Префіксні дерева

Резюме: асоціативні масиви

Резюме: асоціативні масиви

- Асоціативний масив (словник, символьна таблиця, англ: map, dictionary, associative array, symbol table) - це абстрактна структура даних, що дозволяє зберігати пари типу “ключ-значення”, з можливістю доступу до даних за ключем
- Основні операції в асоціативному масиві:
  - Пошук
  - Вставка
  - Видалення

Резюме: асоціативні масиви

- Розглянуті реалізації асоціативних масивів:

Резюме: асоціативні масиви

- Розглянуті реалізації асоціативних масивів:

Структура даних	Пошук		Вставка		Видалення		Підтримка порівняльних операцій
	Середній випадок	Найгірший випадок	Середній випадок	Найгірший випадок	Середній випадок	Найгірший випадок	
Зв'язний список або масив	O(N)	O(N)	O(N)	O(N)	O(N)	O(N)	Немає
Відсортований масив	O(log N)	O(log N)	O(N)	O(N)	O(N)	O(N)	Є
Хеш-таблиця	1	O(N)	1	O(N)	1	O(N)	Немає
Бінарне дерево пошуку	O(log N)	O(N)	O(log N)	O(N)	O(sqrt(N))	O(N)	Є
AVL-дерево	O(log N)	O(log N)	O(log N)	O(log N)	O(log N)	O(log N)	Є

Резюме: асоціативні масиви

- Порівняльні операції (Ordered operations):
  - min()
  - max()
  - floor(key)
  - ceiling(key)
  - rank(key)
  - select(rank)
  - range(min_key, max_key)

Ключі асоціативного масиву

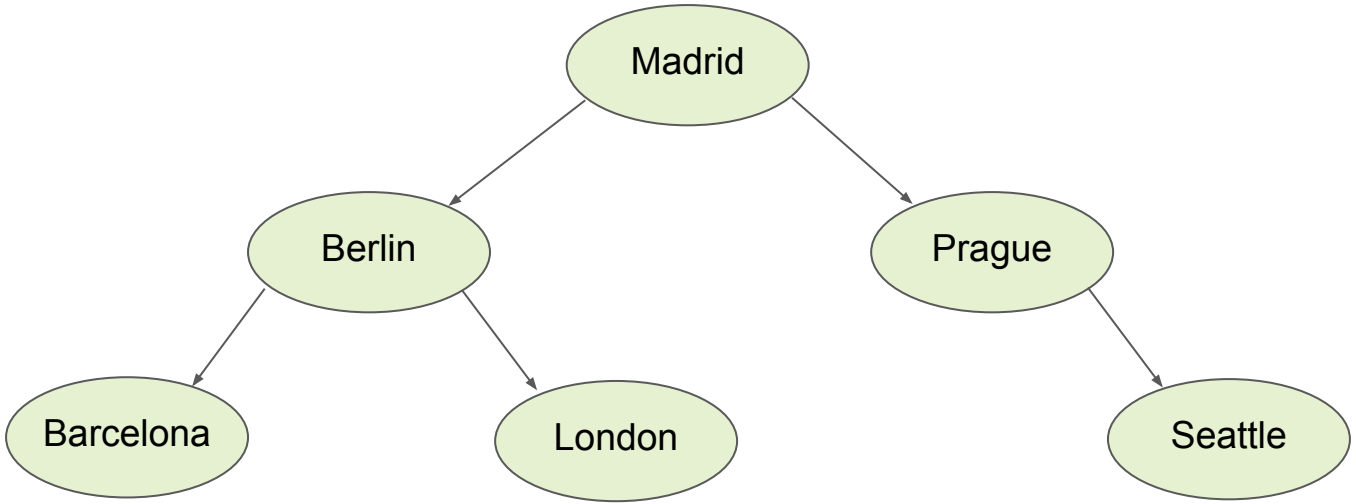
- Ми зазвичай вважаємо, що складність операцій порівняння двох ключів або обчислення хеш-значення ключа є константною. Таке припущення доцільне для числових ключів типу *int*, *float*, а також для *рядків з обмеженою довжиною*
- У випадку, коли ключами є довільні символьні рядки, таке припущення не завжди коректне, так як порівняння відбувається посимвольно:
  - Для порівняння ключів “cat” і “dog” треба зробити меншу кількість операцій, ніж для порівняння ключів “administrator” і “administration”
- Аналогічно і з обчисленням хеш-значення від рядка - потрібно пройтись по всім символам

Дерева з ключами-рядками

- Позначення:
  - N - кількість елементів у дереві
  - L - довжина шуканого рядка
- Складність основних операцій (get, insert, delete) буде залежати як від N, так і від L

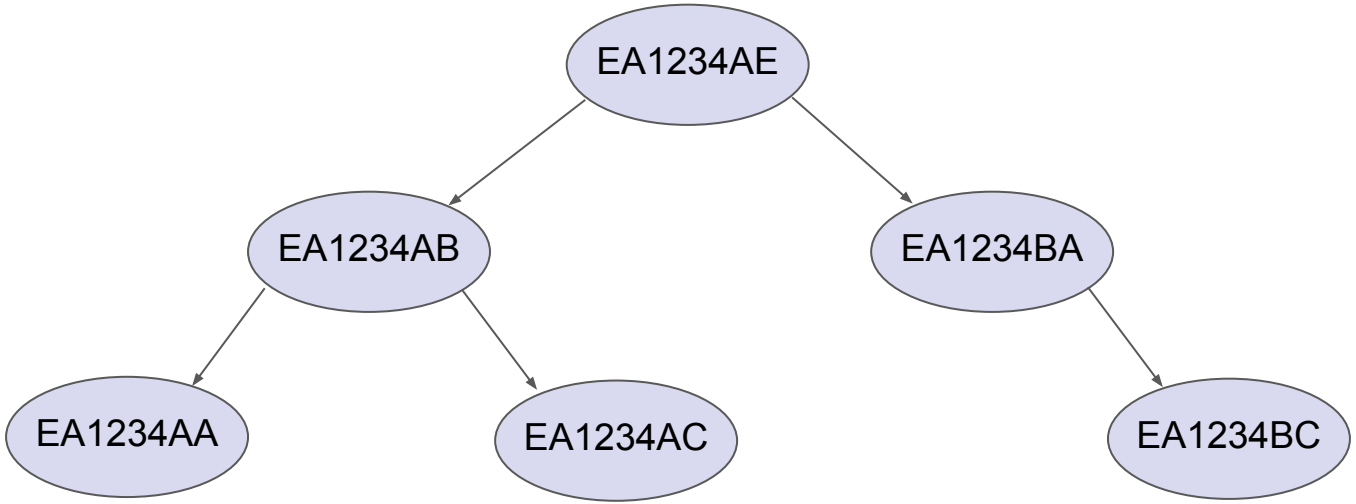
Приклади дерев з ключами-рядками

- Ключ - назва міста, значення - інформація про місто
- Складність операції `get(“London”)` ~ *L + log(N)*



Приклади дерев з ключами-рядками

- Ключ - номерний знак автомобіля, значення - інформація про авто
- Складність операції `get(“EA1234AC”)` ~ *L * log(N)*

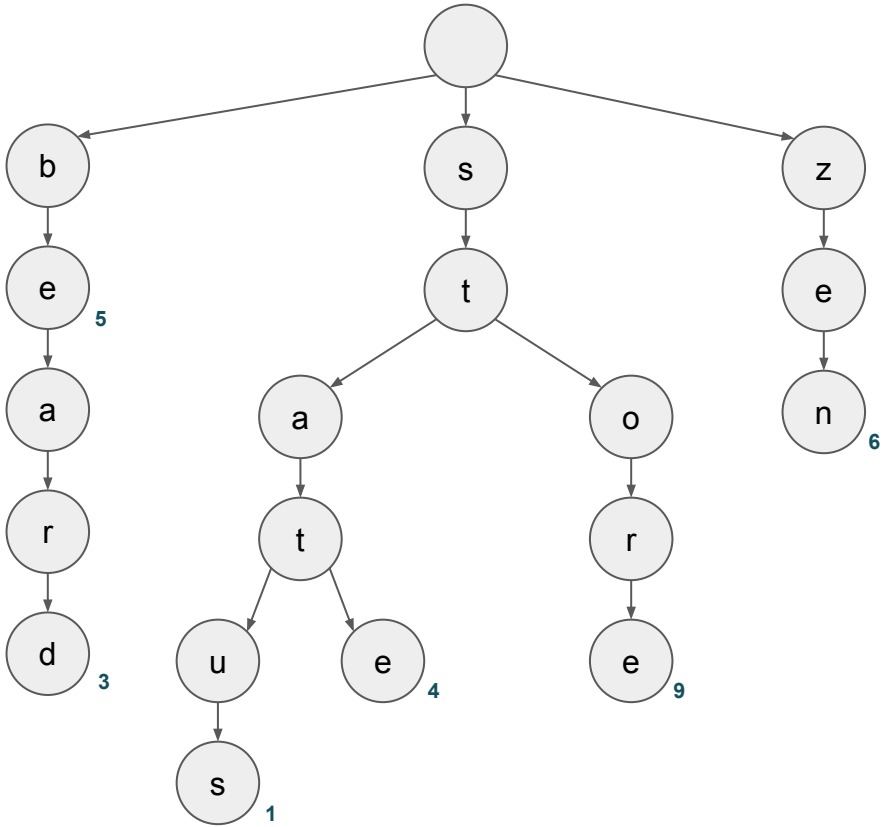


Чи можемо ми краще?

