
```

Для розуміння складності генерації доказу включення розглянемо детальніше алгоритм його створення. Нам треба пройтись по дереву і зібрати потрібні хеші по одному з кожного рівня, а рівнів в БДМ $\lceil \log_2(n) \rceil$, де n – кількість листів. Тоді і $O() = O(log(n))$.

1.3 Розмір доказу включення.

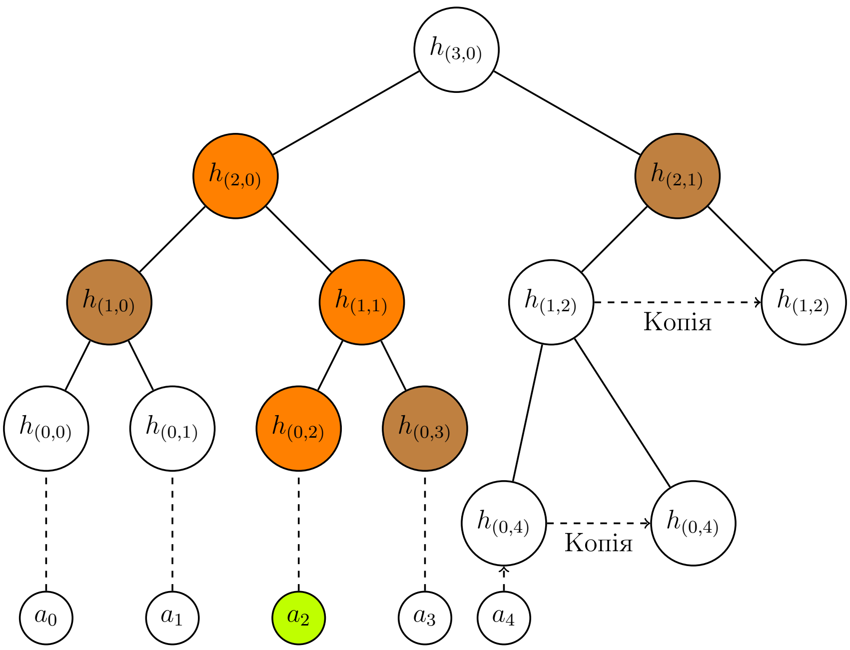
Якщо треба знайти розмір списку (чим за сутністю і є доказ включення) у байтах, переведемо кожний елемент списку у рядок і окремо підрахуємо кількість байтів, а саме:

```
# Розмір у байтах елемента
def bytesize(x):
    string = str(x).encode('utf-8')
    return довжина string

# Підрахунок розміру списку
def proofsize(x):
    size = 0
    for i in range(len(x)):
        size = size + bytesize(x[i])
    return size
```

Так як доказ формується шляхом збору одного хеша на кожному рівні БДМ, окрім останнього, то всього хешів у доведенні буде $\lceil \log_2(n) \rceil - 1$. Кожен хеш це 64 байта, тому довжина доказу у загальному випадку дорівнює $64 * \big(\lceil \log_2(n) \rceil - 1 \big)$.

1.4 Складність верифікації доказу включення.

Для верифікації доказу включення нам треба лише пройтись по дереву і виконувати хешування елементів на кожному рівні. Візуально це виглядає наступним чином (на прикладі a_3):



$$\underbrace{\mathrm{hash}(h_{(0,2)} + h_{(0,3)})}_{h_{(1,1)}} \to \underbrace{\mathrm{hash}(h_{(1,0)} + h_{(1,1)})}_{h_{(2,0)}} \to \underbrace{\mathrm{hash}(h_{(3,1)} + h_{(3,2)})}_{\mathrm{root}}$$

Порівнюємо $\mathrm{hash}(h_{(2,0)} + h_{(2,1)})$ із коренем і якщо вони співпадають, то доказ успішно підтверджено. Код для верифікації має вигляд:

```
def binary_proof_verification(x, y):
    L, n = [], 0

    for i in range(len(leaves)):
        if x == i-й елемент leaves:
            n = 0
            L приймає hash(x)

    for i in range(1, len(BMT)):
        for j in range(len(BMT[i])):
            if x == j-й елемент на i-му рівні BMT:
                n = i
                L приймає x

    if len(L) == 0:
        return 'Немає такого елемента в дереві.'

