INFORMATION RETRIEVAL

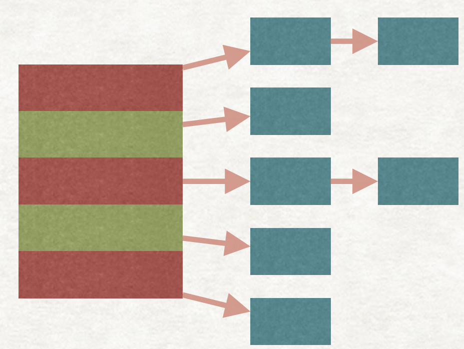
Laura Nenzi
lnenzi@units.it

Lecture 3-4

LECTURE OUTLINE

PRACTICAL PART
A PYTHON IMPLEMENTATION OF A SIMPLE BOOLEAN RETRIEVAL SYSTEM

Data Structures for dictionaries



CATT | SEARCH

DO YOU MEAN “CAT”?

Spelling Correction

+ IMPLEMENTATION

Wildcard Queries

DATA STRUCTURES FOR DICTIONARIES

HOW IS A DICTIONARY ACTUALLY REPRESENTED?

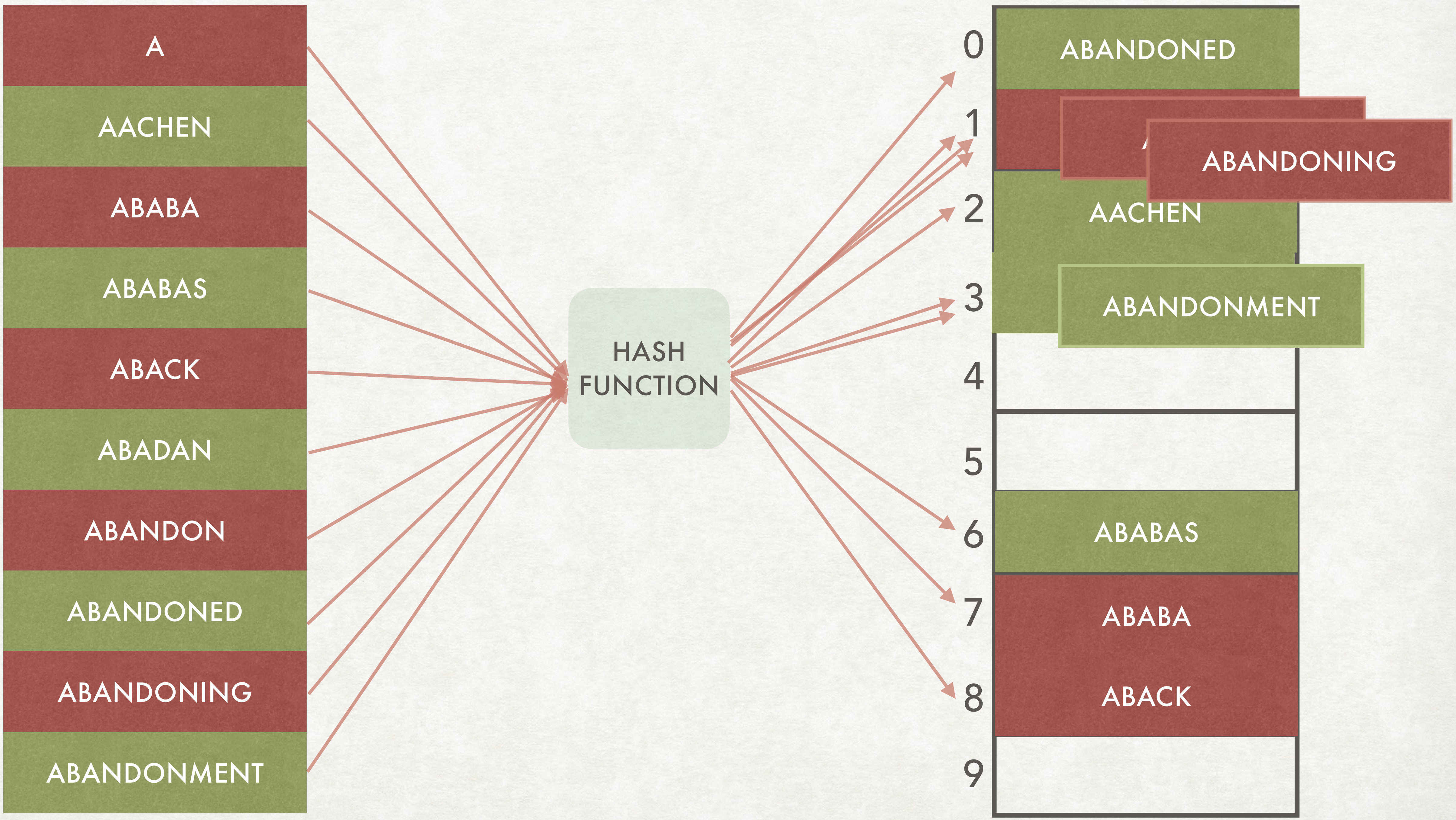
HASH TABLES & TREES

A
AACHEN
ABABA
ABABAS
ABACK
ABADAN
ABANDON
ABANDONED
ABANDONING
ABANDONMENT

- It is necessary to search in a dictionary that can be quite large
- Something more efficient than a linear scan is needed
- Two main approaches:
  - Hash tables
  - Trees (binary trees, b-trees, tries, etc.)

HASH TABLES

A BRIEF RECAP



HASH FUNCTIONS

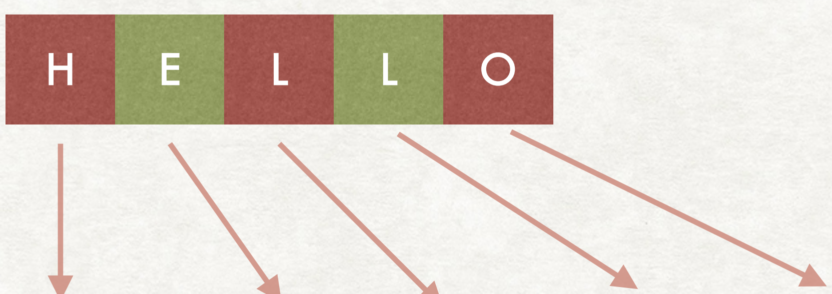
SOME EXAMPLES

Traditional for integers: $h(x) = x \bmod m$ where m is the size of the table

How to manage strings?

Component sum

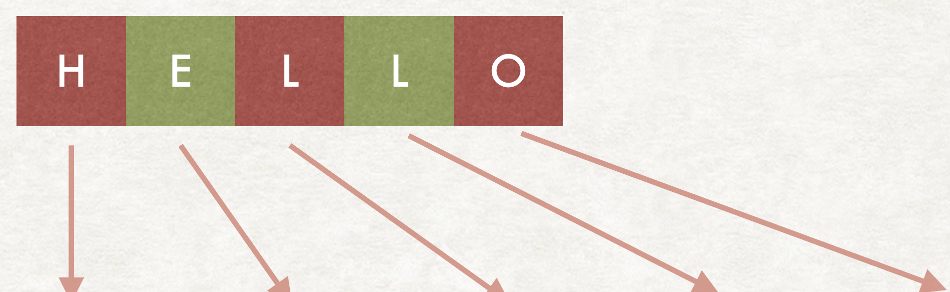
split the string into chunks and sum (or xor) them.



$$104 + 101 + 108 + 108 + 111 = 532$$

Polynomial accumulation

consider each chunk as a coefficient of a polynomial, then evaluate it for a fixed value of the unknown



$$104 + 101x + 108x^2 + 108x^3 + 111x^4$$

for $x = 33$ it evalues to 135639476

HASH TABLES

A BRIEF RECAP

- A hash function assign to each input (term) an integer number, which is the position of the term in a table.
- **Collisions**: sometimes for two different inputs the hash function returns the same value.
- Load factor: $\frac{\text{\# elements}}{\text{size of the table}}$.
  - Lower load factor: higher memory usage but less risk of collisions
  - Higher load factor: lower memory usage but higher risk of collisions

HASH TABLES

MANAGING COLLISIONS

- **Open addressing**. All entries are stored in the table, in case of collision the first free slot according to some probe sequence is found (e.g., linear or quadratic probing).
- **Chaining**. Each "cell" is a list of all entries with the same hash.
- **Perfect hashing**. For a fixed set it is possible to compute an hashing function with no collisions.
- Other collision resolution techniques, like **cuckoo hashing**. It shares some characteristic of perfect hashing while allowing updates.