- Можна використовувати хеш-таблицю, тоді складність схожих операцій була б ~ L
  - Але в хеш-таблиці немає можливості швидко виконувати порівняльні операції типу min, max, range і т.д.
- Інше рішення - префіксне дерево

Префіксне дерево

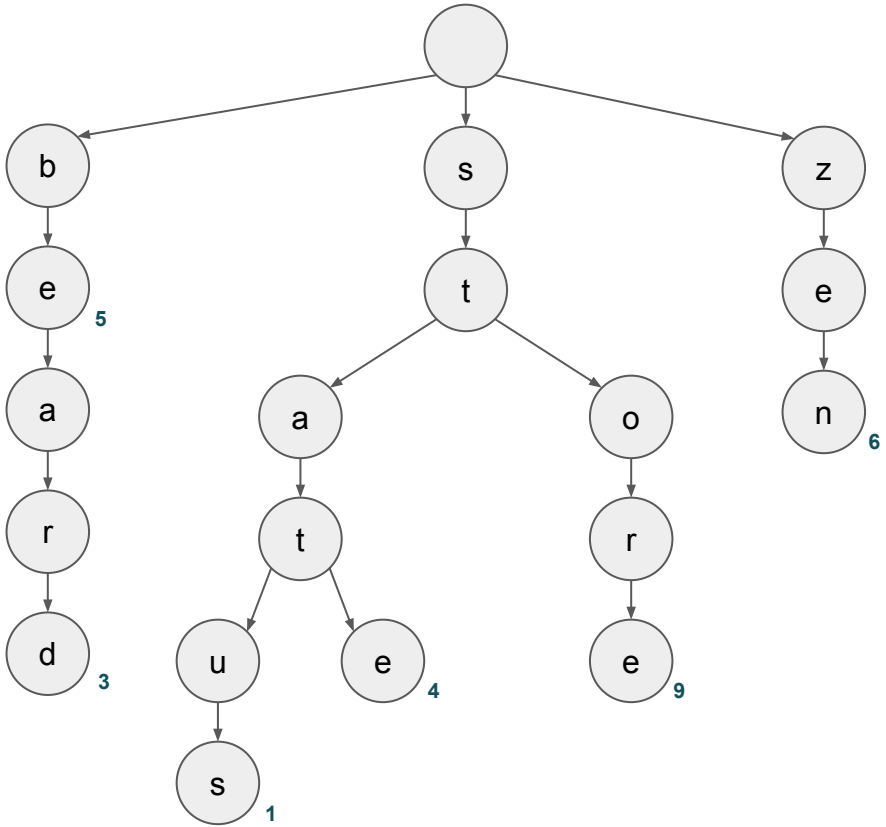
- Префіксне дерево (англ. trie) - це структура даних, яка дозволяє зберігати асоціативний масив, ключами якого є рядки
- Кожен вузол позначає певний символ
- Ключ визначається шляхом від кореня до листка
- У рядків з однаковим префіксом частина шляху є спільною



Префіксне дерево

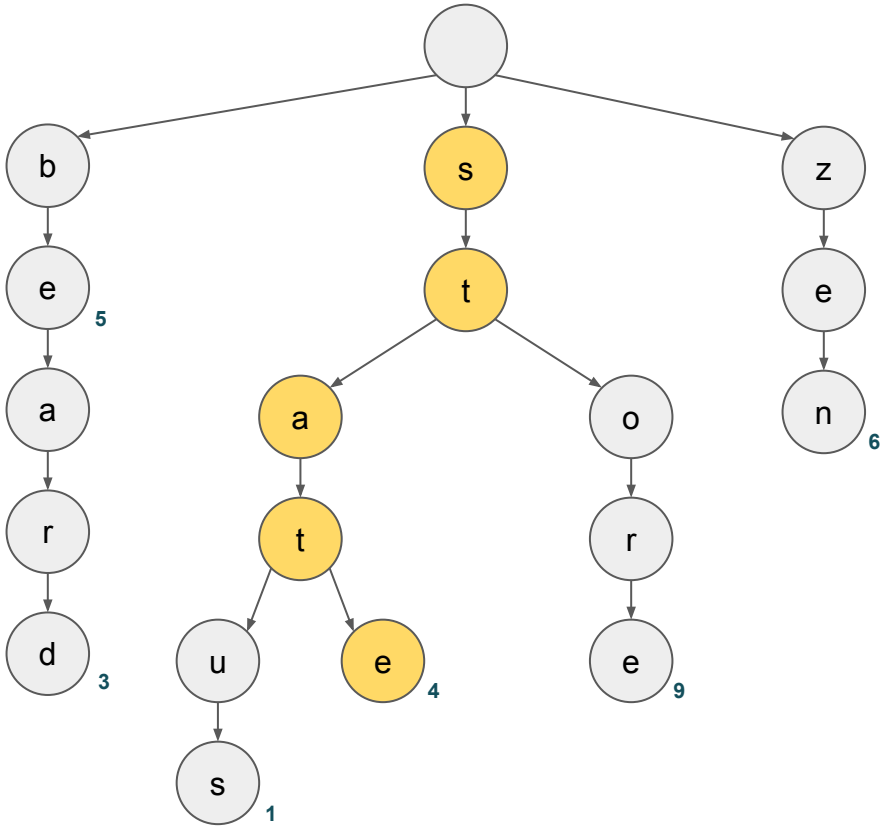
- В даному дереві наявні такі пари “ключ-значення”:

Ключі	Значення
be	5
beard	3
status	1
state	4
store	9
zen	6



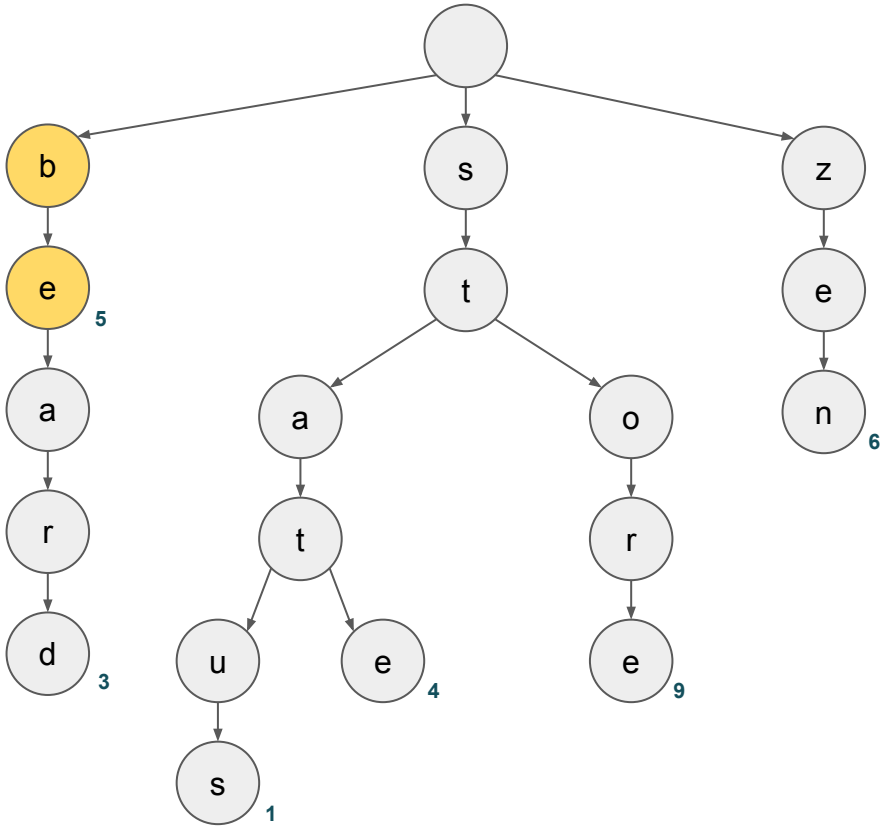
Пошук в префіксному дереві

- `get("state")`
- Повернеться значення “4”



