Trie Data Structure

Drzewa przypomnienie

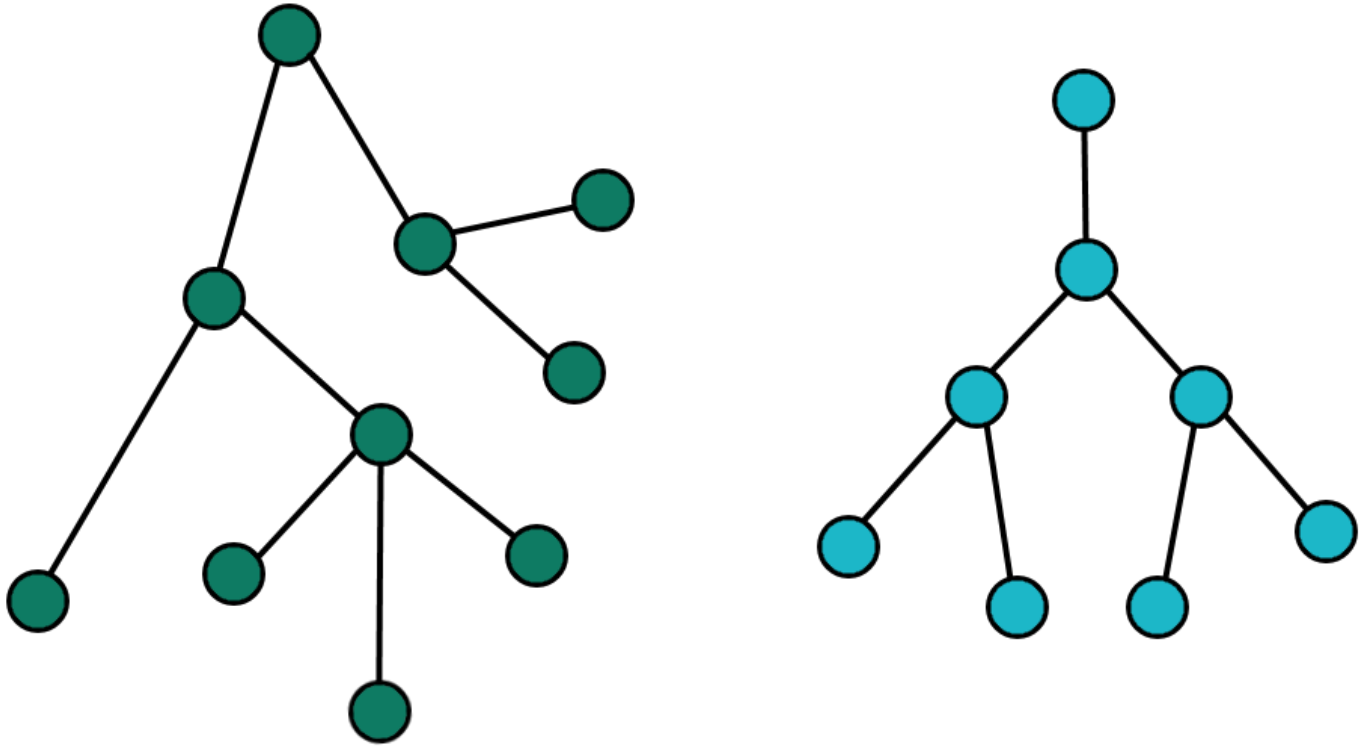
Definicja:

- Struktura danych w postaci grafu acyklicznego, z jednym wyróżnionym węzłem – **korzeniem (root)**.
- Każdy węzeł (poza korzeniem) ma dokładnie **jednego rodzica** i dowolną liczbę dzieci.

Drzewa przypomnienie

- **Ważne pojęcia:**
  - **Korzeń (Root):** pierwszy węzeł drzewa
  - **Liść (Leaf):** węzeł bez dzieci
  - **Krawędź:** połączenie między rodzicem a dzieckiem
  - **Głębokość (Depth):** odległość od korzenia
  - **Wysokość (Height):** maksymalna głębokość drzewa
  - **Poddrzewo (Subtree):** drzewo zakorzenione w dowolnym węźle

Drzewa przypomnienie



Czym jest Trie?

