INFORMATION RETRIEVAL

Laura Nenzi
lnenzi@units.it

Lecture 2

LECTURE OUTLINE

Tokenization,
Stop Words, Normalization,
Stemming & Lemmatization

Phrase Queries, Biwords,
positional postings

Array, linked lists,
and skip lists

Data Structures
for dictionaries

PRACTICAL PART
A PYTHON IMPLEMENTATION
OF A SIMPLE BOOLEAN
RETRIEVAL SYSTEM

IMPROVING THE QUALITY OF RETRIEVAL

BUILDING AN INVERTED INDEX

MAJOR STEPS

1. Collect the documents to be indexed

2. Tokenize the text, turning each document into a list of tokens

3. Do linguistic preprocessing, producing a list of normalized tokens, which are the indexing terms

4. Index the documents that each term occurs in by creating an inverted index, consisting of a dictionary and postings.

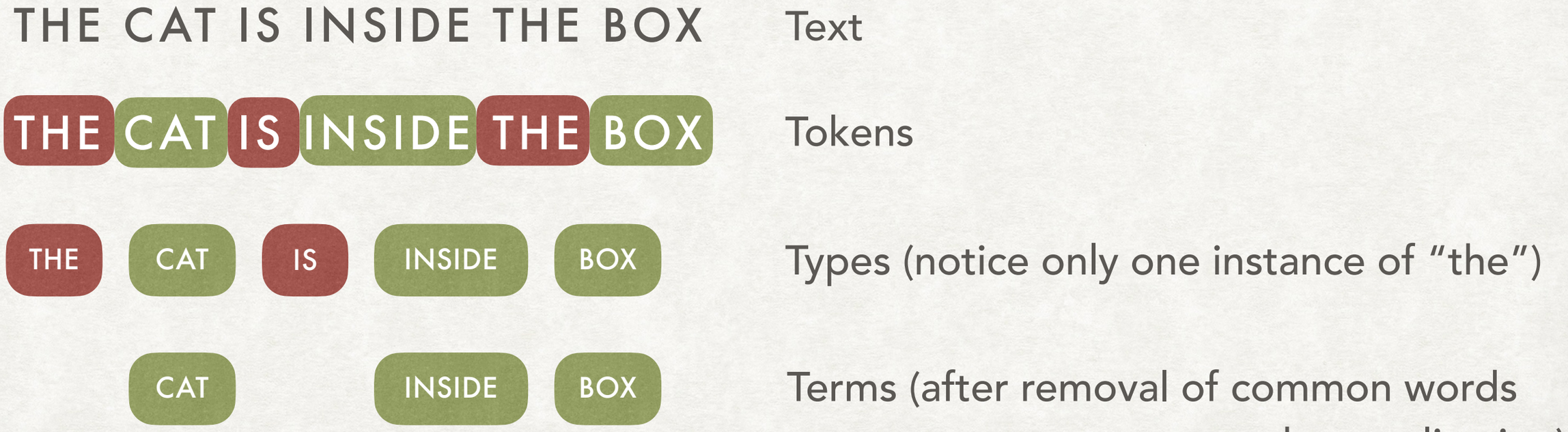
CHOOSING A DOCUMENT UNIT

- First step in the indexing process is to decide what is the granularity of the indexing
- For very long documents, the issue of indexing granularity arises, e.g. emails and attachments and zip files, html files, returning chapters or paragraphs instead of entire books, individual sentences as mini-document, …
- There is a precision/recall tradeoff
- For now, we will henceforth assume that a suitable size document unit has been chose
- The problems with large document units can be alleviated by use of explicit or implicit proximity search

TERMINOLOGY (4)

THIS TIME FOR TOKENIZATION

- **Token**: instance of a sequence of characters
- **Type**: collection of all tokens with the same character sequence
- **Term**: a type that is inserted into the dictionary



TOKENIZATION

SPLITTING THE TEXT IN WORDS

- The second step is to split a text sequence into tokens.
- In some cases deciding where to split the text sequence is simple…
- …but in many others it is not, even in English.
- For others languages it might not even be clear where a word ends and the next one starts.

EXAMPLES OF PROBLEMATIC TOKENIZATION

Text	Possible tokenizations
New York	[New] [York]
File-system	[File] [system], [File-system]
555-1234 567	[555] [1234] [567], [555-1234] [567], [555-1234 567]
Upper case	[Upper] [case]
Uppercase	[Uppercase]
O’Hara	[O] [Hara], [O’Hara]
Aren’t	[Aren][t], [Aren’t]

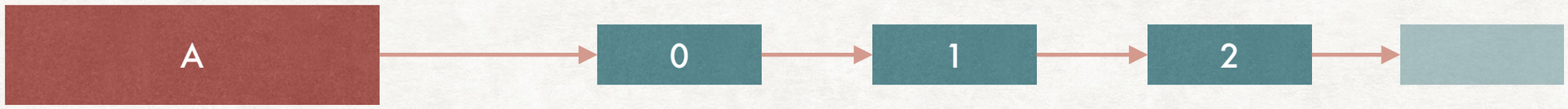
Possible (partial) solutions:

- use the same tokeniser for the documents and the queries
- use a collection of heuristics to decide where to split words

STOP WORDS

DROPPING COMMON TERMS

As anticipated before:



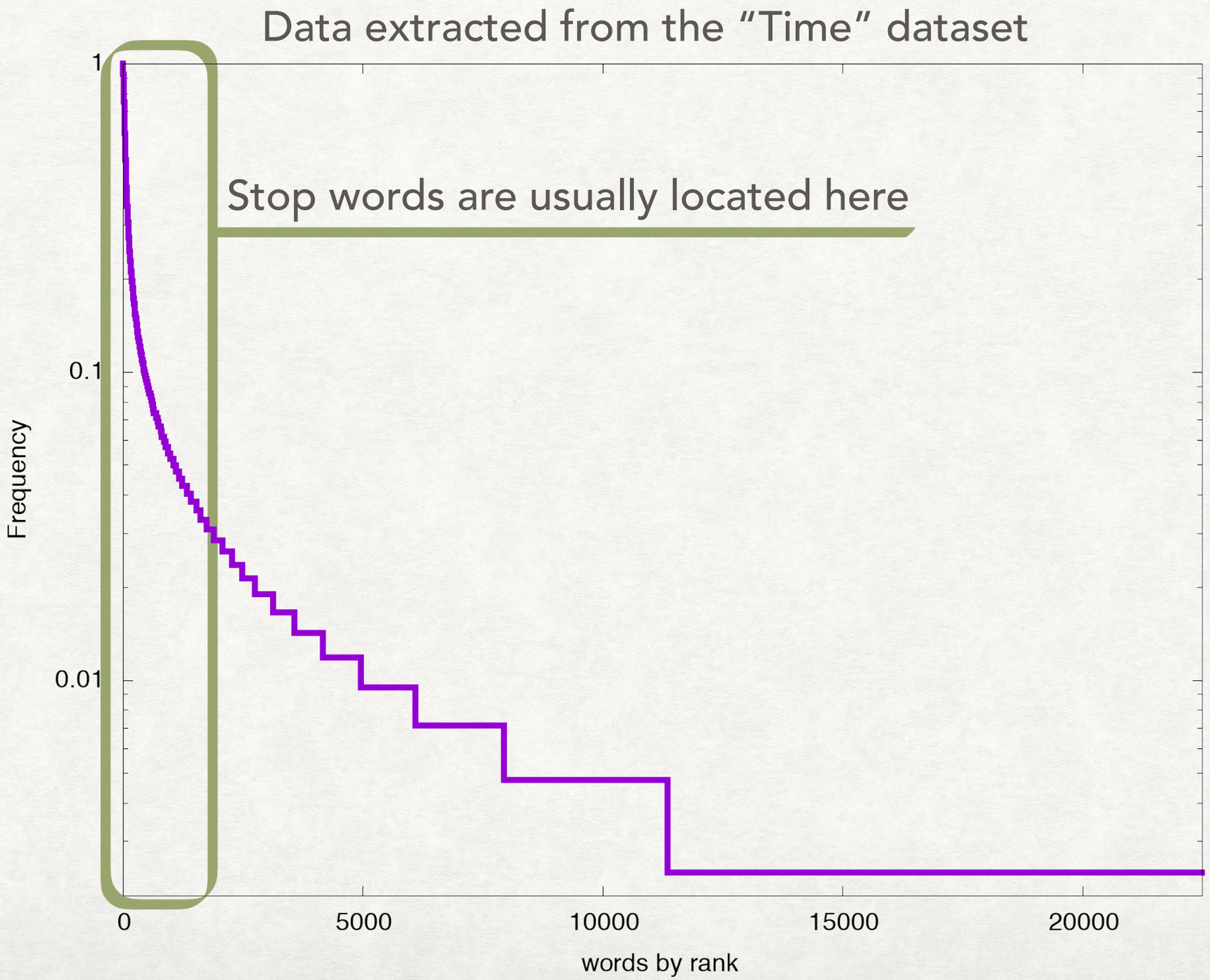
Some terms are not useful: “A” is in all the documents!

- **Stop words**: common words that do not help in selecting a document. They are discarded from the indexing and querying processes

- **Stop list**: list of stop words. Specific for a language/corpus. Usually consists of the most frequent words, curated for their semantic.

DISTRIBUTION OF WORDS

FREQUENCIES OF WORDS IN A CORPUS



STOP WORDS FOR THE ENGLISH LANGUAGE

AND STOP WORDS FOR SPECIFIC TOPICS

- You can find multiple lists of stop words for the English language. They usually include words like:
  - a, about, above, after, again…
  - … the, their, theirs, … , your, yours, yourself, yourselves.
- The list of stop words is language specific: stop words in Italian are different (additional challenge: you might need to infer the language of a document).
- Stop lists can be specific by topic. E.g., in a “books on cats” corpus, the word “cat” might be a stop word.

DROPPING STOP WORD

SOMETIMES IT IS USEFUL

- Using a stop list significantly reduces the number of postings that a system has to store
- And a lot of the time not indexing stop words does little harm
- but..

PROBLEMS WITH STOP WORDS

SOMETIMES STOP WORDS ARE USEFUL

- You now have a IR system that removes all stop words.
- You receive the queries:
  - ~~To be or not to be~~
  - Dr ~~Who~~
  - ~~Do it yourself~~
  - ~~Let it be~~
- Removing stop words can reduce the *recall*.

PROBLEMS WITH STOP WORDS

SOMETIMES STOP WORDS ARE USEFUL

- A single stop word alone can usually be removed…
- …but in a *phrase search* it might be important
- The trend has been from large stop lists (200–300 terms) to very small stop lists (7-12 terms) or no stop word list but:
  - Use compression techniques to reduce the storage requirements
  - Use weighting to limit the impact of stop words
  - Use specific algorithms to limit the runtime impact of stop words

NORMALIZATION

REMOVING SUPERFICIAL DIFFERENCES

- The same word can be written in different ways and it must be normalized to allow the matching to occur.
- The idea is to define *equivalence classes* of terms, for example:
  - By ignoring capitalization (e.g., “HOME”, “home”, “HoMe”), always or just at the beginning of a sentence and titles
  - By removing accents and diacritics (e.g., cliché is considered the same as cliche).
  - Other normalization steps specific to the language, like ignoring spelling differences (e.g., “colors” vs “colours”).

RELATIONS BETWEEN UNNORMALIZED TOKENS

AN ALTERNATIVE TO EQUIVALENCE CLASSES

Sometimes capitalization and other features are important

windows (can mean both the object and the OS) ⇄ Windows (the OS)

This can be solved by saving (possibly asymmetric) relations between token in a *query expansion list*

Query Term	Equivalent terms
Windows	Windows
windows	Windows, windows, window
window	windows, window

RELATIONS BETWEEN UNNORMALIZED TOKENS

AN ALTERNATIVE TO EQUIVALENCE CLASSES

Sometimes capitalization and other features are important

windows (can mean both the object and the OS)  Windows (the OS)

Or it can be solved performing the expansion during index construction.

Term	Indexing
automobile	automobile, vehicle
car	automobile, vehicle, car

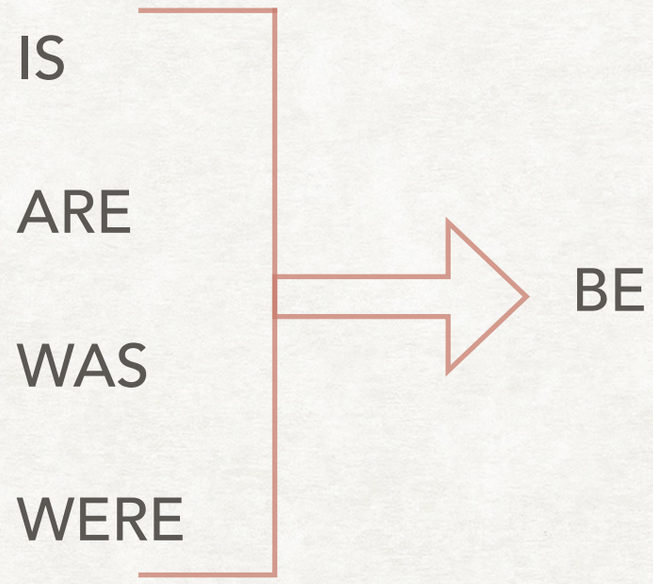
PRO AND CONTRO

WHICH IS IT BETTER?

- Equivalence classing reduce information but has less postings to store and merge
- A *query expansion list* adds a query expansion dictionary and requires more processing at query time; the second method requires more space for storing postings.
- But they are more flexible because the expansion lists can overlap while not being identical (e.g. asymmetry in expansion)
- Traditionally, expanding the space required for the postings lists was seen as more disadvantageous, but with modern storage costs, the increased flexibility that comes from distinct postings lists is appealing.

STEMMING AND LEMMATIZATION

REDUCE WORDS TO A COMMON BASE FORM



Idea
reduce all variants of a word
to a "common root"

Two main ways: stemming and lemmatization

Based on heuristics

Uses a vocabulary and
morphological analysis

PORTER STEMMER

MOST USED STEMMER FOR THE ENGLISH LANGUAGE

Invented in 1979 (published 1980) by Martin Porter,
it is one of the most common stemmers for the English language

Five stages applied sequentially.

Each stage consists of a series of rewriting rules for words,
an example is given here

Rule	
SSES → SS	caressess → caress
IES → I	poinies → poni
SS → SS	caress → caress
S →	cats → cat

Porter Stemmer implementations: https://tartarus.org/martin/PorterStemmer/
(or you can read the original paper and the BCLP implementation)

WHAT CAN GO WRONG

- While it helps a lot for some queries, it equally hurts performance a lot for others
- Stemming increases recall while harming precision
- e.g. the Porter stemmer stems all of the following words:

  operate operating operates operation operative operatives operational -> oper

  losing precision on queries such as:

  operational AND research, operating AND system, operative AND dentistry

WORDNET LEMMATIZER

LEMMATIZER FROM THE NLTK LIBRARY

It uses the WordNet lexical database (Lemmas, Synonyms, Definitions POS info, Semantic relations)

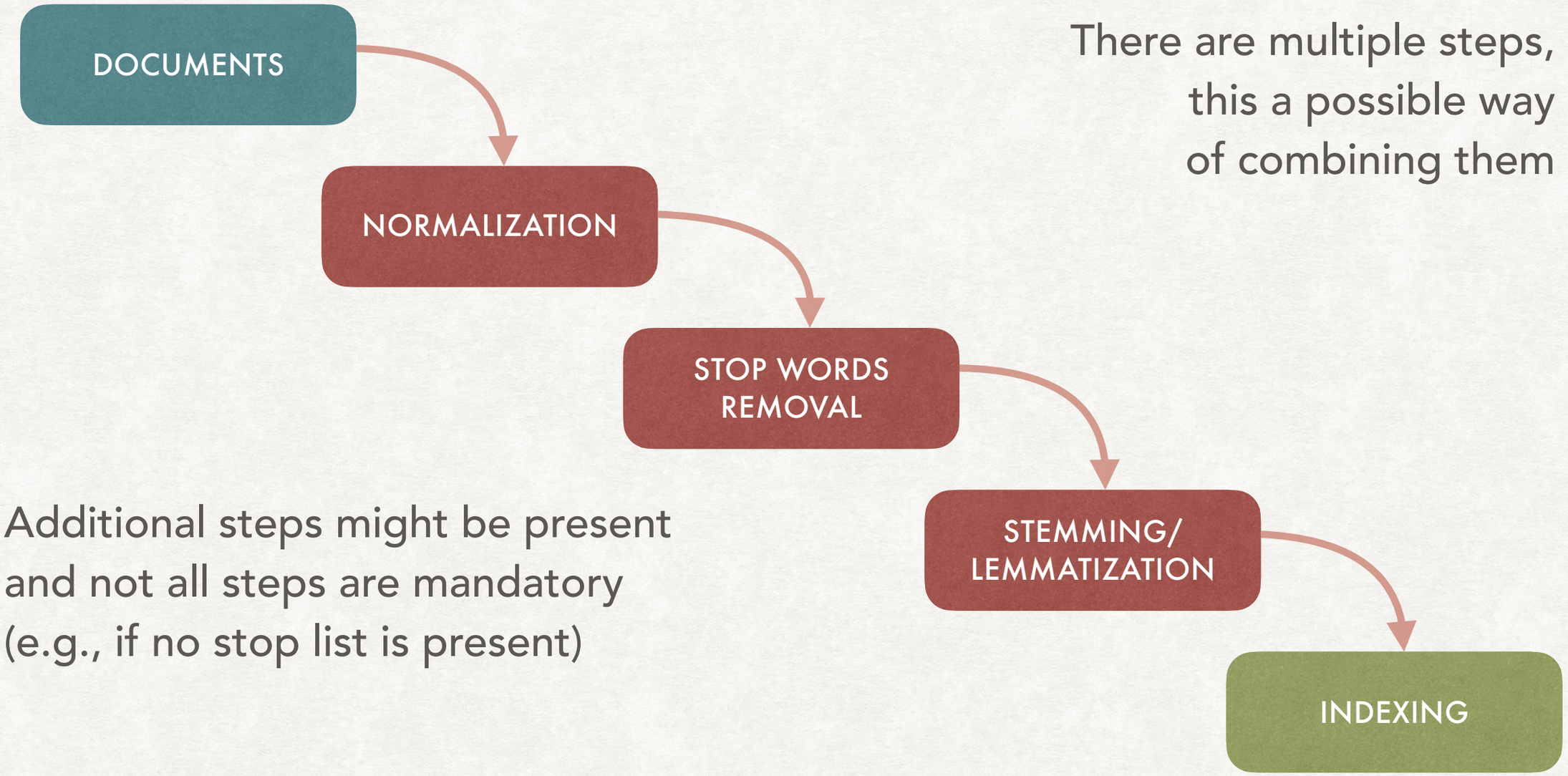
It requires a part of speech (POS) tagging for best results. It returns the base (dictionary) form of a word, depending on its POS.