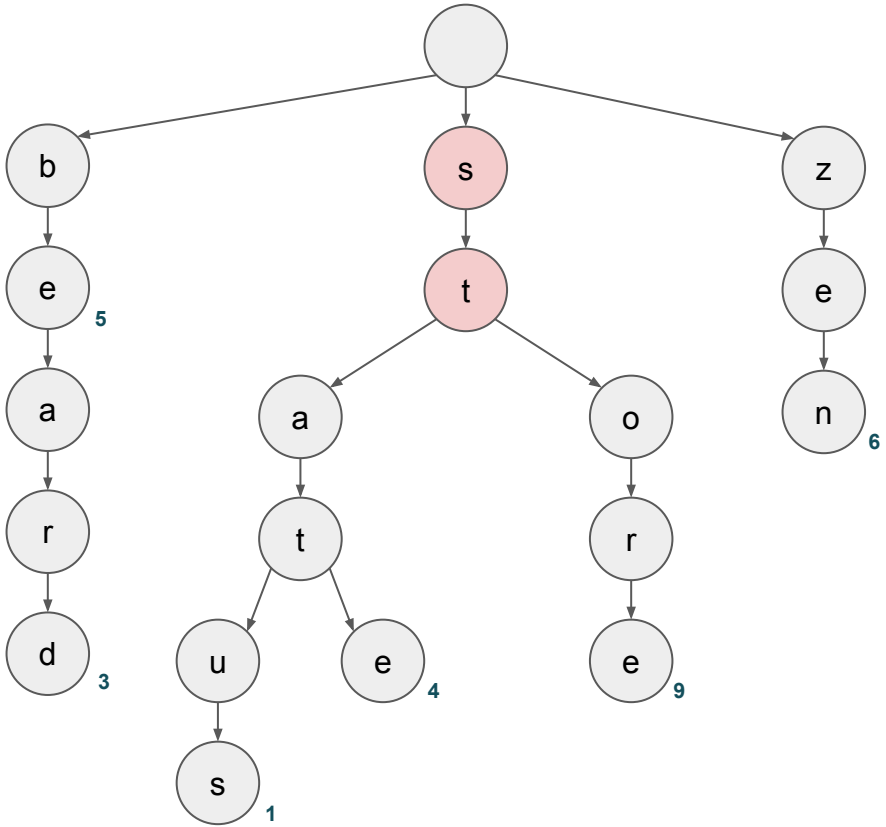
Пошук в префіксному дереві

- `get("be")`
- Повернеться значення “5”



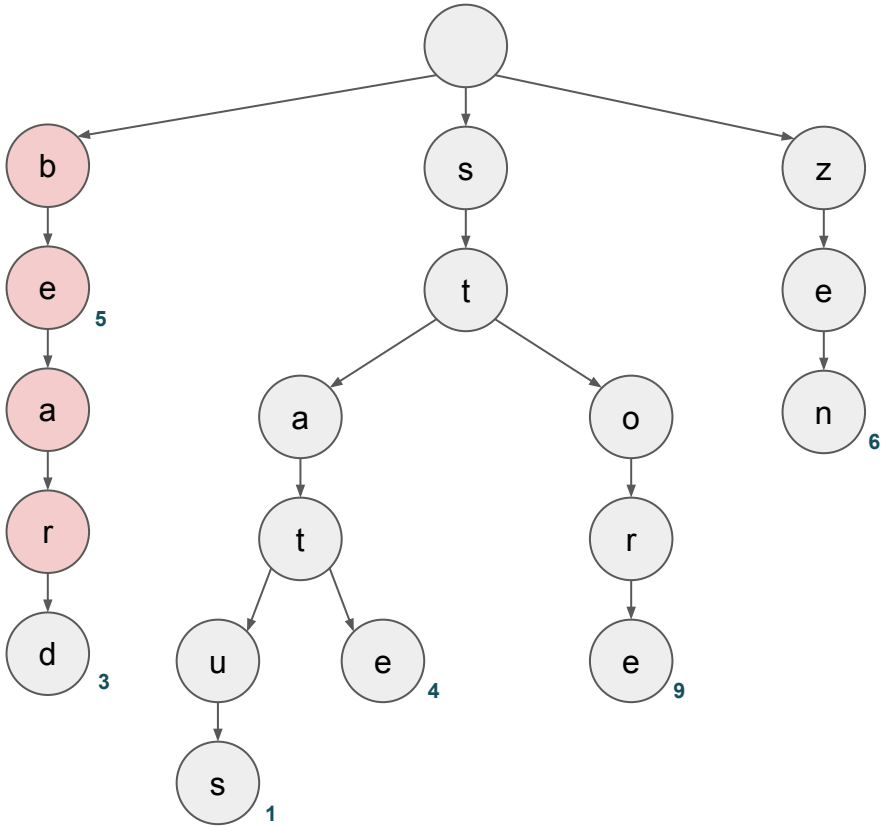
Пошук в префіксному дереві

- `get(“steel”)`
- У вузлі “t” немає посилання на вузол “e”, отже ключ “steel” відсутній у цьому дереві



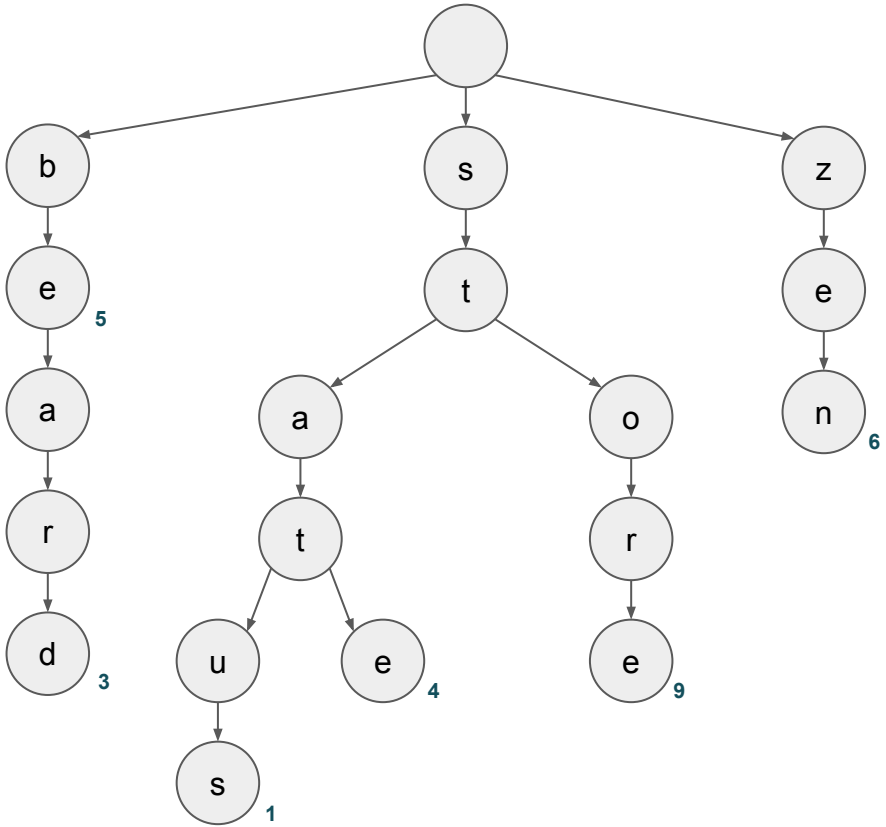
Пошук в префіксному дереві

- `get(“bear”)`
- У дереві є шлях, який утворює ключ “bear”, але в останньому вузлі цього шляху немає значення, тому вважаємо, що пара ключ-значення відсутня