    if n == 0:
        for i in range(len(y)):
            for j in range(len(BMT[i])):
                if i-й елемент y == j-й елемент на i-му рівні BMT and j парний:
                    L приймає hash(i-й елемент y + i-й елемент L)
                    break
                elif i-й елемент y == j-й елемент на i-му рівні BMT and j непарний:
                    L приймає hash(i-й елемент L + i-й елемент y)
                    break
    else:
        for i in range(n, len(y) + 1):
            for j in range(len(BMT[i])):
                if i-n елемент y == j-й елемент на i-му рівні BMT and j парний:
                    L приймає hash(i-n елемент y + останній елемент L)
                    break
                elif i-n елемент y == j-й елемент на i-му рівні BMT and j непарний:
                    L приймає hash(останній елемент L + i-n елемент y)
                    break

    if останній елемент L == корінь BMT:
        return True
    else:
        return False
```

Для аналізу складності пройдемось по кожному етапу коду:

- Пошук елемента в **leaves** займає не більше ніж $O(n)$, де n – кількість елементів в **leaves**;
- Пошук елемента на інших рівнях БДМ займає $\frac{n}{2}, \frac{n}{4}, \frac{n}{8}, \ldots$, в сумі маємо $\frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \cdots = n$, а отже $O(n)$;
- Перевірка доказу і формування списку L в найгіршому випадку займе $2 \cdot n$ часу (пройдемо по всім рівням і всім елементам), тому $O(2 \cdot n) = O(n)$;
- Звірка кореня БДМ з останнім елементом L має складність $O(1)$.

Як результат, маємо $O(n) + O(n) + O(n) + O(1) = O(n)$

2 Розріджене дерево Меркла.

Основна відмінність розрідженого дерева Меркла від бінарного в тому, що у РДМ завжди однакова кількість елементів, абсолютна більшість з яких – порожні. Так як ми використовуємо sha256 для хешування, перший рівень РДМ має 2^{256} елементів, більшість з яких, знову таки, порожні. Рівнів у РДМ 256, а загальна кількість елементів:

$$\sum_{j=1}^{256} 2^j = 2^{256-1} - 1$$

що є дуже великою кількістю, щоб зберігати все дерево у пам'яті комп'ютера, і підрахунок більшості з елементів ніяк не впливає на дерево, але займає великий час. Тому при роботі з РДМ використовують різні методи зберігання лише тієї частини дерева, що використовується. Ми зробимо наступне:

- зрозуміємо, скільки елементів із першого рівня можна відкинути, щоб дерево не змінилось;
- сформуємо список із отриманих елементів;
- побудуємо БДМ, взявши отриманий список як початкові дані;
- зрозуміємо, скільки разів треба захешувати суму кореня отриманого БДМ з хешем None;
- отримуємо корінь РДМ.

Почнемо з першого пункту. Якщо маємо n початкових елементів, то нам треба, щоб список мав у собі $2^{\lceil \log_2(n) \rceil}$ елементів. Тобто:

$$\Big[\underbrace{a_0,\, a_1,\, a_2,\, \ldots,\, a_{n-1},\, \mathrm{None},\, \mathrm{None},\, \ldots,\, \mathrm{None}}_{2^{\lceil \log_2(n) \rceil}}\Big]$$

Інші $2^{256} - 2^{\lceil \log_2(n) \rceil}$ можна відкинути, вони не впливають на дерево, окрім кореня. Це і буде список листків, на яких побудуємо БДМ. Це дерево має висоту $\lceil \log_2(n) \rceil$. Для того, щоб отримати корінь РДМ, треба виконати наступне:

```
for i in range(math.ceil(math.log2(n)), 256):
     root = hash(root + hash(None))
```

Таким чином, у пам'яті зберігається лише потрібна частина РДМ, а нульові елементи, яких величезна кількість, але які ніяк не впливають на хеші, відкидаються. Код для побудови РДМ виглядає наступним чином:

```
def sparse_tree_builder(x):
    height = найближча степінь, у яку треба піднести 2 щоб отримати len(x)
    L = [None] * (2**height)

    for i in range(len(x)):
        i-й елемент L = i-й елемент x

    SMT = binary_tree_builder(L)
    root = SMT[-1][-1]
    j = len(binary_tree_builder(L))

    while j < 256:
        if j == 255:
            root = hash(root + hash(None))
            SMT приймає root
```

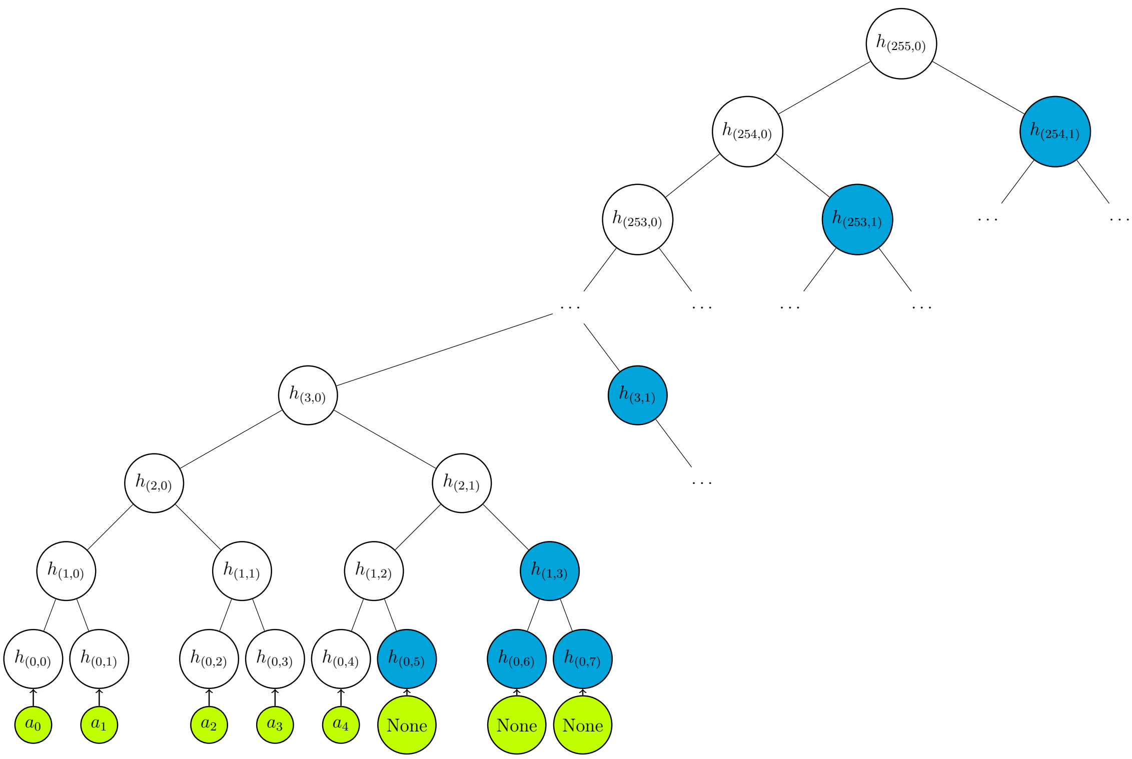
```
    else:
        root = hash(root + hash(None))
        SMT приймає [root, hash(None)]
    j збільшується на 1