It's accurate when the correct POS is specified.

Form	POS	Lemma
running	verb	run
better	adjective	good
cars	noun	car
studies	verb	study

Porter Stemmer implementations: https://tartarus.org/martin/PorterStemmer/
(or you can read the original paper and the BCLP implementation)

THE "PREPROCESSING" PIPELINE



There are multiple steps,
this a possible way
of combining them

Additional steps might be present
and not all steps are mandatory
(e.g., if no stop list is present)

The same steps applied to the queries!

ANSWERING PHRASE QUERIES

OUR GOAL

EXTENDING THE QUERY LANGUAGE

- We want to be able to ask queries consisting of multiple consecutive words:
  - “calico cat”
  - “University of Trieste”
- A common syntax for this kind of queries is to enclose the words in double quotes.
- Two approaches shown: *biword indexes* and *positional indexes*.

BIWORD INDEXES

WORKING ON PAIRS OF WORDS

THE CAT IS INSIDE THE BOX

Text

THE CAT | CAT IS | IS INSIDE | INSIDE THE | THE BOX

Terms

- The terms are pairs of words
- Queries need to be “rewritten”:

“inside the box” → “inside the” AND “the box”

BIWORD INDEXES

POSSIBLE PROBLEMS

Text: INSIDE THE HOUSE THERE IS THE BOX

Original Query: “inside the box” No Match

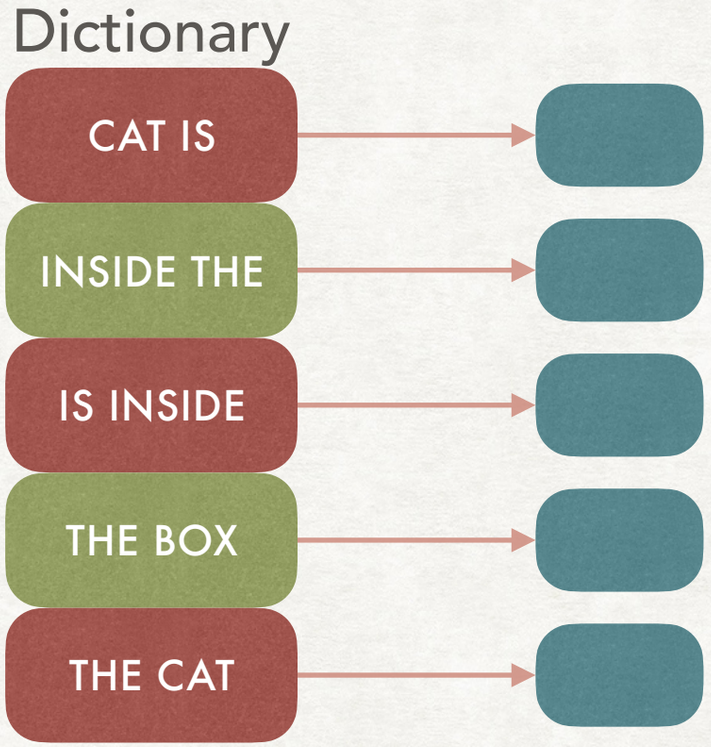
Rewritten Query: “inside the” AND “the box” Match

Rewriting the query might generate false positives
(but it works quite well in practice)

BIWORD INDEXES

POSSIBLE PROBLEMS

To answer the query
we would need to find
all terms containing “cat”



We also need an index of single-word terms!

BIWORD INDEXES

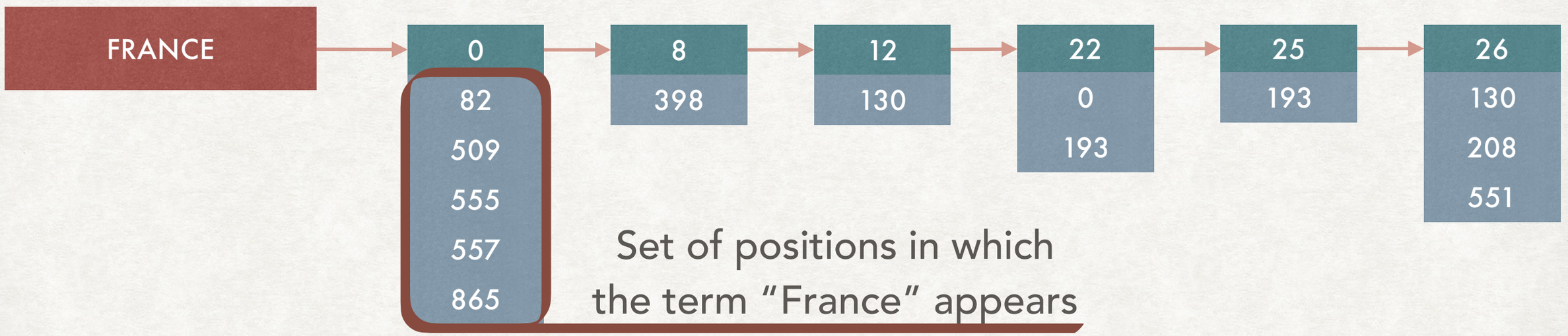
EXTENSIONS AND FURTHER OBSTACLES

- The idea of using pair of words as terms can be extended to any length, reducing the risk of false positives…
- …but increasing the amount of space needed.
- If the number of words in a term is variable it is called *phrase index*.
- It is also possible to “tag” the part of speech (i.e., names, verbs, articles, prepositions, etc.) to add pairs of names separated by articles and prepositions to the index.
  - E.g., in “door at the entrance”, “door entrance” is considered a term

POSITIONAL INDEXES

ADDING POSITIONS TO THE POSTINGS

One way to answer a phrase query is to add, for each posting, the set of positions in which the term appear in the document.

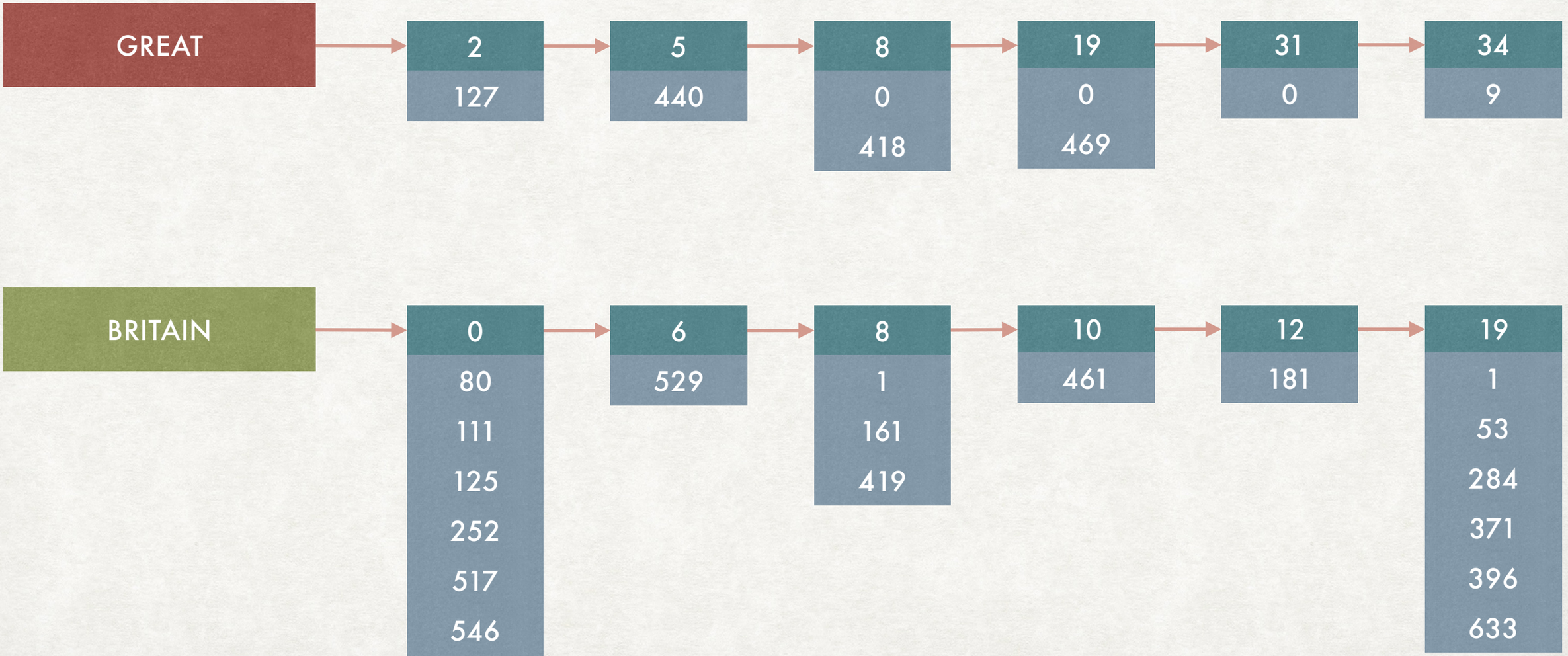


ANSWERING A PHRASE QUERY

WITH POSITIONAL INDEXING

QUERY	"GREAT BRITAIN"

ANSWER

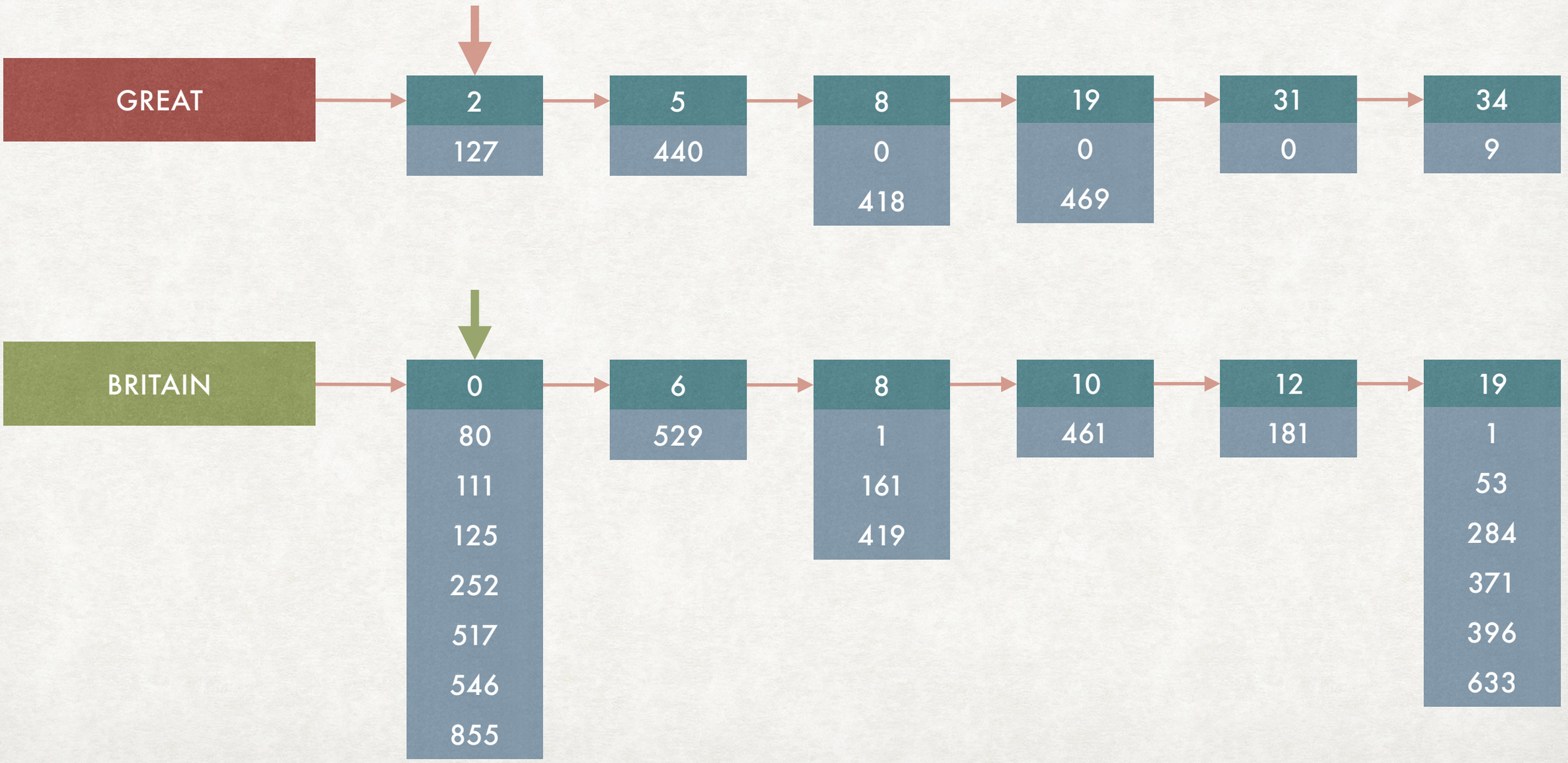


ANSWERING A PHRASE QUERY

WITH POSITIONAL INDEXING

QUERY | "GREAT BRITAIN"

ANSWER

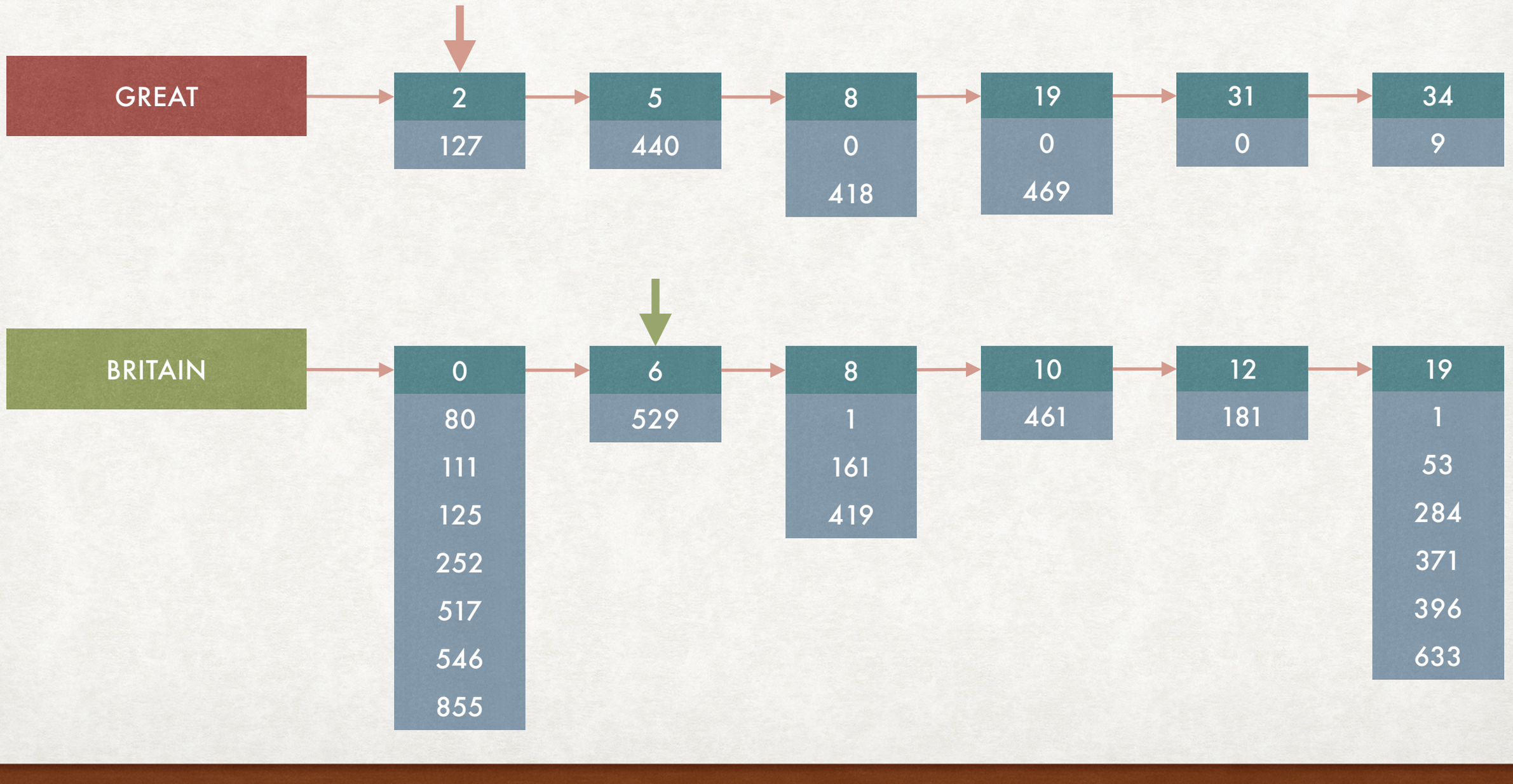


ANSWERING A PHRASE QUERY

WITH POSITIONAL INDEXING

QUERY | "GREAT BRITAIN"