Trie to specjalna struktura danych służąca do przechowywania **zbioru łańcuchów znaków** (np. słów, fraz, ciągów).

Znana również jako **Prefix Tree** – drzewo prefiksowe.

Umożliwia szybkie operacje takie jak:

- Wstawianie słowa
- Wyszukiwanie pełnego słowa
- Wyszukiwanie słów na podstawie **prefiksu**

Etymologia – skąd nazwa Trie?

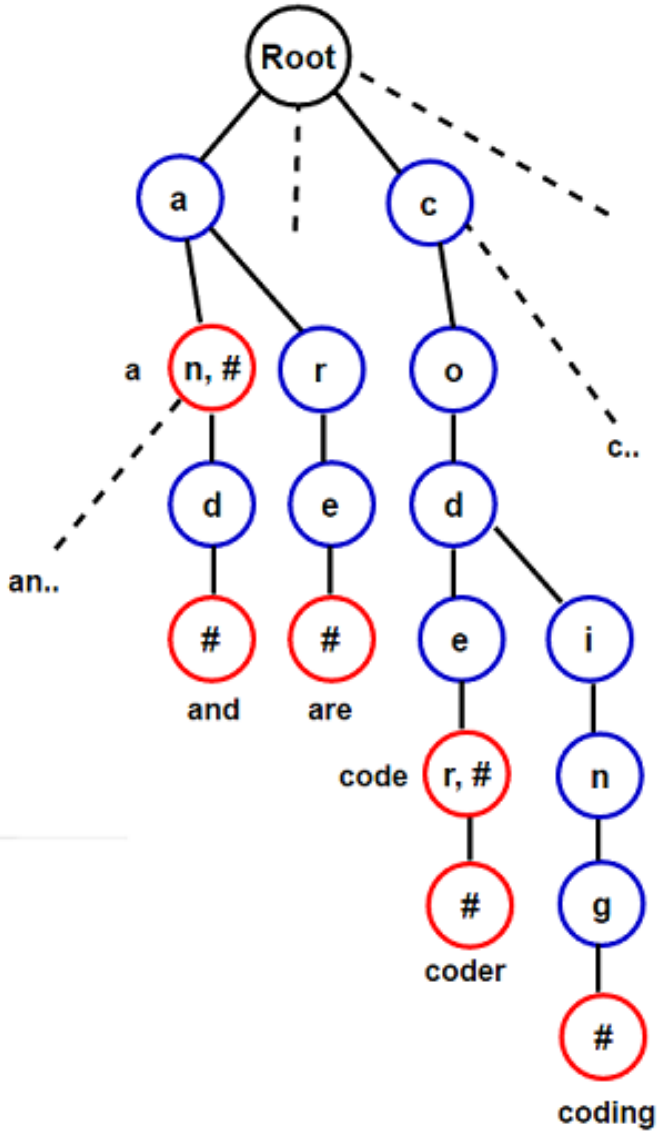
- Nazwa **„trie”** pochodzi od słowa **„retrieval”** (ang. pobieranie, wyszukiwanie), co odzwierciedla główne zastosowanie tej struktury – wydajne wyszukiwanie ciągów znaków.
- Została zaproponowana przez **Edwarda Fredkina** w 1960 roku, który wymyślił termin trie.
- **Wymowa**: Chociaż pisownia sugeruje wyraz „tree”, autor pierwotnie wymawiał ją jako */tri:/* (jak "retrieval"), jednak w praktyce często używa się wymowy */traɪ/* (jak „try”).
- Trie nazywana jest także **„prefix tree”** (drzewo prefiksowe), ponieważ reprezentuje dane w formie drzewa opartego na wspólnych prefiksach.

Czym jest prefix i suffix?

- Prefix to dowolnie długi ciąg kolejnych n liter naszego słowa takich, że n < S
  np. Dla słowa drzewo prefixami będą:
  -d, -dr, -drz, -drze, -drzew, -drzewo
- Suffix to analogicznie dowolnie długi ciąg kolejnych n ostatnich liter naszego słowa
  np. Dla słowa drzewo:
  -o, -wo, -ewo, -zewo, -rzewo, -drzewo

Jak działa Trie?