HASH TABLES

THE GOOD, THE BAD, AND THE UGLY

- Finding an element in a hash table requires $O(1)$ *expected* time.
- In some cases (e.g., perfect hashing) this can also be the worst case time.
- Adding new elements might require *rehashing* (i.e., reinsertion of all elements into a bigger table) which is costly. This is needed to keep the load factor low enough.
- Some kind of searches are not possible, like looking for a prefix. In general anything that requires something different form the exact term.

BINARY TREES

A BRIEF RECAP

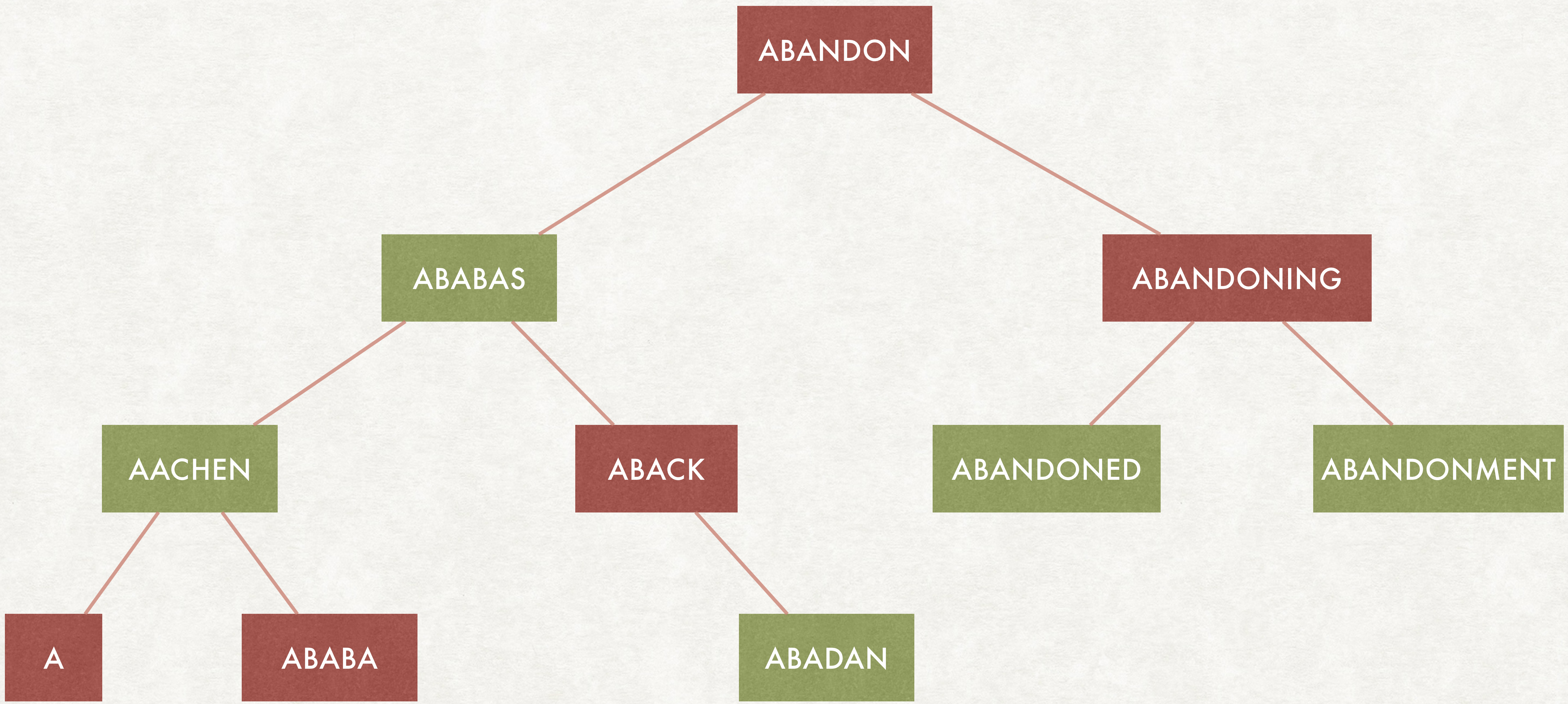
A binary tree is a tree in which each node has at most two children

Each node has an associated value (a term in our case)

A binary *search* tree has the property that the left subtree has only vales smaller than the value in the root and the right subtree only values that are larger.

This means that, if the tree is balanced, search can happen in $O(\log n)$ steps.

AN EXAMPLE OF BINARY SEARCH TREE



BINARY TREES

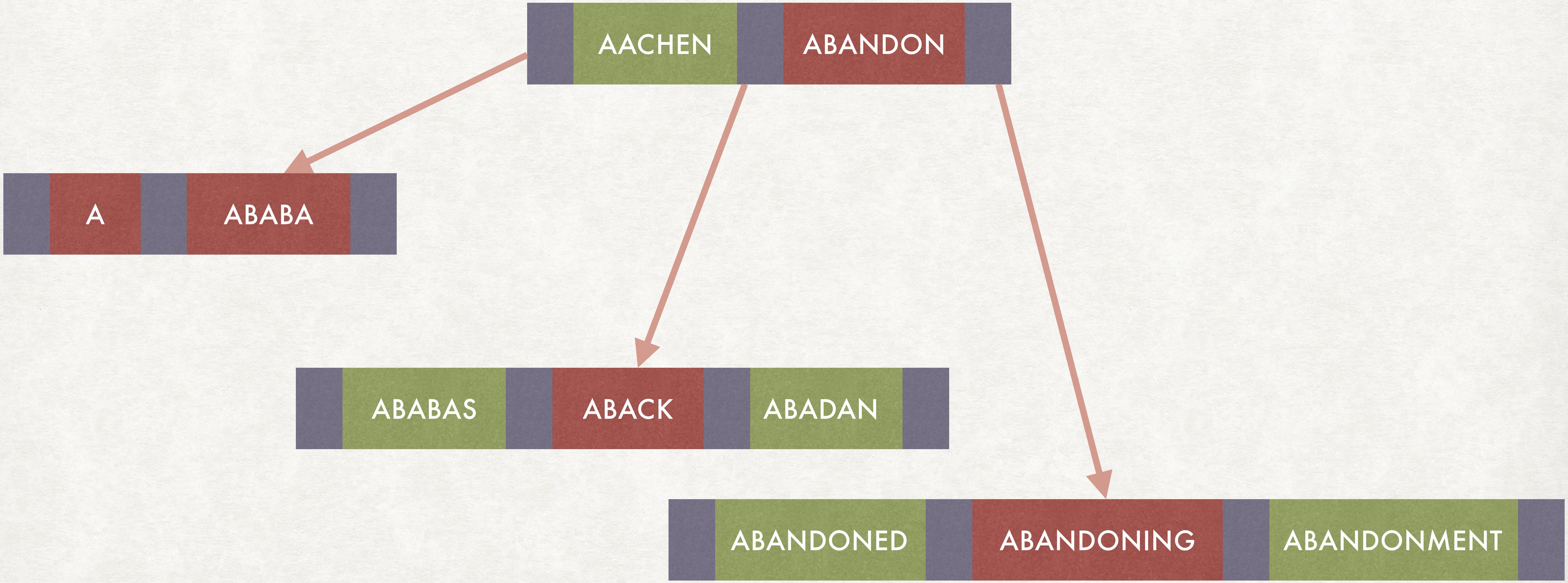
THE GOOD, THE BAD, AND THE UGLY

Binary search trees solve most of the problems of hash tables:

- Insertion (and deletion) are not expensive.
- Searching a prefix is possible.
- As long as the tree is kept balanced, search il efficient.
- But binary trees do no play well with disk access. $O(\log n)$ accesses to the main storage might be costly.
- A way to reduce the number of disk accesses while still using trees is via B-trees.