ANSWER

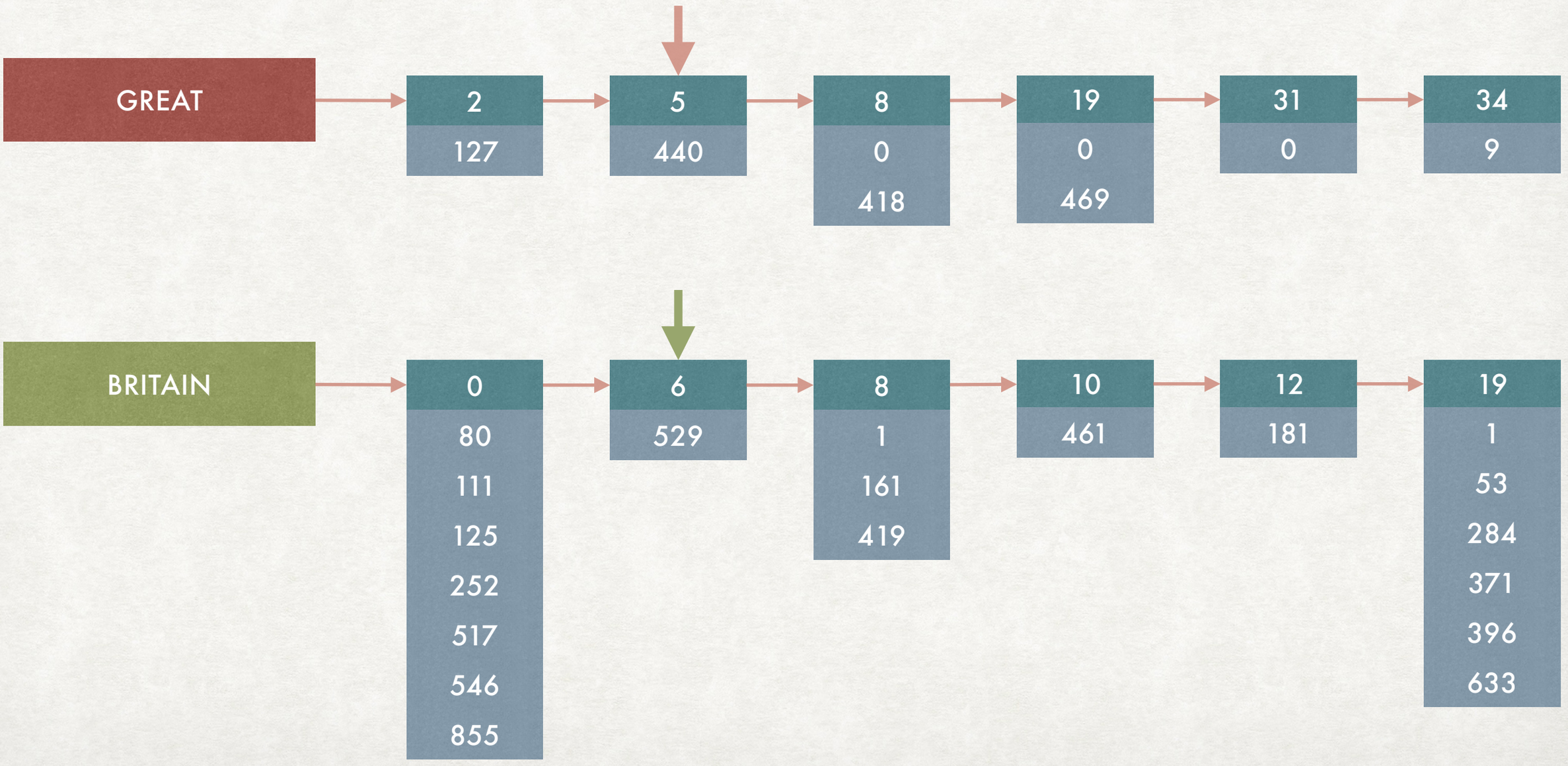


ANSWERING A PHRASE QUERY

WITH POSITIONAL INDEXING

QUERY | "GREAT BRITAIN"

ANSWER

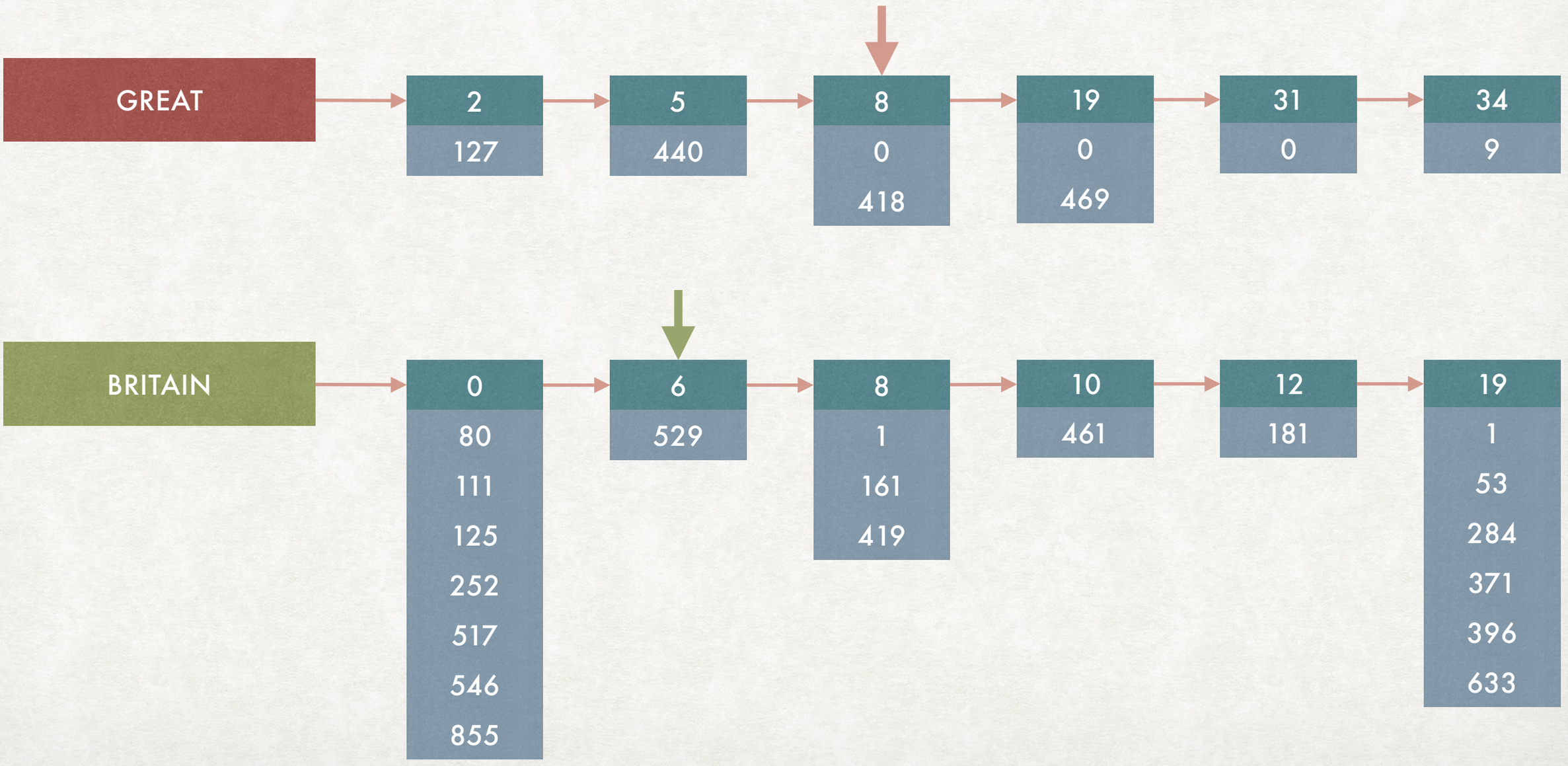


ANSWERING A PHRASE QUERY

WITH POSITIONAL INDEXING

QUERY | "GREAT BRITAIN"

ANSWER

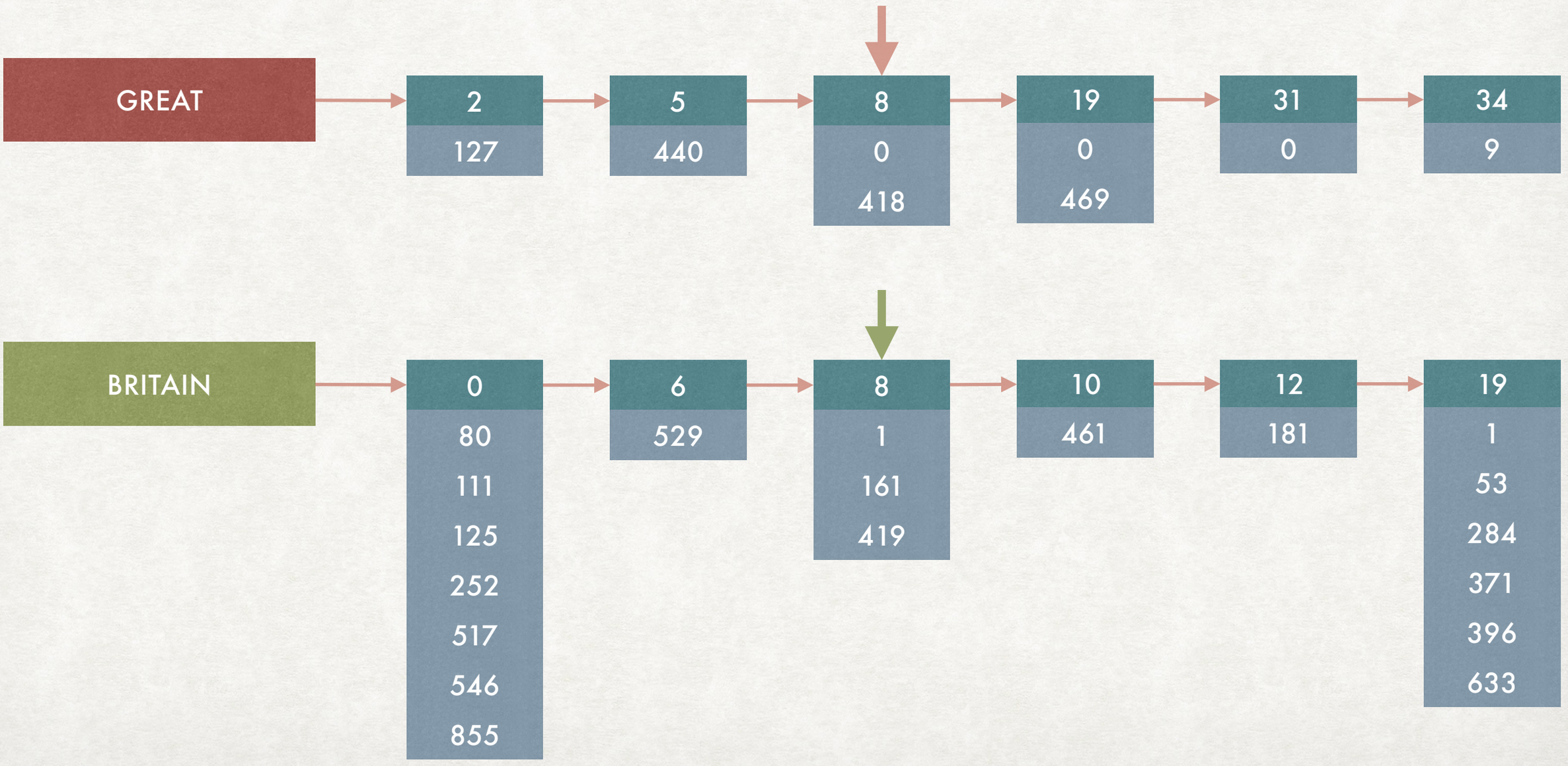


ANSWERING A PHRASE QUERY

WITH POSITIONAL INDEXING

QUERY | "GREAT BRITAIN"

ANSWER

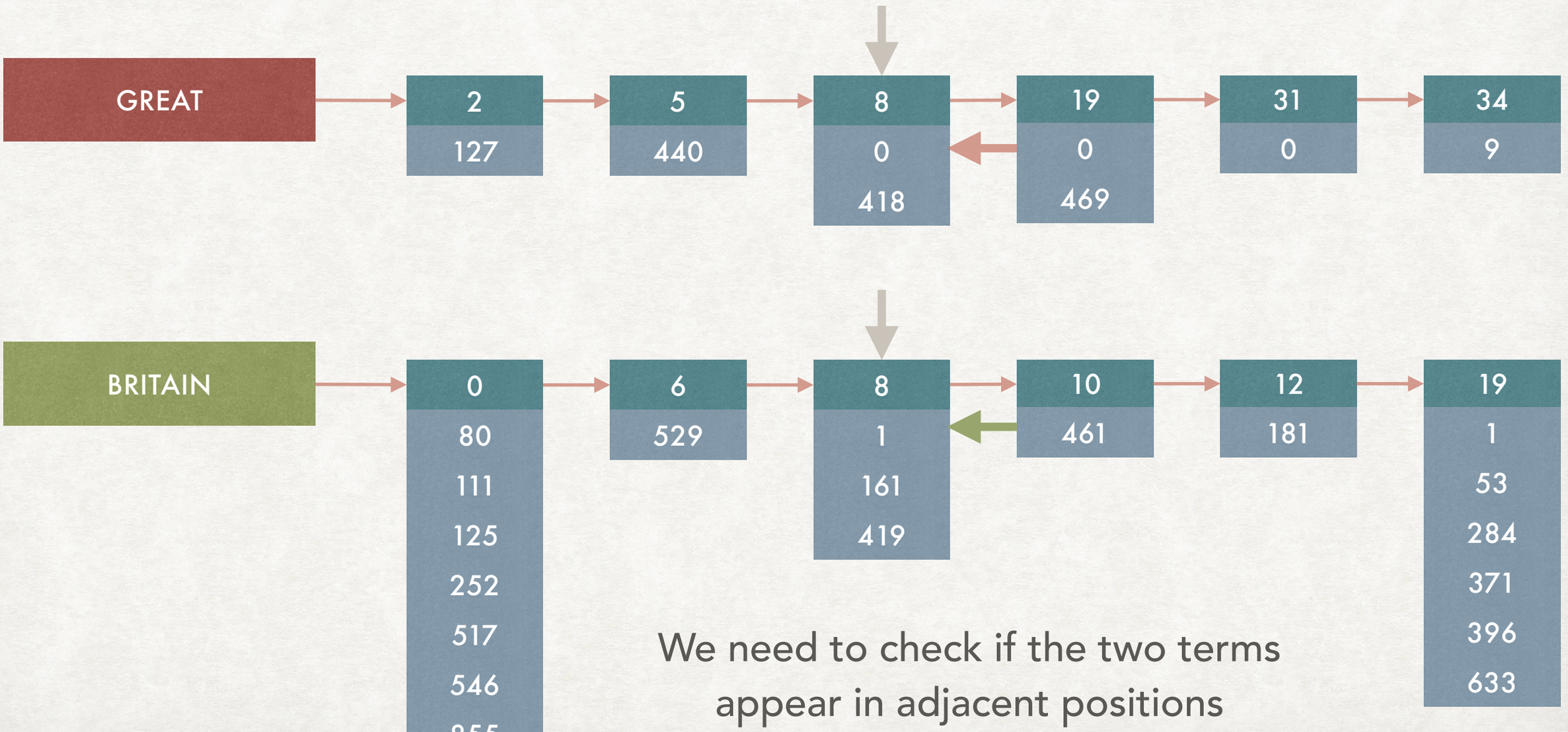


ANSWERING A PHRASE QUERY

WITH POSITIONAL INDEXING

QUERY | "GREAT BRITAIN"