- Trie to struktura danych grafu zbudowana z węzłów wskazujących na inne węzły.
- Każdy **węzeł drzewa** odpowiada jednej **literze**.
- Ścieżka od korzenia do liścia tworzy **słowo**.
- Wspólne prefiksy są przechowywane **wspólnie** – bez powtórzeń.
- Przykład (dla słów *and, are, coder, coding*)



Drzewo trie w różnych językach

- Żaden język nie posiada bazowo zaimplementowanego tego drzewa, ale nadrabiają to odpowiednie biblioteki
- **Python - Biblioteka:** `marisa-trie`
- **Java - Biblioteka:** `org.ahocorasick.trie`
- **C++ - Biblioteka:** `CEDAR`
- **Rust - Biblioteka:** `trie-rs`

Ale jak to działa pod spodem?

- Zaimplementujmy wspólnie drzewo Trie w Pythonie!
- Operacje które będą nas interesować:
  -Stworzenie nowego drzewa
  -Przeszukiwanie(Search)
  -Dodawanie nowych danych(Add)
  -Usuwanie niepotrzebnych ciągów(Remove)

Stworzenie nowego drzewa

Definiujemy klasę TrieNode, której każdy nowy węzeł będzie jej instancją

```
class TrieNode:
    def __init__(self, val=None):
        self.val = val
        self.children = dict()
        self.is_end = False
```

Ustawiamy korzeń który jest bazą naszej struktury

```
class Trie:

    def __init__(self):
        self.rootNode = TrieNode()
```

Dodanie nowych danych

- Złożoność czasowa O(k) k - długość słowa

```python
def add(self, word: str) -> None:
    """
    insert word into the trie
    """
    curr_node = self.rootNode

    for char in word:
        if char not in curr_node.children:
            curr_node.children[char] = TrieNode(char)

        curr_node = curr_node.children[char]

    curr_node.is_end = True
```

Przeszukiwanie

- Złożoność czasowa O(k) k - długość słowa

```python
def search(self, word: str) -> bool:
    """
    check if word exists in the trie
    """
    curr_node = self.rootNode

    for char in word:
        if char not in curr_node.children:
            return False

        curr_node = curr_node.children[char]

    return curr_node.is_end
```

