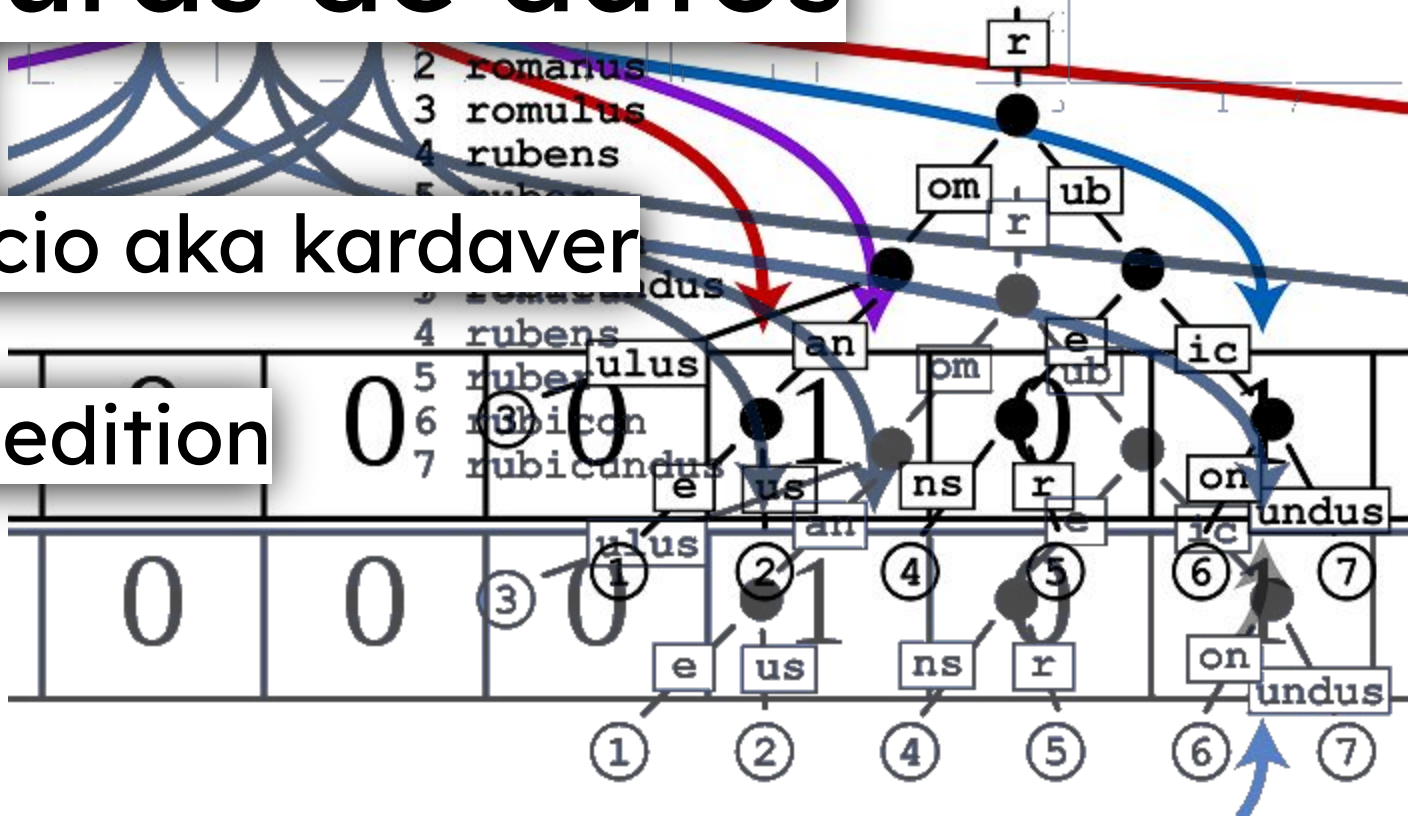
Estructuras de datos *extrañas*

Karen Palacio aka kardaver

2023

jr soft libre edition



```
s3.initCam()
//mouse.y*.003
function thing(){return .05}
src(s3)

//.modulate(src(s3).repeat(2),()=>thing())
.modulate(src(s3).scale(1.2),()=>thing())

.modulate(src(s3).scale(1.6),()=>thing())

.modulate(src(s3).scale(2),()=>thing())
//.diff(o0)
.out()
```

Karen Palacio

instagram.com/kardaver
karen.palacio.1994@gmail.com
https://github.com/karen-pal

AI Software Architect, Tech Lead @ Kunan SA

Programadora Industrial y Creativa

Artista Digital y live coder | VJ

Activista del Software Libre

personalmente,
afectivamente

Las conocí en las ECI

ECI 36 ESCUELA DE CIENCIAS INFORMÁTICAS

24 AL 28 DE JULIO 2023

Se certifica que Karen Araceli Palacio Pastor ha aprobado, con **diez (10) puntos sobre diez**, el curso

"M1: Advanced Data Structures."

de 15 horas de duración, dictado por el *Prof. Conrado Martínez Parra*

como parte de la Escuela de Ciencias Informáticas, entre el 24/07/2023 y el 28/07/2023
en el Departamento de Computación, Facultad de Ciencias Exactas y Naturales,
Universidad de Buenos Aires.

Dr. Sergio Abriola
Presidente de la ECI 2023

Dr. Pablo Barenbaum
Vice-presidente de la ECI 2023

Dr. Juan Pablo Galeotti
Director del Dto. de Computación

Departamento de Computación
Pabellón 0+i – Ciudad Universitaria – C1428EGA – Buenos Aires – Argentina

Red de conocimiento de universidades públicas

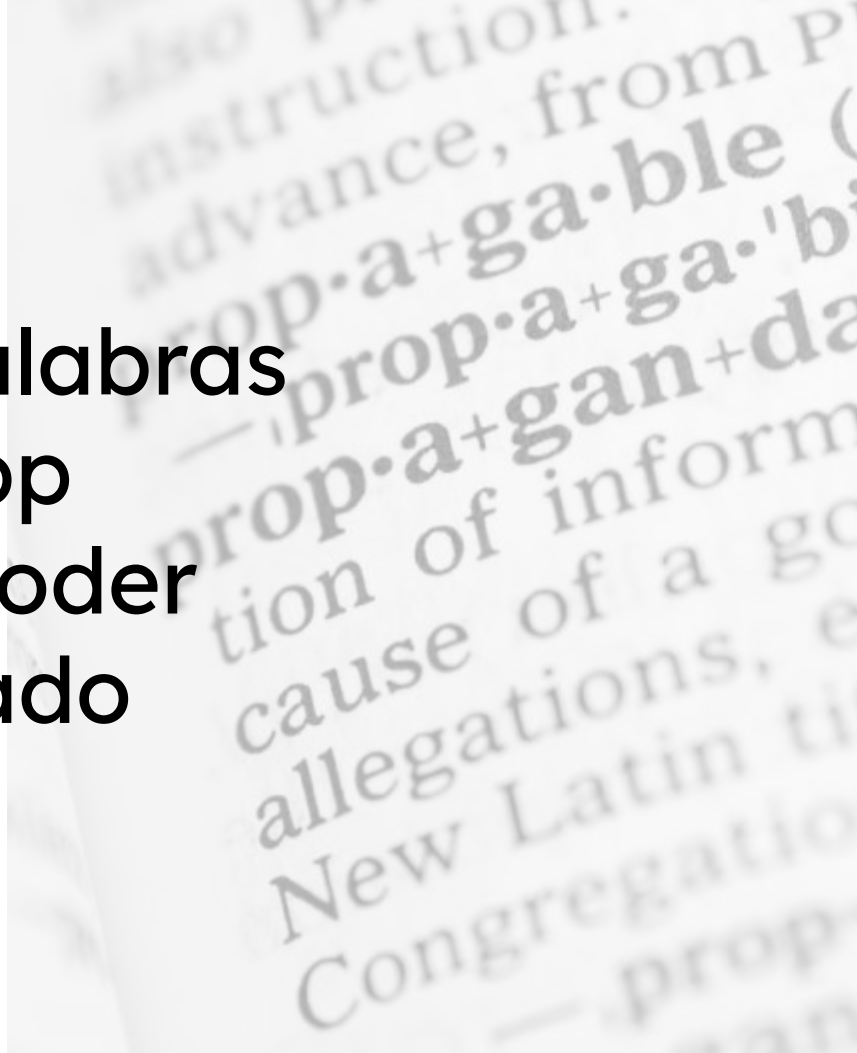
- Componente social de la programación
- Circulación del conocimiento es un derecho humano

JARDIN.HIJA.ALTURA

Mi alias para aquellos que no crean en la educación pública y en octubre voten por arancelar la educación

Voy a mostrar cómo funcionan y cómo se implementan dos estructuras de datos avanzadas: **Bloom Filters y Tries/prefix trees.**

imaginá:
tenemos bocha de palabras específicas en una app nuestra... queremos poder ofrecer autocompletado eficiente!