return SMT
```

Візуально зобразити РДМ теж доволі складно, знову таки через величезну кількість елементів. Але можна зобразити тільки ту частину, з якою працюємо, а саме:

- Розширений список початкових даних;
- БДМ, збудоване на цьому списку;
- "Башта" виду [елемент, hash(None)];
- Корінь РДМ.

Нехай маємо 5 листів. Тоді розширений список початкових даних повинен складатися з $2^{\lceil \log_2(5) \rceil} = 2^3 = 8$ елементів, а саме:

$$\left[a_0,\ a_1,\ a_2,\ a_3,\ a_4,\ \text{None},\ \text{None},\ \text{None}\right]$$



Зелені дані – початкові дані (листя), сині – нульові елементи.

2.1 Складність додавання нового елемента в РДМ.

Для розрахунку складності додавання нового елемента проаналізуємо код поетапно:

- Формування порожнього списку L має довжину $2^{\lceil \log_2(n) \rceil}$, де n – кількість початкових даних. Довжина L не більше ніж у 2 рази більша за n ($n = 3 \Rightarrow 2^{\lceil \log_2(n) \rceil} = 4, \quad n = 5 \Rightarrow 2^{\lceil \log_2(n) \rceil} = 8, \quad n = 478 \Rightarrow 2^{\lceil \log_2(n) \rceil} = 512$), тому $O() \approx O(2 \cdot n) = O(n)$.
- Заміна елементів L елементами з x (leaves по суті) займає $O(n)$ часу, бо в x n елементів;

- Побудова БДМ зі списку L займає $O(n)$ часу (те ж саме додавання елемента в бінарне дерево, але $k = 0$);

- Хешування кореня БДМ з hash(None) виконується $256 - \lceil \log_2(n) \rceil$ раз, тоді $O\big(256 - \lceil \log_2(n) \rceil\big) \leq O(256 - \log_2(n)) \leq O(256) = O(1)$;

- Кінцева складність побудови РДМ $O(n) + O(n) + O(n) + O(1) = O(n)$.

Додавання нових елементів у РДМ супроводжується повним перерахунком усього дерева, тобто складність формування розширеного списку це $O(n + k)$, складність формування РДМ на цьому списку теж $O(n+k)$, тоді складність додавання нових елементів у РДМ $O(n+k)+O(n+k) = O(n+k)$.

2.2 Складність генерації доказу включення.

Генерація доказу включення для РДМ майже нічим не відрізняється від генерації доказу включення, навіть код буде використано дуже схожий. Єдина різниця – нам треба у доказ (що є списком хешів) додати певну кількість нульових хешів. Потреба цієї дії випливає з малюнку РДМ – збудували БДМ на розширеному списку початкових даних і далі для отримання кореня РДМ хешуємо корінь БДМ з нульовим хешем певну кількість разів. Ось це і треба врахувати при формуванні доказу включення для РДМ. Код має вигляд:

```
def sparse_tree_mp(x):
    proof, n, k, found = [], 0, 0, False
    l = копія leaves + [None] * (найближча степінь 2 до len(leaves) - len(leaves))
    tree = binary_tree_builder(l)

    for i in range(len(leaves)):
        if x == i-й елемент leaves:
            k = i
            found = True
            break
        else:
            for i in range(1, len(tree)):
                for j in range(len(tree[idx])):
                    if x == j-й елемент на i-му рівні tree:
                        n, k = i, j
                        found = True
                        break
    if not found:
        return None
    else:
        proof приймає сусіда до k-го елемента на рівні n tree
        for i in range(n + 1, len(tree) - 1):
            proof приймає батька до k-го елемента на рівні n tree
            for j in range(len(tree[i])):
                if останній елемент proof == j-й елемент на i-му рівні tree:
                    k = j

        for i in range(len(tree) - 1, 256):
            proof приймає hash(None)
        return proof
```

Говорячи про складність генерації доказу включення, маємо наступне:

- Копіювання списку листів займає $O(n)$, додавання порожніх елементів $\approx O(n)$, тоді для цього кроку маємо $O(n)$;

- Побудова БДМ займає $O(n)$;

- Пошук елемента в листі або на інших рівнях дерева все був проаналізований і займає $O(n)$;
- Формування доказу займає $O(\log(n))$, бо в гіршому випадку проходимо по кожному рівню дерева і забираємо по одному елементу;
- Додавання порожніх елементів теж було проаналізоване і займає $O(1)$.

Як результат, маємо $O(n) + O(n) + O(n) + O(\log(n)) + O(1) = O(n + \log(n))$.

2.3 Розмір доказу включення.

Доказ включення – це список хешів. Як вже було з'ясовано, один хеш має довжину 64 байта, а доказ включення для РДМ завжди містить 255 елементів, то і довжина фіксована, а саме $64 * 255 = 16320$ байт.

2.4 Складність верифікації доказу включення.

Верифікація доказу включення для РДМ суттєво мало чим відрізняється від аналогічного для БДМ. У функцію верифікації передається елемент, наявність якого підтверджуємо, і сформований доказ для нього. Копіюємо листя і доповнюємо його елементами None до найближчої степені двійки, будуємо БДМ на цьому списку. Далі шукаємо наш елемент спочатку в листі, а потім на інших рівнях (якщо не знайшли у листі). Якщо після всього пошуку в L немає елементів, то поданого у функцію значення в дереві немає. А далі все залежить від того на якому рівні було знайдено наш елемент, але алгоритм майже однаковий – певну кількість раз складаємо і хешуємо елементи із дерева і **sparse_tree_mp** в певній послідовності, а потім хешуємо суму останнього елемента із L з hash(None) len(Tree) - len(sparse_tree_mp) раз. У вигляді коду маємо наступне:

```
def sparse_proof_verification(x, y):
    if y == None:
        return None

    L, n = [], 0

    l = копія leaves + [None] * (найближча степінь 2 до len(leaves) - len(leaves))
    tree = binary_tree_builder(l)

    for i in range(len(leaves)):
        if x == i-й елемент leaves:
            n = 0
            L приймає hash(x)

    if len(L) == 0:
        for i in range(len(SMT)):
            for j in range(len(i-й рівень SMT)):
                if x == j-й елемент на i-му рівні SMT:
                    n = i
                    L приймає x
    else:
        pass

    if len(L) == 0:
        return 'Немає такого елемента в дереві.'

    if n == 0:
        for i in range(len(tree)):
            for j in range(len(i-й рівень tree)):
                if i-й елемент y == j-й елемент на i-му рівні tree and j парний:
```

```
                    L приймає hash(i-й елемент y + i-й елемент L)
                    break
                elif i-й елемент y == j-й елемент на i-му рівні tree and j непарний:
                    L приймає hash(i-й елемент L + i-й елемент y)
                    break
    else:
        for i in range(n, len(tree) + 1):
            for j in range(len(i-й рівень SMT)):
                if i - n елемент y == j-й елемент на i-му рівні SMT and j парний:
                    L приймає hash(i - n елемент y + останній елемент L)
                    break
                elif i - n елемент y == j-й елемент на i-му рівні SMT and j непарний:
                    L приймає hash(останній елемент L + i - n елемент y)
                    break

    for i in range(len(tree), len(y)):
        L приймає hash(i - 1 елемент L + hash(None))