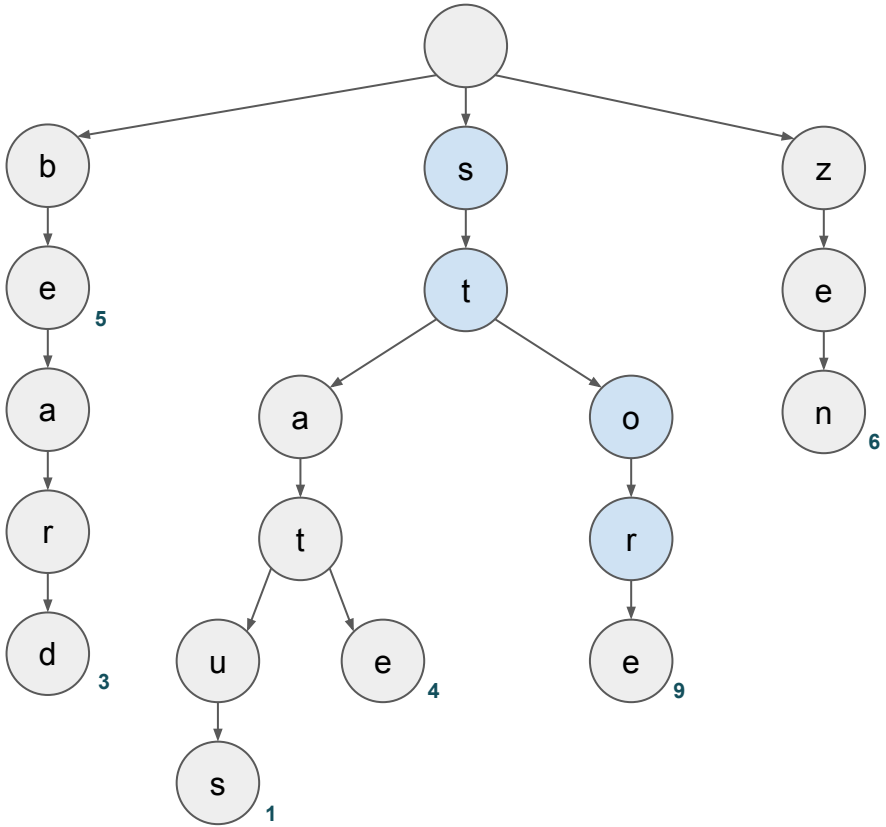
Вставка в префіксне дерево

- `put(“story”, 8)`



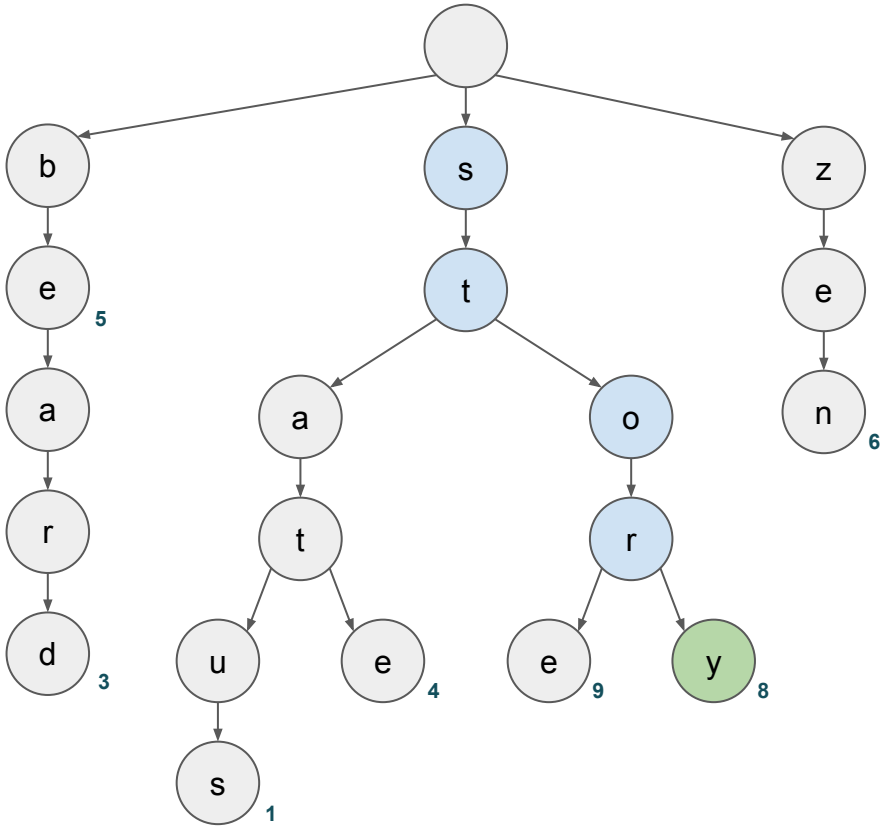
Вставка в префіксне дерево

- `put(“story”, 8)`
- Найдовшим префіксом ключа “story”, що існує в дереві, є “stor”



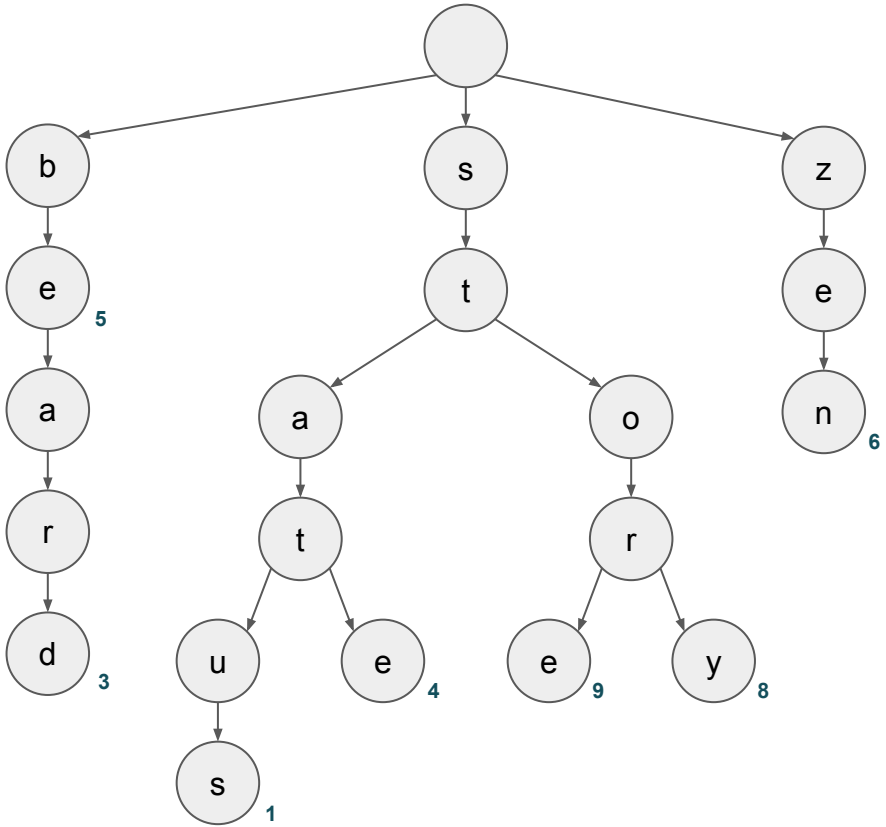
Вставка в префіксне дерево

- `put(“story”, 8)`
- Найдовшим префіксом ключа “story”, що існує в дереві, є “stor”
- В останньому вузлі цього префіксу додаємо вузол-нащадок з літерою “y” і значенням “8”