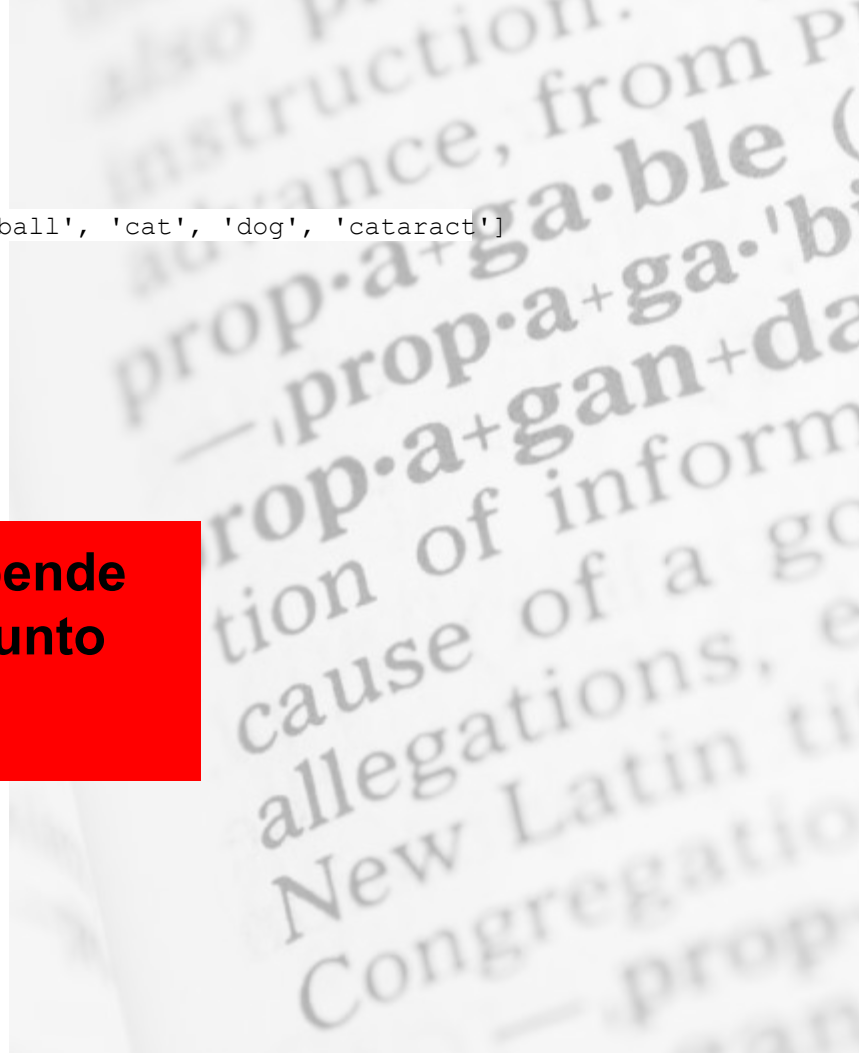
```
words = ['apple', 'apricot', 'banana', 'apartment', 'apex', 'ball', 'cat', 'dog', 'cataract']
```

```
fun naive_lookup(SEARCHED_WORD):
    for word in words:
        if SEARCHED_WORD == word:
            return True
    return False
```

```
words = ['apple', 'apricot', 'banana', 'apartment', 'apex', 'ball', 'cat', 'dog', 'cataract']
```

```
fun naive_lookup(SEACHED_WORD):
    for word in words:
        if SEARCHE
            return
    return False
```