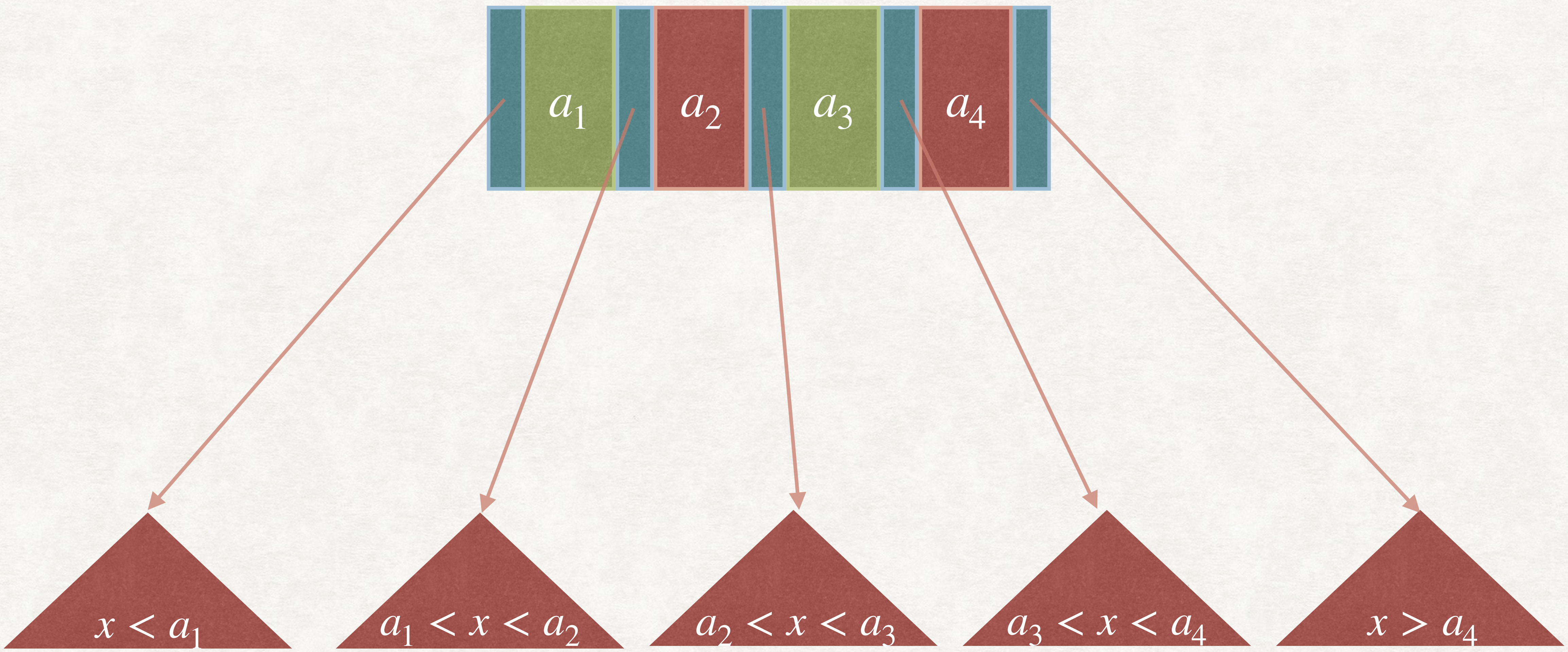
B-TREES

B-trees can be seen as a generalisation of binary search trees in which each node has between a and b children.



STRUCTURE OF A B-TREE NODE

The size of a node is usually selected to be a "block"



The node can contain up to four values
and five pointers to subtrees each respecting
a "generalised" version of the BST property

WHY B-TREES?

AND NOT SIMPLY BINARY SEARCH TREES?

- If you have to search across 10^6 elements then you need to go through at most:
  - $\lceil \log_2(10^6) \rceil = 20$ nodes in a binary search tree.
  - $\lceil \log_B(10^6) \rceil$ nodes in a B-tree, where B is the size of the block. Suppose $B = 100$, then $\lceil \log_{100}(10^6) \rceil = 3$.
  - This number corresponds to the number of disk accesses, which are the ones dominating the running time.

TRIES

ALSO KNOWN AS PREFIX TREES

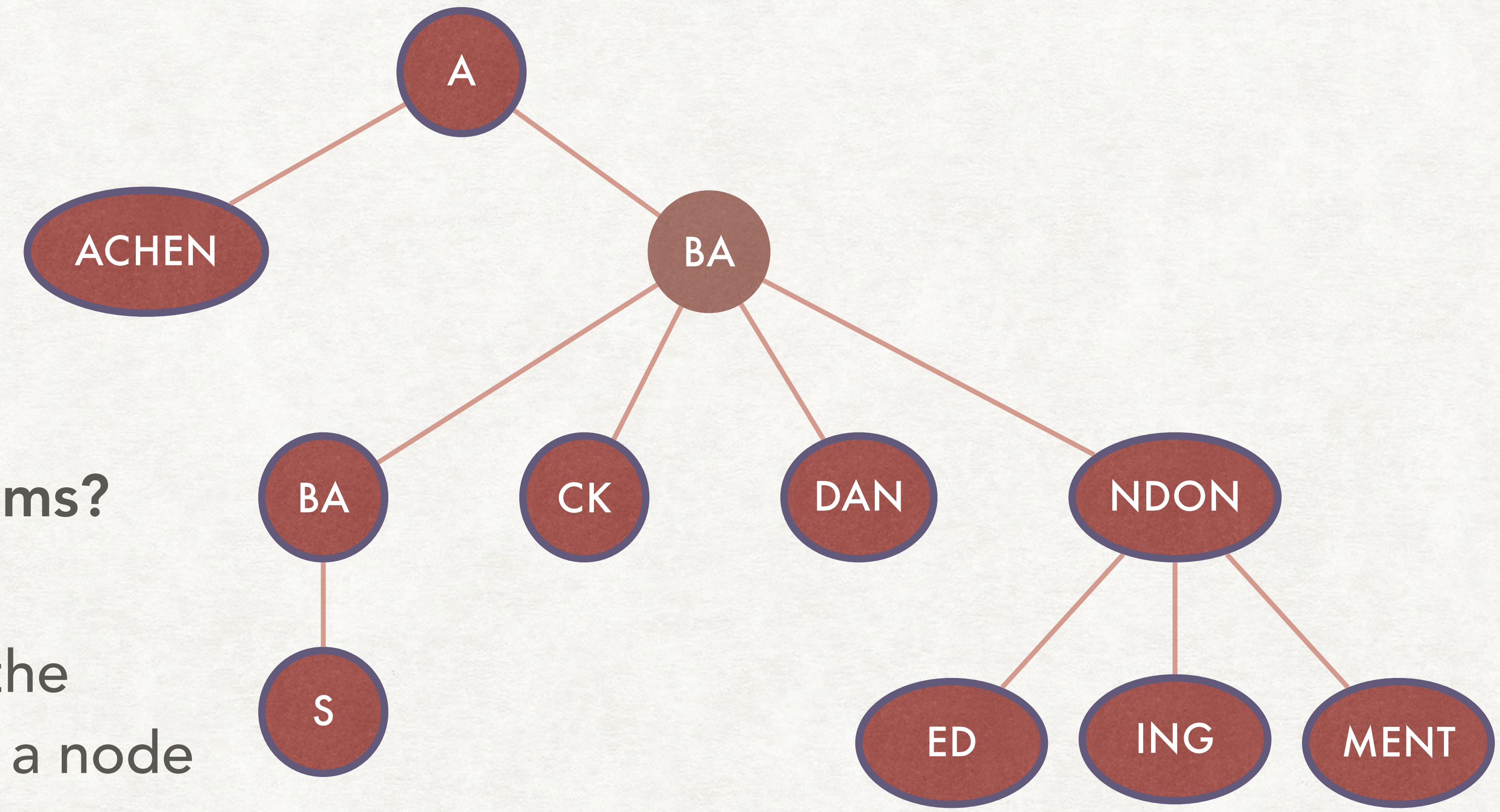
A **trie** is a special kind of tree based on the idea of searching by looking at the prefix of a key