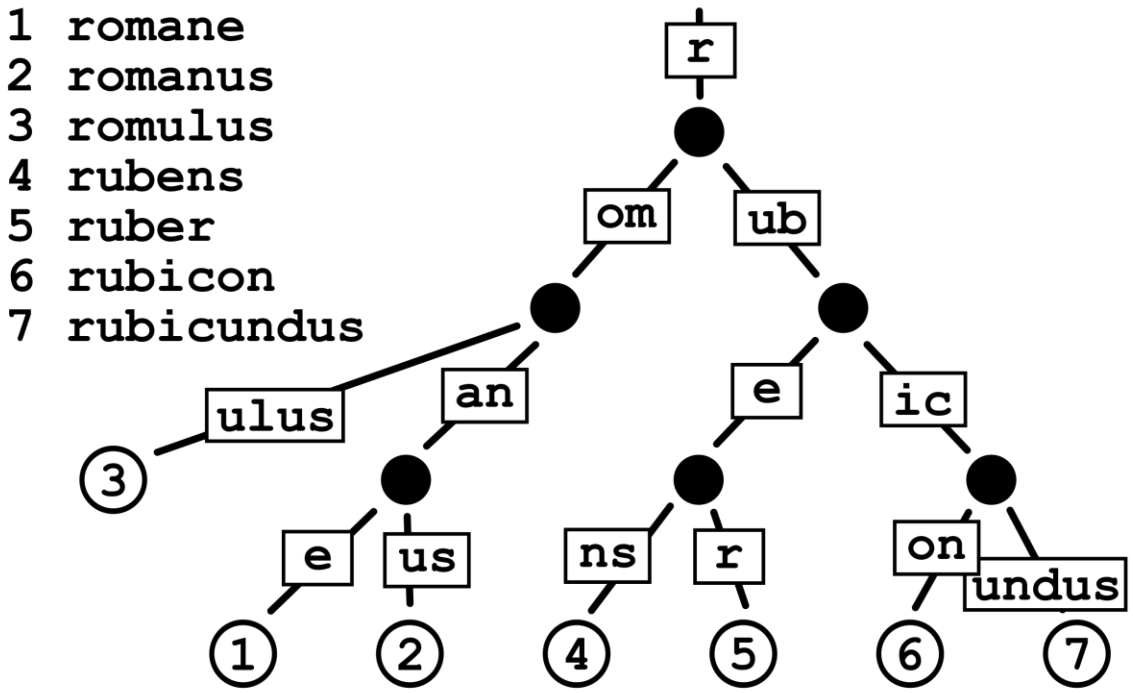
Usuwanie danych

- Złożoność czasowa O(k) k - długość słowa

```python
def remove(self, word: str) -> None:
    """
    Remove word from the trie.
    """
    curr_node = self.rootNode
    branch = [curr_node]

    for char in word:
        curr_node = curr_node.children[char]
        branch.append(curr_node)

    curr_node.is_end = False

    n = len(branch)
    for i in range(n-2, -1):
        j = i + 1
        if not branch[j].children:
            del branch[i].children[branch[j].val]

    return None
```

Drzewo Radix

- Radix tree, znane też jako compressed trie lub patricia tree, to struktura danych będąca zmodyfikowaną wersją trie, w której ciągi znaków z pojedynczymi dziećmi są łączone w jeden węzeł.



Zastosowania Trie

- Autouzupełnianie (autocomplete)
- Wyszukiwarki słów i fraz
- Korekta pisowni
- Kompresja danych (np. słowniki Huffmana)
- Implementacja słowników w edytorach tekstu
- Analiza DNA, kodów genetycznych

Trie w autouzupełnianiu (Autocomplete)

Tries są powszechnie wykorzystywane do implementacji mechanizmów autouzupełniania w edytorach tekstu, wyszukiwarkach internetowych i innych aplikacjach, w których użytkownicy muszą szybko znaleźć i wybrać jedną z wielu możliwych opcji.

Dzięki przechowywaniu słownika możliwych słów w strukturze Trie:

- System może błyskawicznie przeszukać strukturę pod kątem prefiksu,
- Zwrócić wszystkie możliwe uzupełnienia zaczynające się od danego ciągu znaków,
- Wydajność nie zależy od liczby słów, a od długości wpisanego prefiksu.

Przykład: Użytkownik wpisuje: **"har"**
System sugeruje: **"Harry", "Harper", "Haruki", "Hardy"**

Trie w sprawdzaniu pisowni (Spell Checking)

Tries mogą być także używane do implementacji algorytmów sprawdzających poprawność pisowni. Dzięki strukturze drzewa zawierającej zbiór poprawnych słów:

Możemy:

- Błyskawicznie sprawdzić, czy dane słowo znajduje się w słowniku,
- Identyfikować literówki i błędy na podstawie nieznalezionych słów,
- Wyszukiwać podobne słowa poprzez algorytmy różnicy edycyjnej (Levenshtein distance, fuzzy search).

Przykład: Wpisano: **"bananna"**
Trie nie zawiera takiego słowa → system sugeruje poprawkę: **"banana"**

Trie w kompresji Huffmana

Huffman coding to algorytm kompresji danych, który przypisuje **krótsze kody binarne** znakom występującym **częściej** w tekście, a dłuższe — tym **rzadziej** spotykanym.

Struktura **Trie** (drzewo prefiksowe) może być użyta do:

- reprezentowania **rozkładu częstości występowania znaków**,
- budowy **drzewa kodowania Huffmana**, gdzie każdy liść reprezentuje znak,
- szybkiego kodowania i dekodowania danych tekstowych.

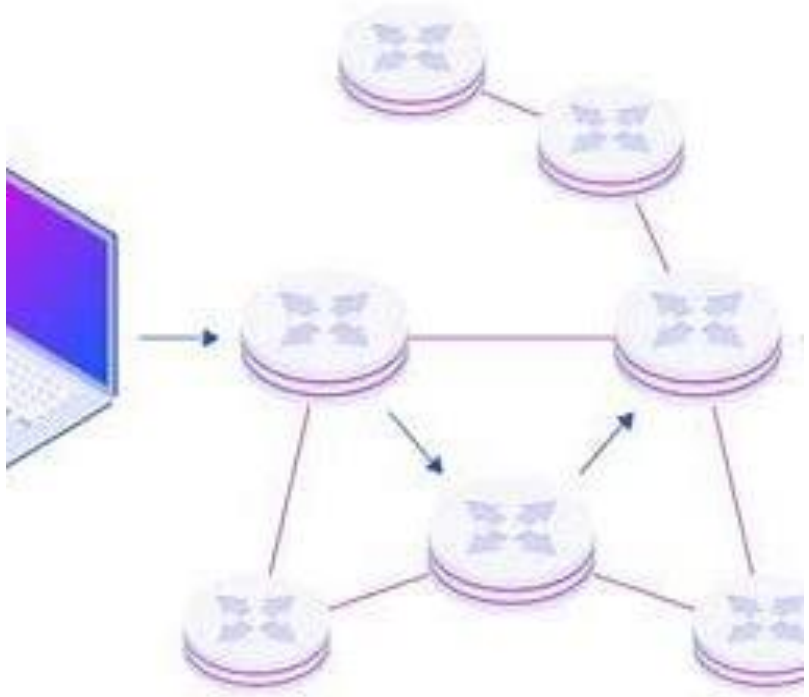
Efekt: Zmniejszenie rozmiaru danych przy zachowaniu możliwości pełnego odtworzenia oryginalnej treści.

Trie w IP Routing

IP Routing – to proces określania ścieżki, jaką powinny przebyć pakiety danych w sieci, aby dotrzeć do docelowego urządzenia.

Rola Trie w IP Routing:

- Przechowywanie adresów IP
- Proces wyszukiwania (lookup)
- Określanie ścieżki (next hop)
- Zarządzanie tablicą routingu



Trie vs Lista / Tablica

Lista / Tablica:

- Szukanie pasujących słów (np. z prefiksem `"ca"`) wymaga przeszukiwania całej listy.
- Złożoność czasowa: **O(n * m)** gdzie n to liczba słów, `m` to długość prefiksu.
- Działa dobrze dla małej liczby słów.

Trie:

- Każdy znak prowadzi do kolejnego poziomu drzewa.
- Szukanie prefiksu: **O(m)**, niezależnie od liczby słów!
- Doskonałe do dynamicznego uzupełniania na żywo.

Wniosek: Trie jest znacznie szybsze przy dużej liczbie słów i krótkich prefiksach.

Trie vs Słownik (dict / hash map)

Słownik:

- Szybki dostęp: **O(1)** dla dokładnego dopasowania (np. `"car"`).
- Ale nie wspiera efektywnego **wyszukiwania po prefiksie** – trzeba iterować po wszystkich kluczach.
- Brak logicznego uporządkowania.

Trie:

- Naturalnie wspiera wyszukiwanie po prefiksie.
- Można łatwo znaleźć wszystkie słowa z danego zakresu (np. "ca...").
- Może być bardziej pamięciożerny niż dict.

Wniosek: Gdy potrzebujesz operować na częściach słów – Trie wygrywa z haszowaniem.

Złożoności czasowe

m – długość słowa/prefiksu, n – liczba słów, k – liczba wyników

Operacja	Lista	Słownik (dict)	Trie
Wstawianie słowa	O(1)	O(1)	O(m)
Dokładne wyszukanie	O(n)	O(1)	O(m)
Wyszukanie prefiksu	O(n*m)	O(n*m)	**O(m)**
Autouzupełnianie	O(n*m)	O(n*m)	**O(m + k)**

Materiały na Github

https://github.com/bbieda/pattern-searching-using-a-trie-of-all-suffixes

Zadania

Zadania znajdują się na github