coste lineal que depende del tamaño del conjunto de búsqueda

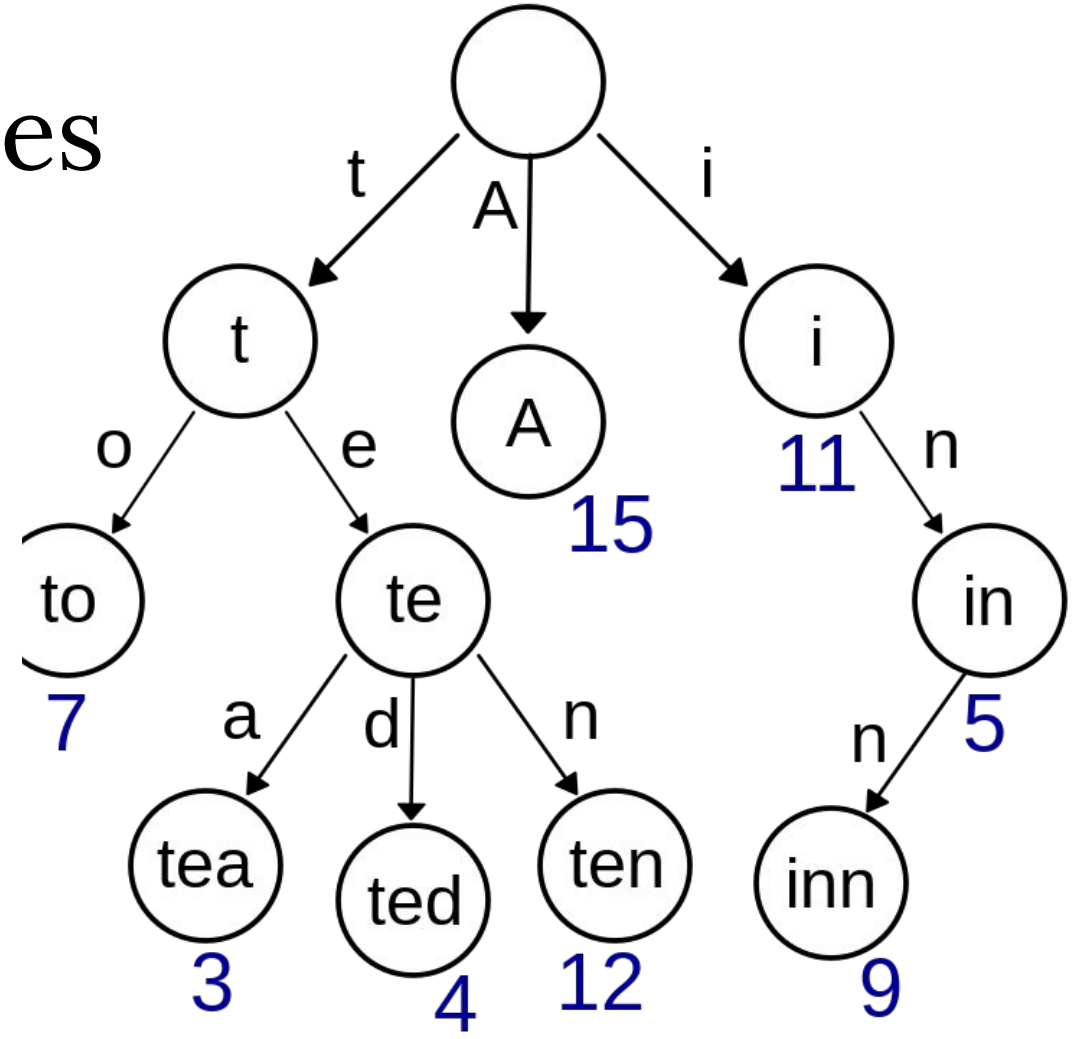


Tries/prefix trees

En vez de guardar palabra por palabra,

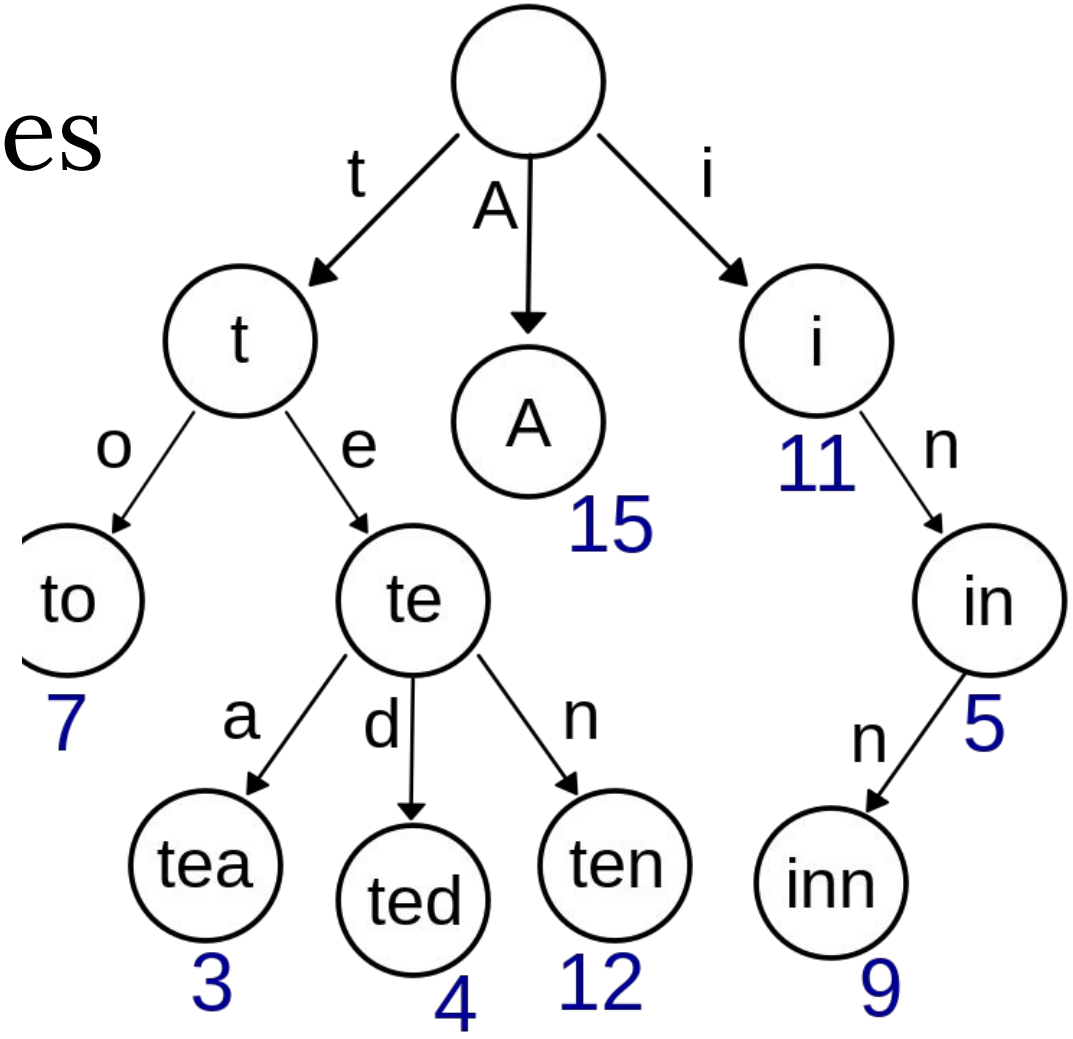
guardamos “átomos” de estos

de tal forma que **los prefijos comunes están implícitos en la estructura**

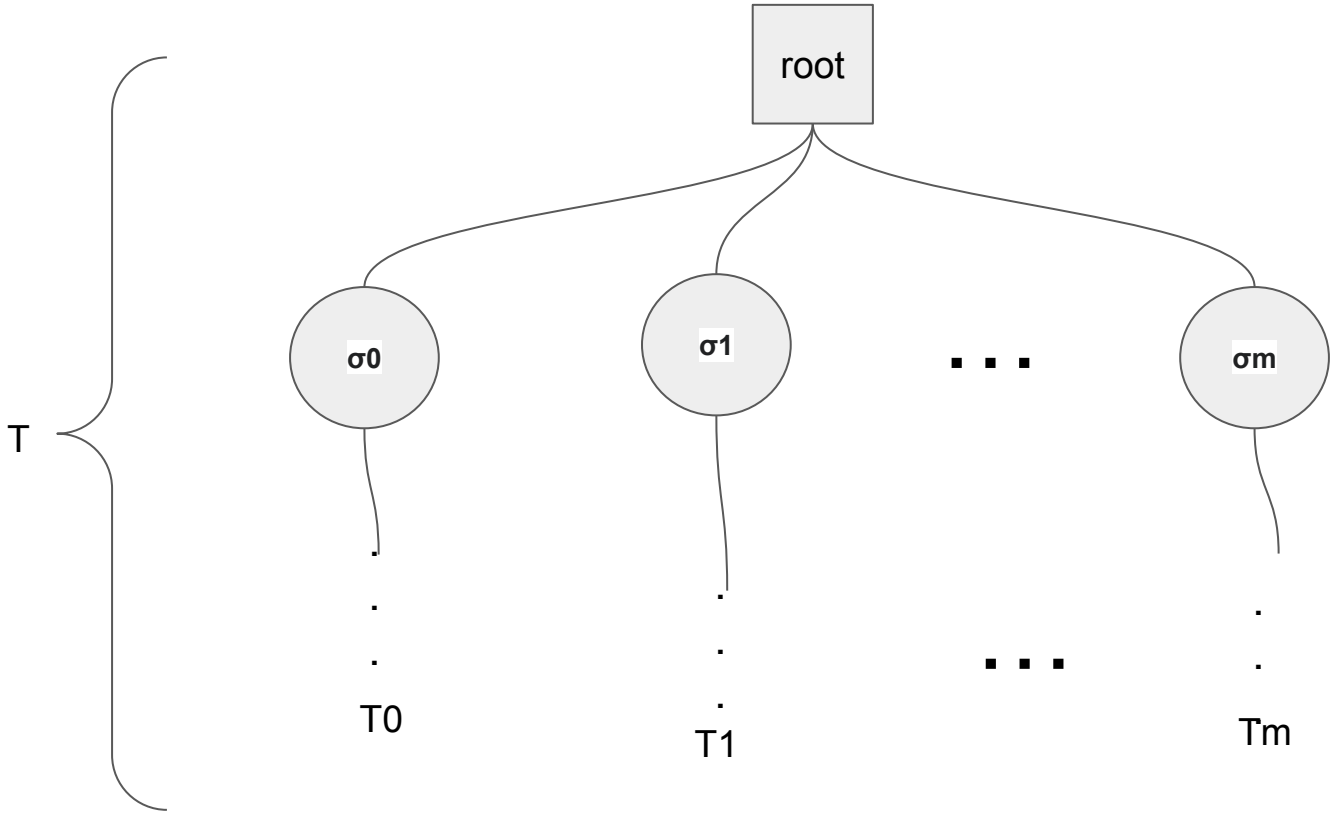


Tries/prefix trees

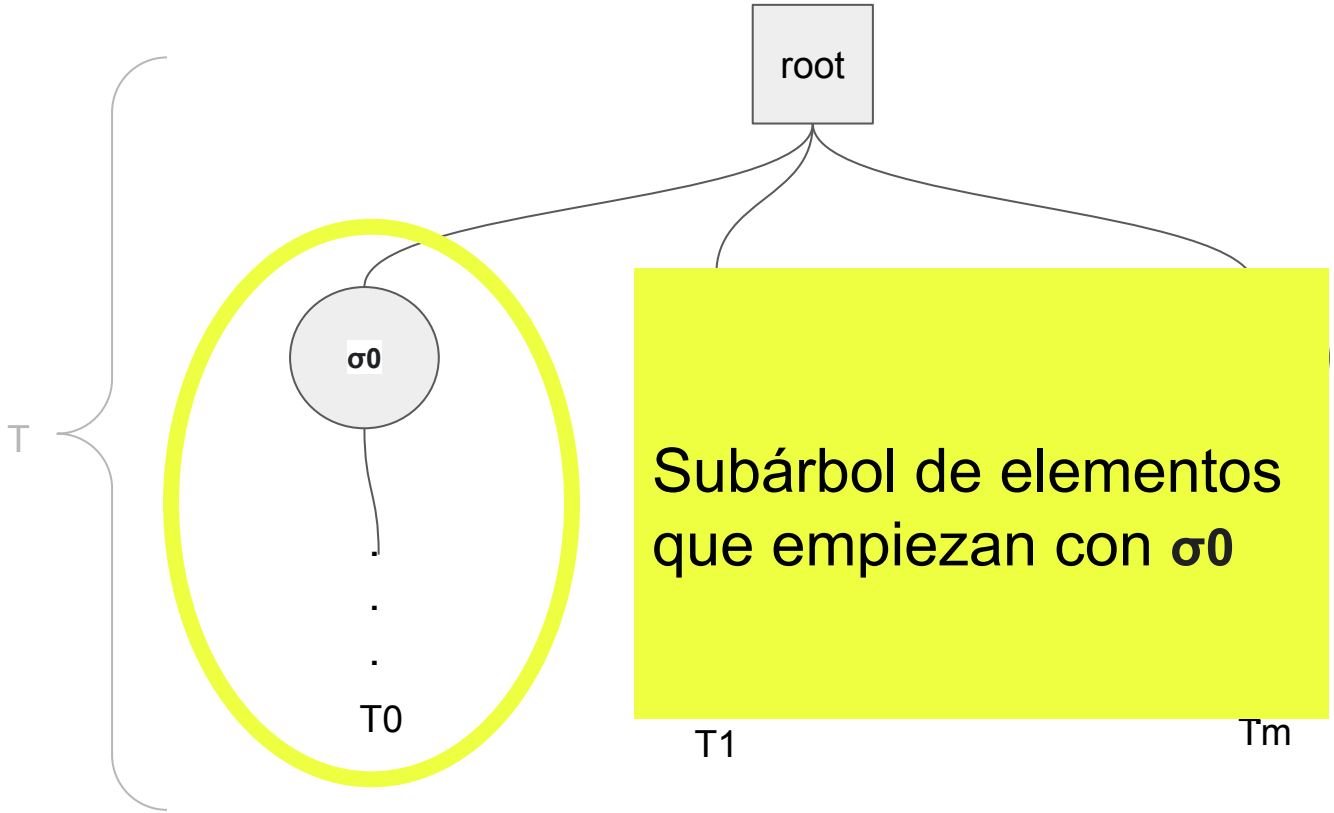
los hijos de un nodo representan las distintas posibilidades de símbolos diferentes que pueden continuar al símbolo representado por el nodo padre



Tries/prefix trees



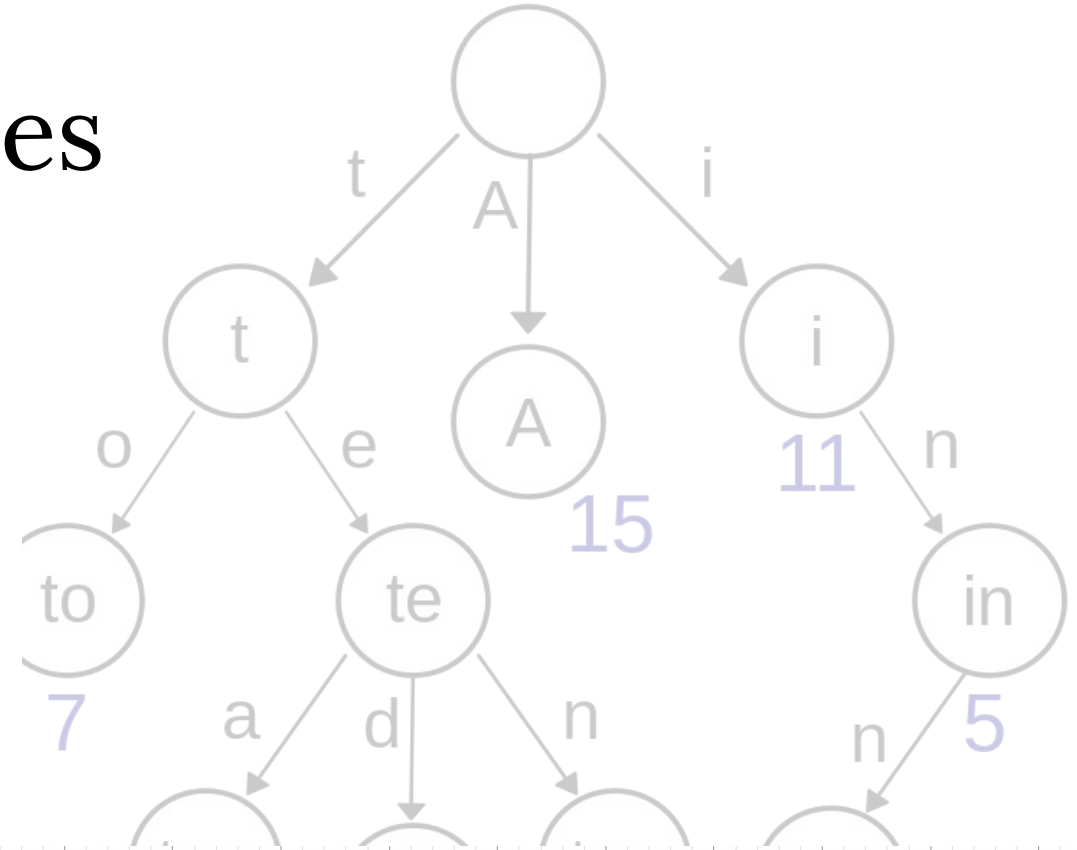
Tries/prefix trees

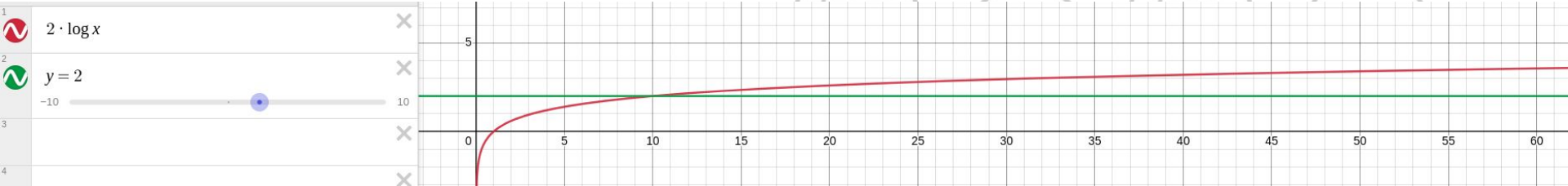


Tries/prefix trees

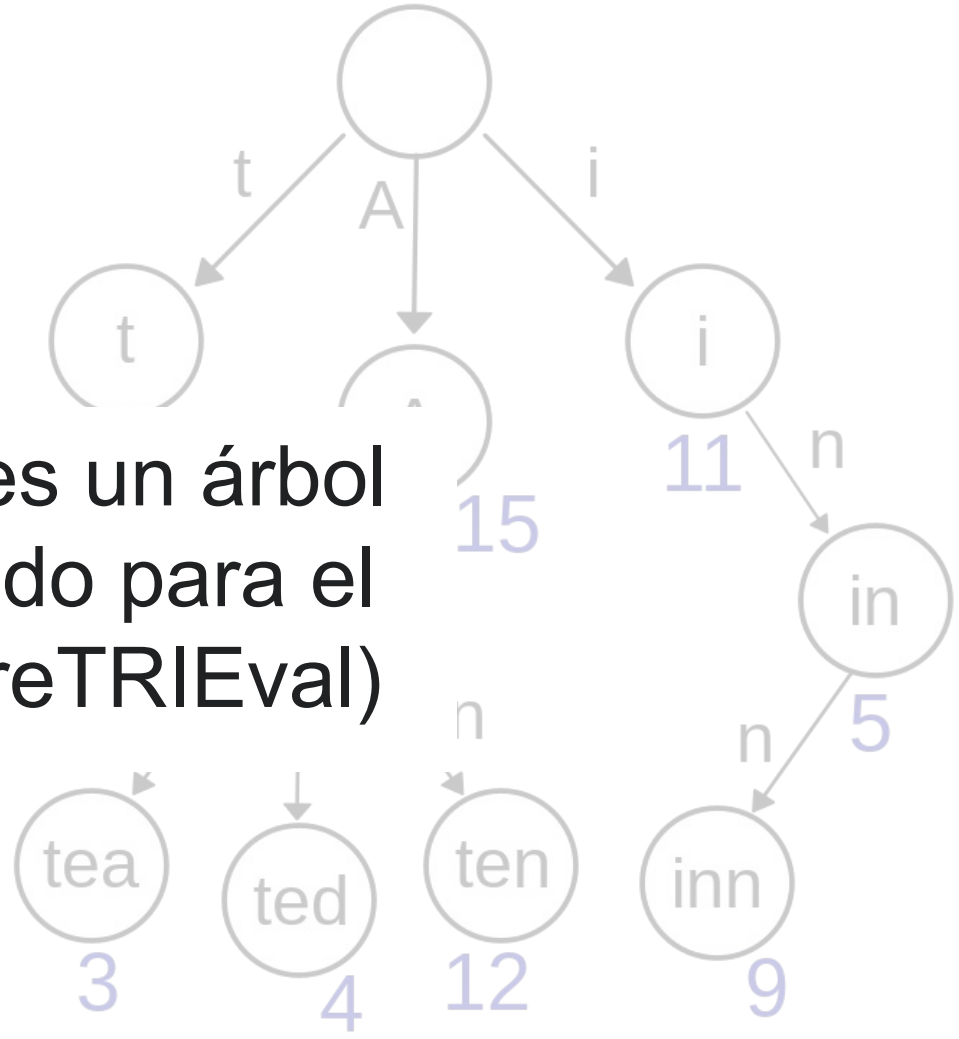
La búsqueda de una **clave de longitud m** tendrá en el peor de los casos un coste de **O(m)**

*Un árbol binario de búsqueda tiene un coste de O(m*logn) siendo n el número de elementos del árbol, ya que la búsqueda depende de la profundidad del árbol, logarítmica con el número de claves.*





Tries/prefix trees



Un Trie es un árbol optimizado para el lookup (reTRIEval)

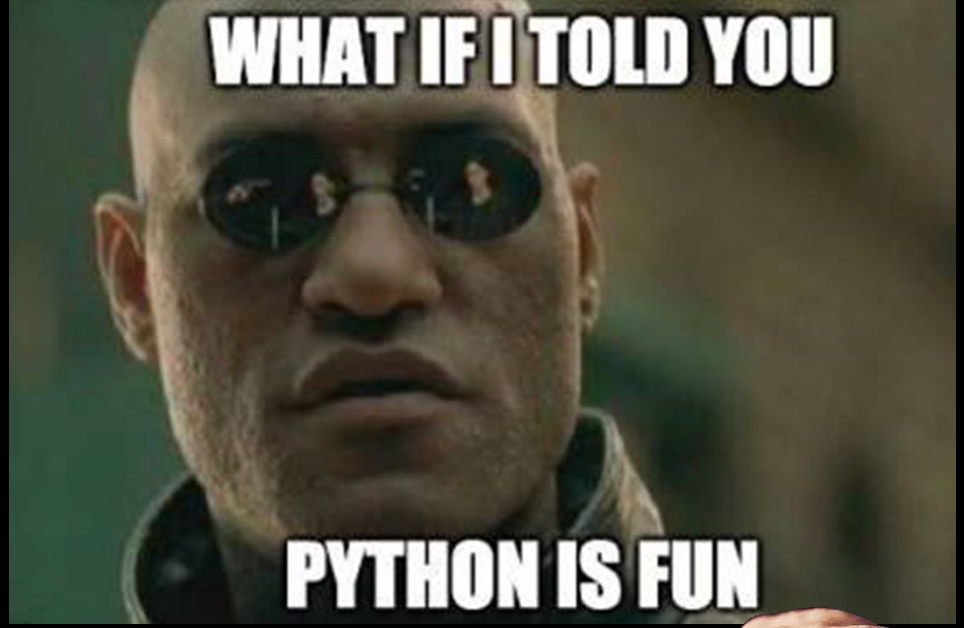
Consider a finite alphabet $\Sigma = \{\sigma_1, \ldots, \sigma_m\}$ with $m \geq 2$ symbols. We denote Σ^*, as usual in the literature, the set of all strings that can be formed with symbols from Σ. Given two strings u and v in Σ^* we write $u \cdot v$ for the string which results from the concatenation of u and v. We will use λ to dente the empty string or string of length 0.

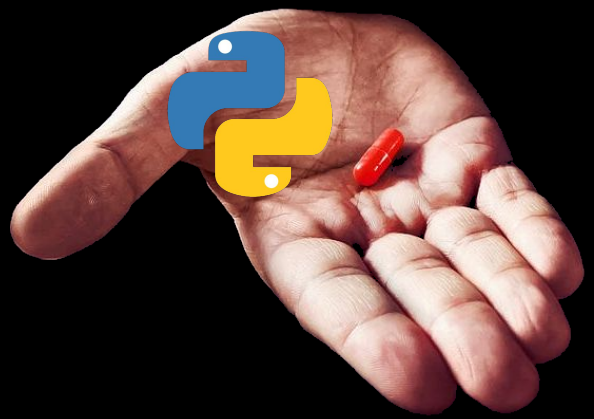
Definition

Given a finite set of strings $X \subset \Sigma^*$, all of identical length, the *trie* T of X is an m-ary tree recursively defined as follows:

1. If X contains a single element x or none, then T is a tree consisting on a single node that contains x or is empty.
2. If $|X| \geq 2$, let T_i be the trie for the subset

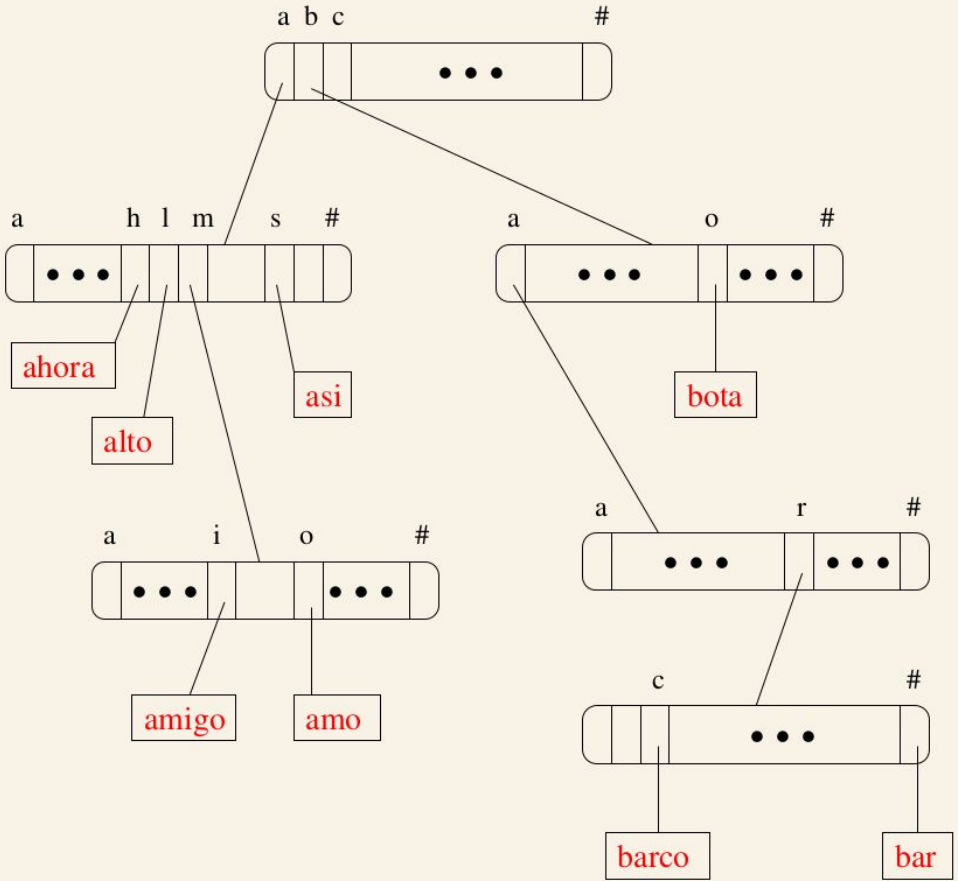
$$X_i = \{y \,|\, x = \sigma_i \cdot y \in X \wedge \sigma_i \in \Sigma\}$$

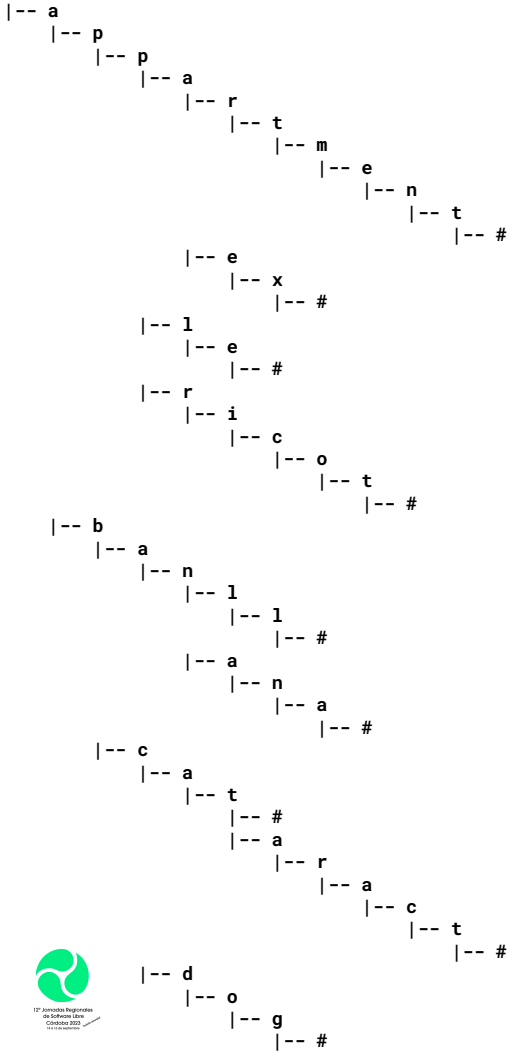

WHAT IF I TOLD YOU
PYTHON IS FUN



Tries

X = {ahora, alto, amigo, amo, asi, bar, barco, bota, ...}





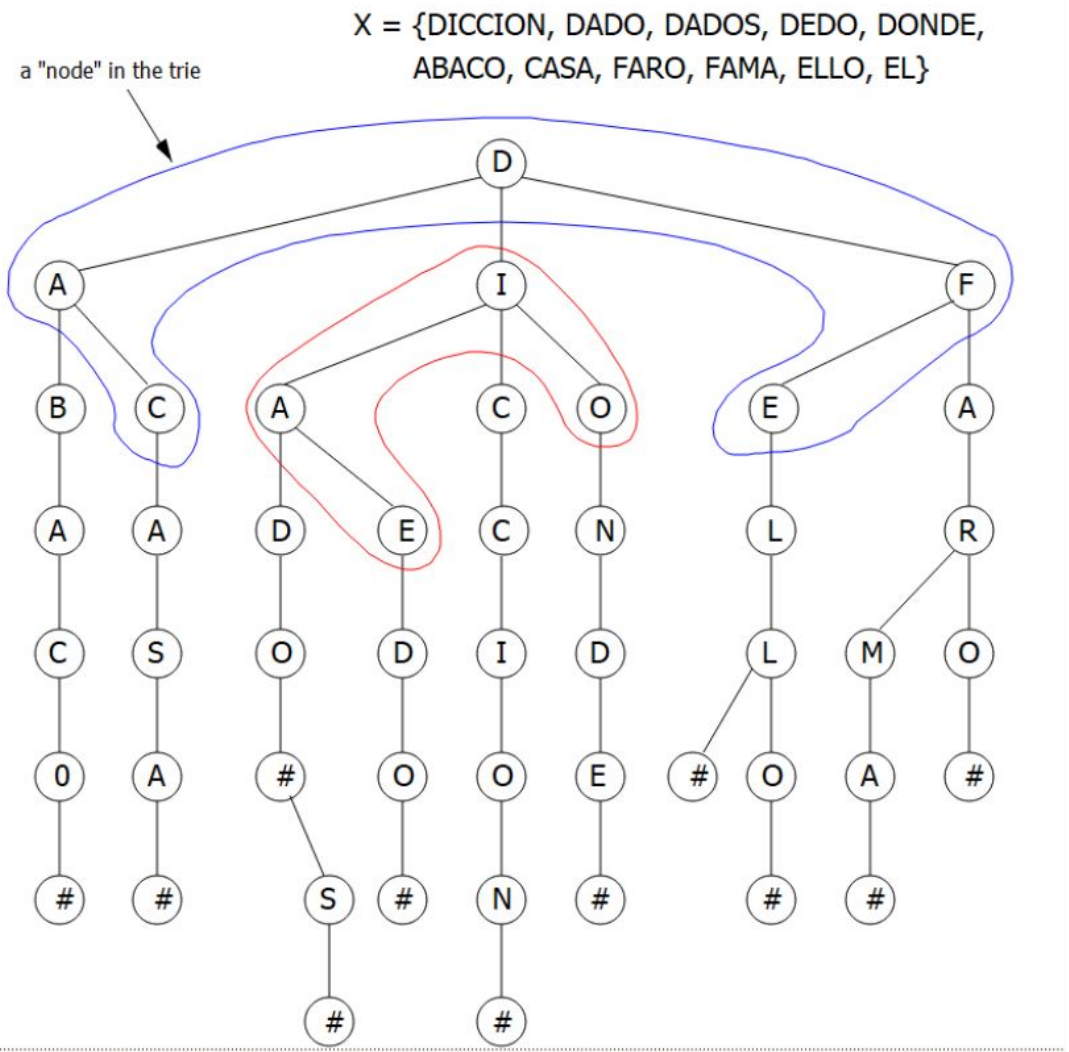
Ternary search tree

→ como BST

→más eficiente respecto a uso de memoria que otros tipos de prefix tries. Un poco más lentos

Words with prefix 'ap': ['apartment', 'apex', 'apple', 'apricot']
Words with prefix 'ba': ['ball', 'banana']
Words with prefix 'cata': ['cataract']

```python
class TSTNode:
    def __init__(self, char):
        self.char = char
        self.is_end_of_word = False
        self.left = None
        self.middle = None
        self.right = None
        self.words = []
```


X = {DICCION, DADO, DADOS, DEDO, DONDE,
ABACO, CASA, FARO, FAMA, ELLO, EL}
a "node" in the trie
D
A
I
F
B
C
A
C
O
E
A
A
A
D
E
C
N
L
R
C
S
O
D
I
D
L
M
O
0
A
#
O
O
E
#
O
A
#
#
#
S
#
N
#
#
#
#
#
#

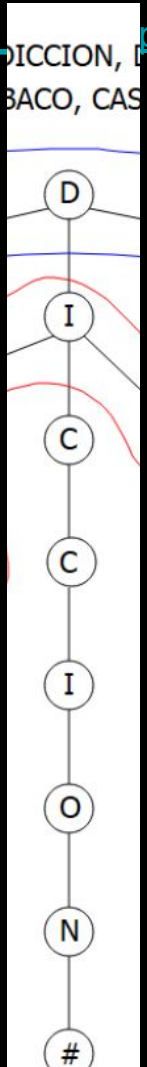
```
class TST:

    def __init__(self):
        self.root = None
    def insert(self, word):
    def list_words_with_prefix(self,
prefix):
```

https://github.com/karen-pal/advanced_data_structures/blob/main/tst_prefix_search.py

```
def insert(self, word):
        def insert_recursive(node, char_index):
            if char_index == len(word):
                return

            char = word[char_index]

            if not node:
                node = TSTNode(char)

            if char < node.char:
                node.left = insert_recursive(node.left, char_index)
            elif char > node.char:
                node.right = insert_recursive(node.right, char_index)
            else:
                if char_index == len(word) - 1:
                    node.is_end_of_word = True
                else:
                    node.middle = insert_recursive(node.middle, char_index + 1)

            return node

        self.root = insert_recursive(self.root, 0)
```

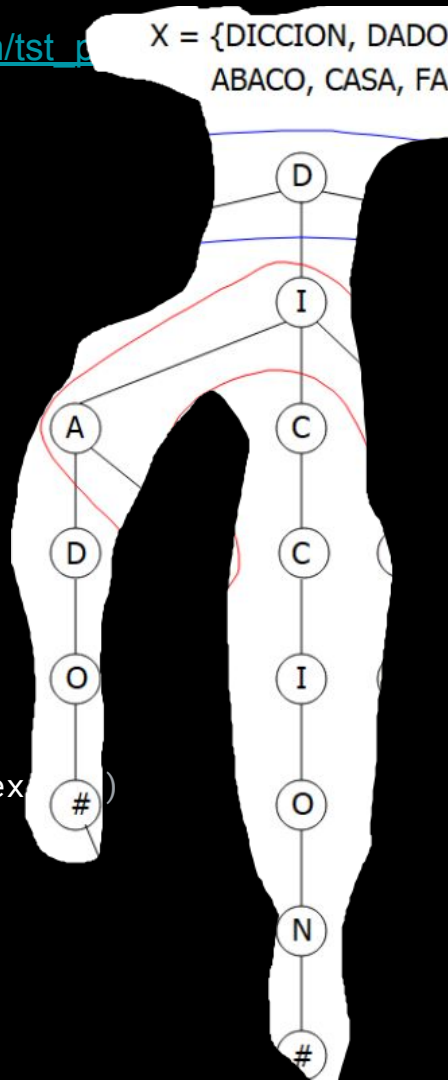
https://github.com/karen-pal/advanced_data_structures/blob/main/tst_prefix_[illegible].py

```python
def insert(self, word):
        def insert_recursive(node, char_index):
            if char_index == len(word):
                return

            char = word[char_index]

            if not node:
                node = TSTNode(char)

            if char < node.char:
                node.left = insert_recursive(node.left, char_index)
            elif char > node.char:
                node.right = insert_recursive(node.right, char_index)
            else:
                if char_index == len(word) - 1:
                    node.is_end_of_word = True
                else:
                    node.middle = insert_recursive(node.middle, char_index + 1)

            return node

        self.root = insert_recursive(self.root, 0)
```



https://github.com/karen-pal/advanced_data_structures/blob/main/tst_p

```python
def insert(self, word):
        def insert_recursive(node, char_index):
            if char_index == len(word):
                return

            char = word[char_index]

            if not node:
                node = TSTNode(char)

            if char < node.char:
                node.left = insert_recursive(node.left, char_index)
            elif char > node.char:
                node.right = insert_recursive(node.right, char_index)
            else:
                if char_index == len(word) - 1:
                    node.is_end_of_word = True
                else:
                    node.middle = insert_recursive(node.middle, char_index + 1)

            return node

        self.root = insert_recursive(self.root, 0)
```

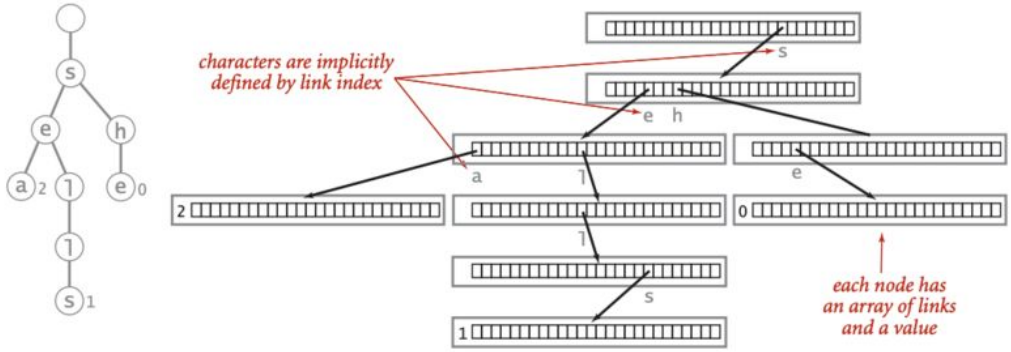


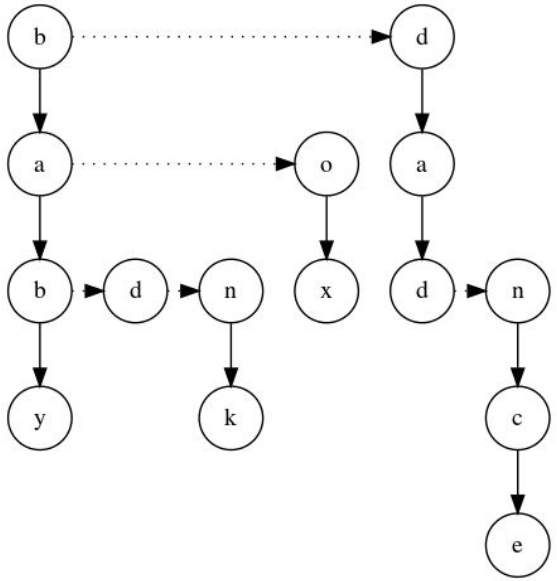
Tries/prefix trees

Conclusión:

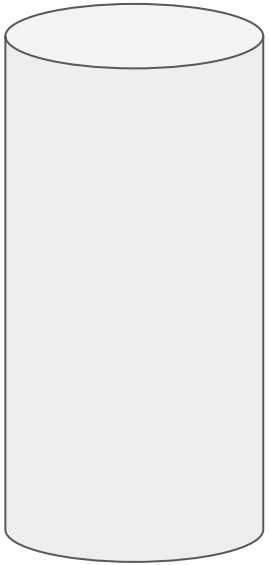
existen muchas variedades de tries - c/u con cambios que impactan la relación tradeoff entre espacio/costo

me encantan



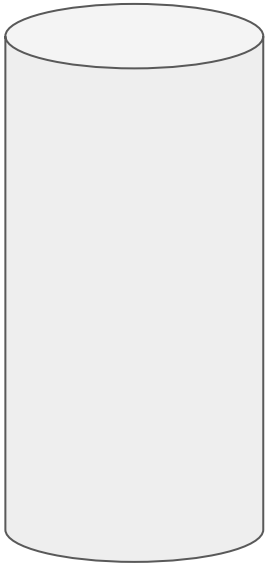


imaginá:
tenemos una bbdd
que por una cuestión u otra
es muy costosa de “queriar”.
Queremos **minimizar**
entonces **los hits**.

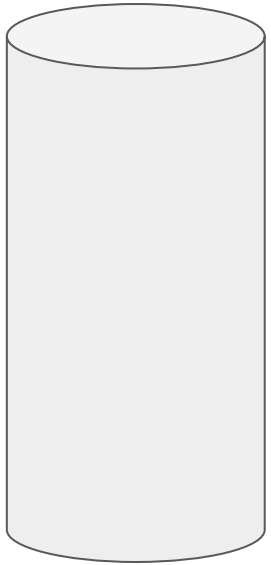


por qué?

- condiciones geográficas agregan latencia por networking
- dependencias en la lógica de la app hacen que este problema sea una cascada de problemas



solución:
descartemos casos en donde ya sabemos que no tenemos que ir a buscar a esta bbdd



Sabías que...

hay una estructura en donde la consulta por pertenencia tiene **costo constante** !

(independiente del tamaño del conjunto)

solo un detallito…
da falsos positivos

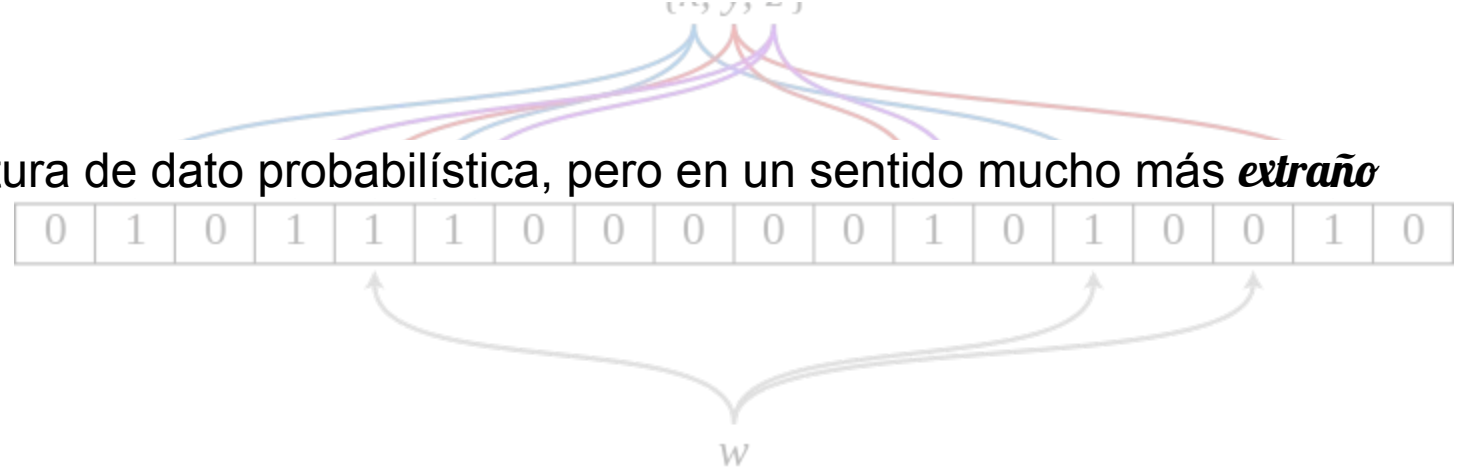
Si nos dice que x **no pertenece** a F, **lo sabemos con certeza** (ie, no da falso negativo)

Si nos dice que x **pertenece** a F, existe una **probabilidad de que esto sea mentira** (falso positivo)

en ese caso
buscaremos en esta
estructura en vez de
en la bbdd!
si la key no está, listo.

Bloom Filters

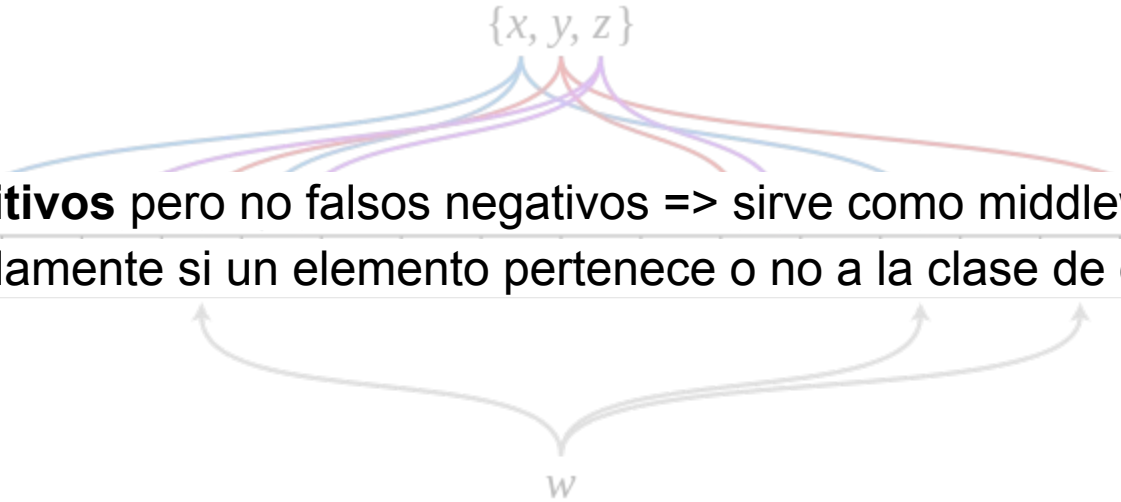
En situaciones donde importa MUCHO la performance



estructura de dato probabilística, pero en un sentido mucho más ***extraño***

Bloom Filters

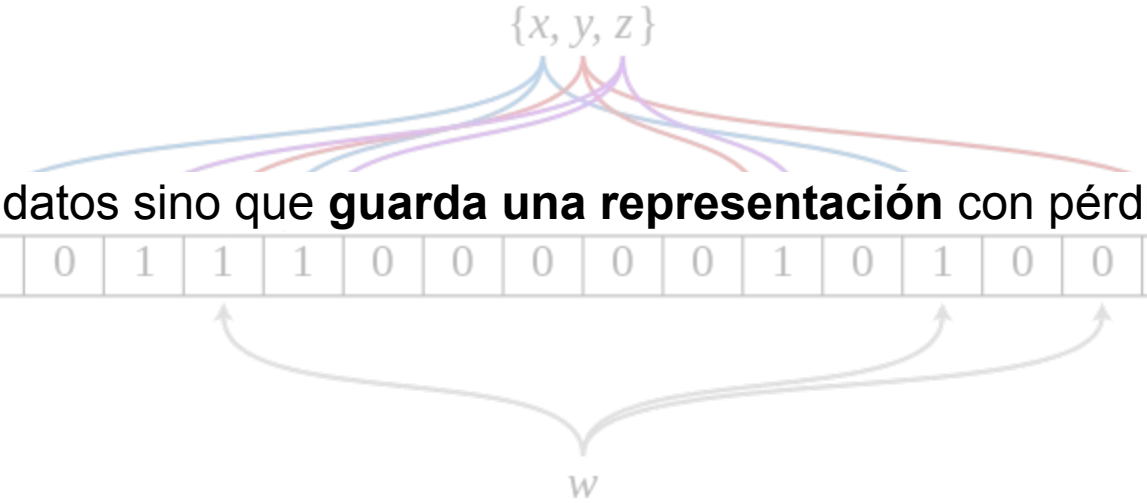
extraño:



da falsos positivos pero no falsos negativos => sirve como middleware para descartar rápidamente si un elemento pertenece o no a la clase de datos que nos interesa

Bloom Filters

extraño:



No almacena datos sino que **guarda una representación** con pérdida.

Bloom Filters

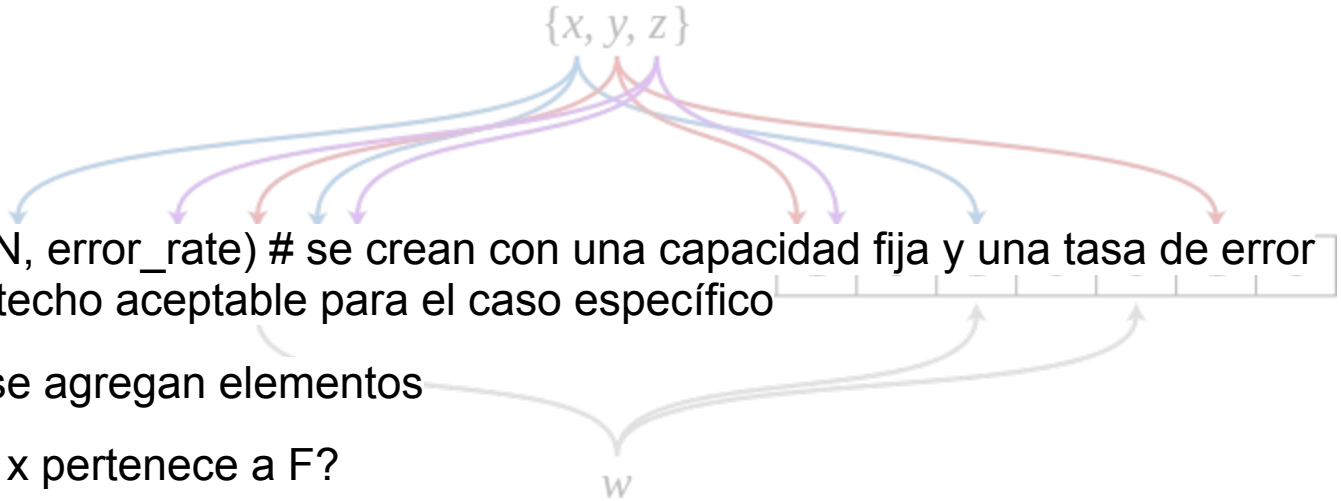
A Bloom Filter is a probabilistic data structure representing a set of items; it supports:

- Addition of items: $F := F \cup \{x\}$
- Fast lookup: $x \in F$?

Bloom filters do require very little memory and are specially well suited for unsuccessful search (when $x \notin F$)

Bloom Filters

En concreto:



F = crearBF(N, error_rate) # se crean con una capacidad fija y una tasa de error considerada techo aceptable para el caso específico

F.insert(x) # se agregan elementos

F.lookup(x) # x pertenece a F?

- False, entonces es cierto y x ∉ F
- True, puede estar mal con una probabilidad <= error_rate

Bloom Filters

- Un conjunto x1,...,xn de n elementos de un universo X

Bloom Filters

- Un conjunto x1,...,xn de n elementos de un universo X

["apple", "banana", "cherry", "date", "elderberry"]

Bloom Filters

- Un conjunto x1,...,xn de n elementos de un universo X
- Un vector S, de m bits. Inicialmente 0

Bloom Filters

- Un conjunto x1,...,xn de n elementos de un universo X
- Un vector S, de m bits. Inicialmente 0

0	0	0	0	0	0	0	0	0	0

m = 10

Bloom Filters

- Un conjunto x1,...,xn de n elementos de un universo X
- Un vector S, de m bits. Inicialmente 0
- Un conjunto de k funciones hash diferentes h1,..,hk. Cada una de las cuales dado un valor de X devuelve un valor en el dominio {1,..,m} (una posición del vector S).

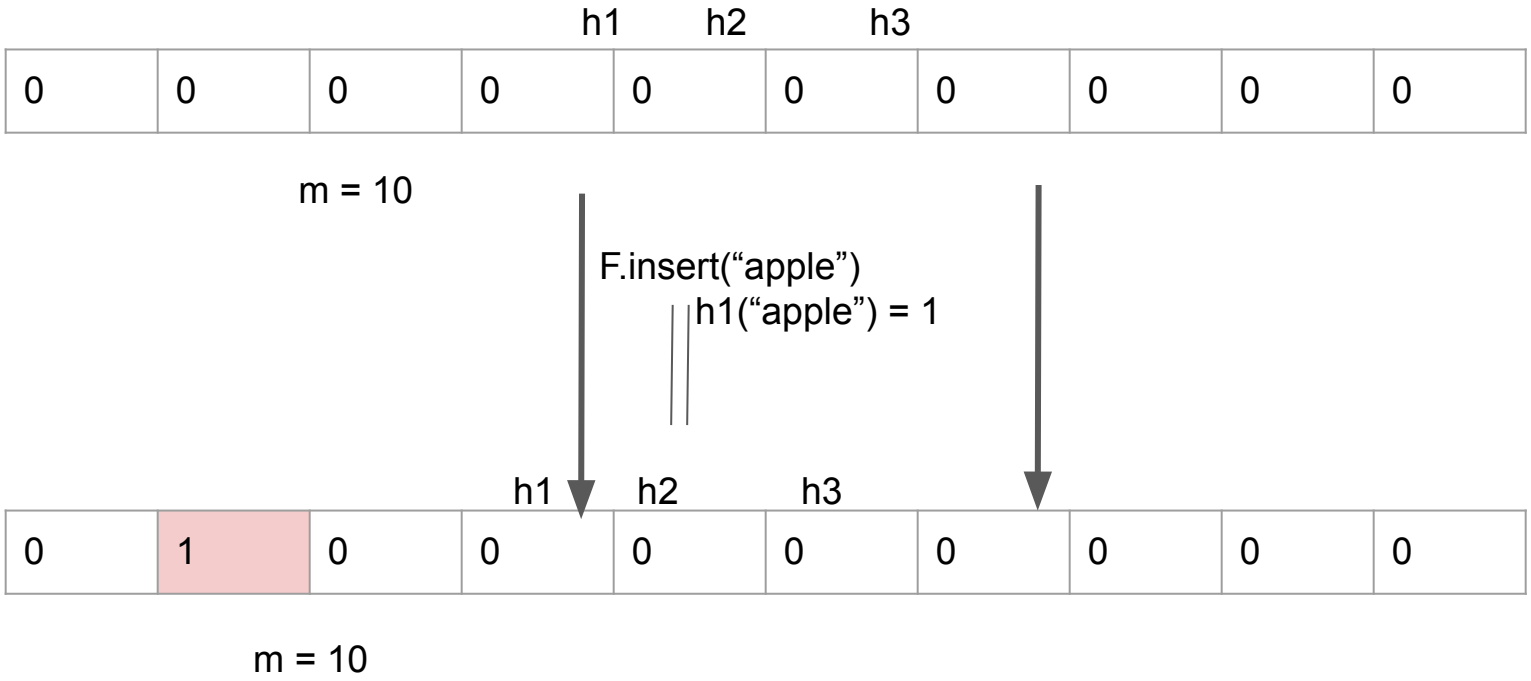
h1 h2 h3

0	0	0	0	0	0	0	0	0	0

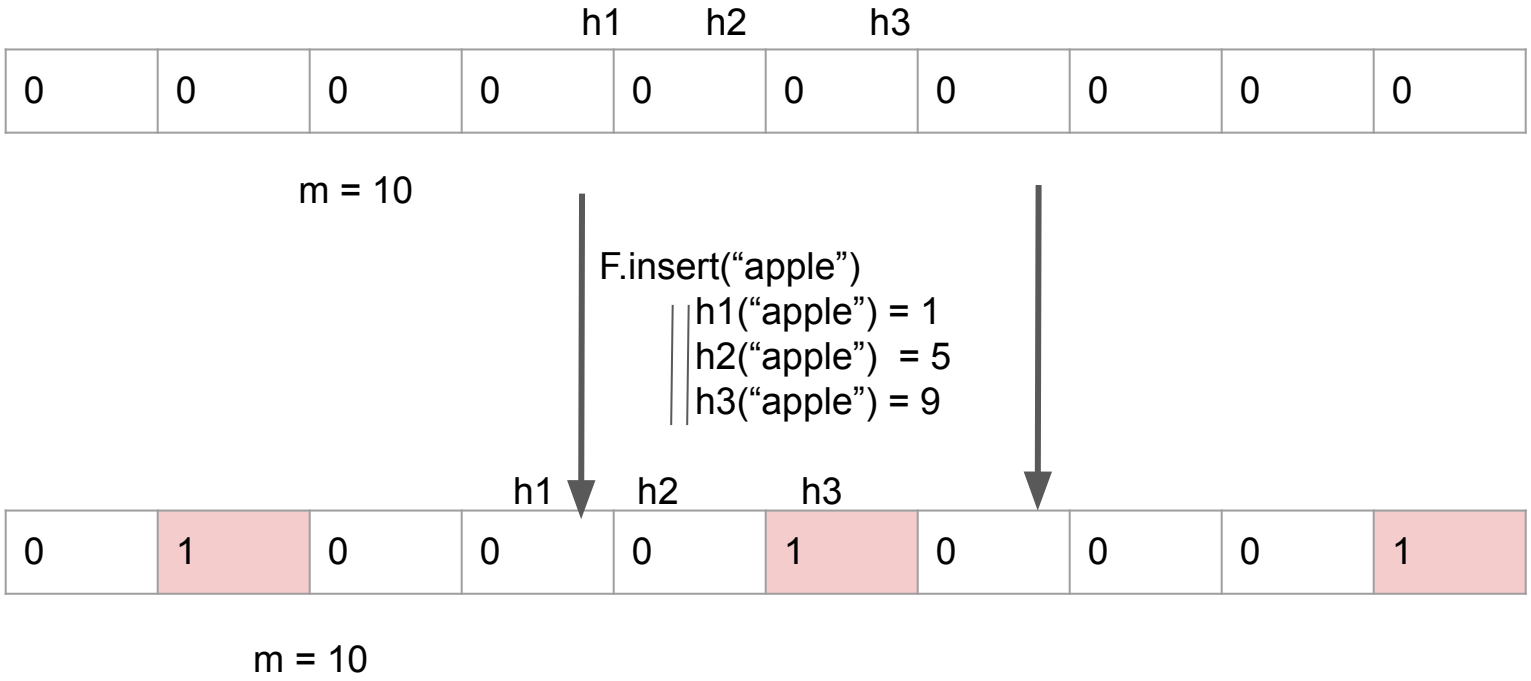
m = 10

cada hi relaciona un elemento de X con una representación de 1 bits…

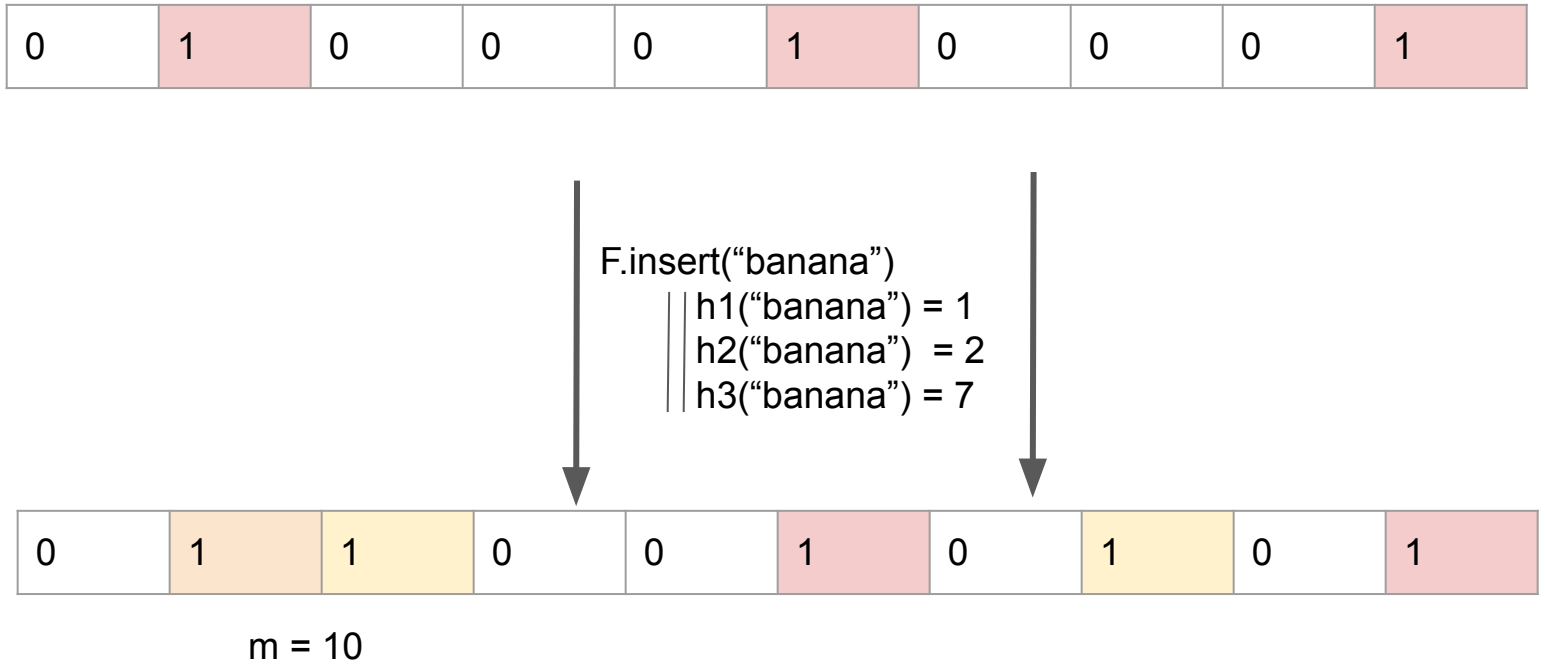
Bloom Filters



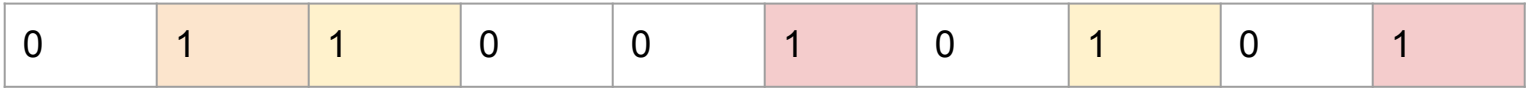
Bloom Filters



Bloom Filters



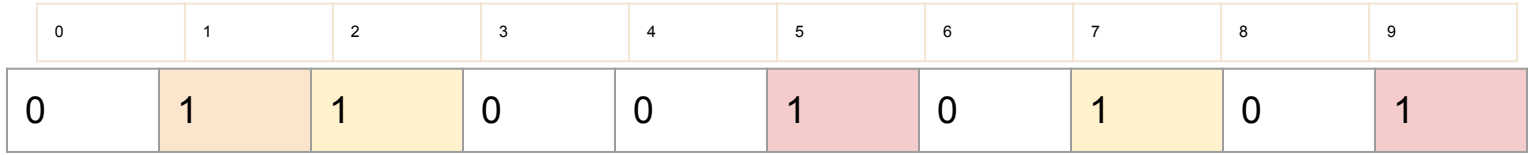
Bloom Filters



F.lookup(“apple”) = ?

(S[h*i*(apple)] == 0) para todo i? False : True

Bloom Filters



F.lookup(“asado”) = ?
Imaginemos que justo

h1[“asado”] = 1,
h2[“asado”] = 5,
h3[“asado”] = 7,

entonces
S[hi[“asado”]] = 1 para todo i => F.lookup(“asado”) = True…

Listo, ***falso positivo***

Bloom Filters

Optimal parameters for Bloom filters

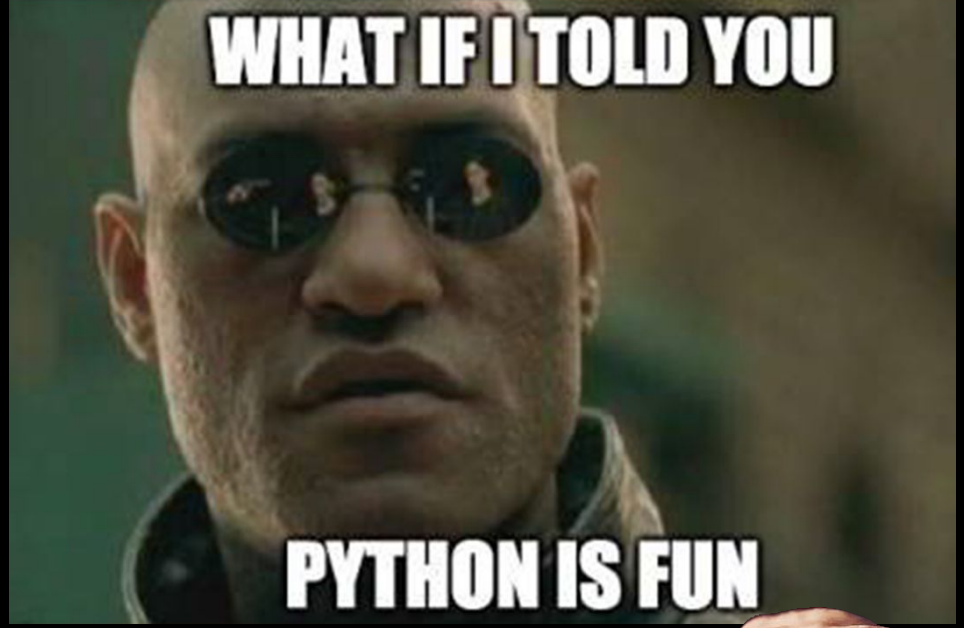
- Suppose that you want the probability of false positive $p^* = p(k^*)$ to remain below some bound P

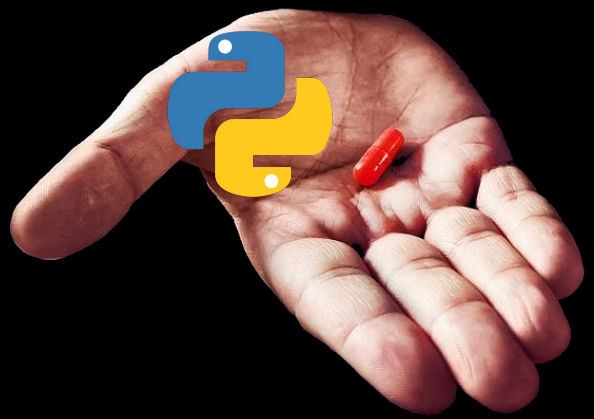
$$p^* \leq P \implies \ln p^* = -\frac{M}{n}(\ln 2)^2 \leq \ln P$$

$$\frac{M}{n}(\ln 2)^2 \geq -\ln P = \ln(1/P)$$

$$\frac{M}{n} \geq \frac{1}{\ln 2}\log_2(1/P) \approx 1.44 \log_2(1/P)$$

$$M \geq 1.44 \cdot n \cdot \log_2(1/P)$$


WHAT IF I TOLD YOU
PYTHON IS FUN

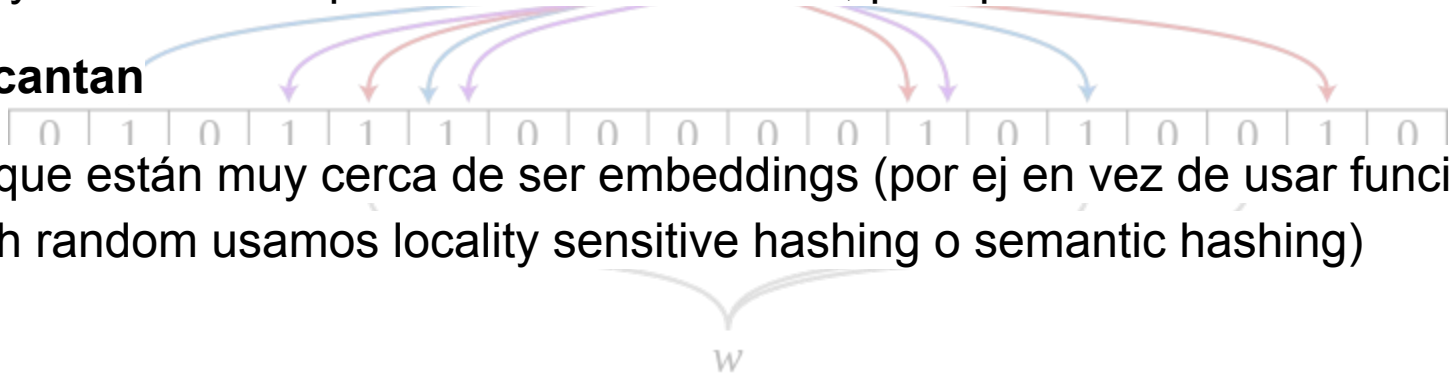


Bloom Filters

Conclusión:

son muy eficientes respecto a uso de memoria, pero pierden certeza

me encantan



siento que están muy cerca de ser embeddings (por ej en vez de usar funciones de hash random usamos locality sensitive hashing o semantic hashing)

espero que les sirva para meterse en estructuras de datos avanzadas!

12° Jornadas Regionales de Software Libre
Córdoba 2023
14 a 16 de septiembre
haciendo comunidad

gracias

JARDIN.HIJA.ALTURA

Mi alias para aquellos que no crean en la educación pública y en octubre voten por arancelar la educación

KD trees

Estructura de dato k-dimensional.

Particiona recursivamente espacios… crea espacialidad en el dato/regiones