The key itself (the term in our case) provides the path along the edges of the trie

Access time: worst case $O(m)$ where m is the size of the key. This is optimal because we must read the key.

Insertion is still possible and efficient.

TRIES: AN EXAMPLE



Where are the terms?

They are encoded in the paths from the root of the tree to a node

There **is** a key corresponding to the path from the root to this node

There **isn't** key corresponding to the path from the root to this node

TRIES: PROS AND CONS

- Tries have access time that is as good as hash tables (the $O(1)$ time for hash tables assumes a constant-length key)
- Differently from hash tables, there cannot be collisions.
- Insertion is still efficient.
- Search inside a range of key is very efficient.
- There can still be problems of too many accesses to disk.
- There ara variants of tries for external storage that mitigate the problem

WILDCARD QUERIES

WHAT ARE WILDCARD QUERIES?

SEARCHING AN ENTIRE SET OF WORDS

- Examples of wildcard queries:
  - **Car***: captures “**car**”, “**cars**”, “**car**t”, “**car**bon”, etc.
  - ***e*a***: captures “fl**ea**”, “**ea**r”, “h**ea**d”, “**Eva**”, etc.
- The uses might use wildcard queries when he/she:
  - Is uncertain of the spelling of a word.
  - Knows that a word has multiple spellings.
  - Want to catch all variants of term
    (which might also be “captured” by stemming).

TRAILING WILDCARDS

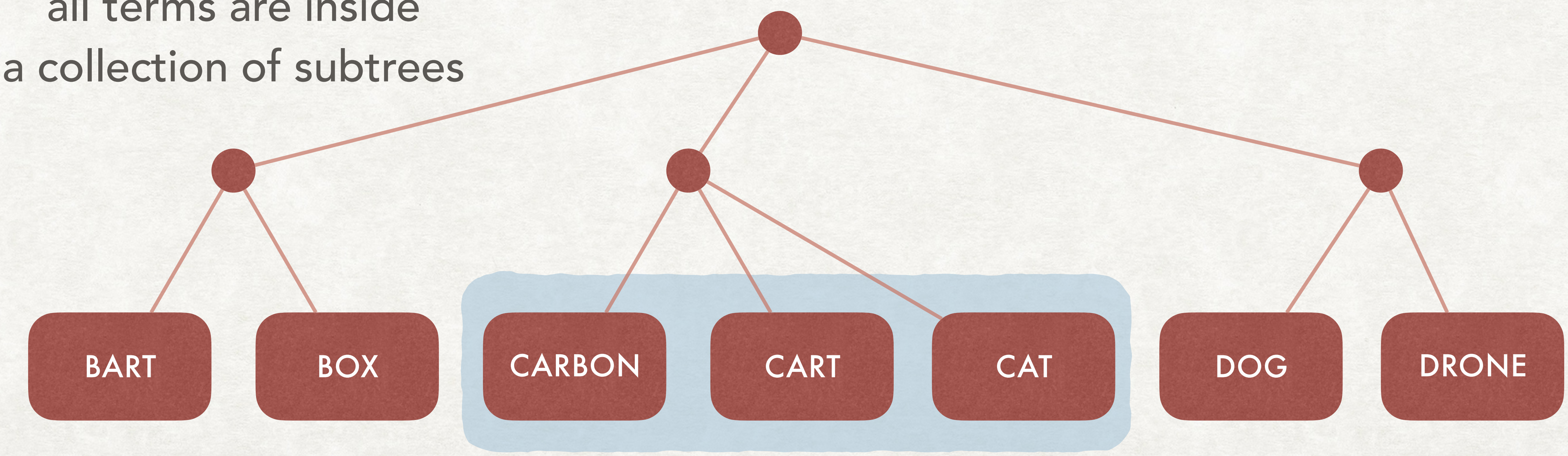
THE SIMPLEST CASE

term* **Trailing wildcard**
there is only one wildcard
and it is at the end of the word

Let us consider the query **CA***

In a binary tree/b-tree or
a variant (as shown below)
all terms are inside
a collection of subtrees

We can retrieve the posting lists
of all of them and perform
a union of the results