    if останній елемент L == корінь SMT:
        return True
    else:
        return False
```

Для оцінки складності проаналізуємо код покроково:

- Перевірка y на порожність і оголошення змінних має складність $O(1)$;
- Формування списку l вже було проаналізоване і має складність $O(n)$, як і будування БДМ на цьому списку. Кінцева складність для цього етапу $O(n) + O(n) = O(n)$;
- Пошук елемента в листі має складність $O(n)$, а пошук на інших рівнях займає $O(\log(n))$ для проходу по кожному рівню і $O(n)$ для проходу по всім елементам рівня, отже кінцева складність для пошуку елемента в гіршому випадку становить $O(n) + O(\log(n)) + O(n) = O(n + \log(n))$;
- Перевірка L на порожність займає $O(1)$;
- Проходження по дереву і хешування в гіршому випадку відбувається з самого листя і до кореня, а робиться одна операція на одному рівні, тому складність становить $O(\log(n))$;
- Хешування елемента з L з hash(None) має складність $O(1)$, як було з'ясовано вище;
- Звірка елементів займає $O(1)$.

Як кінцевий результат маємо $O(1)+O(n)+O(n)+O(1)+O(\log(n))+O(1)+O(1) = O(n+\log(n))$.

3 Індексоване дерево Меркла.

Індексоване дерево Меркла (ІДМ) схоже на БДМ за вийнятком того, що якщо в БДМ для елемента не знаходилось пари для хешування їх суми, то він дублювався і додавався в кінець списку, а в ІДМ "одинокий"елемент просто переноситься без змін на наступний рівень. Також відмінність у тому, що за кожним елементом закріплений свій індекс, що спрощує взаємодію з деревом і окремими елементами. Візуалізуємо невелике дерево для наочності.

Нехай є список листів leaves $= [a_0, a_1, a_2, a_3, a_4, a_5, a_6]$. Тоді у хешованому вигляді будемо мати перший рівень дерева

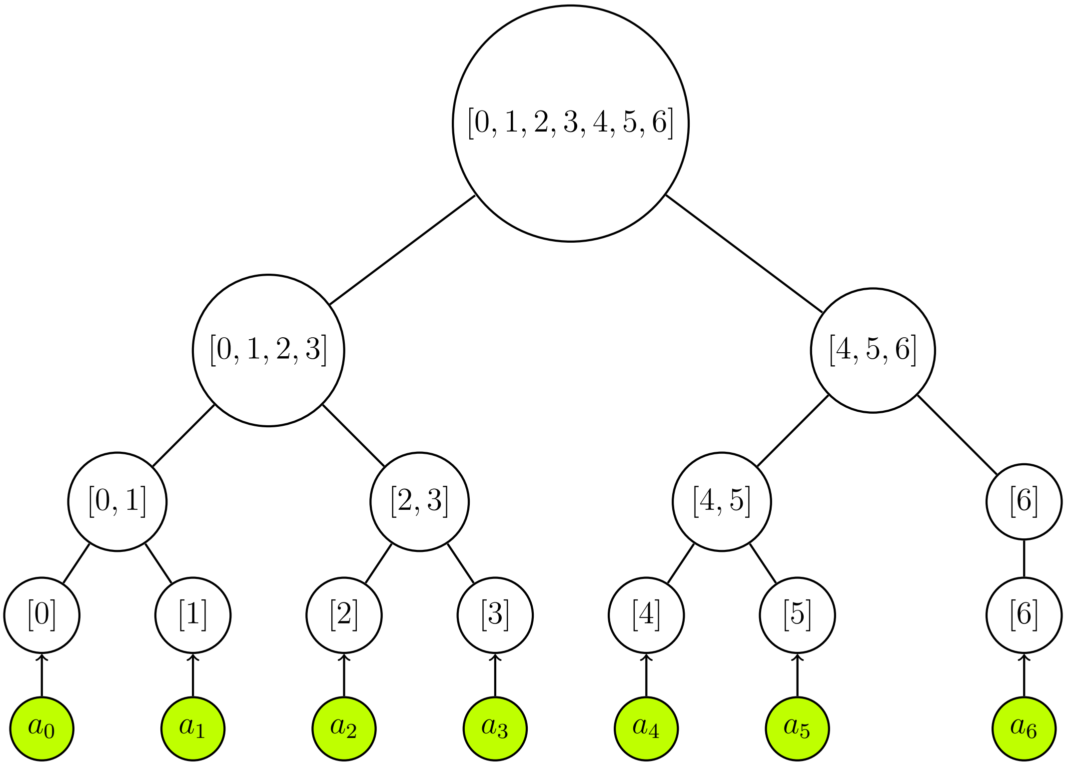
$$\Big[\,\big[h_{(0,1)},\,[0]\big],\,\big[h_{(0,1)},\,[1]\big],\,\big[h_{(0,2)},\,[2]\big],\,\big[h_{(0,3)},\,[3]\big],\,\big[h_{(0,4)},\,[4]\big],\,\big[h_{(0,5)},\,[5]\big],\,\big[h_{(0,6)},\,[6]\big]\,\Big]$$

Наступні рівні формуються так само як і раніше, тільки враховуємо індекси, які змінюються наступним чином:

$$h_{(1,0)} = \Big[\underbrace{\text{hash}(h_{(0,0)} + h_{(0,1)})}_{\text{Новий елемент ІДМ}},\ \underbrace{[0,1]}_{\text{Індекс}}\Big], \qquad h_{(1,1)} = \Big[\underbrace{\text{hash}(h_{(0,2)} + h_{(0,3)})}_{\text{Новий елемент ІДМ}},\ \underbrace{[2,3]}_{\text{Індекс}}\Big]$$

$$h_{(2,0)} = \Big[\underbrace{\text{hash}(h_{(1,0)} + h_{(1,1)})}_{\text{Новий елемент ІДМ}},\ \underbrace{[0,1,2,3]}_{\text{Індекс}}\Big]$$

Для візуалізації ІДМ опустимо написи $h_{(n,k)}$, а запишемо лише індекси:



Код для побудови ІДМ з заданими початковими даними виглядає наступним чином:

```
def indexed_tree_builder(x):
    tree, Level = [], []
    for i in range(len(x)):
        Level приймає [hash(i-й елемент x), [i]]

    tree приймає Level
    for i in range(1, math.ceil(math.log2(len(x))) + 1):
        Level = []

        for j in range(0, 2 * math.floor(len(tree[i - 1]) / 2), 2):
            Level приймає [hash(0-вий елемент j-го списку на i-1 рівні tree +
            0-вий елемент j+1-го списку на i-1 рівні tree), [об'єднання їх індексів]]

        if len(i-1 -й рівень tree) непарна:
            Level приймає (останній елемент i-1 -го рівня tree)
        else:
            pass

        tree приймає Level

    return tree
```

3.1 Складність додавання нового елемента в ІДМ.

Додавання нового елемента в ІДМ відбувається абсолютно так само, як і в БДМ/РДМ – формується новий список із листя і додаткових даних і він передається у **indexed_tree_builder**. У вигляді коду маємо наступне:

```
def extra_indexed_tree_builder(data, extra_data):
    l = [об'єднання списків data i extra_data]
    return indexed_tree_builder(l)
```

Для оцінки складності розберемо покроково алгоритм:

- Хешування всіх елементів списку і додавання їх індексів має складність $O(n)$;
- При формуванні нового рівня кількість елементів зменшується приблизно в 2 рази, тому складність буде $O(n) + O\left(\frac{n}{2}\right) + O\left(\frac{n}{4}\right) + \ \ldots \ = O(2 \cdot n) = O(n)$;
- Кінцева складність будування дерева зі списку із n елементів буде $O(n) + O(n) = O(n)$;
- Якщо треба додати k нових елементів, то формування нового списку із листів і додаткових даних має складність $O(n+k)$ і побудова дерева на такому списку буде мати складність $O(n+k)$, тому кінцева складність додавання нових елементів має вигляд $O(n+k) + O(n+k) = O(n+k)$.

3.2 Складність генерації доказу включення.

Генерація доказу включення для ІДМ має майже ту ж саму структуру, що і для БДМ, але треба враховувати "одинокі" елементи. Це не дуже змінює код, але зауважити треба. Сам код:

```
def indexed_tree_mp(x):
    proof, n, k = [], 0, 0

    if len(leaves) непарна and x == останній елемент leaves:
        k = len(leaves) - 1
        while len(індекс останнього елемента на n+1 рівні IMT) == 1:
            n збільшується на 1
            k = len(n-й рівень IMT) - 1
    else:
        for i in range(len(leaves)):
            if x == i-й елемент leaves:
                k = i
                break
            else:
                for i in range(1, len(IMT)):
                    for j in range(len(i-й рівень IMT)):
                        if x == перший елемент j-го списку на i-му рівні IMT:
                            n, k = i, j

    proof приймає сусіда до k-го елемента на n рівні IMT

    for i in range(n + 1, len(IMT) - 1):
        proof приймає батька k-го елемента на i-му рівні IMT
        for j in range(len(IMT[i])):
            if 0-й елемент останнього списку proof == 0-й елемент j-го списку на i-му рівні IMT:
                k = j

    return proof
```

Для оцінки складності розберемо покроково алгоритм:

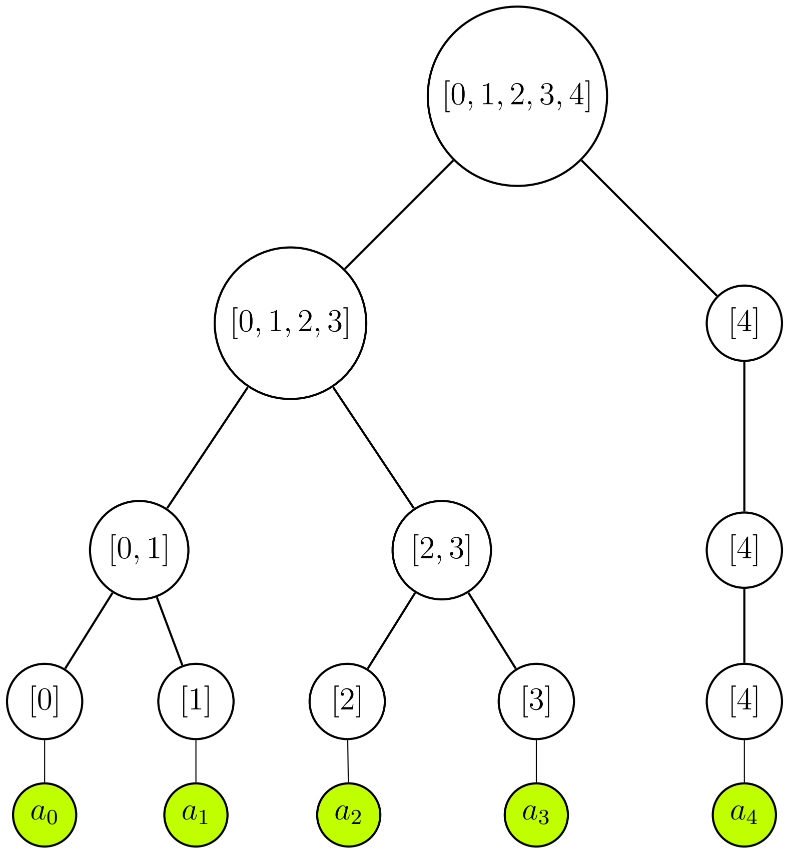
- В найгіршому випадку треба пройти всі рівні дерева, що має складність $O(\log(n))$ і всі елементи на кожному рівні, тобто $O(n)$;

- Для формування доказу в найгіршому випадку треба пройтись по кожному рівню дерева і зібрати по одному хешу на кожному, що має складність $O(\log(n))$.

Кінцева складність становить $O(n) + O(\log(n)) + O(\log(n)) = O(n + \log(n))$.

3.3 Розмір доказу включення.

Доказ включення може мати різний розмір через "одинокі" елементи. Як можна побачити на малюнку ІДМ, для доказу наявності елементу з індексом 6 достатньо двох хешів, а для доказу елемента з індексом 3 – три хеші. Якщо розглянути дерево з п'яти початковими даними, то будемо мати:



Для доказу елемента з індексом 4 достатньо лише одного елемента. На цьому прикладі бачимо, що розмір доказу включення ≥ 64 байта, але $\leq 64 \cdot (\lceil \log_2(n) \rceil - 1)$.

3.4 Складність верифікації доказу включення.

Верифікація доказу включення елемента в ІДМ взагалі не відрізняється від аналогічного для БДМ. В цьому випадку було вирішено подавати у функцію верифікації не тільки елемент і сформований для нього доказ, а і індекс елемента. Це спрощує код, зменшує кінцевий час роботи програми, але призводить до іншого недоліку – неможна просто подати елемент і програма сама знайде його індекс. Код виглядає наступним чином:

```
def indexed_proof_verification(x, k, y):
    L = []

    if x in leaves:
        L.append([hash(x), [k]])
        for i in range(len(y) - 1):
            if int(y[i][1][-1]) < int(L[i][1][-1]):
```

```
                L.append([hash(y[i][0] + L[i][0]), [*y[i][1], *L[i][1]]])
            else:
                L.append([hash(L[i][0] + y[i][0]), [*L[i][1], *y[i][1]]])

        if L[-1][0] == IMT[-2][0][0]:
            L.append([hash(L[-1][0] + y[-1][0]), [*L[-1][1], *y[-1][1]]])
        else:
            L.append([hash(y[-1][0] + L[-1][0]), [*y[-1][1], *L[-1][1]]])
    else:
        L.append([x, k])
        for i in range(len(y) - 1):
            if int(y[i][1][-1]) < int(L[i][1][-1]):
                L.append([hash(y[i][0] + L[i][0]), [*y[i][1], *L[i][1]]])
            else:
                L.append([hash(L[i][0] + y[i][0]), [*L[i][1], *y[i][1]]])

        if L[-1][0] == IMT[-2][0][0]:
            L.append([hash(L[-1][0] + y[-1][0]), [*L[-1][1], *y[-1][1]]])
        else:
            L.append([hash(y[-1][0] + L[-1][0]), [*y[-1][1], *L[-1][1]]])

    if L[-1][0] == IMT[-1][0][0]:
        return True
    else:
        return False
```

Для оцінки складності цього коду, пройдемо по ньому поетапно:

- Пошук x серед leaves займає в гіршому випадку $O(n)$ часу;
- Далі іде хешування елементів на кожному рівні, в гіршому випадку це займає $O(\log(n))$ часу
- Перевірка індексів має складність $O(1)$.

Як результат, маємо $O(n) + O(\log(n)) = O(n + \log(n))$.

Порівняльна таблиця.

	БДМ	РДМ	ІДМ
Додавання нового елемента	$O(n+k)$	$O(n+k)$	$O(n+k)$
Генерація доказу включення	$O(\log(n))$	$O(n+\log(n))$	$O(n+\log(n))$
Розмір доказу включення	$64 \cdot \big(\lceil \log_2(n) \rceil - 1\big)$	16320	$\leq 64 \cdot \big(\lceil \log_2(n) \rceil - 1\big)$
Верифікація доказу включення	$O(n)$	$O(n+\log(n))$	$O(n+\log(n))$