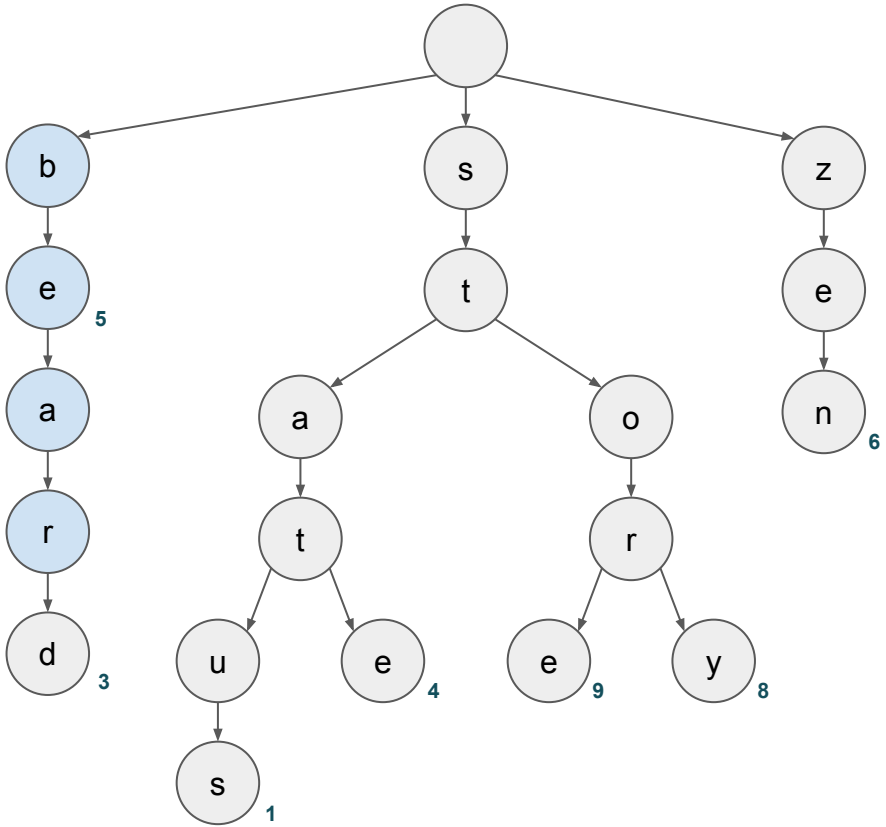
Вставка в префіксне дерево

- `put(“bear”, 4)`



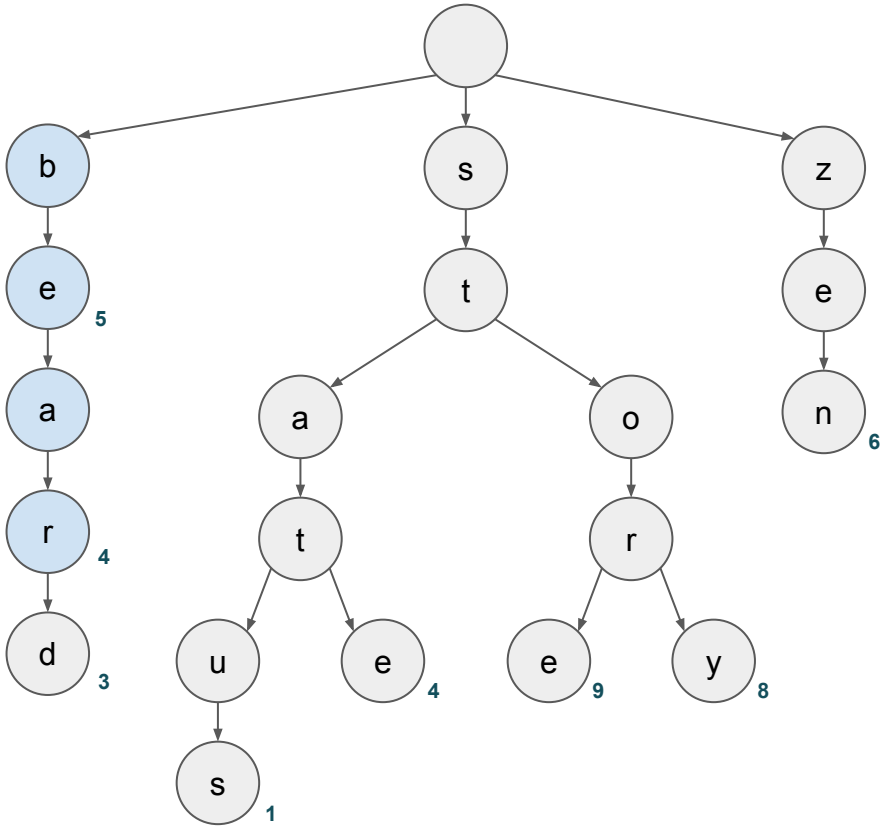
Вставка в префіксне дерево

- `put(“bear”, 4)`
- В дереві шлях “bear” вже існує



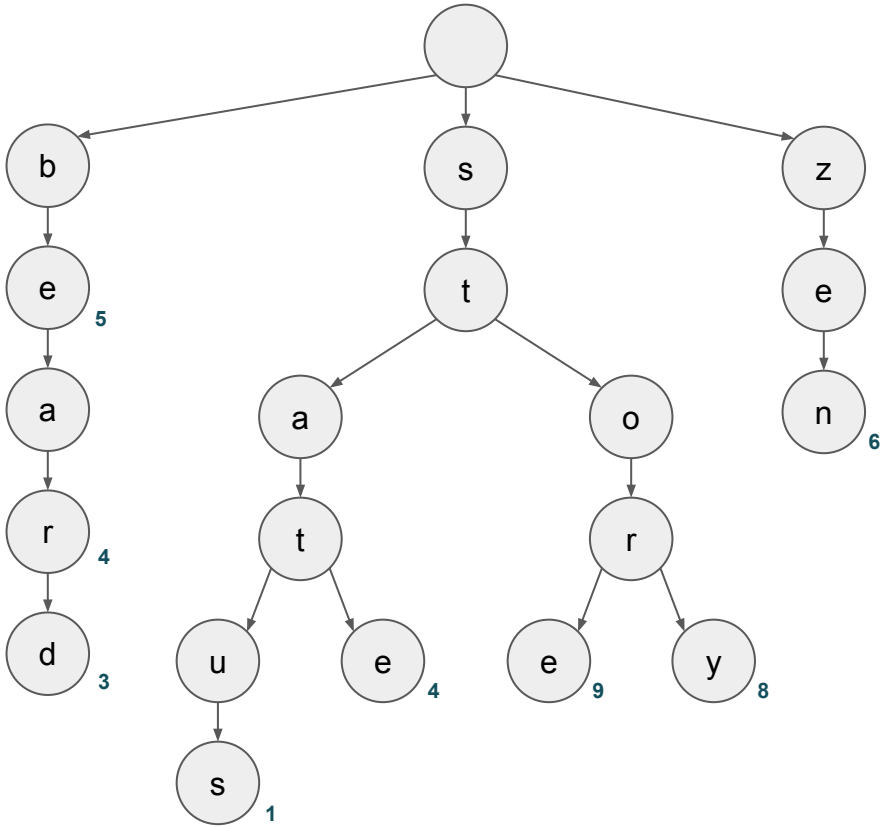
Вставка в префіксне дерево

- `put(“bear”, 4)`
- В дереві шлях “bear” вже існує
- В останньому вузлі цього шляху додаємо значення “4”



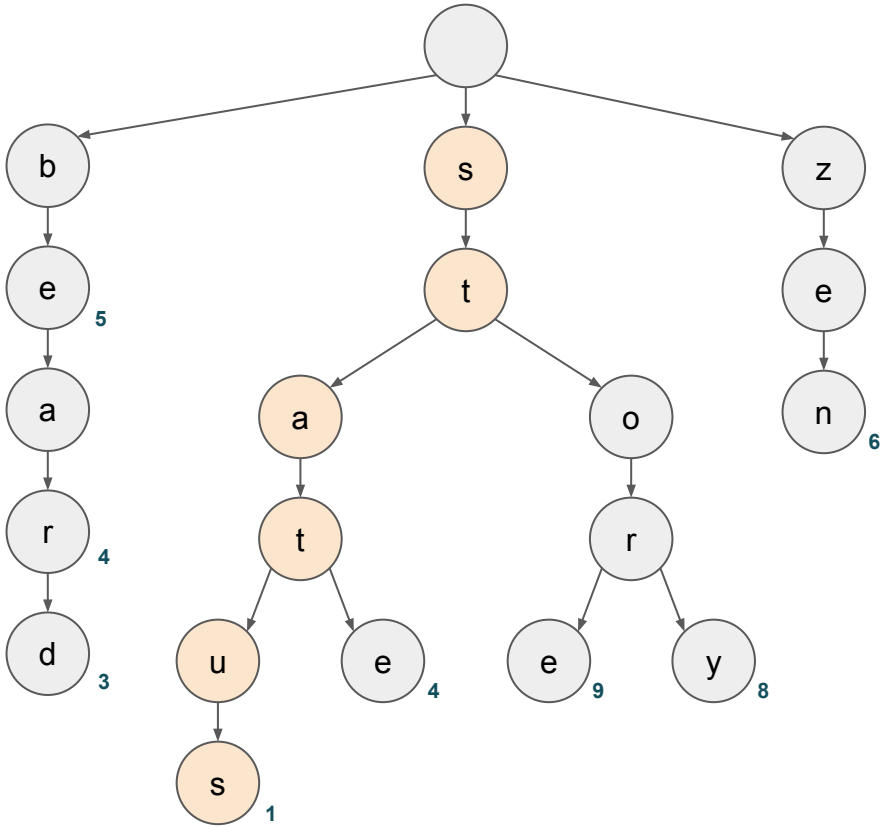
Видалення з префіксного дерева

- `remove("status")`



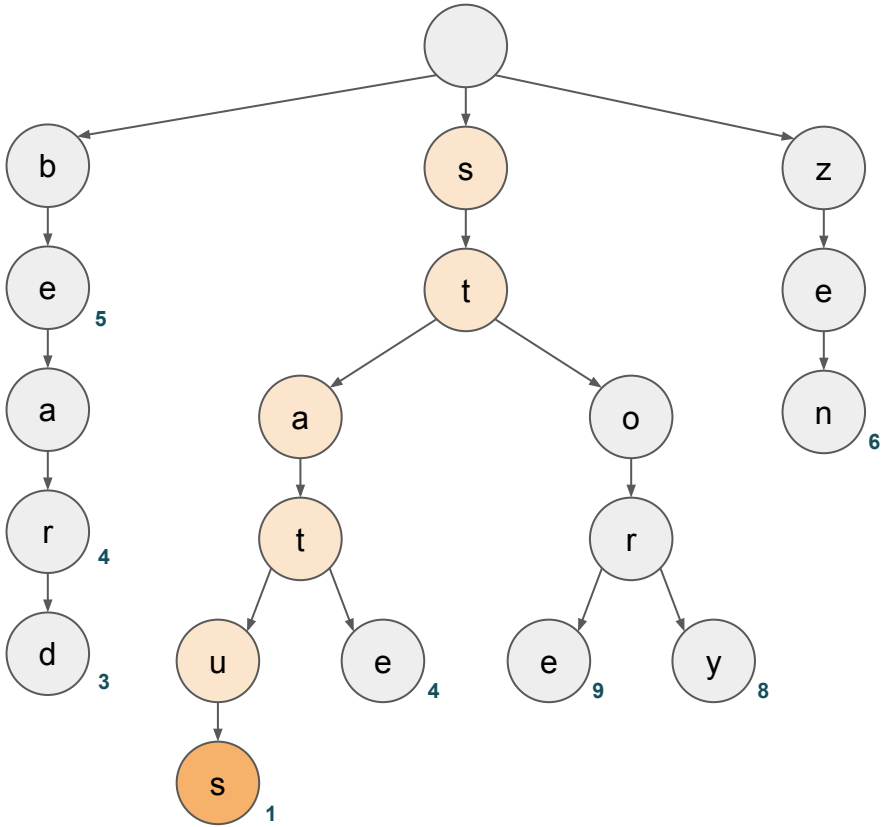
Видалення з префіксного дерева

- `remove("status")`
- Знаходимо останній вузол шляху “status”



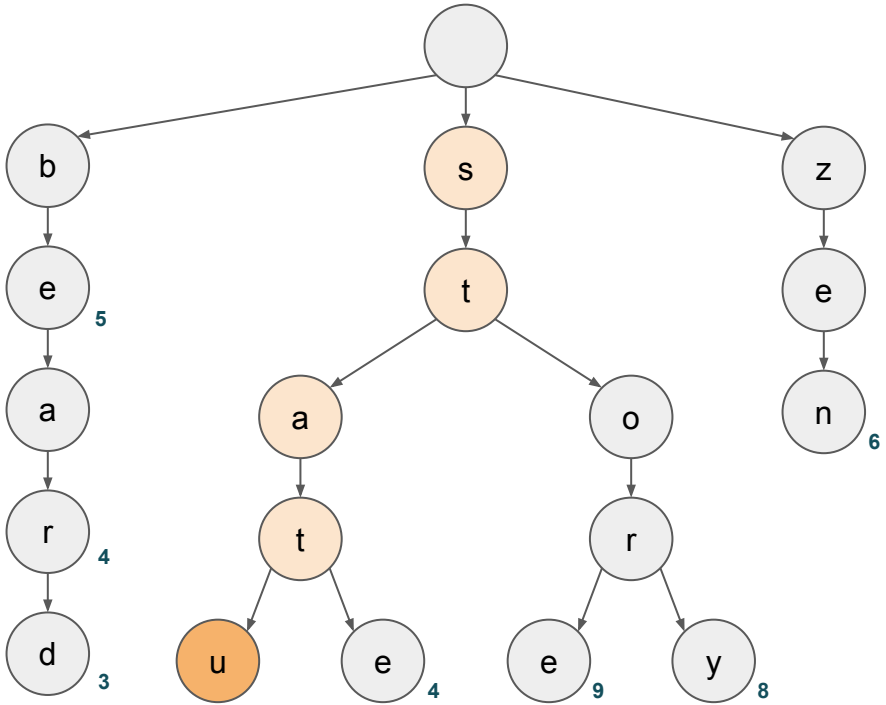
Видалення з префіксного дерева

- `remove("status")`
- Знаходимо останній вузол шляху “status”
- Видаляємо вузол “s”, так як в нього немає посилань на інші вузли