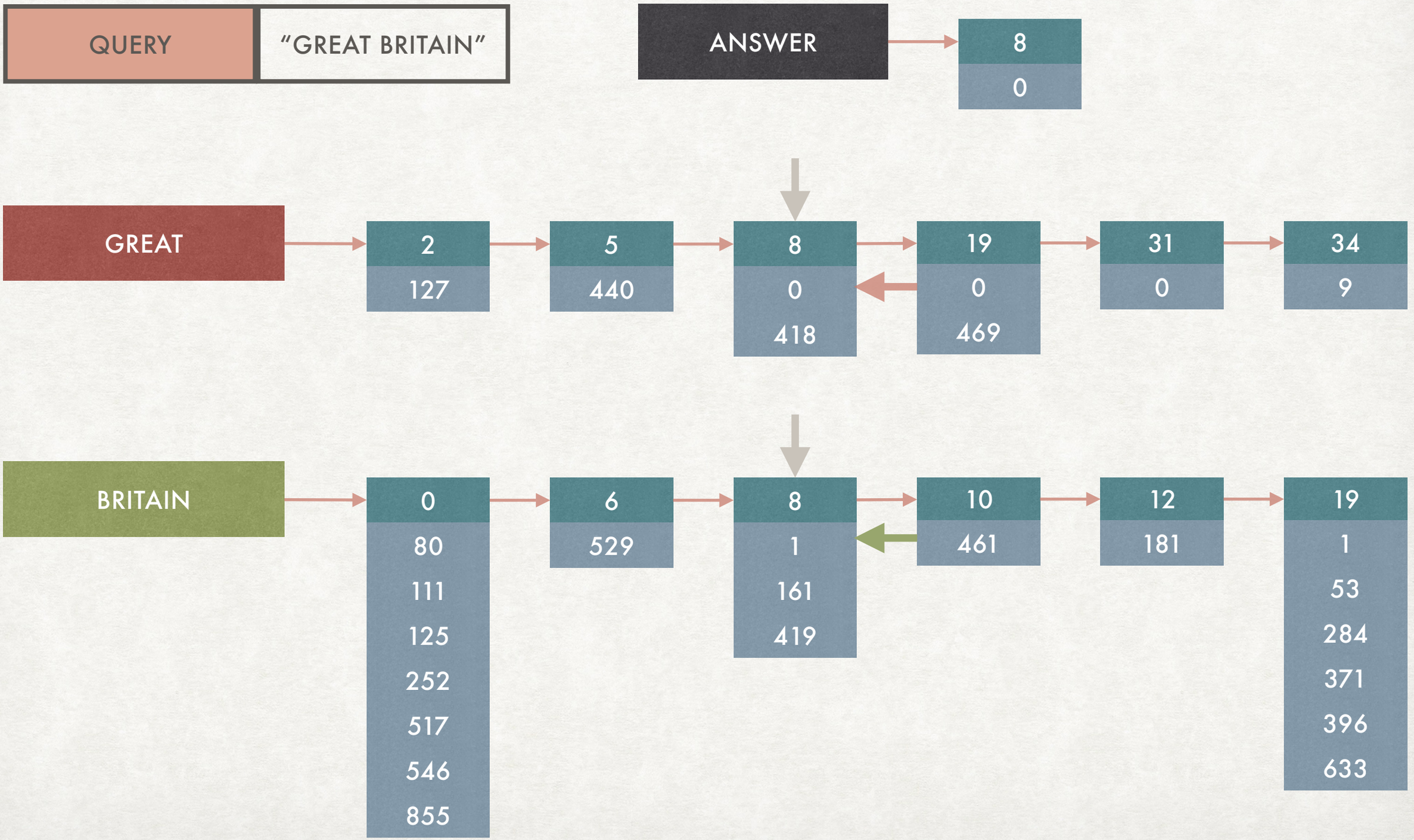
ANSWER



We need to check if the two terms appear in adjacent positions

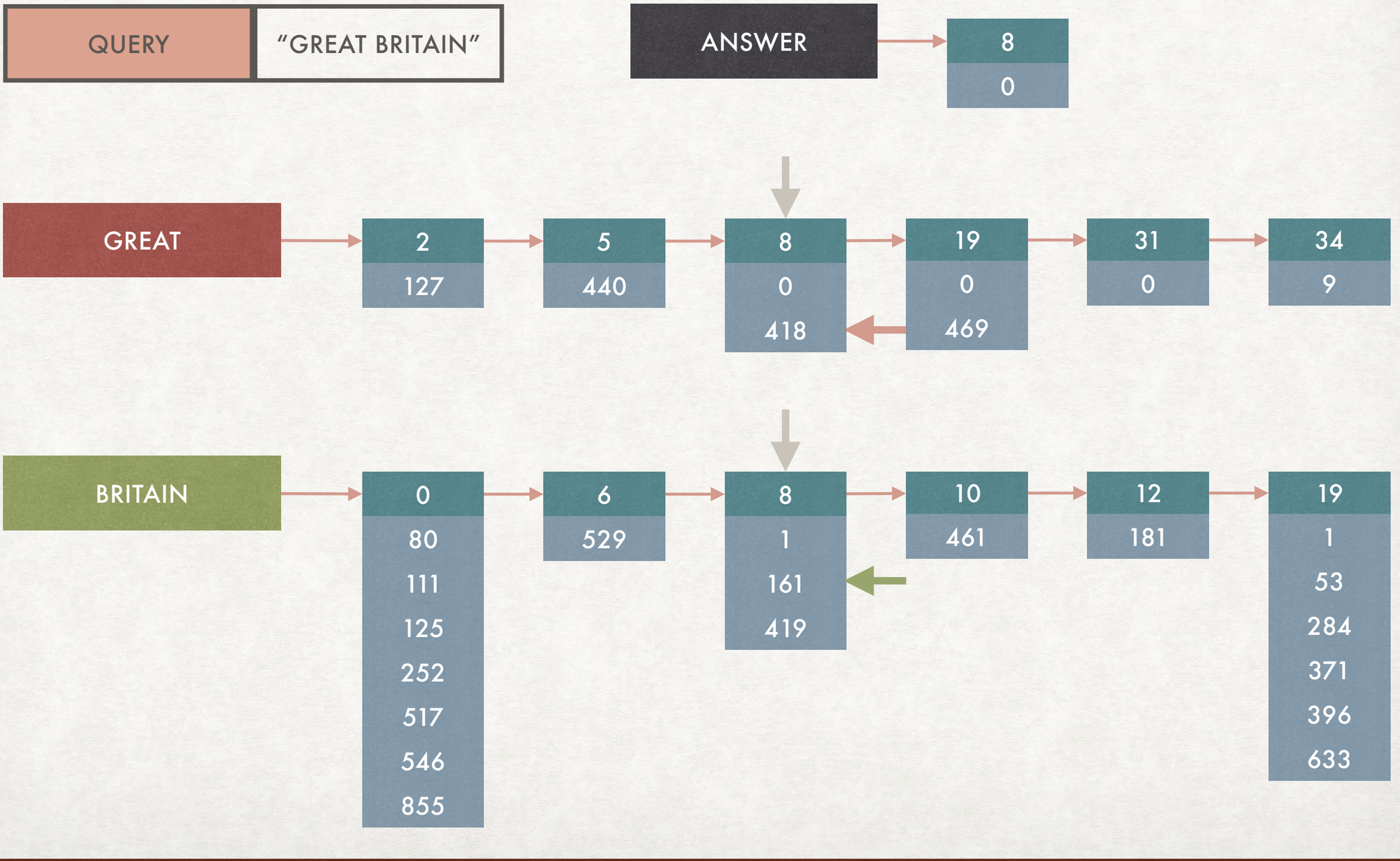
ANSWERING A PHRASE QUERY

WITH POSITIONAL INDEXING



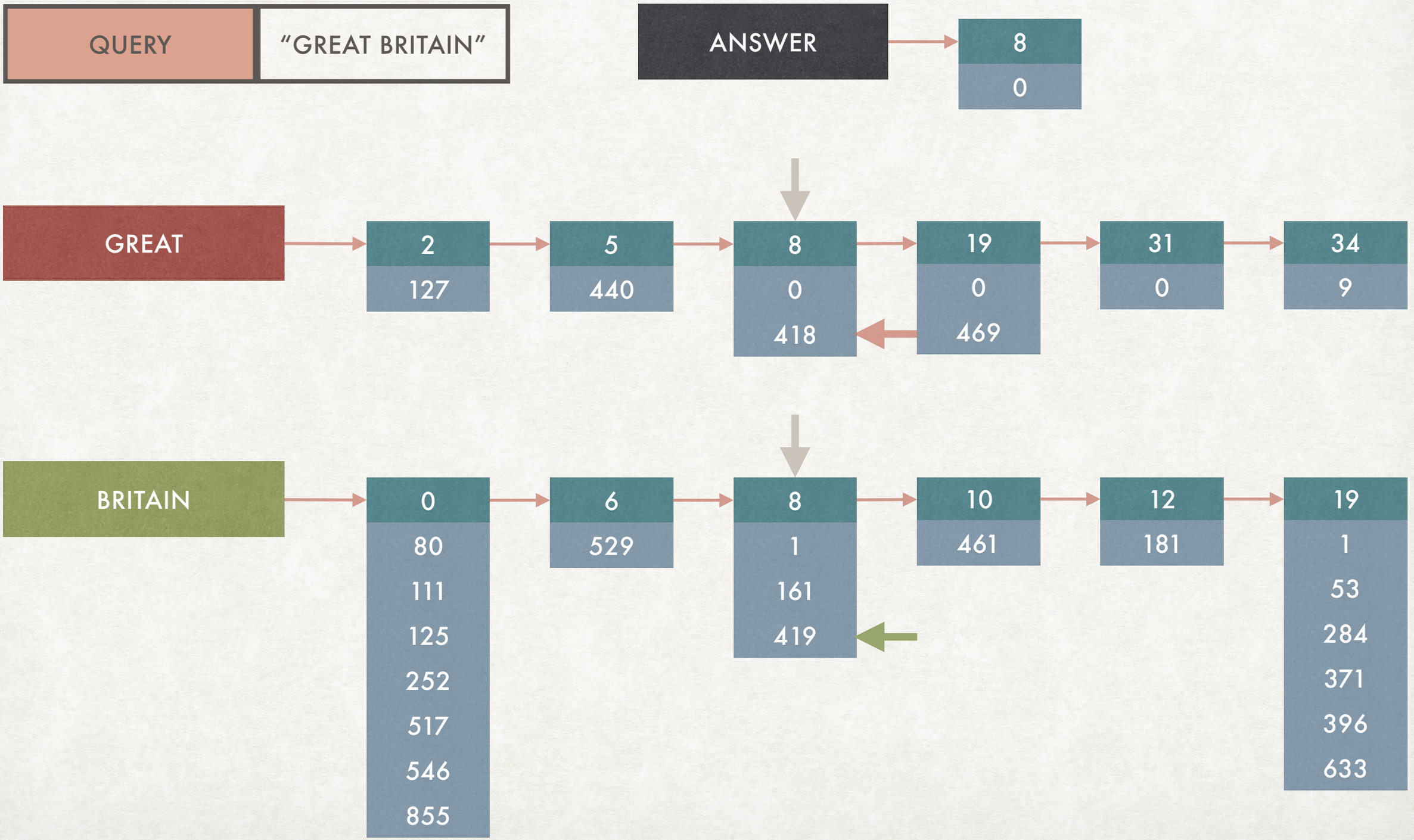
ANSWERING A PHRASE QUERY

WITH POSITIONAL INDEXING



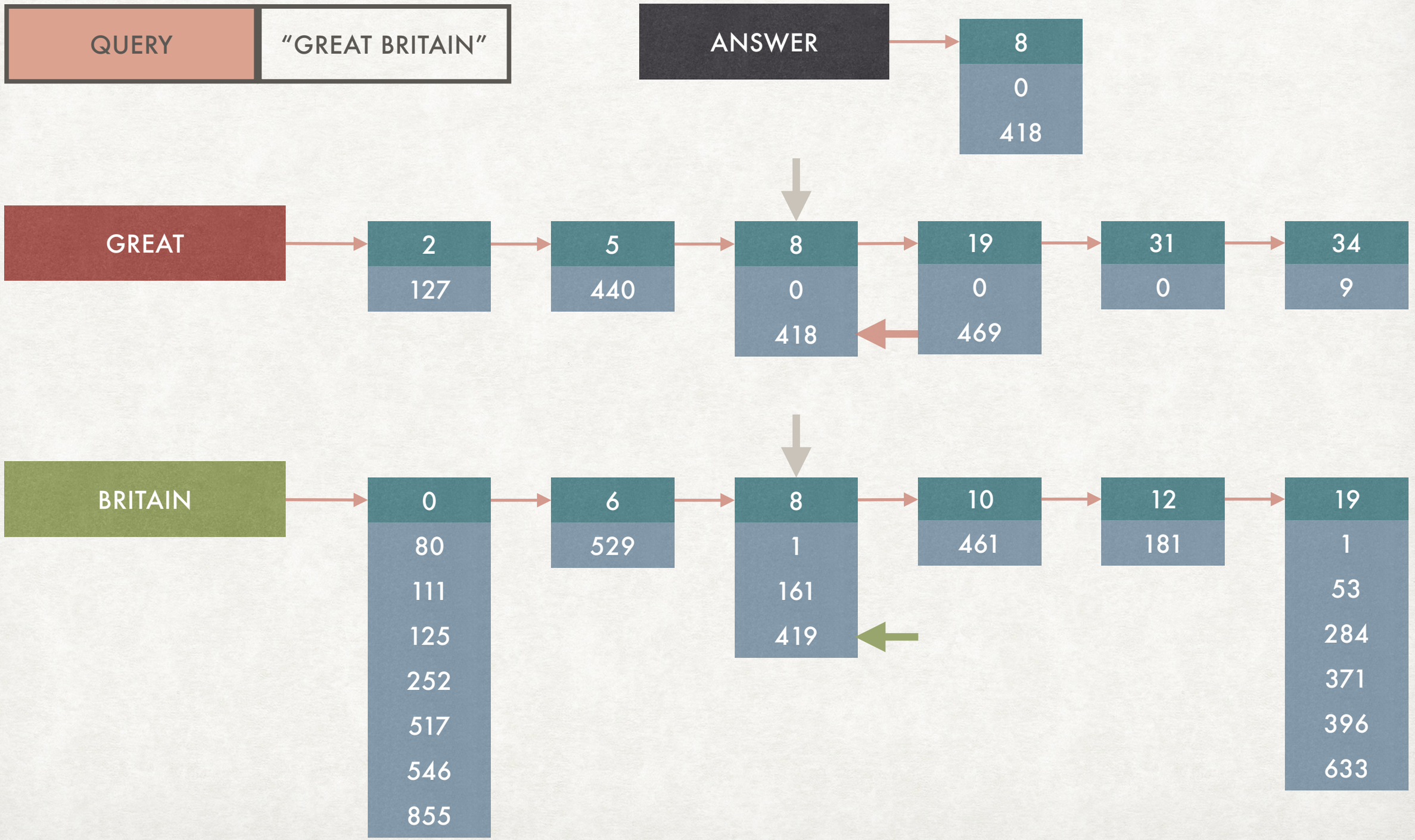
ANSWERING A PHRASE QUERY

WITH POSITIONAL INDEXING



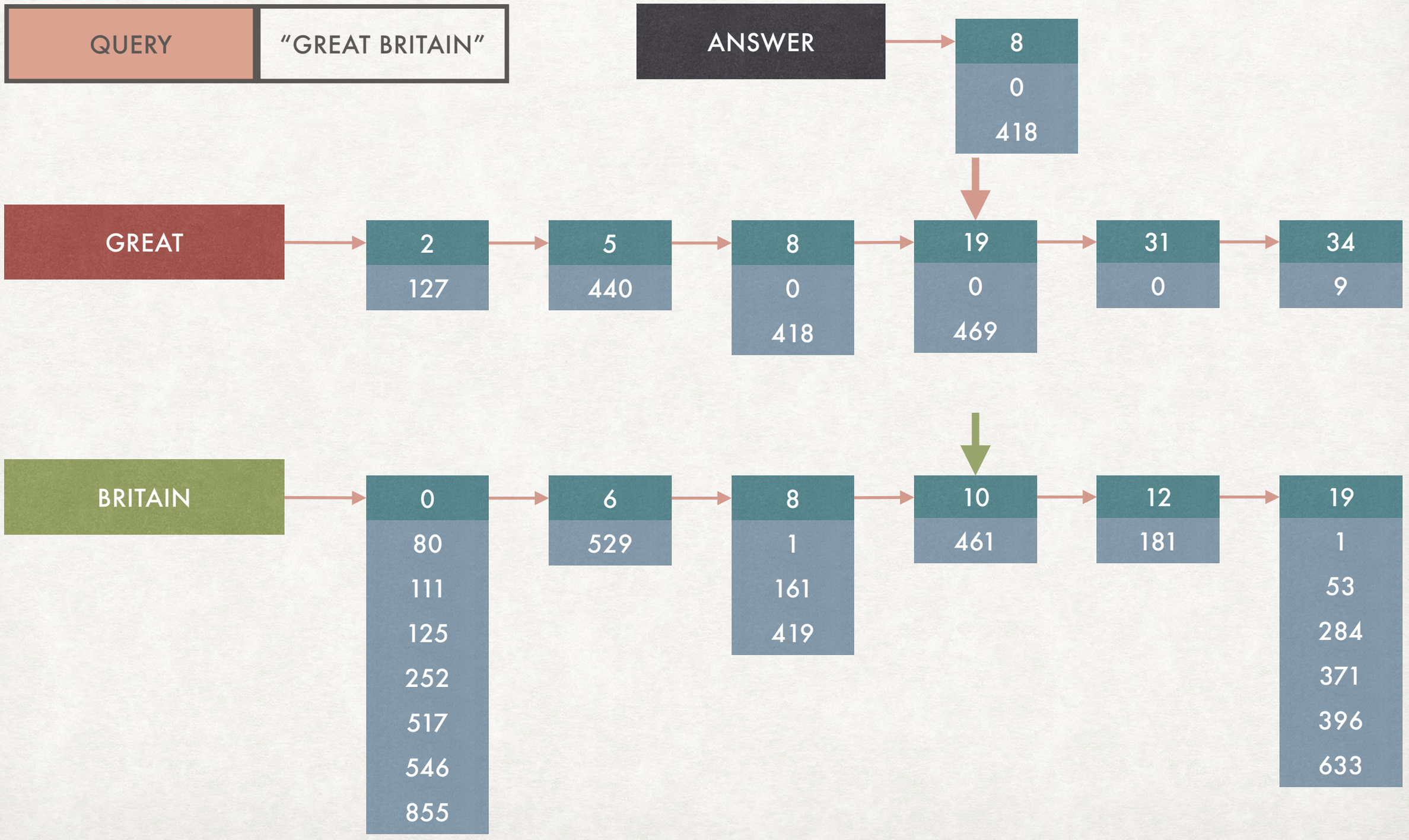
ANSWERING A PHRASE QUERY

WITH POSITIONAL INDEXING



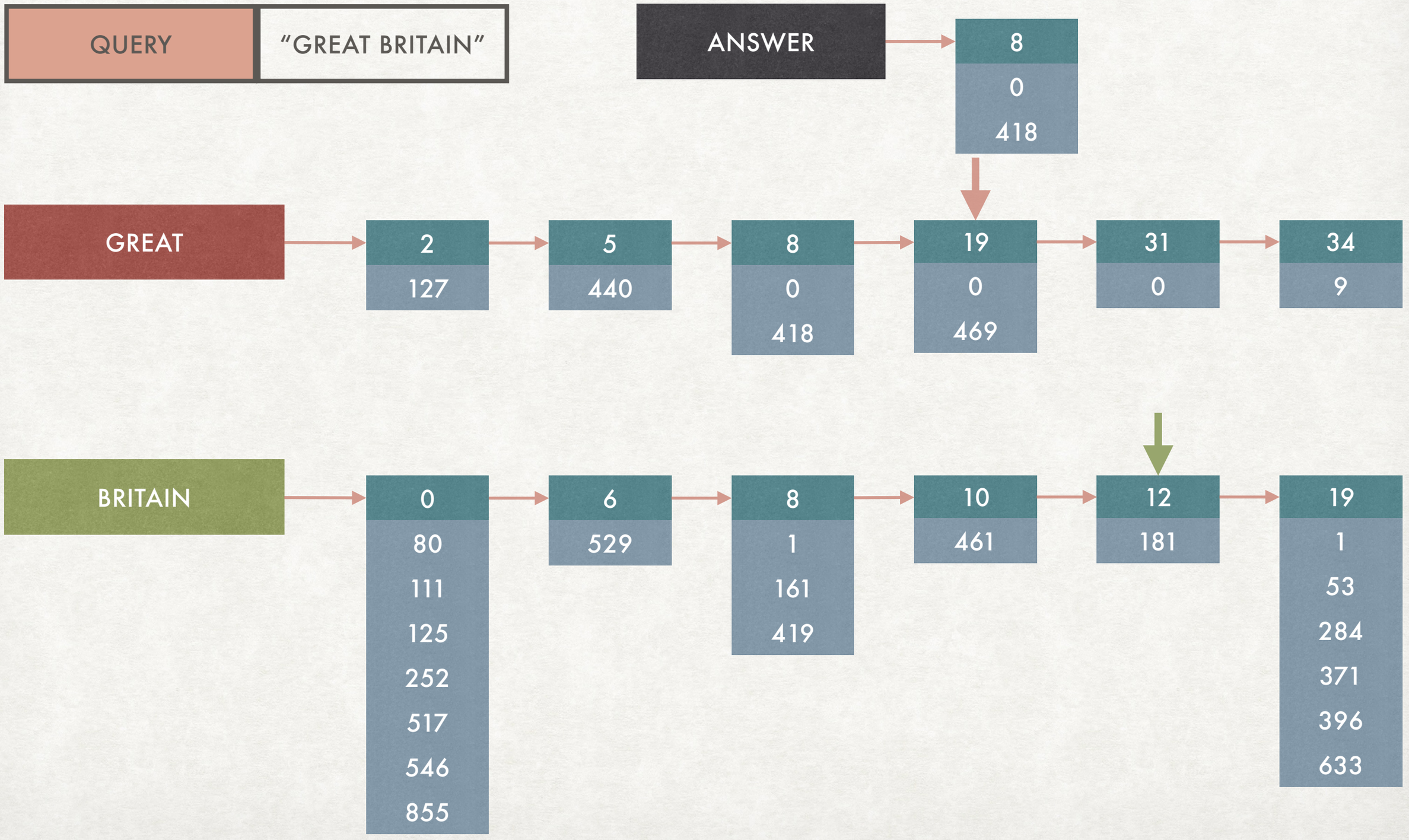
ANSWERING A PHRASE QUERY

WITH POSITIONAL INDEXING



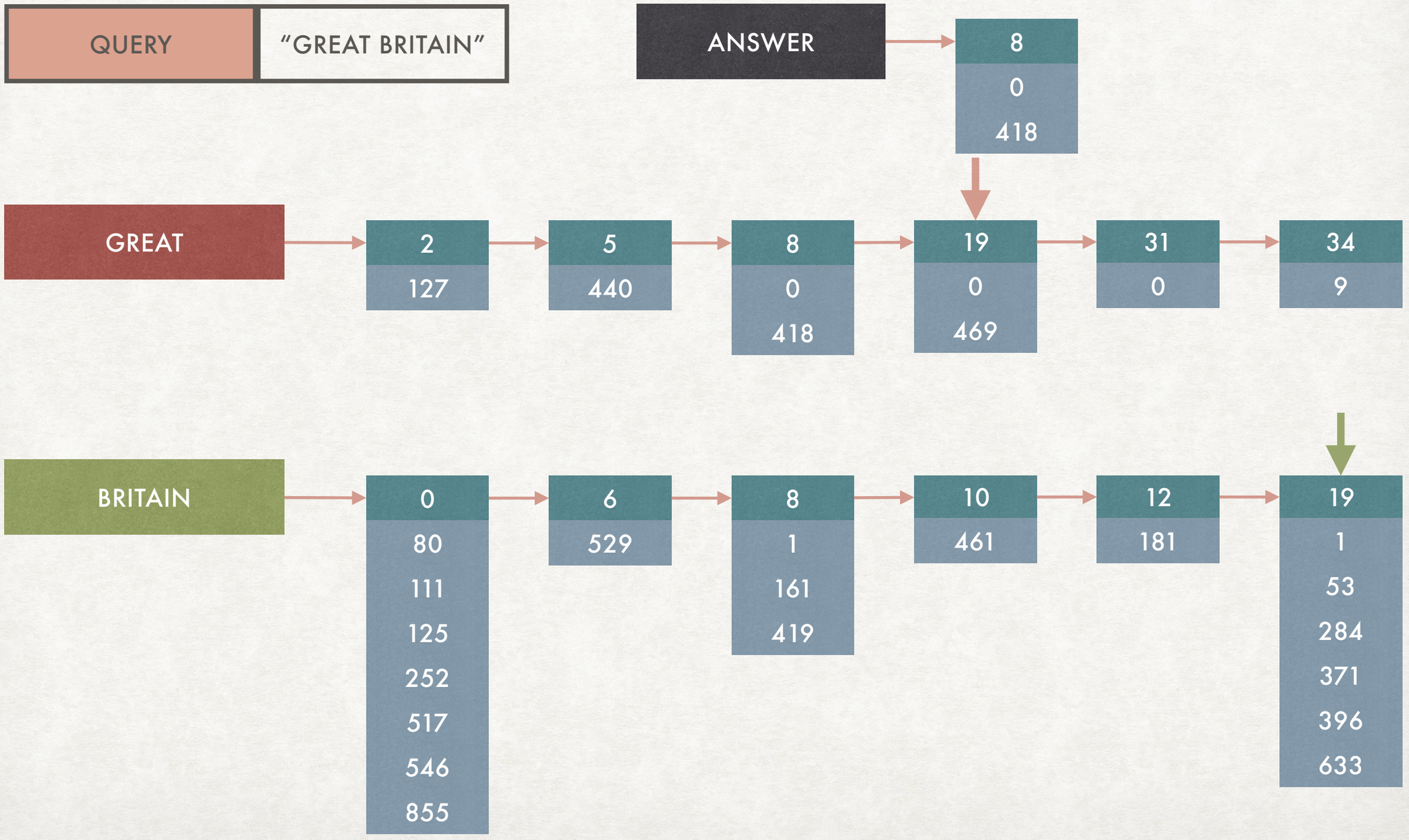
ANSWERING A PHRASE QUERY

WITH POSITIONAL INDEXING



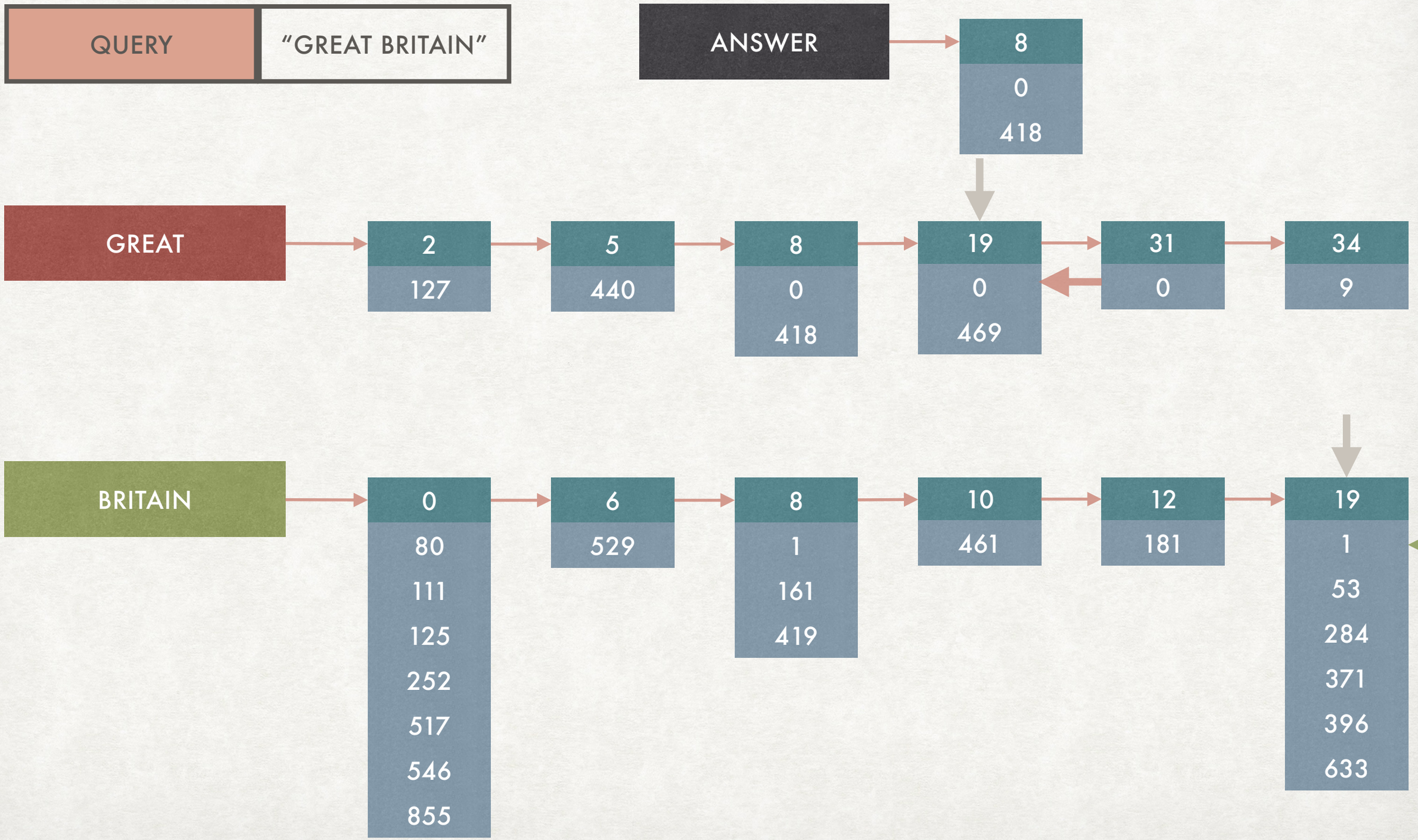
ANSWERING A PHRASE QUERY

WITH POSITIONAL INDEXING