LEADING WILDCARDS

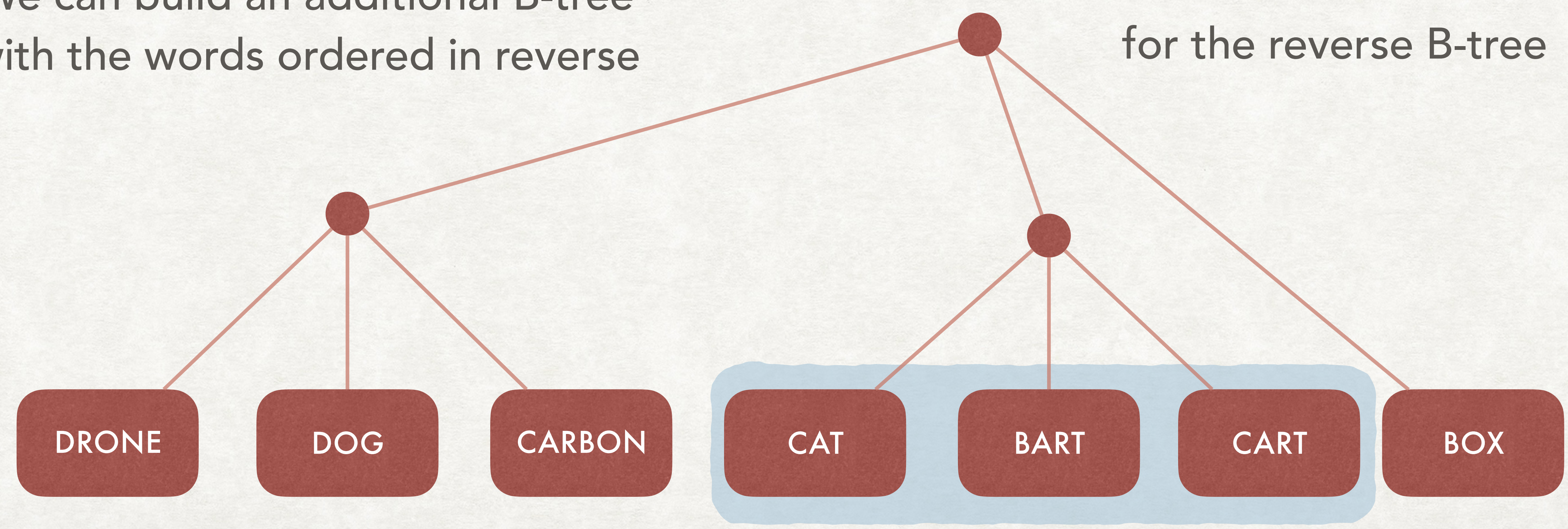
AND REVERSE (B-)TREES

***term** Leading **wildcard**
there is only one wildcard
and it is at the beginning of the word

Let us consider the query ***T**

We can build an additional B-tree
with the words ordered in reverse

Then the "leading wildcard" is
like an "inverse wildcard"
for the reverse B-tree



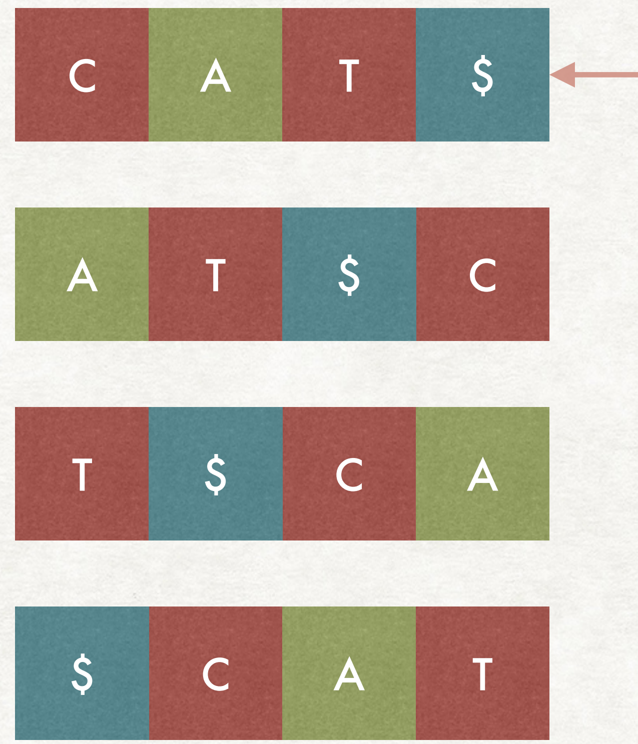
PERMUTERM INDEX

MANAGING GENERAL WILDCARD QUERIES

- Now we can answer all queries with leading and trailing wildcards.
- What about queries like “$word_1$*$word_2$”?
- Can we reformulate the problem of “one wildcard” as a leading or trailing wildcard problem?
- Yes, using the “permuterm index”
- We can also extend the solution to queries with more than one wildcard.

PERMUTERM INDEX

MANAGING GENERAL WILDCARD QUERIES



Special "*end of word*" symbol

Rotations of the word

We insert all the rotations of the word (including the "end of word") in the dictionary.

All the rotations of the same word points to the **same** postings list

PERMUTERM INDEX

MANAGING GENERAL WILDCARD QUERIES

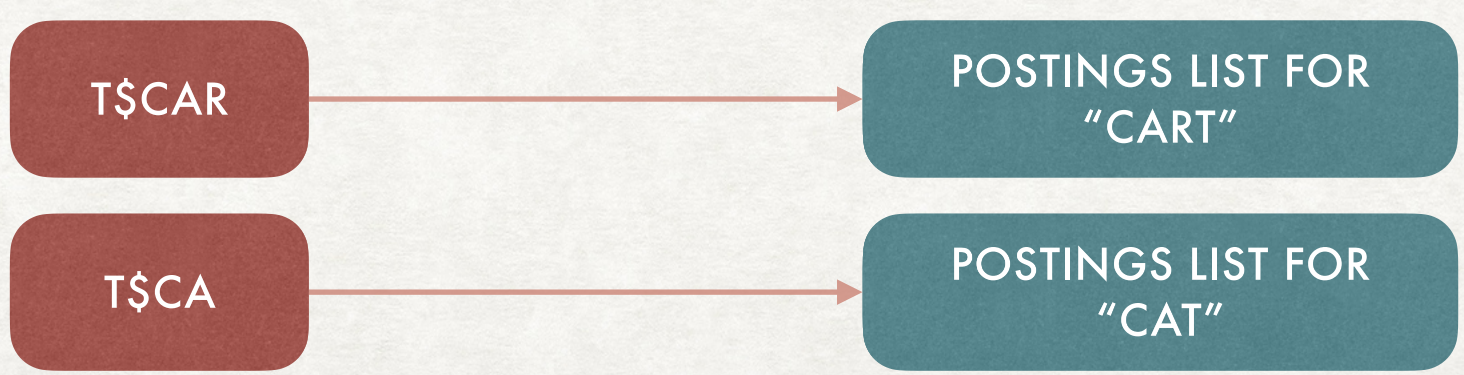
Our query: **C*T**

C*T$ Put the “end of word” at the end

T$C* Rotate the word to have the wildcard at the end

We can have a trailing wildcard, that we know how to solve!

Term in the dictionary



PERMUTERM INDEX

WHAT ABOUT MULTIPLE WILDCARDS?

Our query: *A*T

*A*T$ Put the “end of word” at the end

*T$ Consider the more general query where everything between the first and last wildcard is “folded” inside a single wildcard

T$* Rotate to have a trailing wildcard query

BART ~~BORT~~ CART CAT

Collect all the terms matching the simplified query

Scan the list to remove the ones **not** matching the original query

PERMUTERM INDEX

ADVANTAGES AND DISADVANTAGES

- We can now answer wildcard queries with any number of wildcards!
- Even if for more than one wildcard a linear scan of a list of terms is still needed.
- There is an interesting interplay between the algorithm that we use and the data structures employed.
- The main problem of permuterm indices: the amount of space needed to store all rotations of a word. A word with n letters will have $n + 1$ rotations (due to the “end of word” symbol).

K-GRAM INDEXES

ANOTHER WAY TO MANAGE WILDCARD QUERIES

k-gram: a sequence of k characters

DRONE

DRO
RON
ONE

3-grams of "DRONE"