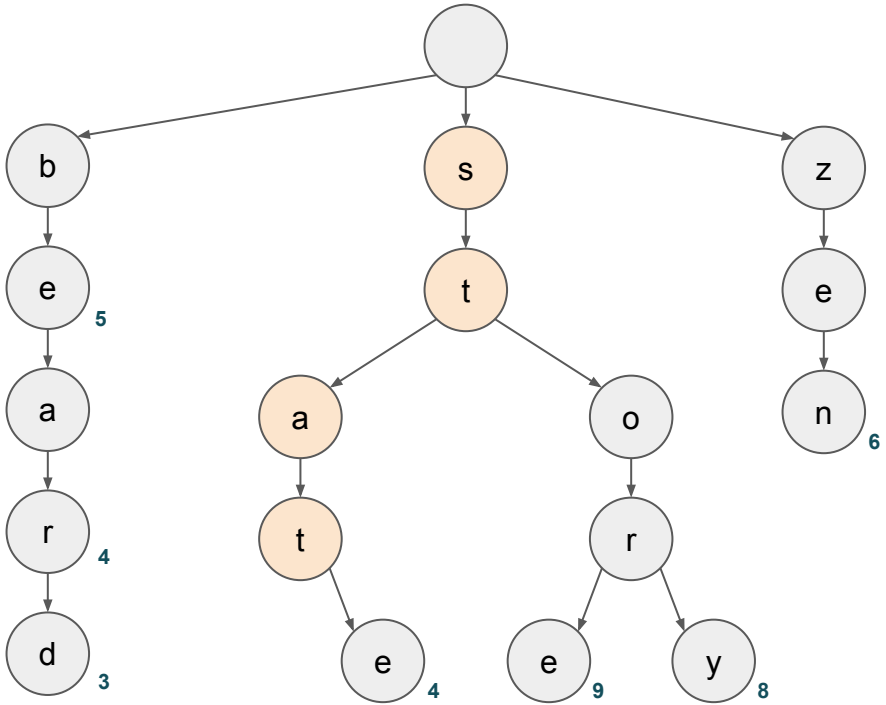
Видалення з префіксного дерева

- `remove("status")`
- Знаходимо останній вузол шляху “status”
- Видаляємо вузол “s”, так як в нього немає посилань на інші вузли
- Видаляємо вузол “u”, так як в ньому немає ні значення, ні посилань на інші вузли



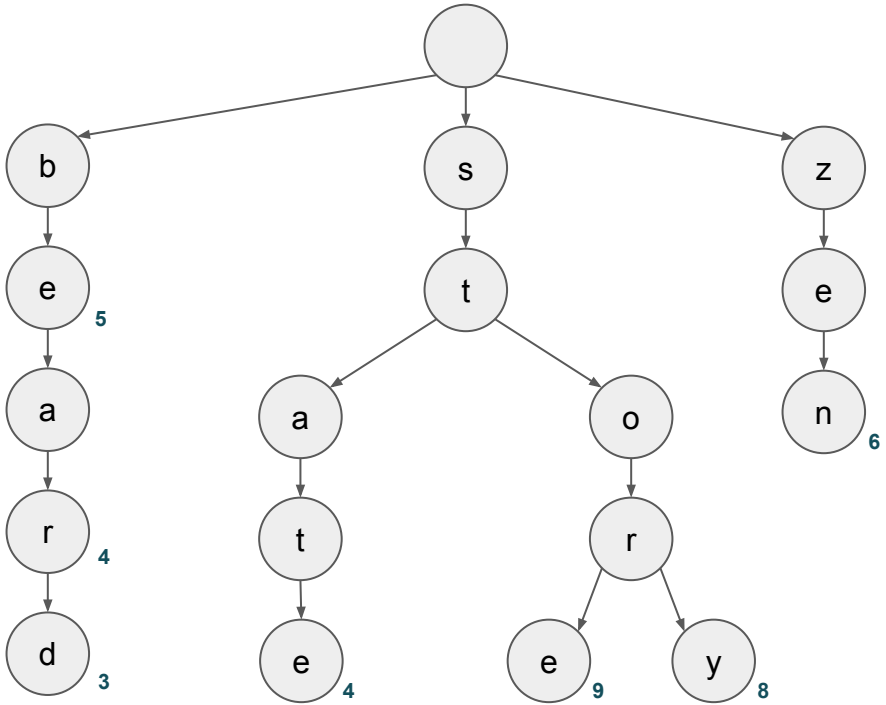
Видалення з префіксного дерева

- `remove("status")`
- Знаходимо останній вузол шляху “status”
- Видаляємо вузол “s”, так як в нього немає посилань на інші вузли
- Видаляємо вузол “u”, так як в ньому немає ні значення, ні посилань на інші вузли
- У вузлі “t” немає значення, але є посилання на інший вузол, тому на ньому і зупиняємось



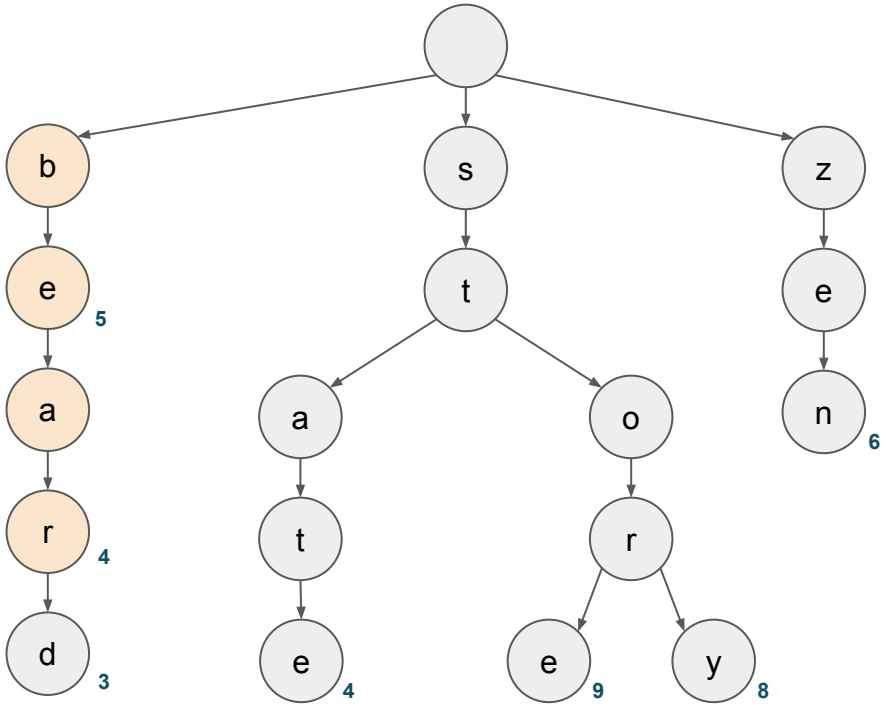
Видалення з префіксного дерева

- `remove("bear")`



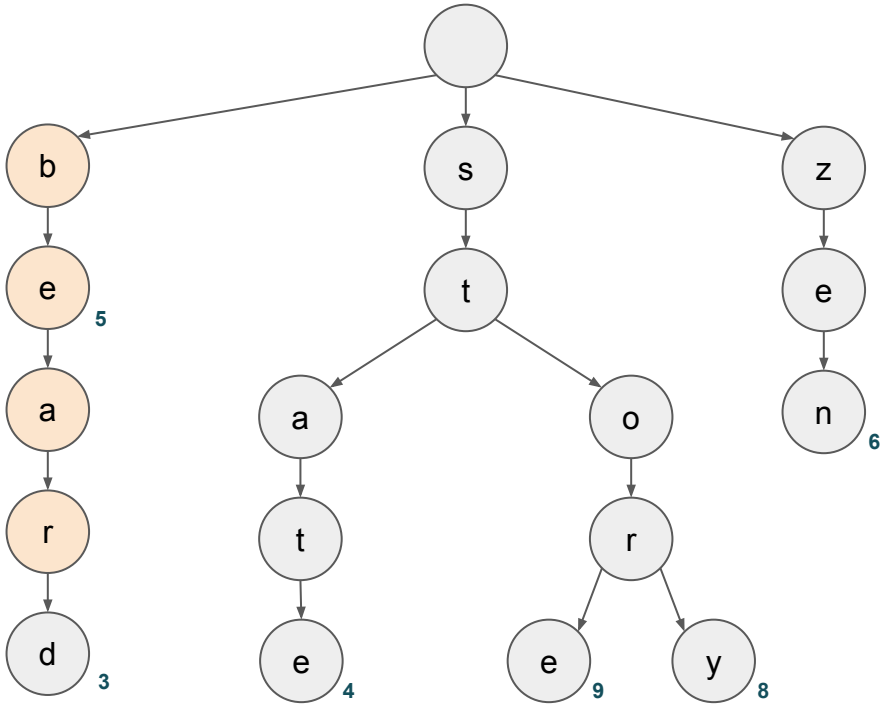
Видалення з префіксного дерева

- `remove("bear")`
- Знаходимо останній вузол шляху “bear”