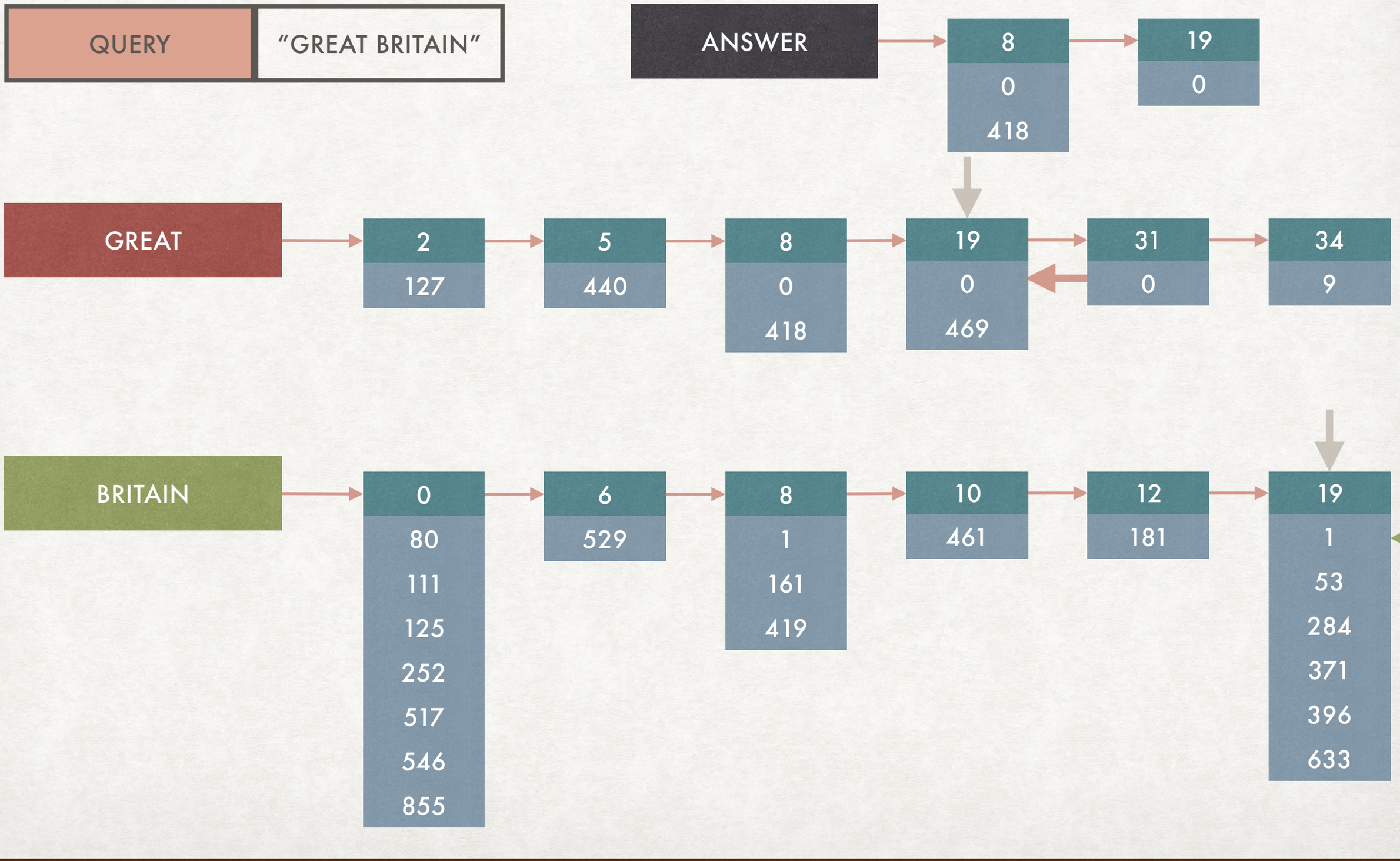
ANSWERING A PHRASE QUERY

WITH POSITIONAL INDEXING



ANSWERING A PHRASE QUERY

WITH POSITIONAL INDEXING

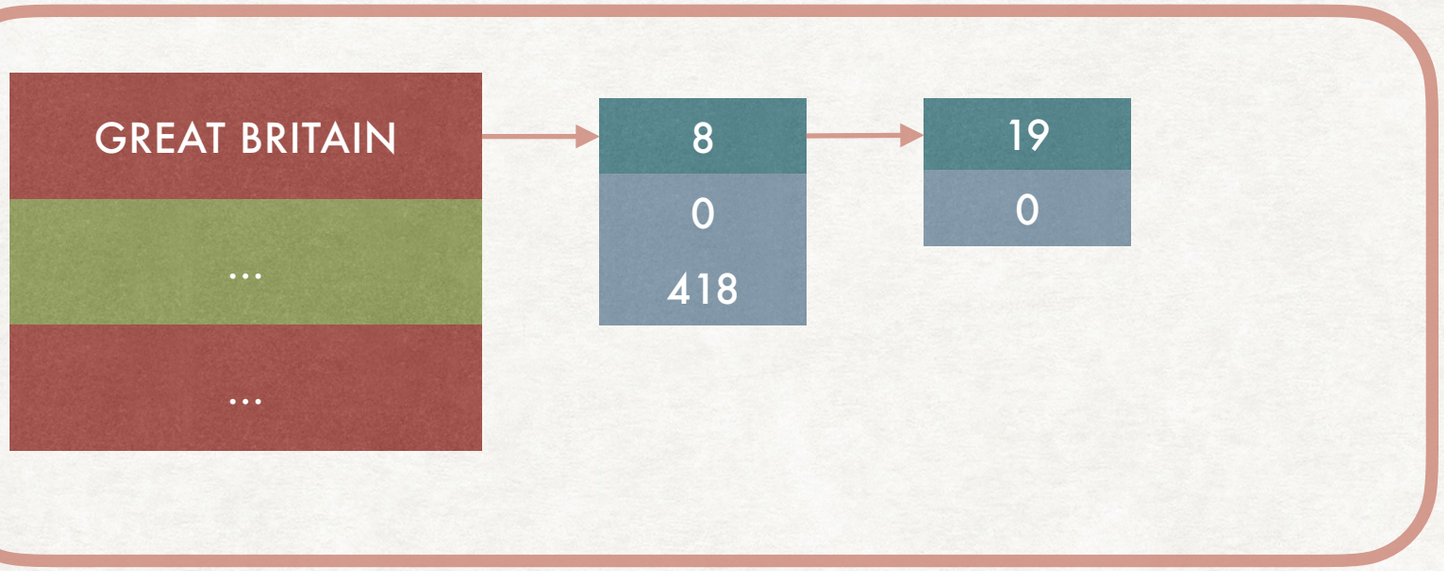


POSITIONAL INDEXING: SUMMARY

THE GOOD, THE BAD, AND THE UGLY

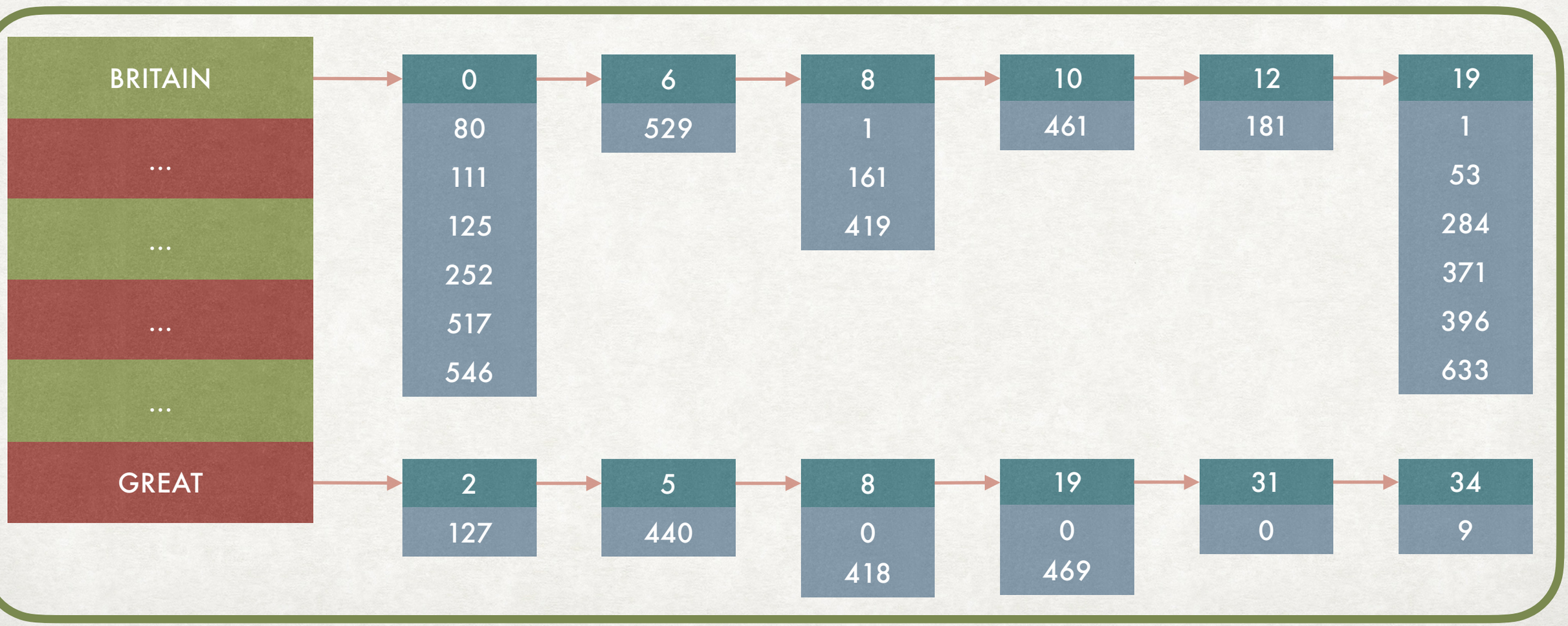
- The positional index can be used to support the operators of the form "$term_1$ /k $term_2$" with k an integer indicating the maximum number of words that can be between $term_1$ and $term_2$.
- The complexity of performing a query is not bounded anymore by the number of documents, but by the number of terms
- The size of the index now depends on the average document size.

COMBINING BIWORD AND POSITIONAL INDEXES



Phrase index for
frequently asked queries

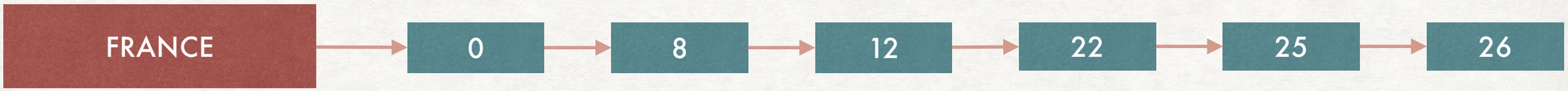