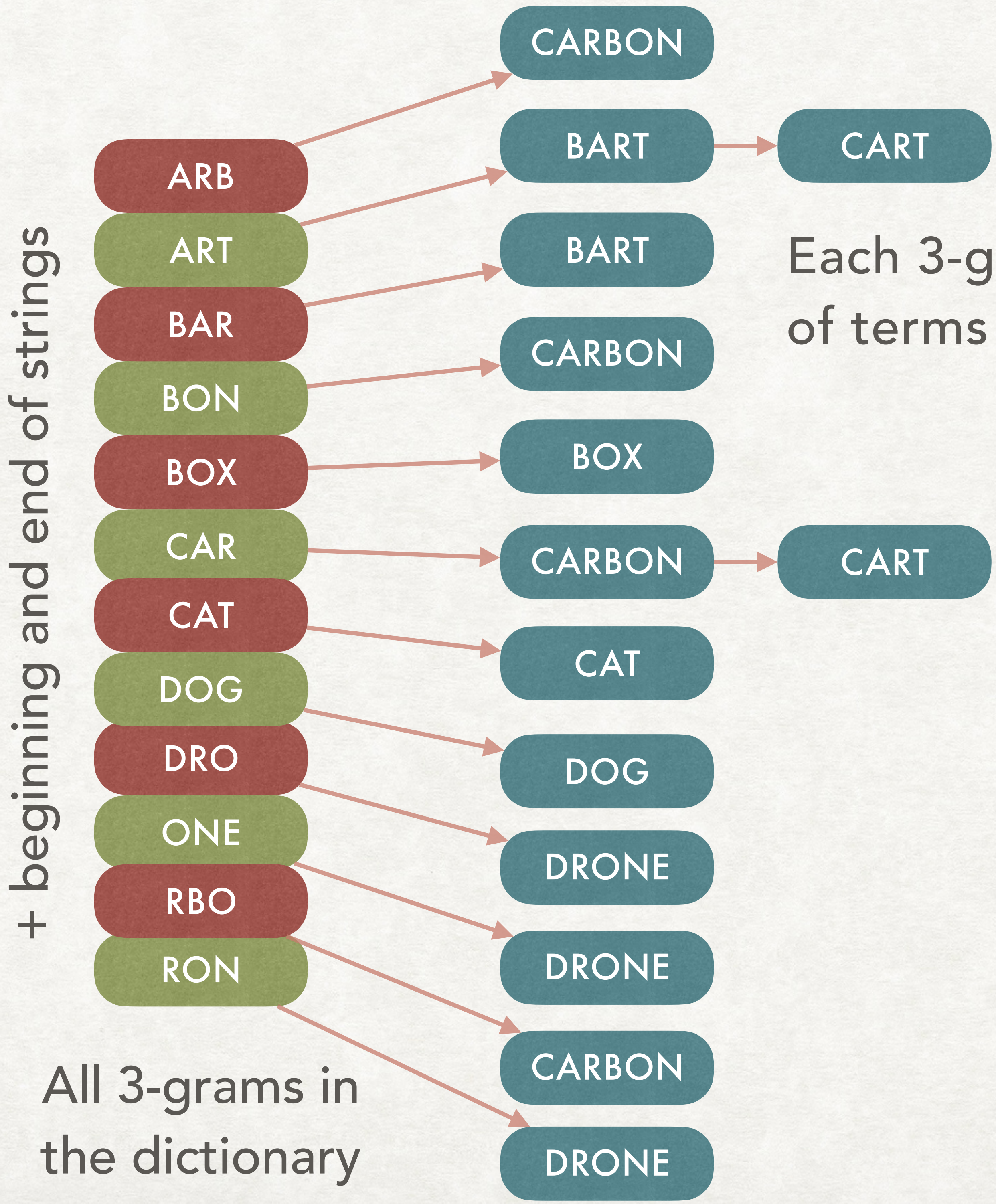
$DR
DRO
RON
ONE
NE$

We actually use the "$" symbol to denote the beginning and end of the word

We create a dictionary of k-grams obtained from all the terms

K-GRAMS INDEXES

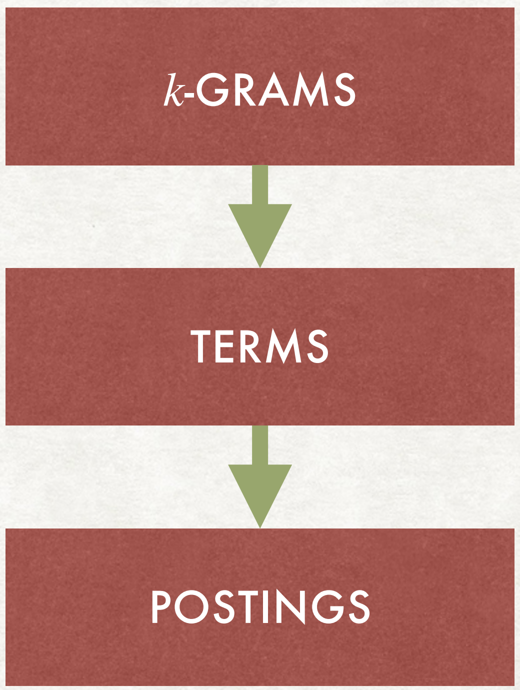
AN EXAMPLE



All 3-grams in the dictionary

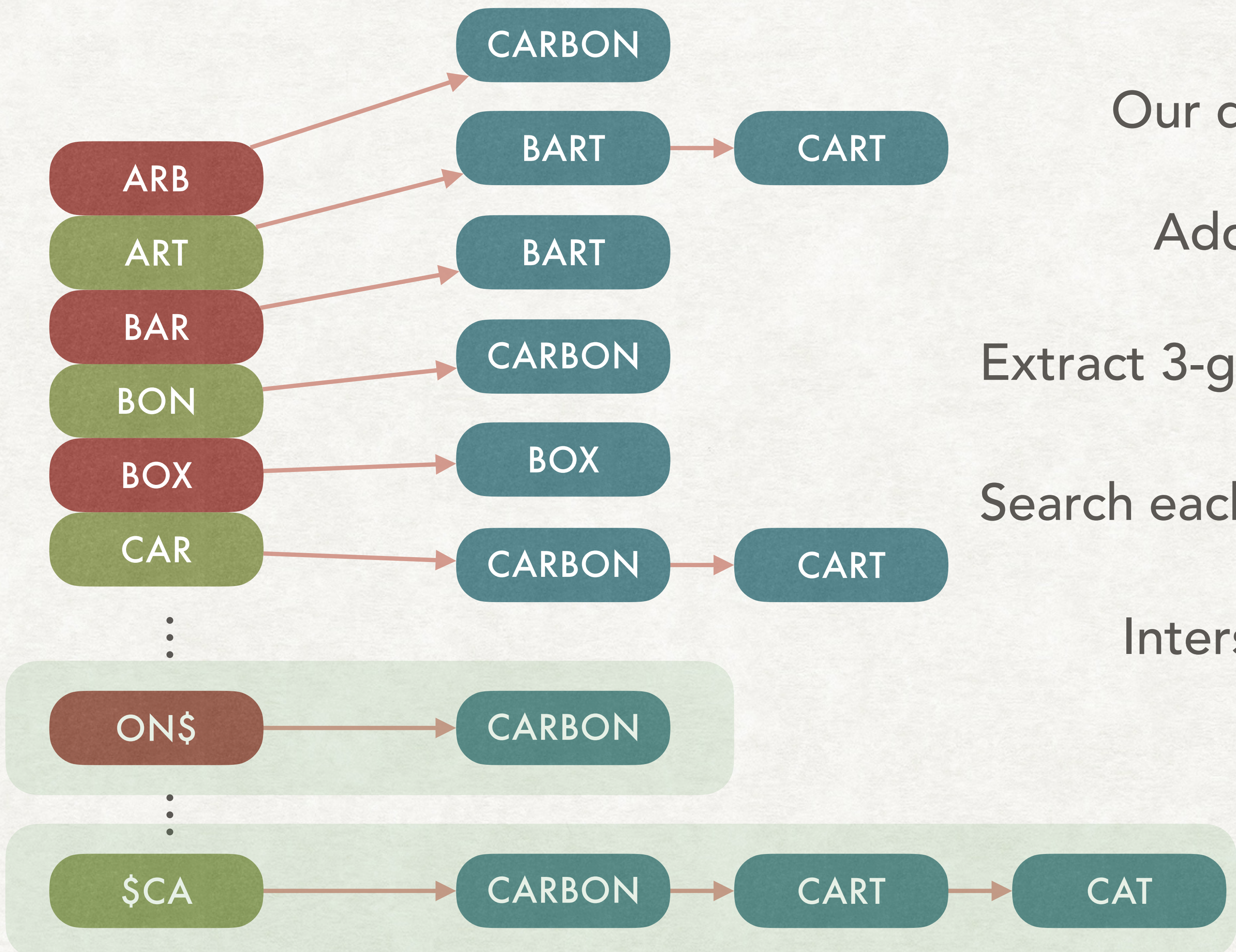
Each 3-gram points to the list of terms containing it.