Видалення з префіксного дерева

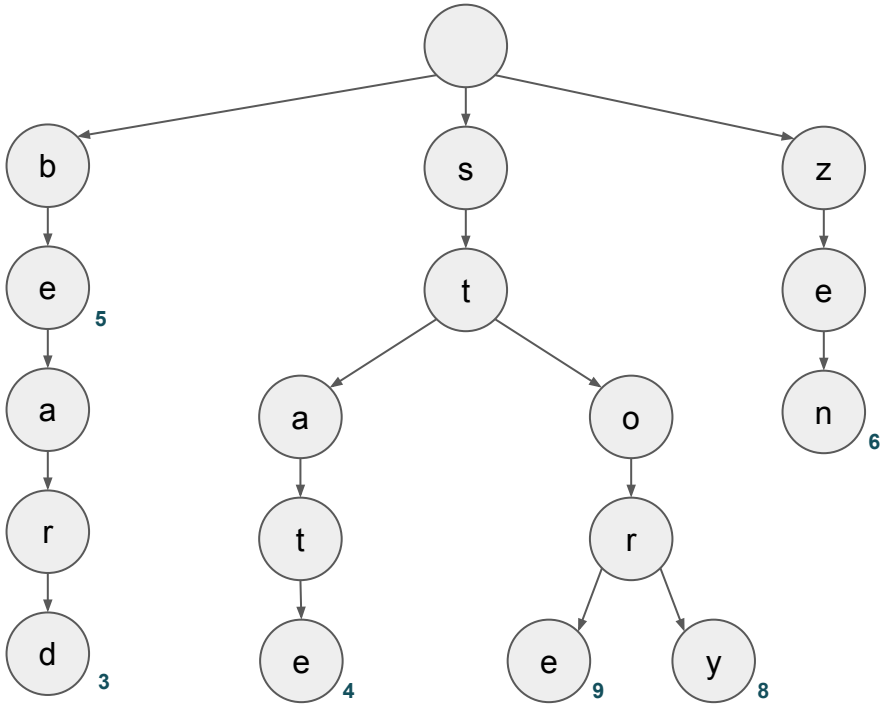
- `remove("bear")`
- Знаходимо останній вузол шляху “bear”
- Видаляємо з вузла “r” значення, але сам вузол залишається, так як в нього є посилання на інший вузол



Префіксне дерево

- Після виконаних дій, пари “ключ-значення” виглядають так:

Ключі	Значення
be	5
beard	3
state	4
store	9
story	8
zen	6



Реалізація на Python

- В класі `Node` (вузол) ми будемо зберігати значення і посилання на наступні `R` вузлів
- Кожен вузол також асоціюється з певною літерою, але вона явним чином не зберігається. Її можна визначити за індексом в масиві `next` вузла-предка

```
class Node:
    def __init__(self):
        self.value = None
        self.next = [None] * Trie.R
```

Реалізація на Python

```
class Trie:
    R = 26  # lower case latin letters

    def __init__(self):
        self.root = Node()

    def put(self, key, value):
        self.__put(self.root, key, value, 0)

    def __put(self, node, key, value, d):
        if node is None:
            node = Node()
        if d == len(key):
            node.value = value
            return node
        index = ord(key[d]) - ord("a")
        node.next[index] = self.__put(node.next[index], key, value, d + 1)
        return node
```

Реалізація на Python

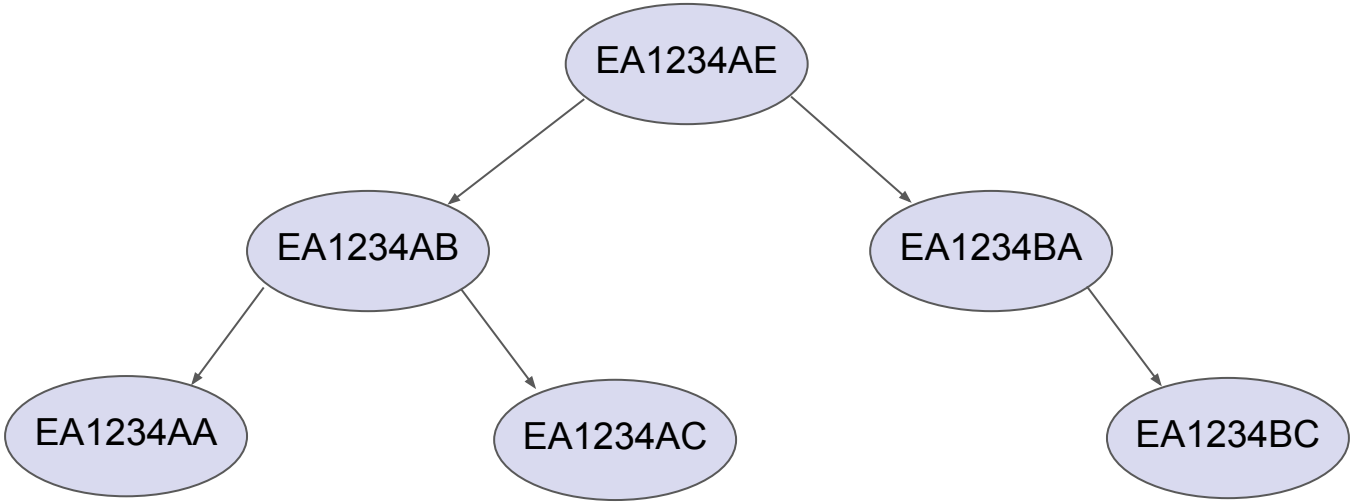
```
class Trie:

    # __init__(), put()

    def get(self, key):
        return self.__get(self.root, key, 0)

    def __get(self, node, key, d):
        if node is None:
            return None
        if d == len(key):
            return node.value
        index = ord(key[d]) - ord("a")
        return self.__get(node.next[index], key, d + 1)
```

Складність по часу і пам’яті

- Якщо шуканий ключ наявний в дереві, то складність буде дорівнювати L, так як необхідно розглянути всі символи ключа
- Якщо шуканий ключ відсутній, то часто вже перші символи можуть бути відсутніми в дереві, але в найгіршому випадку складність буде також L
- В кожному листку буде R пустих посилань, в проміжних вузлах їх також буде багато. Внаслідок цього дарма витрачається багато пам’яті
- В цілому, префіксне дерево дозволяє робити швидкий пошук, але потребує багато пам’яті. Крім цього, воно також підтримує порівняльні операції

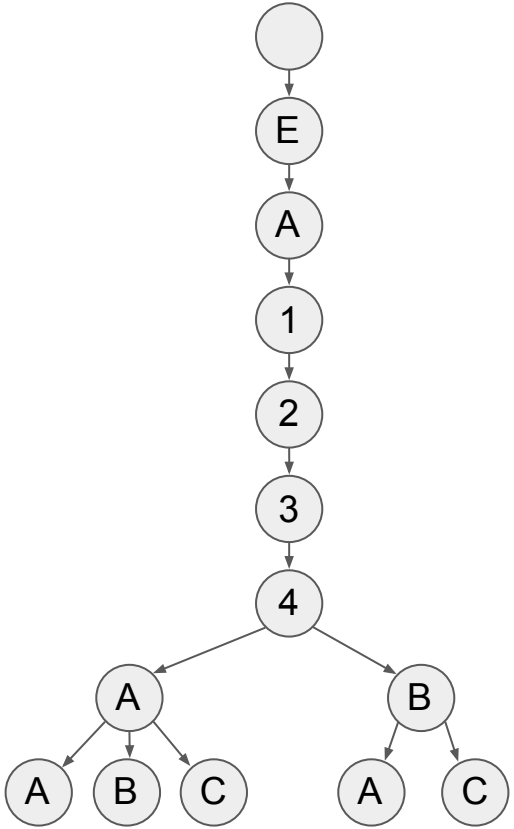
Повернемось до прикладу з бінарним деревом

- Ключ - номерний знак автомобіля, значення - інформація про авто



Побудуємо префіксне дерево

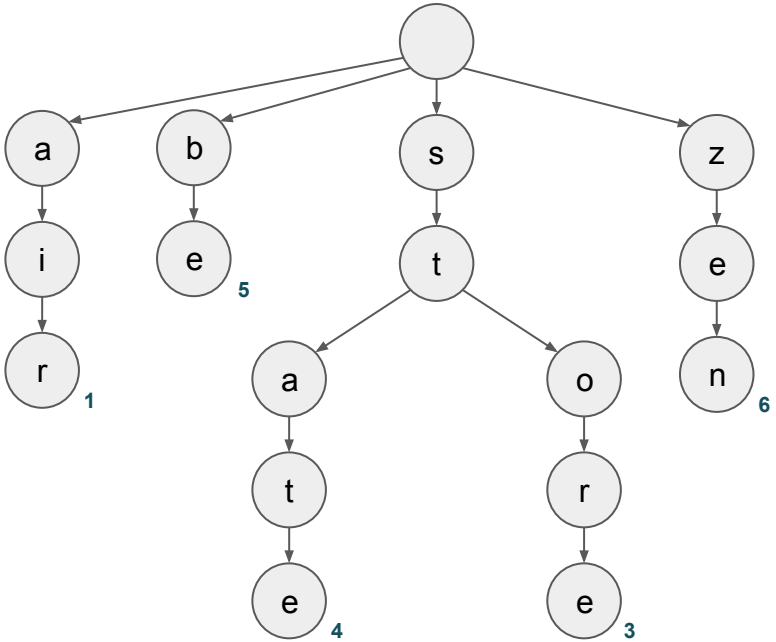
- Складність операції `get(“EA1234AC”)` ~ L