Positional index for
all other queries



IMPROVING THE INVERTED INDEX

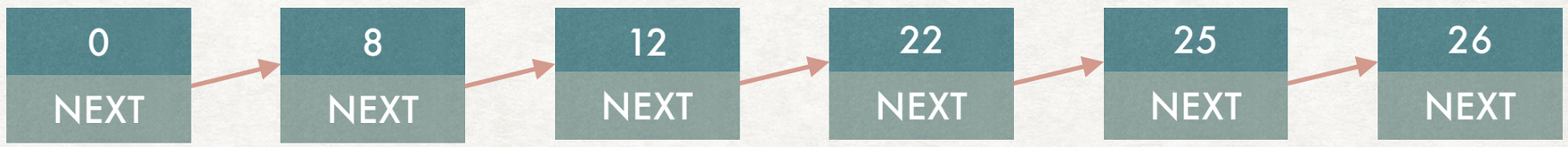
ARRAYS OR LINKED LISTS?

WHAT TO USE FOR THE POSTING LISTS?



Which data structures should we actually use for the postings list?

Singly linked lists



cheap insertion and updates

pointer overhead, poor memory locality (pointers chasing)

Variable length arrays

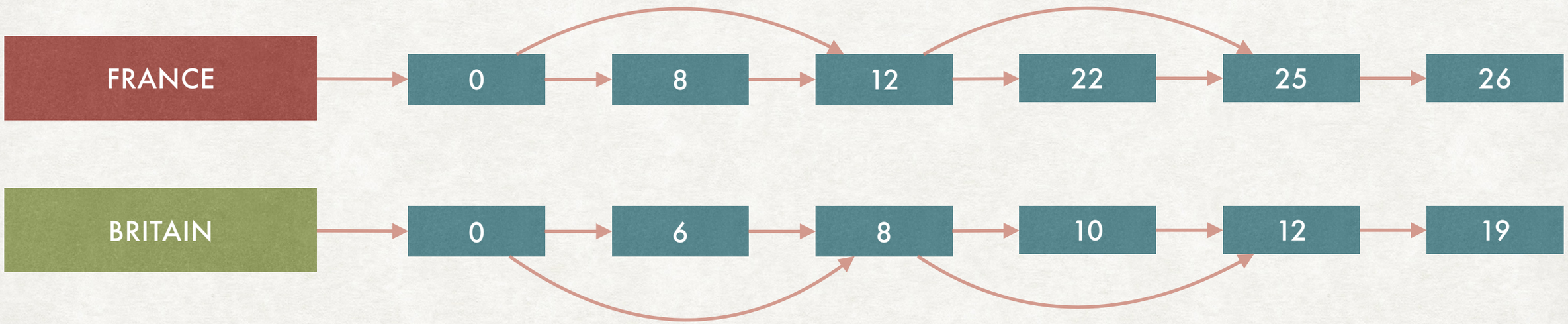
no pointers overhead, contiguous memory

difficult to update

SKIP LISTS

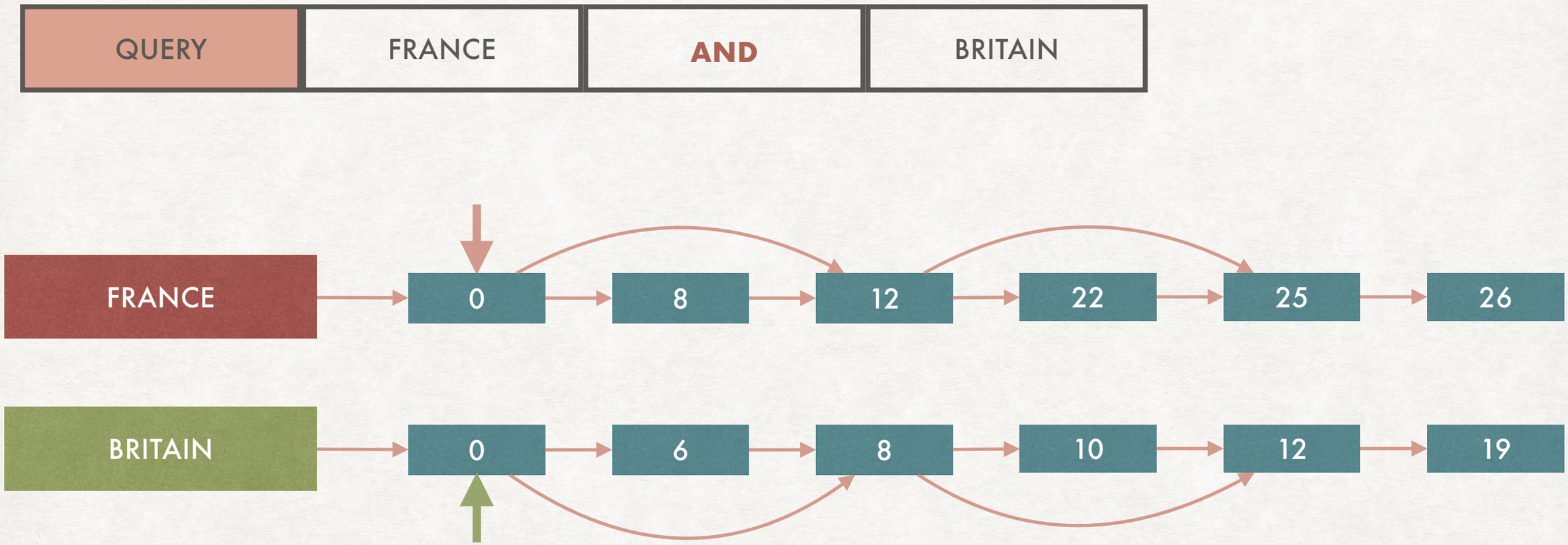
FASTER INTERSECTION

- We add additional forward pointers every k postings inside a list. The forward pointer "skips" a certain number of postings.
- A rule of thumb is, for a postings list of P postings to use $\sqrt{P}$ evenly spaced skip pointers



AN EXAMPLE QUERY

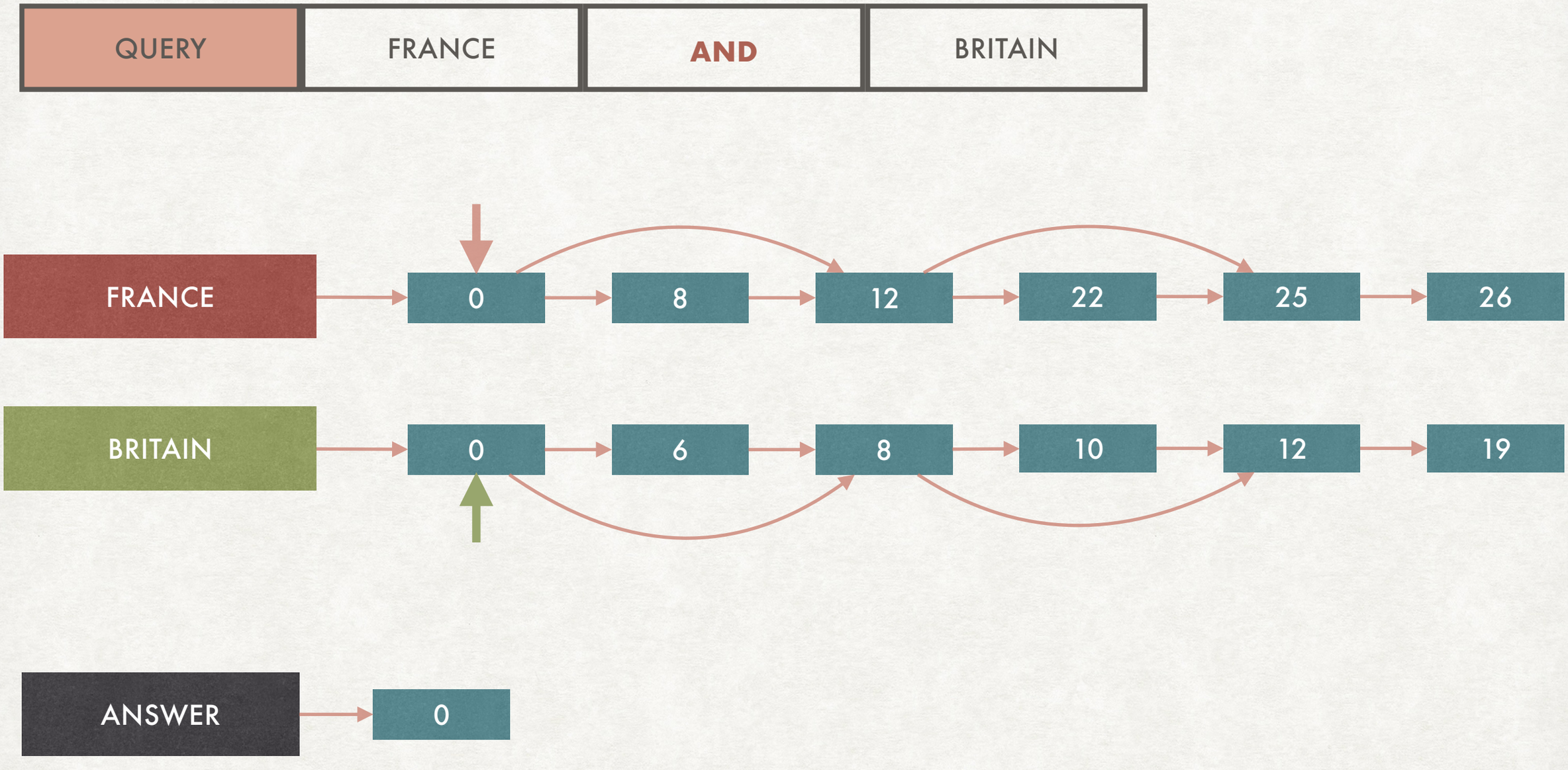
SAVING TIME WITH SKIP LISTS



ANSWER

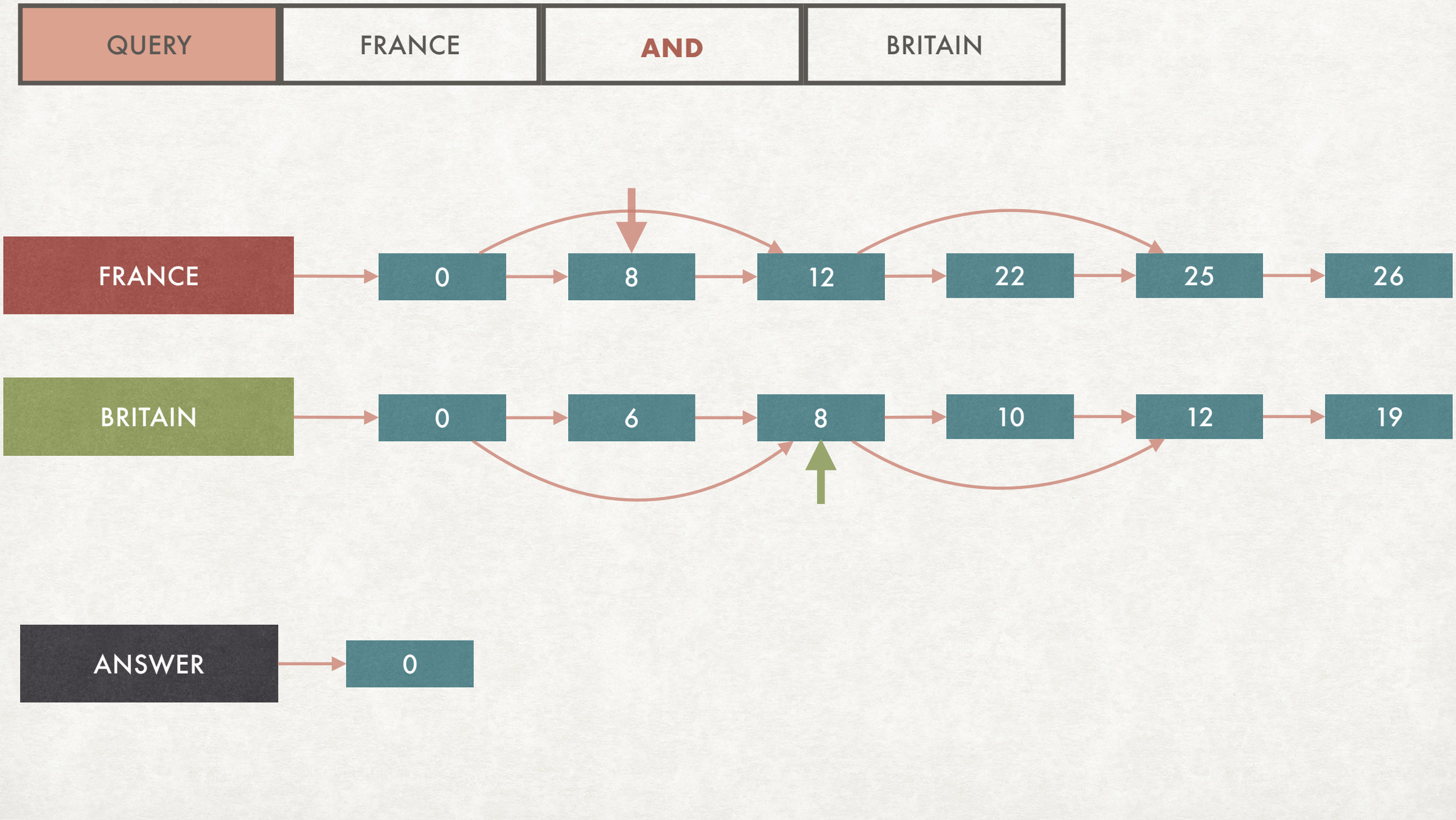
AN EXAMPLE QUERY

SAVING TIME WITH SKIP LISTS



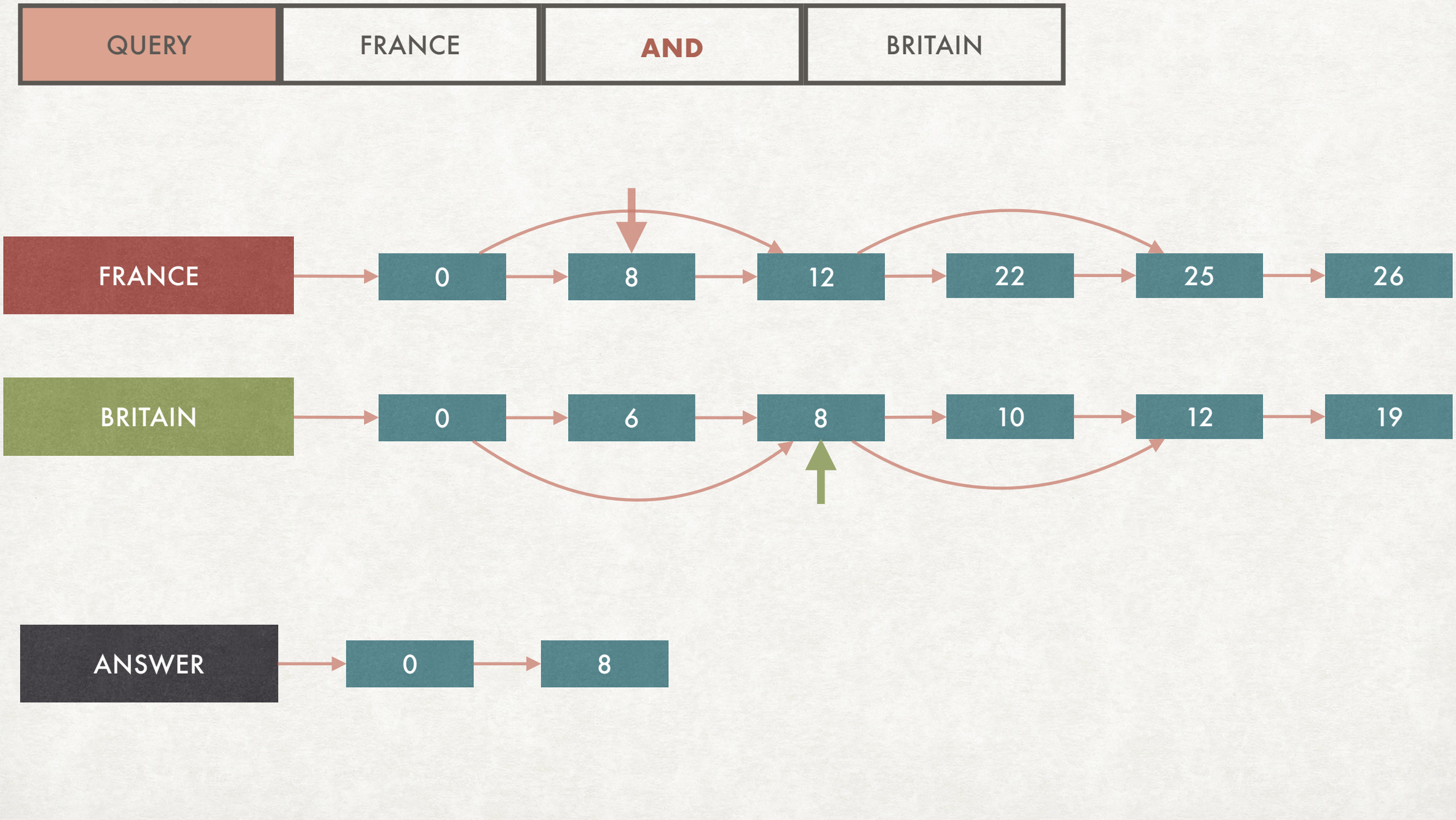
AN EXAMPLE QUERY

SAVING TIME WITH SKIP LISTS



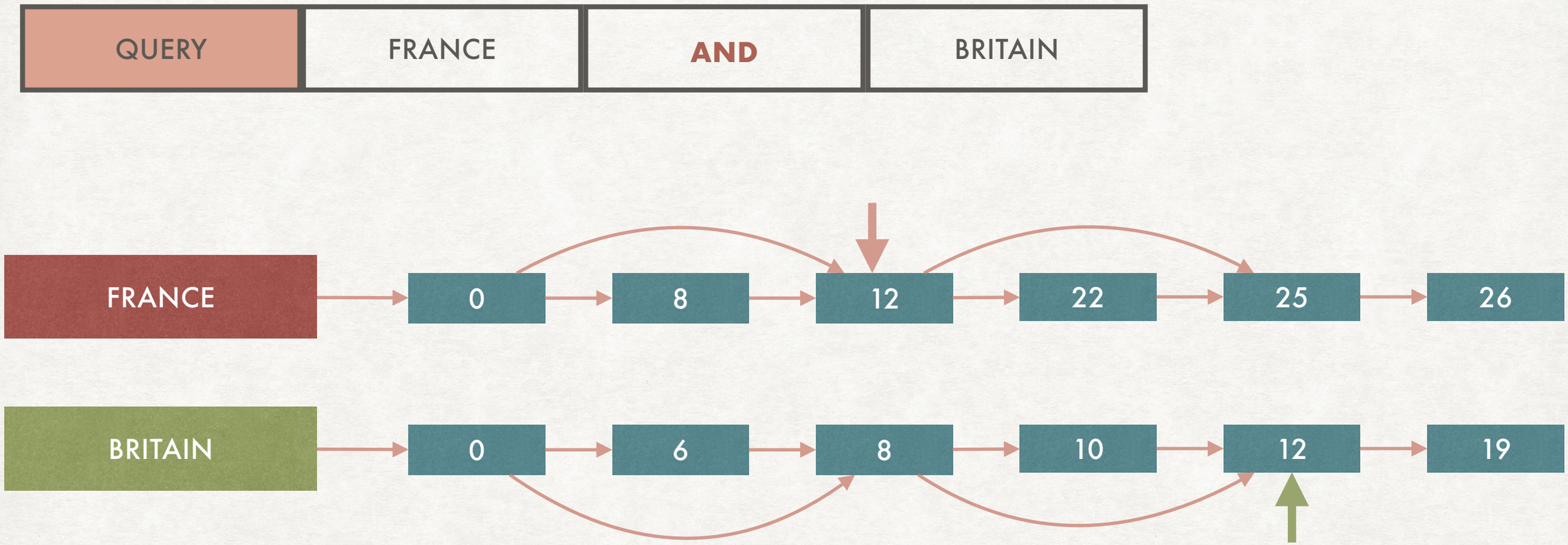
AN EXAMPLE QUERY

SAVING TIME WITH SKIP LISTS



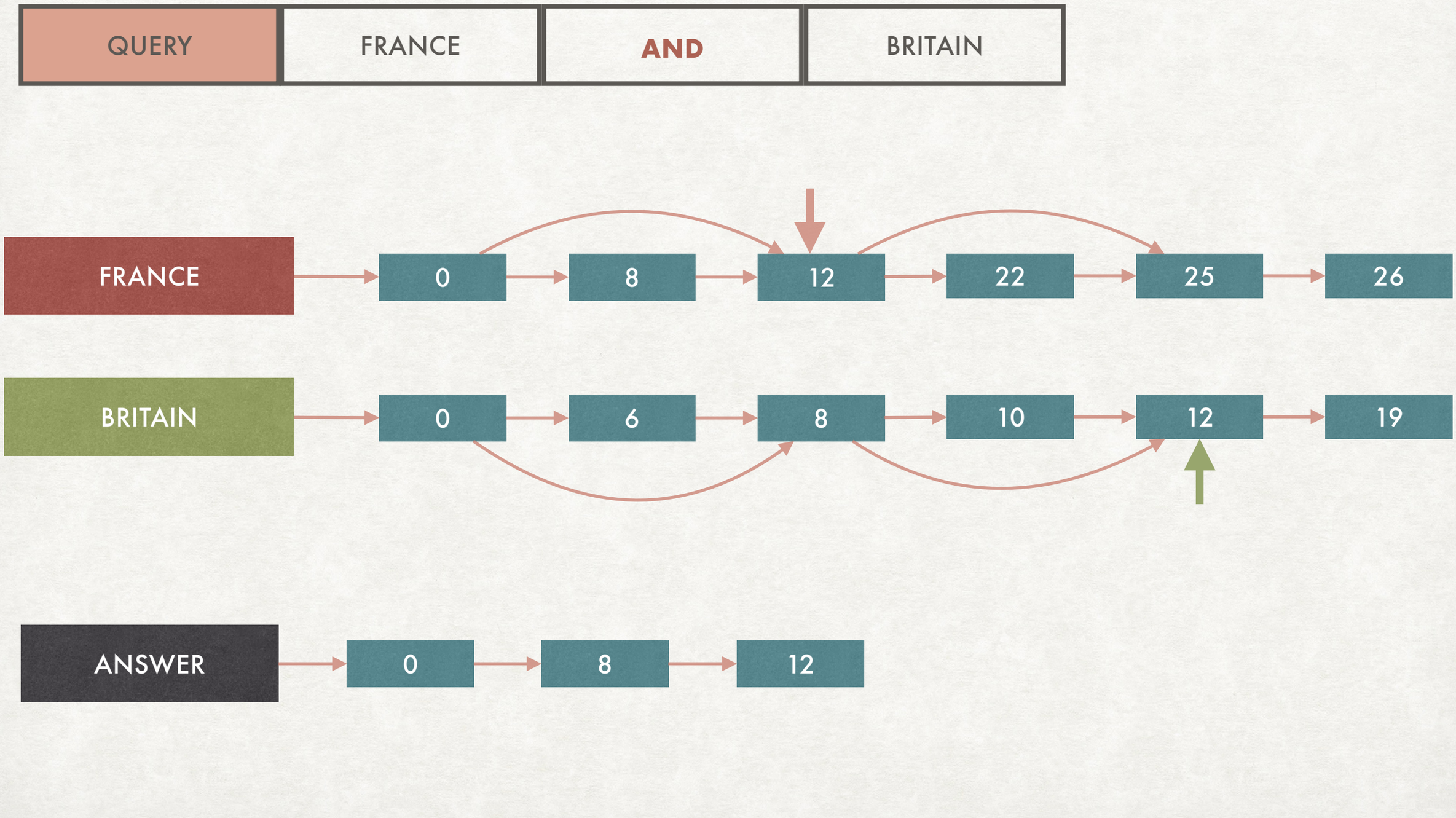
AN EXAMPLE QUERY

SAVING TIME WITH SKIP LISTS



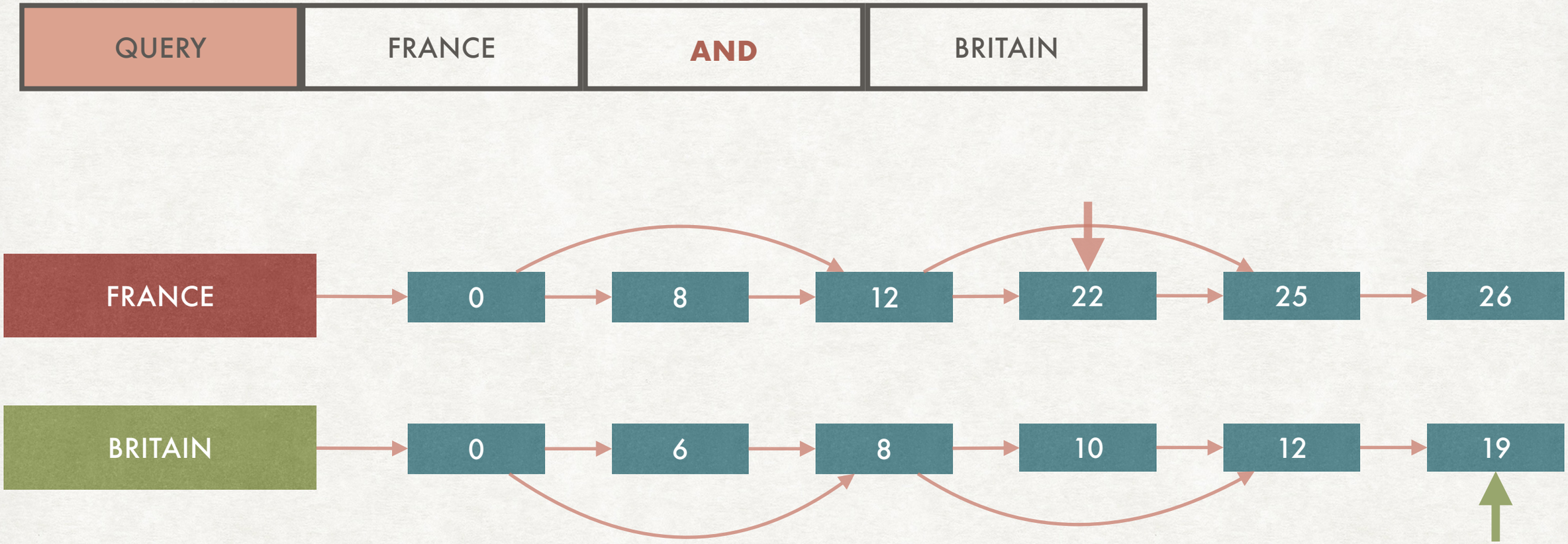
AN EXAMPLE QUERY

SAVING TIME WITH SKIP LISTS



AN EXAMPLE QUERY

SAVING TIME WITH SKIP LISTS



In some situations
we might only need
$O\left(\sqrt{P}\right)$ steps
to traverse a list

EXERCISES

THE PRACTICAL PART

- We are going to implement some of the algorithms and data structure described in this course
- We use Python 3, but you can follow along with any other programming language
- While IR systems must be efficient, we will sometimes allow for inefficiencies for the sake of more readable code
- Dataset that we use: http://www.cs.cmu.edu/~ark/personas/, more than 42k movie descriptions

DATA STRUCTURES FOR DICTIONARIES

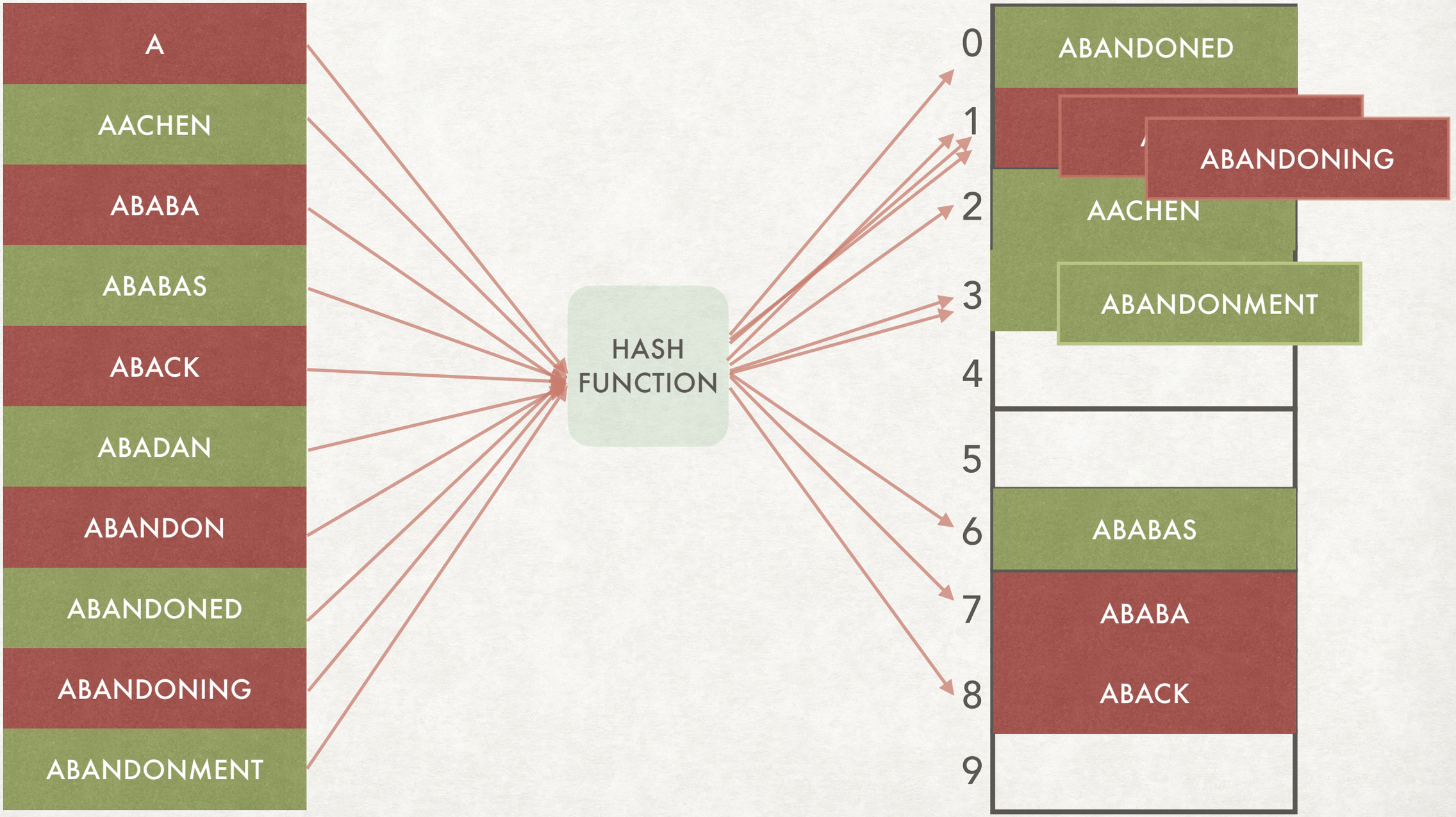
HOW IS A DICTIONARY ACTUALLY REPRESENTED?

HASH TABLES & TREES

- 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



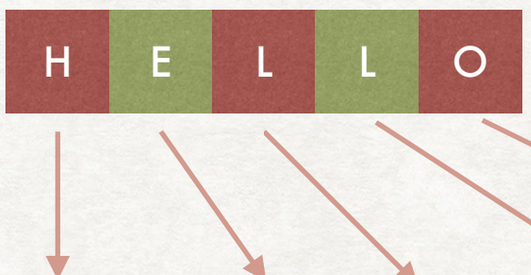
HASH FUNCTIONS

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

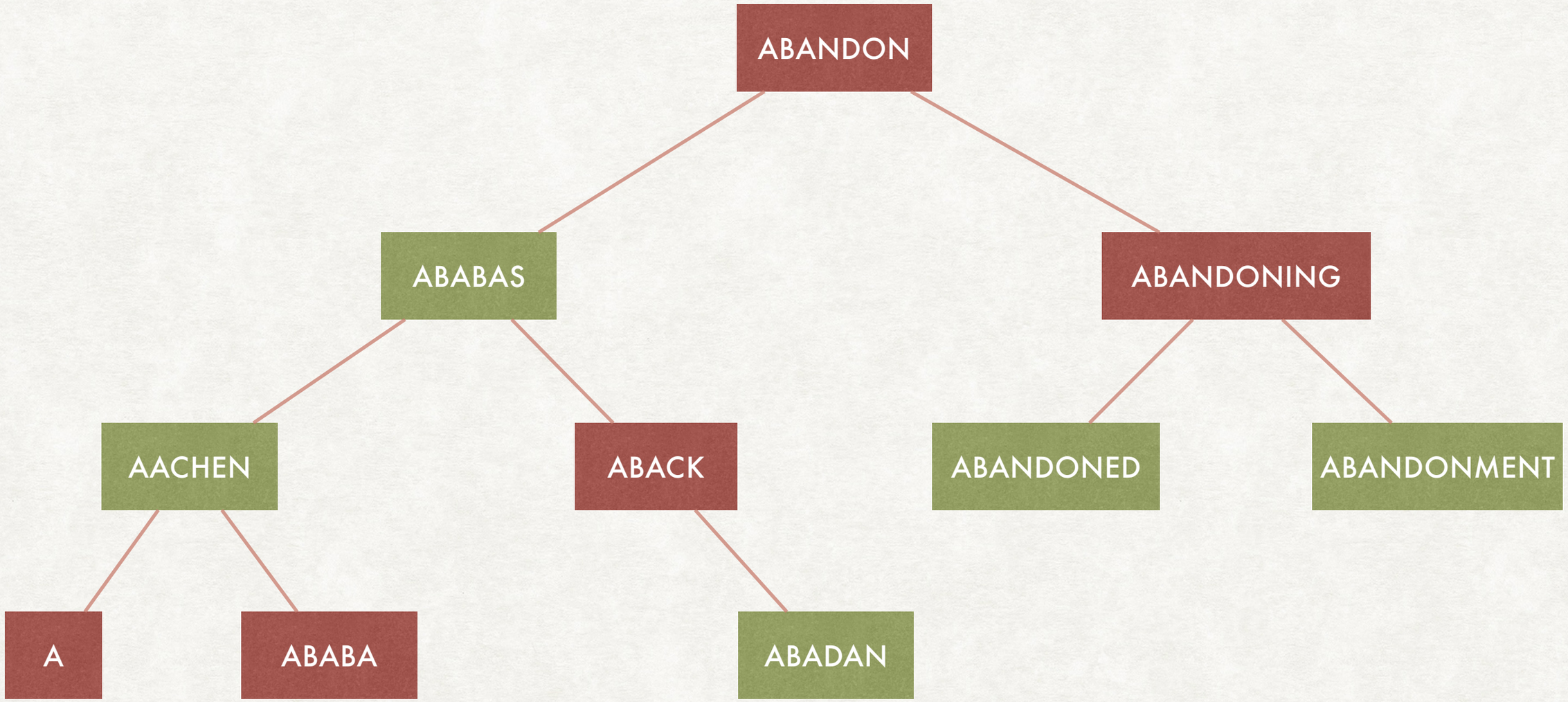
A binary tree is a tree in which each node has at most two children

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

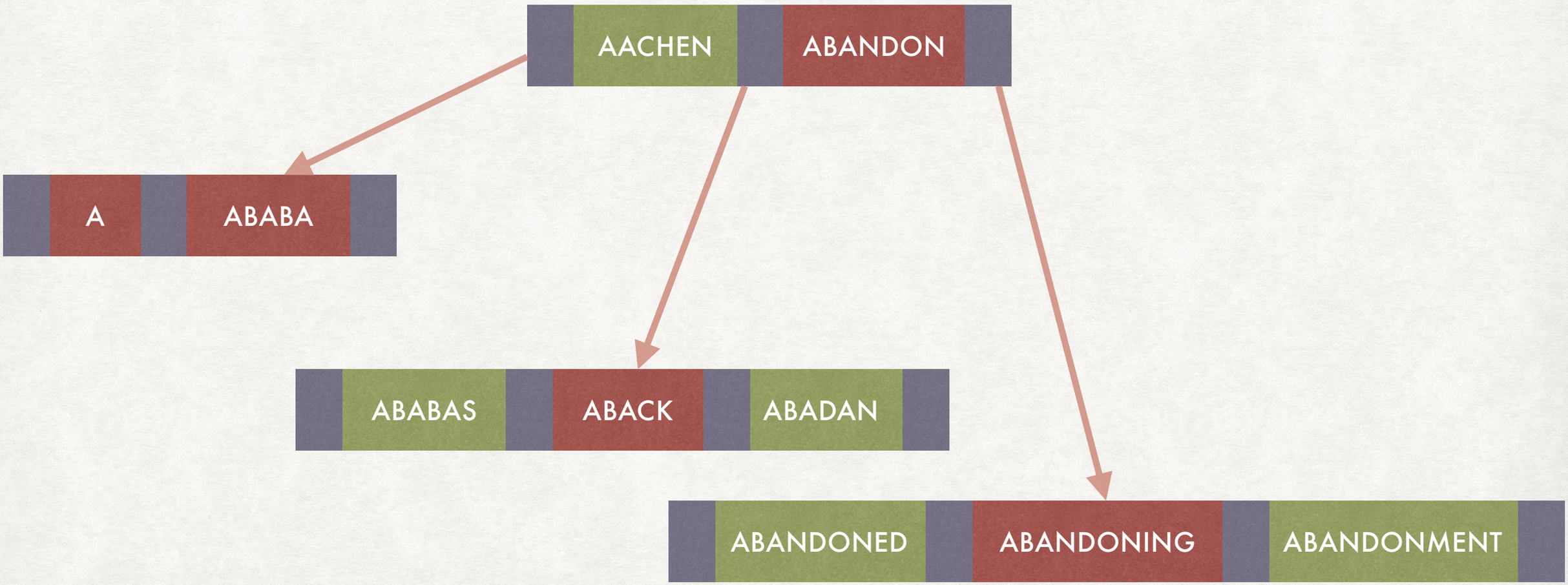
THE GOOD, THE BAD, AND THE UGLY

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.

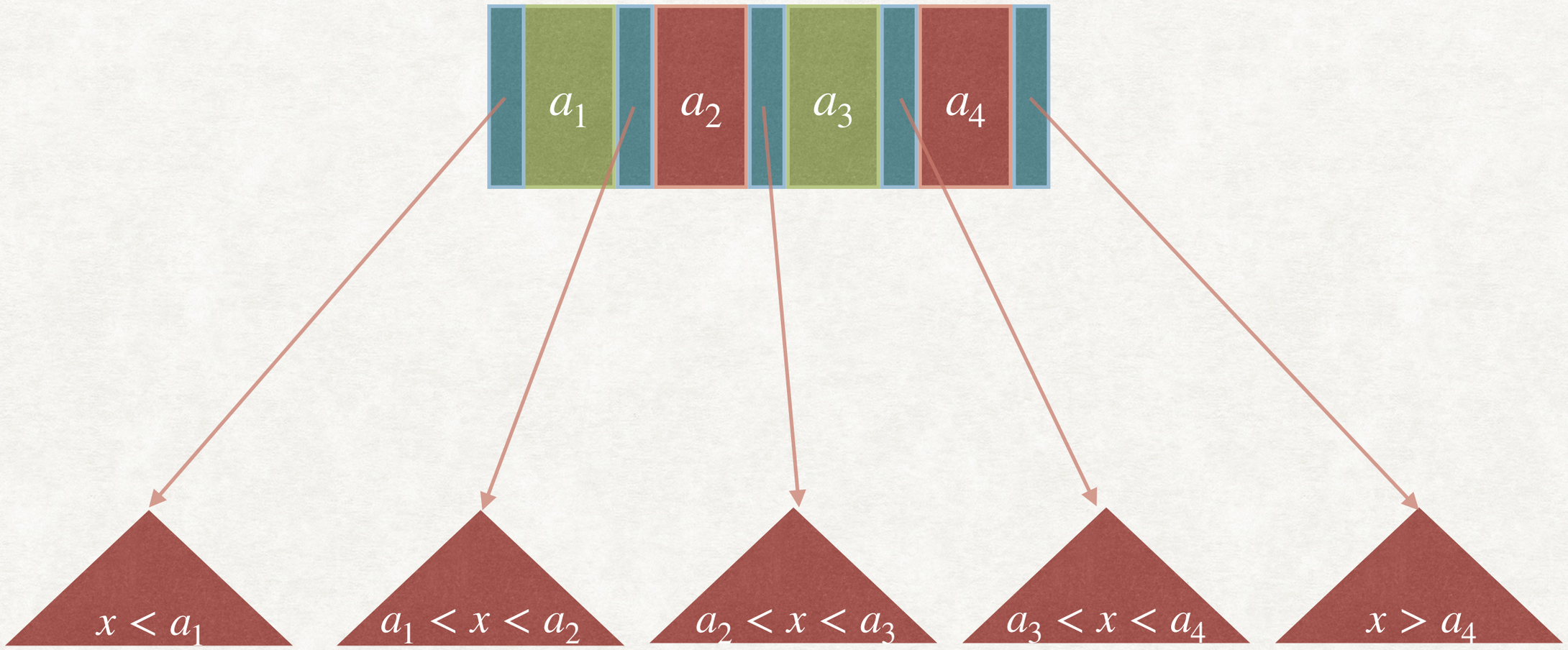
B-TREES

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

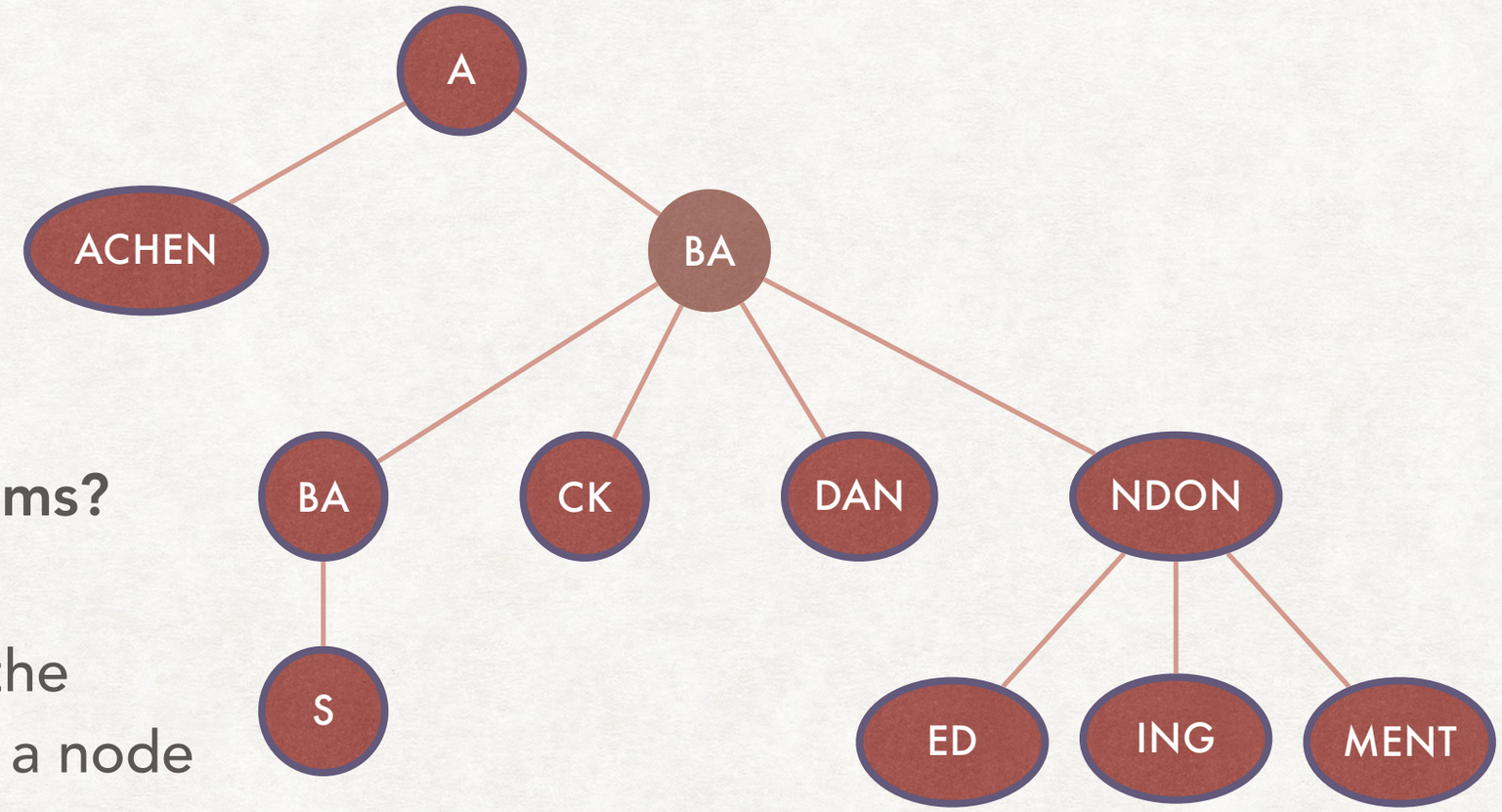
A **trie** is a special kind of tree based on the idea of searching by looking at the prefix of a key

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