The current structure of the system:



K-GRAMS INDEXES

HOW TO USE THEM TO ANSWER QUERIES



Our query: **CA*ON**

Add "$": **$CA*ON$**

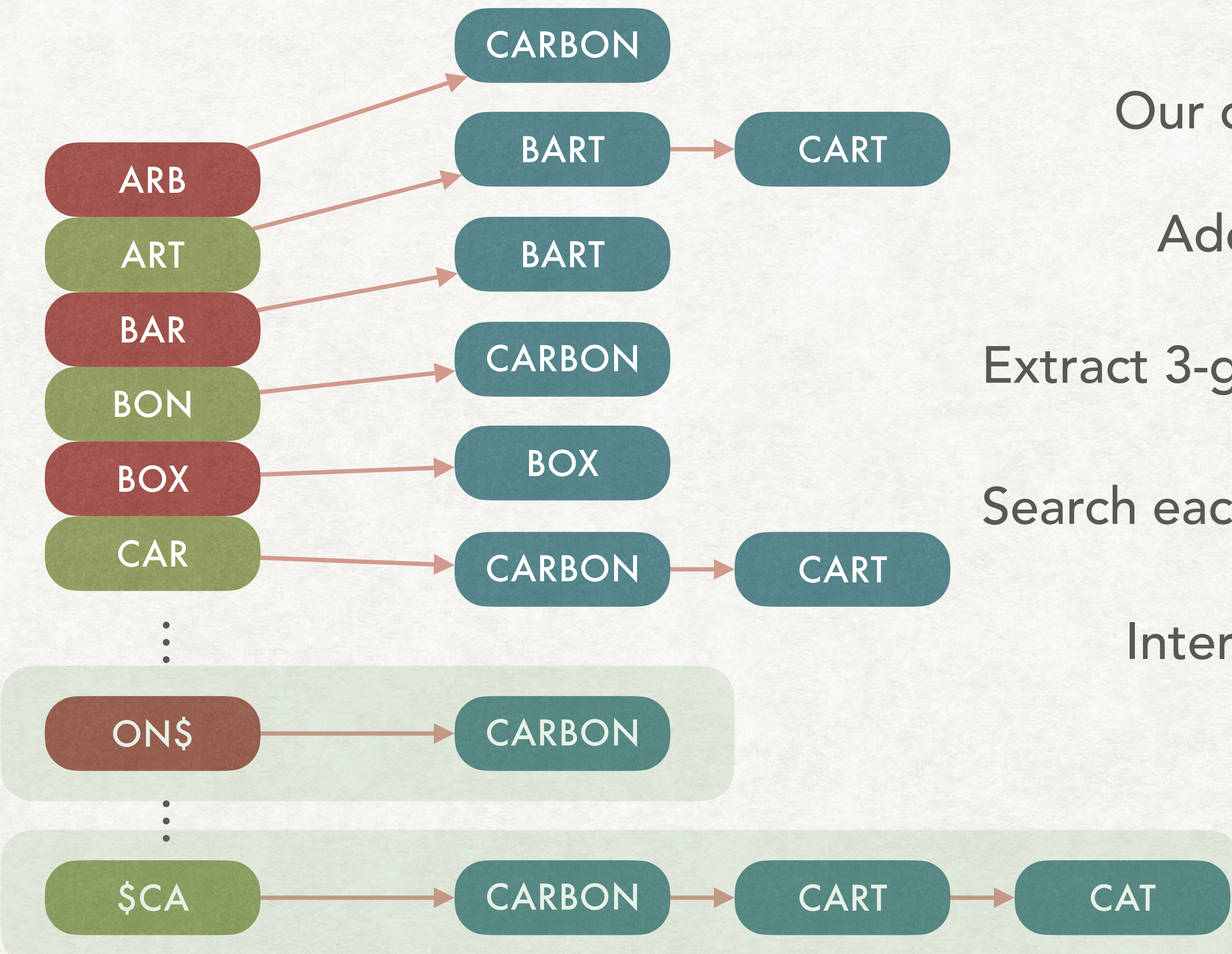
Extract 3-grams: **$CA ON$**

Search each one of the 3-grams

Intersect the results: CARBON

K-GRAMS INDEXES

HOW TO USE THEM TO ANSWER QUERIES



Our query: **CA*ON**

Add “$”: **$CA*ON$**

Extract 3-grams: **$CA ON$**

Search each one of the 3-grams

Intersect the results: CARBON

K-GRAMS

ADVANTAGES AND DISADVANTAGES

- They allow to answer wildcard queries
- A filtering step might still be needed:
  - Query: **GOL***
  - 3-grams: **$GO** and **GOL**
  - Possible element of the intersection: GOGOL, which does not respect the original query.
- k-grams can also be used to help in spelling correction
- Most commonly, the capability is hidden behind an interface (say an "Advanced Query" interface) that most users never use

SPELLING CORRECTION

BASICS OF SPELLING CORRECTION

- There are two main principle behind spelling correction:
  - If a word is misspelled, then find the nearest one.
  - If two or more words are tied (or nearly tied) select the most frequent word (in the collection).
- Which means that we need to define what "nearest" means.
- Two main approaches for addressing the isolated-term correction:
  - Edit (or Levenshtein) distance
  - k-grams overlap

EDIT DISTANCE

AKA LEVENSHTEIN DISTANCE

- The idea is that the distance between two words w_1 and w_2 is given by the *smallest* number of edit operations that must be performed to transform w_1 in w_2.
- The possible edit operations are:
  - *Insert* a character in a string (e.g, from **brt** to **bart**).
  - *Delete* a character from a string (e.g., from **caar** to **car**).
  - *Replace* a character in a string (e.g., from **arx** to **art**).

COMPUTING THE EDIT DISTANCE

WITH DYNAMIC PROGRAMMING

- How to compute efficiently the edit distance?
- There is a classical dynamic programming algorithm the runs in time $O(|w_1| \times |w_2|)$, where $|\cdot|$ denotes the length of a word.
- We are now going to detail the idea formally and then with an example

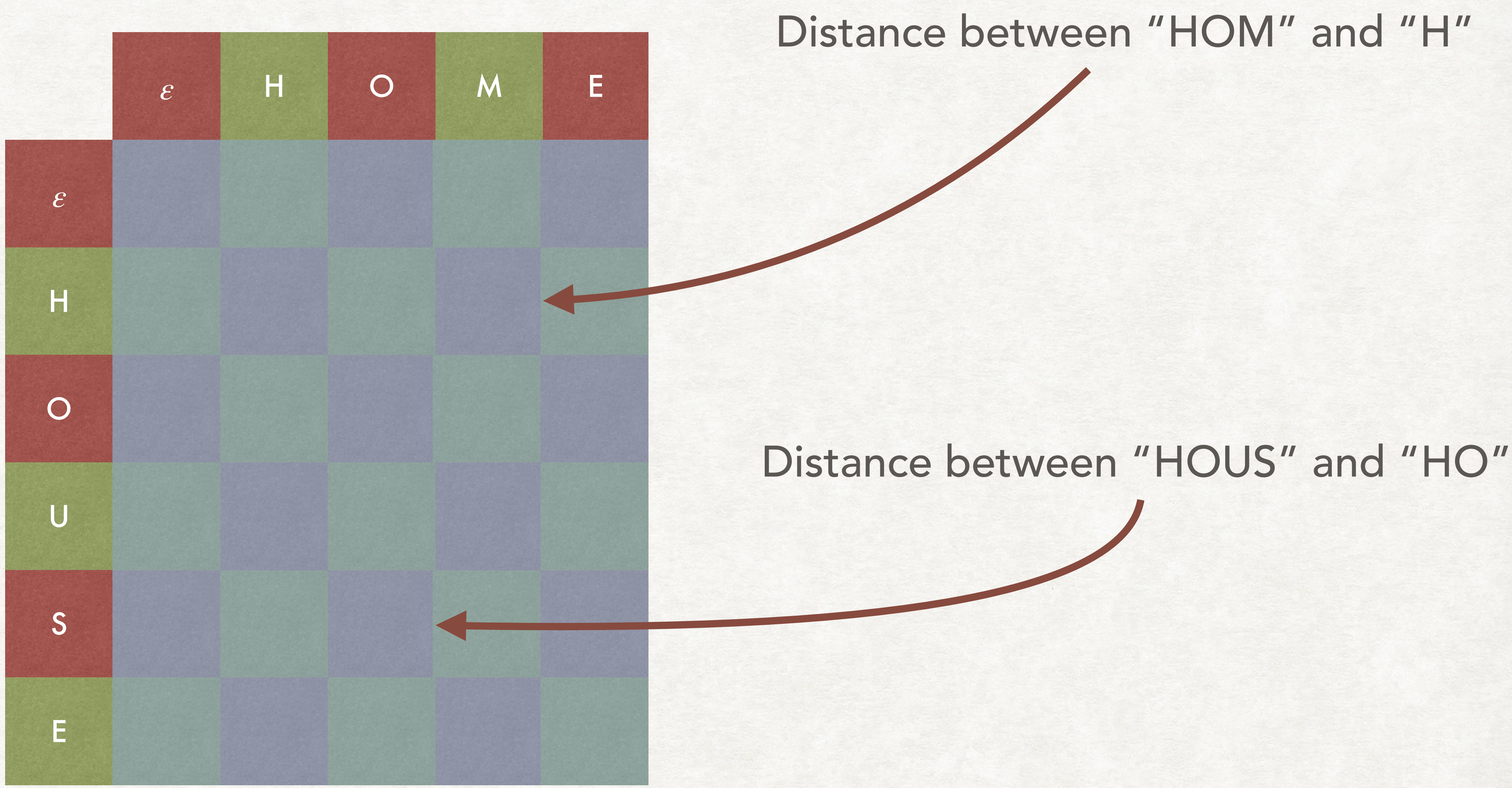
COMPUTING THE EDIT DISTANCE

WITH DYNAMIC PROGRAMMING

- Let $w_1 = v_1 a$ and $w_2 = v_2 b$ with a, b characters and v_1, v_2 words.
- The main idea is that you know the edit distance $d(w_1, w_2)$ between w_1 and w_2 is the minimum between:
    - $d(v_1, v_2) + 1$ if $a \neq b$ (i.e., we replace a by b)
    - $d(v_1, v_2)$ if $a = b$ (i.e., the distance does not increase)
    - $d(v_1, v_2 b) + 1$ (i.e., we remove a from the first word)
    - $d(v_1 a, v_2) + 1$ (i.e., we add b in the second word)

COMPUTING THE EDIT DISTANCE

WITH DYNAMIC PROGRAMMING



COMPUTING THE EDIT DISTANCE

WITH DYNAMIC PROGRAMMING

	ε	H	O	M	E
ε	0	1	2	3	4
H	1				
O	2				
U	3				
S	4				
E	5				

The distance between a word and an empty string is simply the length of the word

COMPUTING THE EDIT DISTANCE

WITH DYNAMIC PROGRAMMING

	ε	H	O	M	E
ε	0	1	2	3	4
H	1	0			
O	2				
U	3				
S	4				
E	5				

This is the minimum between:

$d(\varepsilon, H) + 1 = 2$

$d(H, \varepsilon) + 1 = 2$

$d(\varepsilon, \varepsilon) + 0 = 0$

COMPUTING THE EDIT DISTANCE

WITH DYNAMIC PROGRAMMING

	ε	H	O	M	E
ε	0	1	2	3	4
H	1	0	1		
O	2				
U	3				
S	4				
E	5				

This is the minimum between:

$d(HO, \varepsilon) + 1 = 3$

$d(H, H) + 1 = 1$

$d(H, \varepsilon) + 1 = 2$

COMPUTING THE EDIT DISTANCE

WITH DYNAMIC PROGRAMMING

	ε	H	O	M	E
ε	0	1	2	3	4
H	1	0	1	2	3
O	2	1	0	1	2
U	3	2	1	1	2
S	4	3	2	2	2
E	5	4	3	3	2

We compute each element of the matrix

The result is in the bottom right corner of the matrix

Computing the value for one cell requires constant time…

…and there are $O(|w_1| \times |w_2|)$ cells

THE EDIT DISTANCE

ADVANTAGES AND DISADVANTAGES

- By computing the edit distance we can find the set of words that are the closest to a misspelled word.
- However, computing the edit distance on the entire dictionary can be too expensive.
- We can use some heuristics to limit the number of words, like looking only at words with the same initial letter (hopefully this has not been misspelled).
- Or we can use k-grams to retrieve terms with low edit distance from the misspelled word.

K-GRAM INDEXES

THIS TIME FOR SPELLING CORRECTION

- We can try to retrieve terms with “many” k-grams in common with a word.
- We hypothesise that having “many” k-grams in common is indicative of a low edit distance.
- This might not be true. Consider the the word “*cata*”:
  - it has all of its 2-grams in common with “*catastrophic*”, but it is not a “good” correction.
  - “*cats*”, which has has fewer 2-gram in common, is a more reasonable correction

THE JACCARD COEFFICIENT

MEASURING THE OVERLAP OF TWO SETS

The Jaccard coefficient of two sets A and B is defined as:

$$\frac{|A \cap B|}{|A \cup B|}$$

We can use the Jaccard coefficient to select the terms obtained by looking at the k-grams in common.

In this “cata” and “catastrophe” have a Jaccard coefficient of $3/10$, while “cata” and “cats” of $1/2$.

To compute the Jaccard coefficient, we only need the length of the strings, n_common/(n_bg1 + n_bg2 - n_common).

EDIT DISTANCE AND K-GRAMS

IN PRACTICE

- For **Smaller Datasets**: Edit distance might be preferred due to its accuracy in finding closely related words.
- For **Larger Datasets** or Faster Performance: K-grams might be chosen for their efficiency and ability to handle a broader range of misspellings.
- **Hybrid Approaches**: Some systems use both methods in conjunction, first using k-grams to narrow down the list of candidate corrections and then applying edit distance to find the best match among those candidates.

EDIT DISTANCE vs K-GRAMS

IN PRACTICE

- For **Smaller Datasets**: Edit distance might be preferred due to its accuracy in finding closely related words.
- For **Larger Datasets** or Faster Performance: K-grams might be chosen for their efficiency and ability to handle a broader range of misspellings.
- **Hybrid Approaches**: Some systems use both methods in conjunction, first using k-grams to narrow down the list of candidate corrections and then applying edit distance to find the best match among those candidates.

CONTEXT-SENSITIVE CORRECTION

SOMETIMES CONTEXT IS IMPORTANT

- Sometimes all the words of a query are spelled correctly…
  …but one is actually the wrong word.
- Consider “Flights *form* Malpensa”.
  The correct query should have been “Flights *from* Malpensa”.
- How can we mitigate the problem?
- Substitute one at a time the words of the query with the most similar in the dictionary, perform the modified queries and look at the variants with most results.
- Can be expensive, but some heuristics can help (e.g., looking at common pairs of words)

PHONETIC CORRECTION

WHEN A WORD IS WRITTEN "AS IT SOUNDS"

- Sometimes the user does not know how to spell a word…
- …so he/she tries to write it based on the sound…
- …and gets the result wrong.
- We can try to correct this kind of error by using specific algorithms that tries to put similar-sounding words in the same equivalence class.
- These algorithms are language-specific (or, at least, non universal).
- For English we will see the **Soundex** algorithm.

SOUNDEX ALGORITHM

MARSHMALLOW	Keep the first letter unchanged through the algorithm
M0RS0M0LL00	Change all occurrences of A, E, I, O, U, H, W, Y to 0
M0620504400	Convert the letters according to the following table: 1) B, F, P, V 2) C, G, J, K, Q, S, X, Z 3) D,T 4) L 5) M, N 6) R
M6254400000	Remove all occurrences of 0 and pad the string with 0
M625	Return the first four positions (1 letter, 3 digits)

THE SOUNDEX ALGORITHM

HOW TO USE IT

- We can search for words with the same "phonetic hash" as the ones in the query.
- The mains ideas that make the Soundex algorithm work are:
    - Vowels are seen as interchangeable.
    - Consonants are assigned to different equivalence classes depending on how they sound.
- The algorithm, however, is not perfect. There can be words that sound similar with different "phonetic hashes" and vice versa.