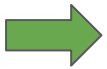


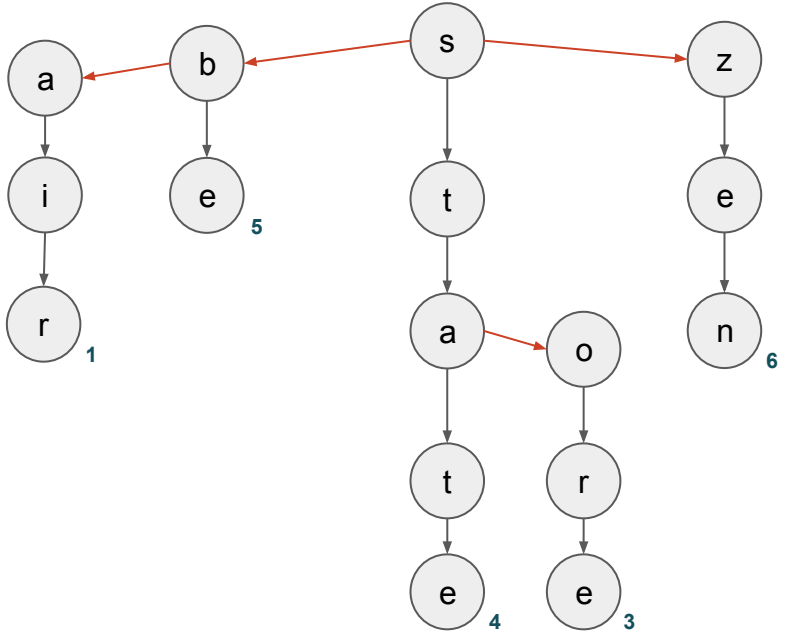
Тернарне префіксне дерево

- В кожному вузлі зберігається символ, значення і три посилання на інші вузли:
  - `left` - в нього потрібно перейти, якщо поточний символ ключа менший за символ у вузлі
  - `middle` - в нього потрібно перейти, якщо поточний символ ключа дорівнює символу у вузлі
  - `right` - в нього потрібно перейти, якщо поточний символ ключа більший за символ у вузлі

Тернарне префіксне дерево







а) Префіксне дерево

б) Тернарне префіксне дерево

Складність по часу і пам’яті

- В тернарному префіксному дереві в загальному випадку складність по часу буде пропорційна логарифму від кількості елементів:
  - *~ L + log(N)*
- Але воно є доволі оптимізованим з точки зору використання пам’яті

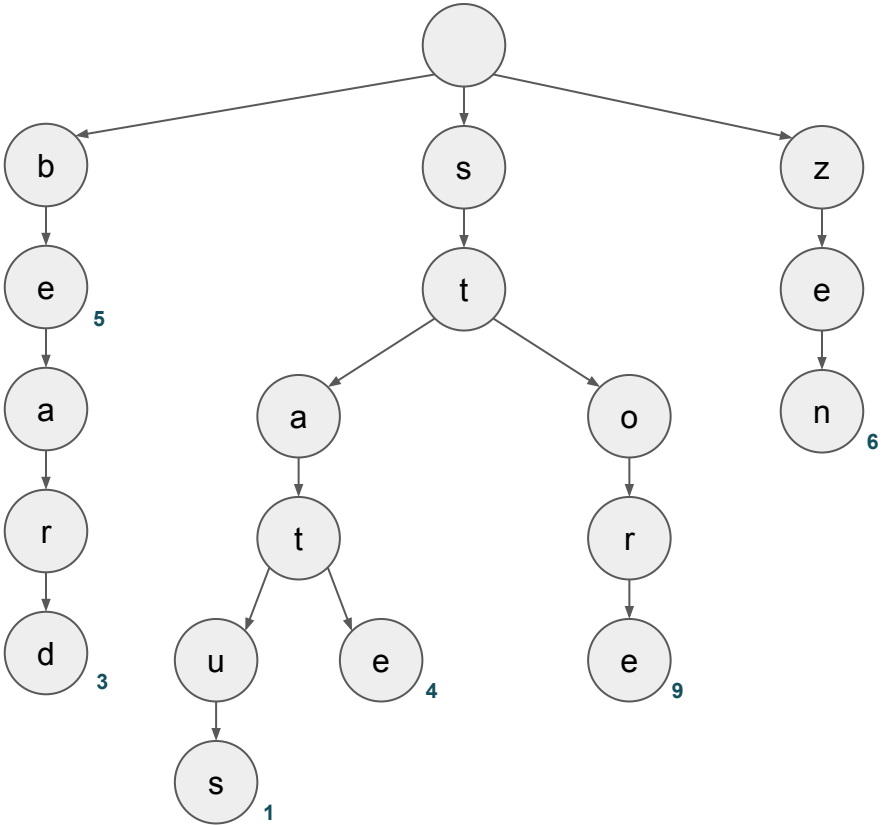
Гібридне префіксне дерево

- Можна також спробувати об’єднати звичайне префіксне дерево з тернарним, наприклад таким чином:
  - На 2-3 верхніх рівнях використовувати звичайне префіксне дерево
  - На нижні рівня використовувати тернарне префіксне дерево

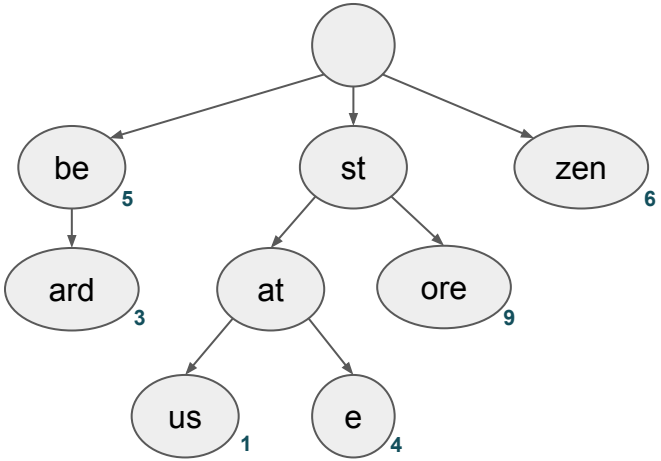
Базисне дерево

- Базисне дерево (англ. radix trie, compact prefix tree) - є оптимізованим по пам’яті варіантом префіксного дерево
- В базисному дереві, якщо у вузла є тільки один нащадок - то такі два вузла зливаються

Базисне дерево



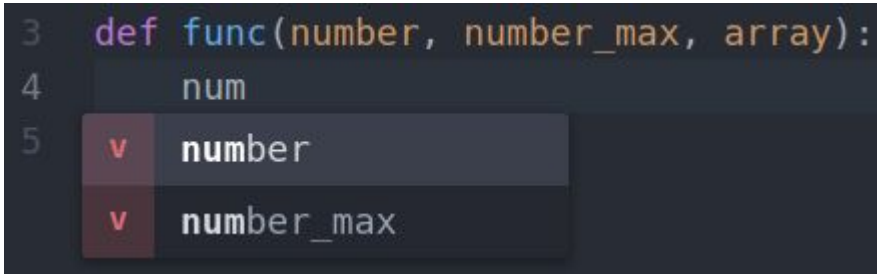
а) Префіксне дерево

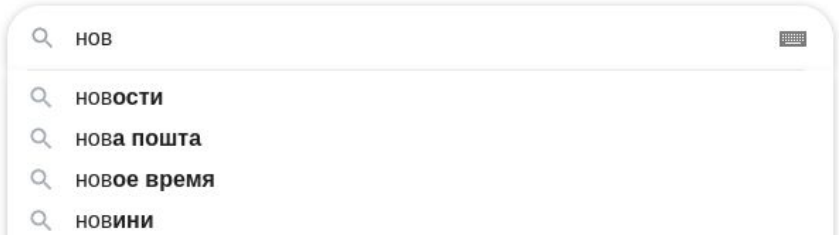


б) Базисне дерево

Використання префіксних дерев

- Для автодоповнення в текстових редакторах
- В маршрутизаторах для визначення найдовшого префікса в IP-адресах
- В базах даних
- В реалізаціях файлових систем





Посилання на матеріали

1. https://prometheus.org.ua/cs50/week6.html#trees_and_tries
2. https://www.coursera.org/lecture/algorithms-part2/r-way-tries-CPVdr
3. https://www.coursera.org/lecture/algorithms-part2/ternary-search-tries-yQM8K
4. https://www.coursera.org/lecture/algorithms-part2/character-based-operations-jwNmV
5. https://uk.wikipedia.org/wiki/%D0%91%D0%B0%D0%B7%D0%B8%D1%81%D0%BD%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE