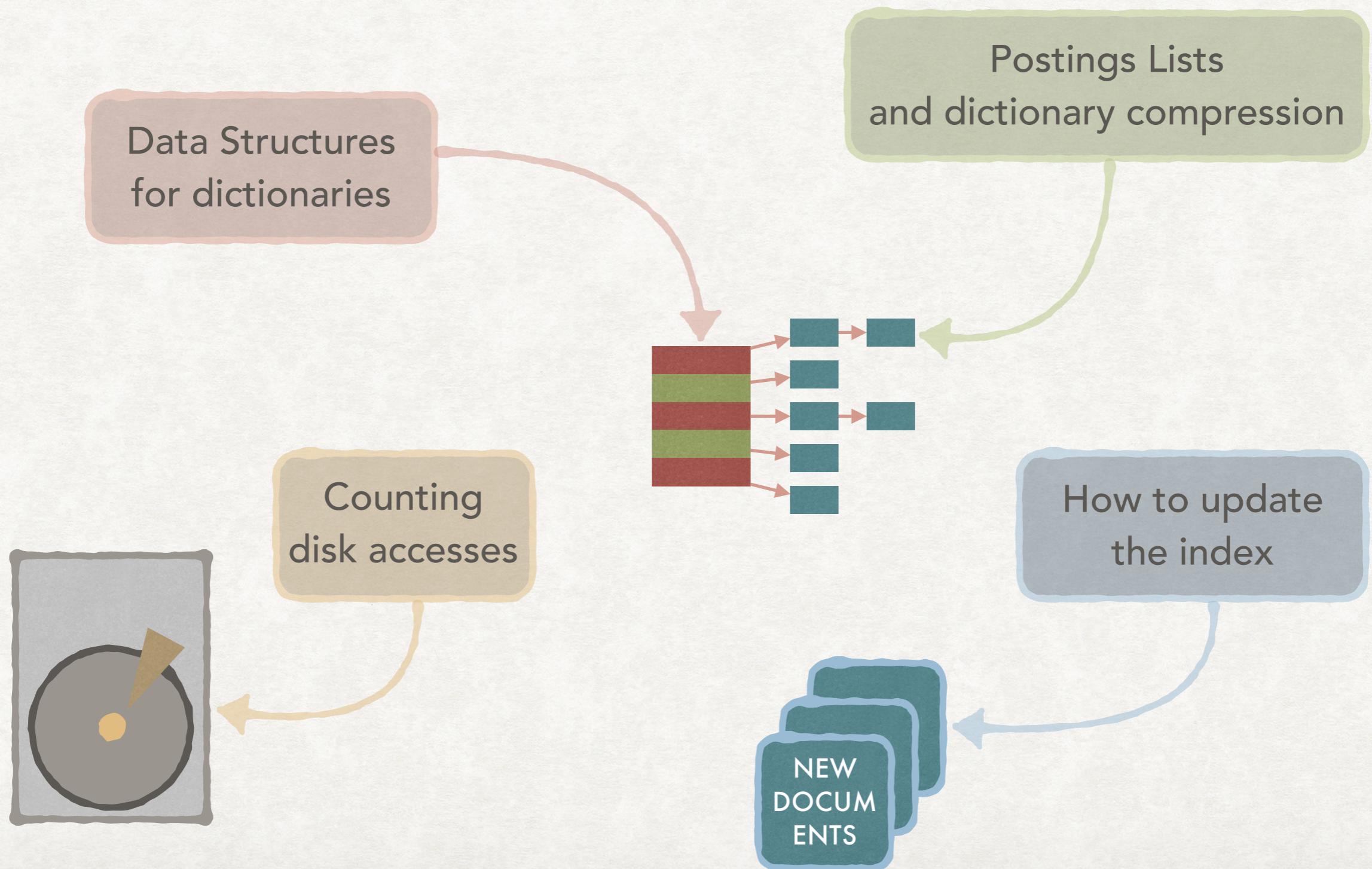


# INFORMATION RETRIEVAL

Luca Manzoni  
[lmanzoni@units.it](mailto:lmanzoni@units.it)

# LECTURE OUTLINE

\* NOW WITHOUT ASBESTOS



# EXTERNAL STORAGE

## MAIN MEMORY AND EXTERNAL STORAGE

- When analysing the complexity of algorithms each step is considered as having the same cost.
- This is a reasonable assumption in many cases, especially if all the data fit in the main memory.
- When accessing storage this assumption is stretched too thin: the costs can be orders of magnitude greater than accessing the main memory.
- Similar considerations holds when using the network.

# TIMING OF SOME STANDARD OPERATIONS

## FROM NANOSECONDS DI MILLISECONDS

|                                     |               |
|-------------------------------------|---------------|
| execute typical instruction         | 1 ns          |
| fetch from L1 cache memory          | 0.5 ns        |
| branch misprediction                | 5 ns          |
| fetch from L2 cache memory          | 7 ns          |
| Mutex lock/unlock                   | 25 ns         |
| fetch from main memory              | 100 ns        |
| send 2K bytes over 1Gbps network    | 20,000 ns     |
| read 1MB sequentially from memory   | 250,000 ns    |
| fetch from new disk location (seek) | 8,000,000 ns  |
| read 1MB sequentially from disk     | 20,000,000 ns |

# COMPLEXITY: COUNTING DISK ACCESSES

- We should take into account the number of disk accesses.
- Some effects of this choice are:
  - We might want to transfer data “in bulk”. Since each read is costly, we want to read more than strictly necessary.
  - While asymptotic results are important, we might want to be do a finer analysis.
  - Since the access to external storage is expensive, we might decide to do more work “in memory” to minimise the number of accesses (e.g., compress and decompress data).

# DATA STRUCTURES FOR DICTIONARIES

# HOW IS A DICTIONARY ACTUALLY REPRESENTED?

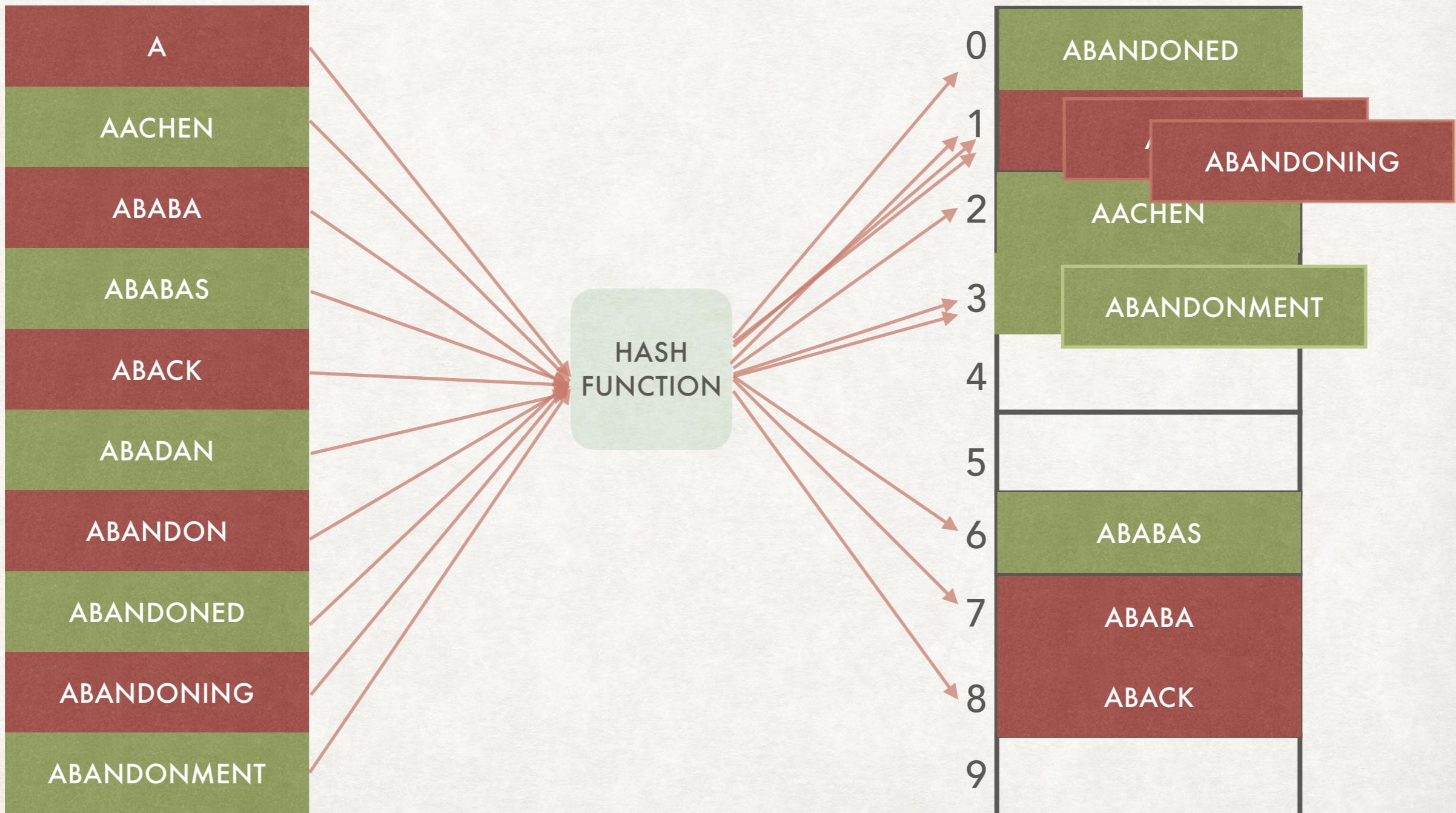
## HASH TABLES & TREES



- 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 evaluates 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 from the exact term.

# BINARY TREES

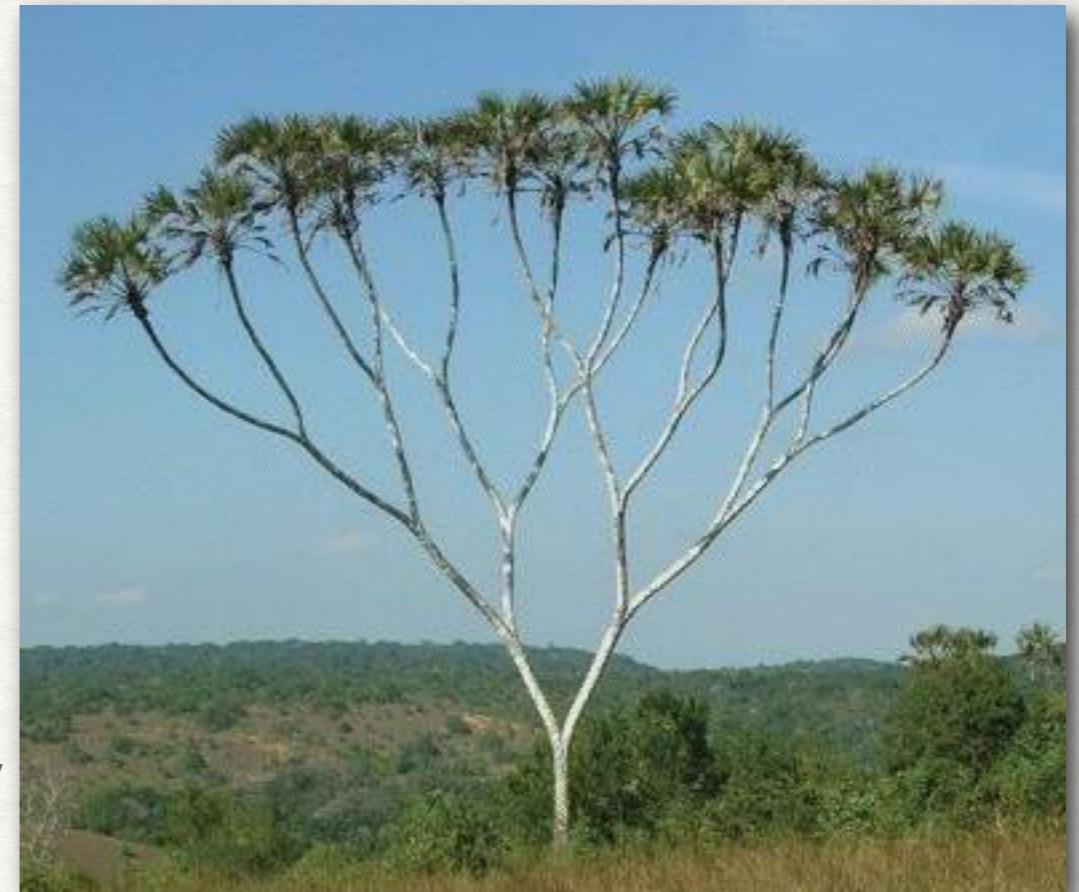
## A BRIEF RECAP

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

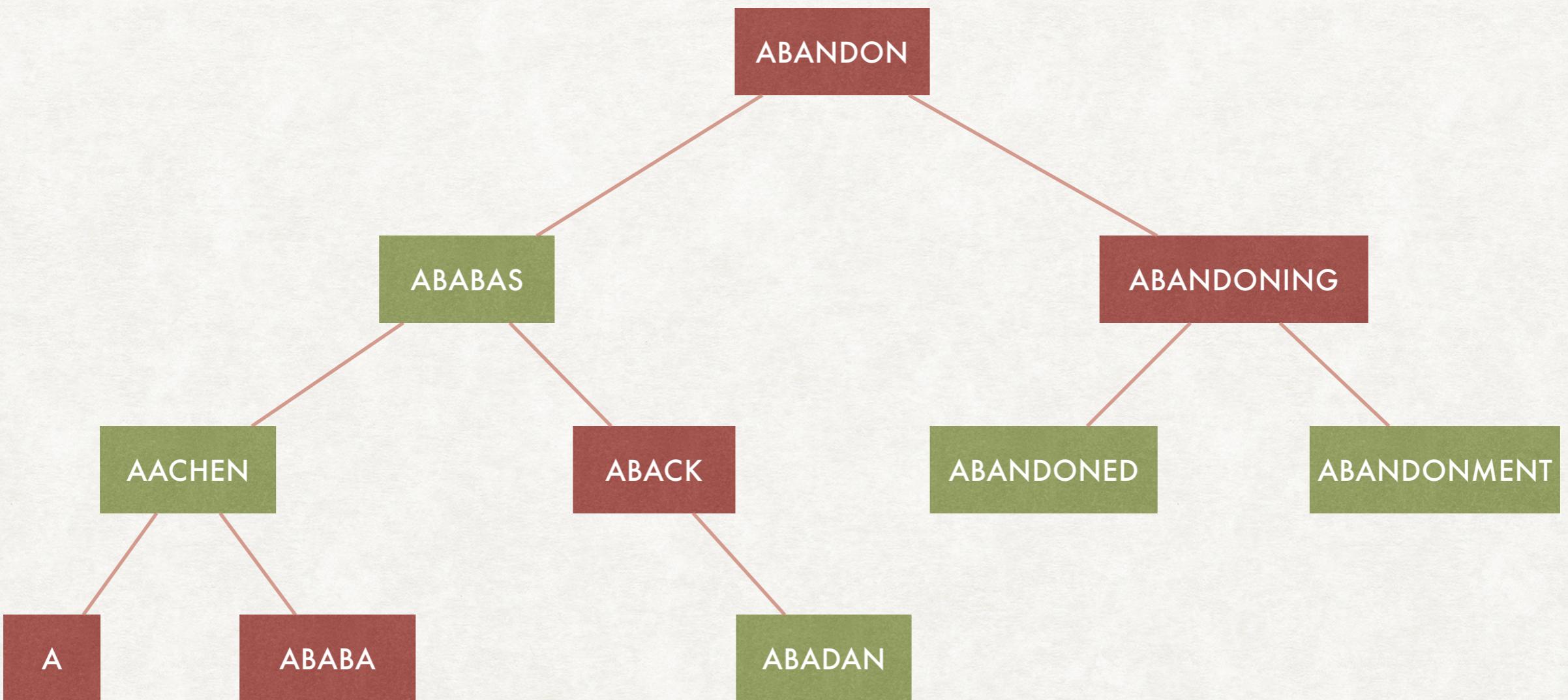
Each node has an associated value (a term in our case)

A binary *search* tree has the property that the left subtree has only values 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

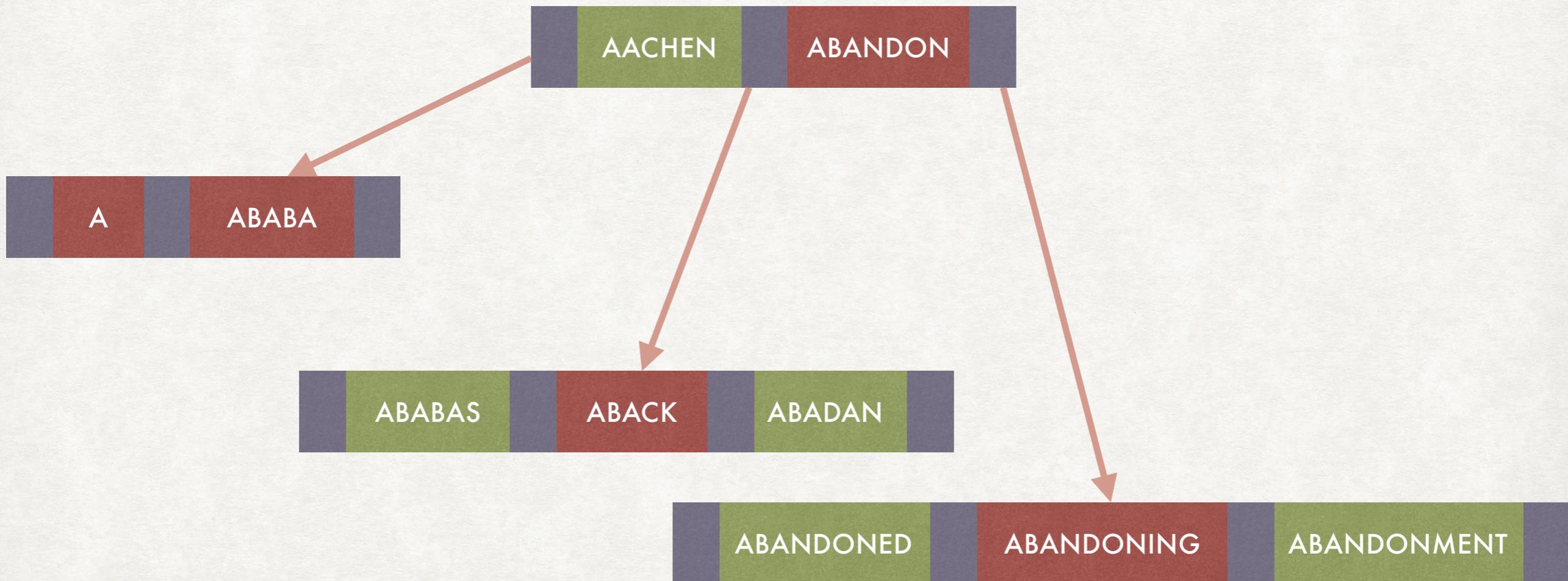
## THE GOOD, THE BAD, AND THE UGLY

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 is efficient.
- But binary trees do not 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.

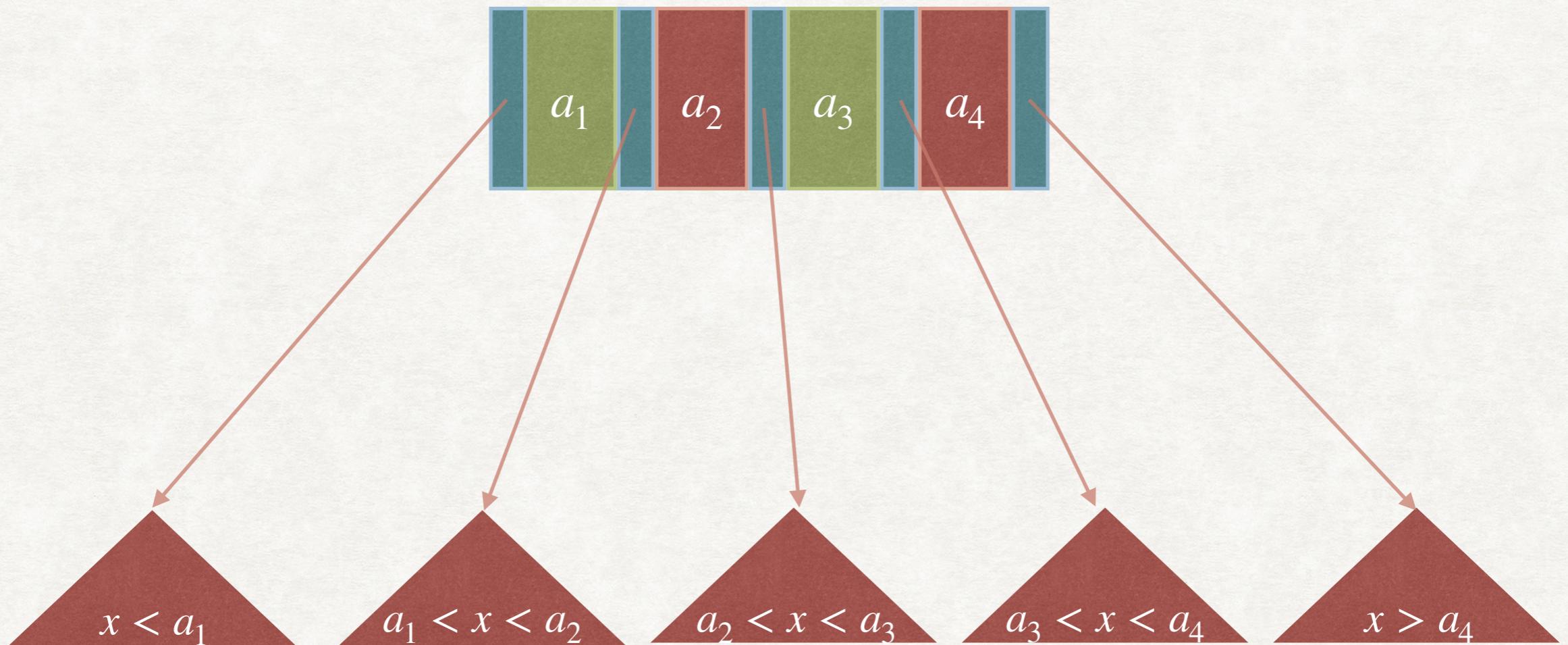
# B-TREES

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

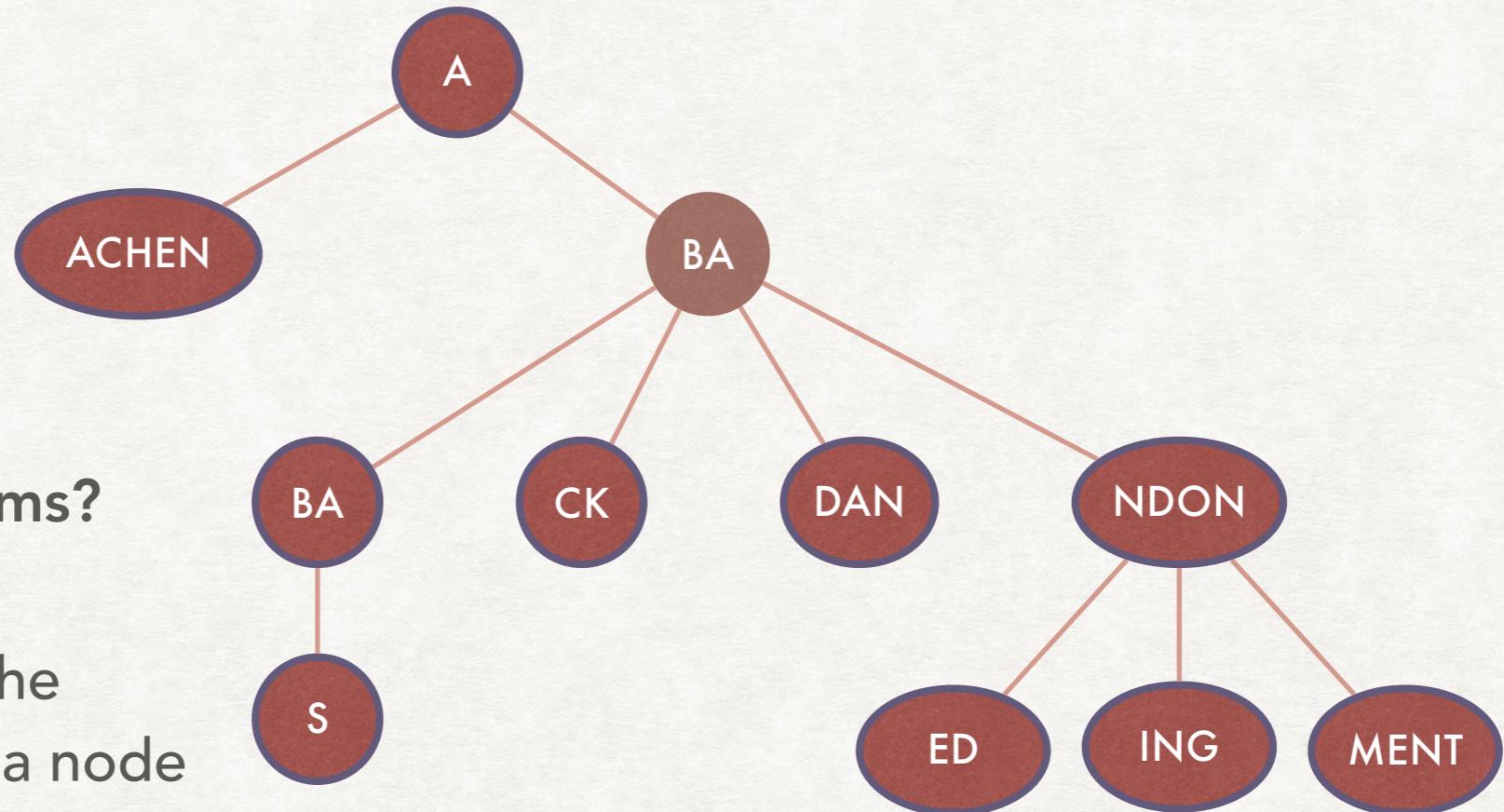
**Access time:** worst case  $O(m)$  where  $m$  is the size of the key.  
This is optimal because we must read the key.

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 are variants of tries for external storage that mitigate the problem

# **DICTIONARY AND INDEX COMPRESSION**

# WHY COMPRESSION?

## SPEED IMPLICATIONS

- We can compress two things: the dictionary and the postings
- Why using compression?
  - To save disk space.
  - To keep the entire dictionary in memory.
  - To keep more data in the main memory.
  - It might be faster to read less data from disk and decompress it in memory than to read the non-compressed data.

# ESTIMATION OF THE NUMBER OF TERMS

## HEAPS' LAW

In a collection with  $T$  tokens the estimated size of the vocabulary is:

$$M = kT^b$$

Typical values for  $k$  are between 30 and 100

Usually,  $b \approx 0.5$

HEAPS' LAW SUGGESTS THAT THE SIZE OF THE DICTIONARY  
INCREASES WITH THE SIZE OF THE COLLECTION

# DISTRIBUTION OF TERMS

## ZIPF'S LAW

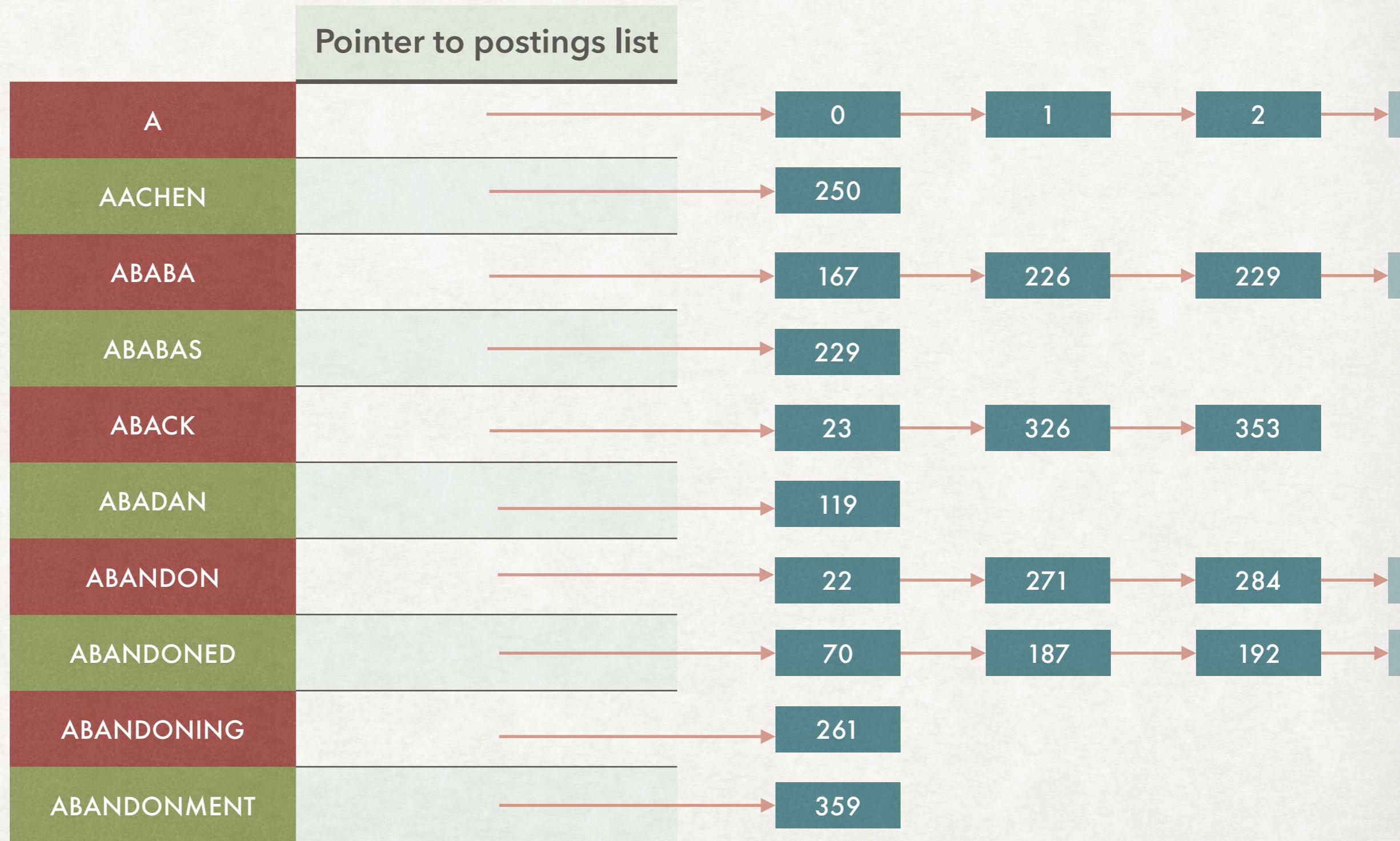
The  $i$ -th most common term in a collection appears with a frequency  $\text{cf}_i$  proportional to  $\frac{1}{i}$ :

$$\text{cf}_i \propto \frac{1}{i}$$

In other words, frequency of terms decreases rapidly with the rank.

Equivalent formulation:  $\text{cf}_i = ci^k$  for some constants  $c$  and for  $k = -1$ .

# DICTIONARY AS FIXED-WIDTH ENTRIES

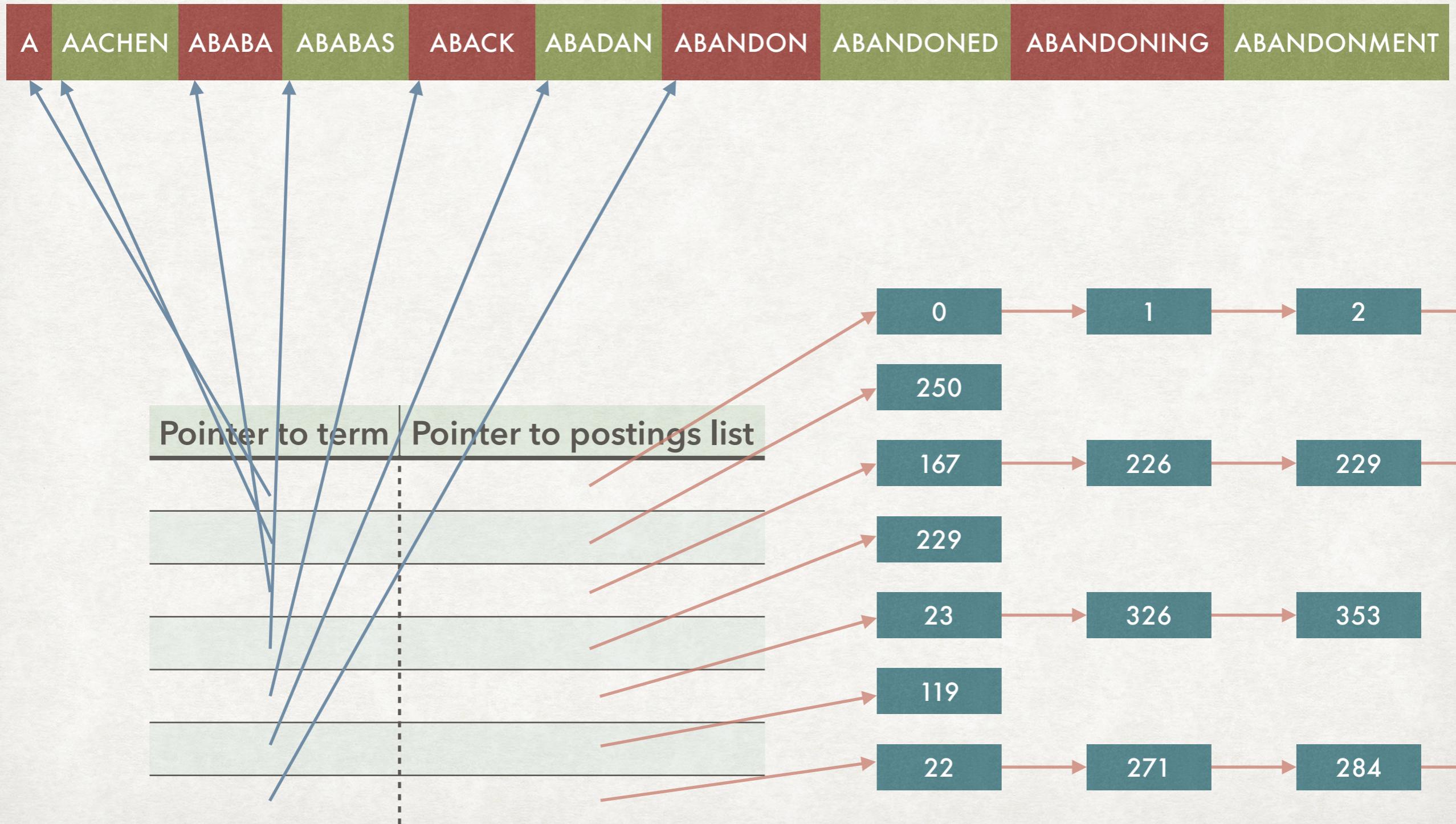


# DICTIONARY AS FIXED-WIDTH ENTRIES

## ADVANTAGES AND DISADVANTAGES

- Each entry consists of an entry of  $m$  characters and a pointer to the postings list.
- Words of length at most  $m$  can be stored.
- If we save a 40 characters string (i.e.,  $m \geq 40$ ) then every entry will require 40 characters, wasting a lot of space for short words.

# DICTIONARY AS A SINGLE STRING



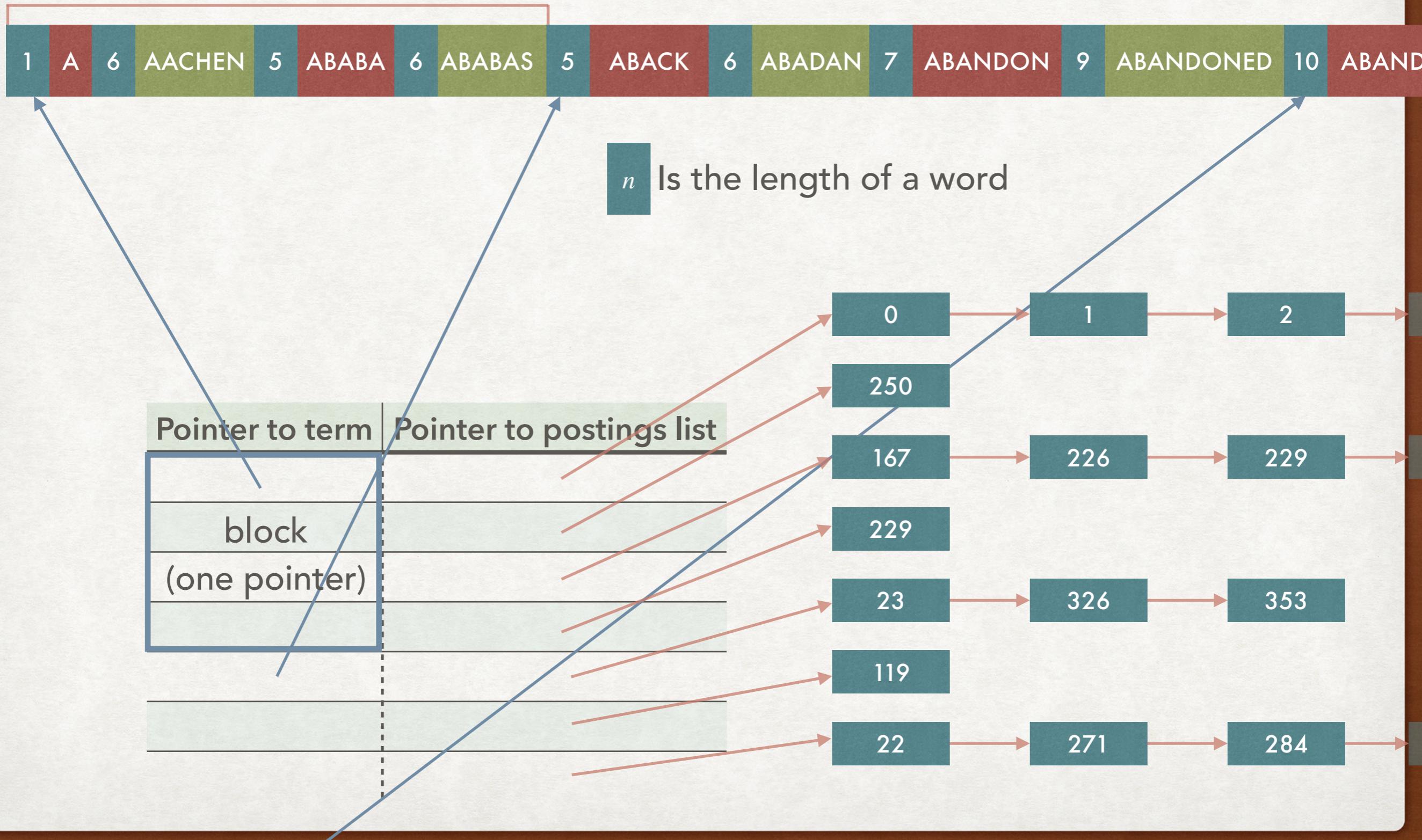
# DICTIONARY AS A SINGLE STRING

## DESCRIPTION AND ADVANTAGES

- Each entry consists of a pointer to the term that is part of a contiguous string and to the pointer to the postings list
- To know the end of the word it is necessary to look at the next pointer.
- There is no wasted space for the strings...
- ...but it is necessary to keep an additional pointer for each entry.
- We can reduce the space used for the pointers by using *blocked storage*.

# BLOCKED STORAGE

Block of 4 terms



# BLOCKED STORAGE

## ADVANTAGES AND DISADVANTAGES

- Now only one every  $k$  entries (the block size) has a pointer.
- The value of the others entries is determined by a linear scan of the block.
- The addition of the length of the string is needed to know where a string end.
- It is a trade-off between space and access time (which is increased due to the linear scanning inside a block).
- We are still not using the fact that the words in the dictionary are ordered alphabetically.

# FRONT CODING



In red the prefix shared by each word  
with the previous one in the block:

A  
**A**ACHEN  
**A**BABA  
**A**BABAS

ABACK  
**A**BADAN  
**A**NDON  
**A**BANDONED

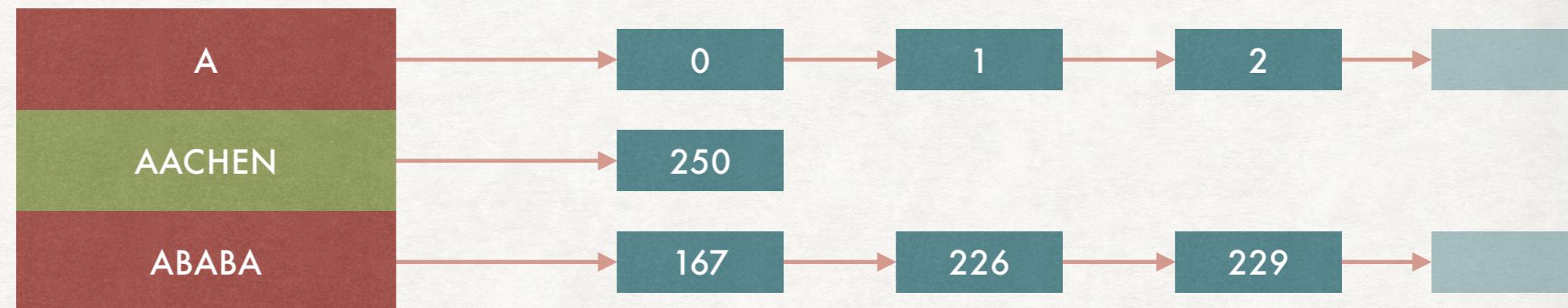
This works because the  
dictionary is ordered alphabetically,  
thus many words share a prefix

# FRONT CODING

## ADVANTAGES AND DISADVANTAGES

- Used inside a block reduces the size of the dictionary.
- Uses the fact that the dictionary is ordered.
- Requires, in addition to the linear scanning, a decoding phase.
- As before a trade-off between reducing size and incrementing the cost of retrieving a term.

# POSTING FILE: ENCODING DIFFERENCES AND THE USE OF VARIABLE BYTE CODES

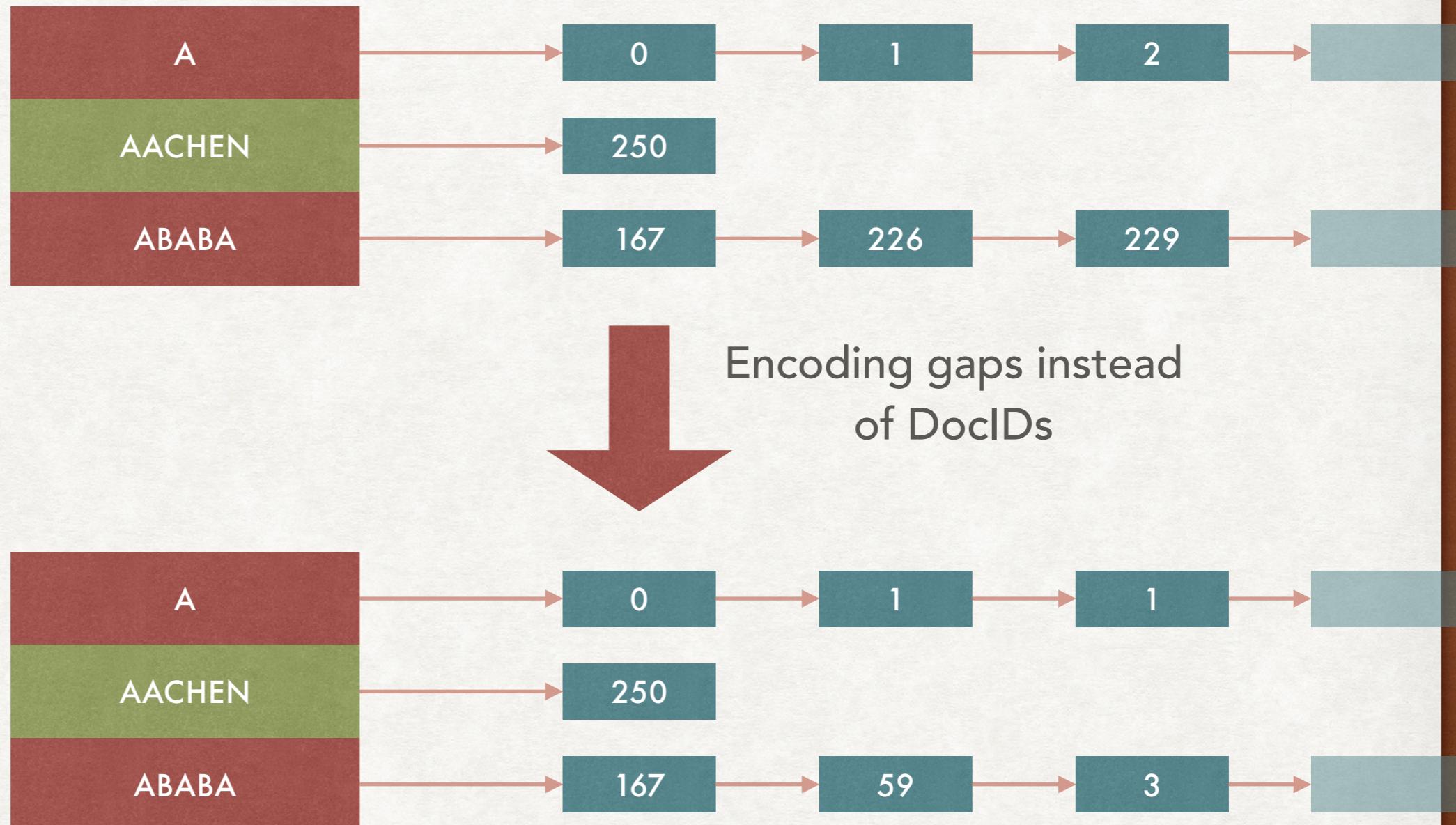


Can we use the fact that a postings list is ordered?

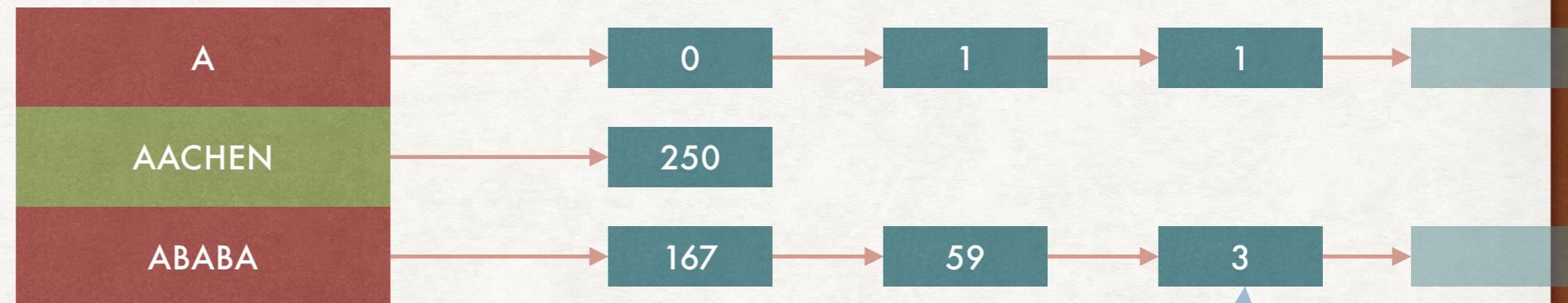
YES

If we have a sequence  $(a_0, a_1, a_2, a_3, \dots)$  with  $a_i < a_{i+1}$  for all  $i$ , we can also encode it as a sequence of differences of consecutive terms:  $(a_0, a_1 - a_0, a_2 - a_1, a_3 - a_2, \dots)$

# POSTING FILE: ENCODING DIFFERENCES AND THE USE OF VARIABLE BYTE CODES



# POSTING FILE: ENCODING DIFFERENCES AND THE USE OF VARIABLE BYTE CODES



How can we recover this DocID?

$$167 + 59 + 3 = 229$$

In general, to recover the  $k$ -th DocID in list we sum all the values up to the  $k$ -th one:

$$a_0 + \sum_{i=0}^k (a_i - a_{i-1}) = a_0 + a_1 - a_0 + a_2 - a_1 + \dots = a_k$$

# POSTING FILE: ENCODING DIFFERENCES

## MOTIVATIONS FOR ENCODING THE GAPS

- The DocID can be arbitrarily large...
- ...but most of the gaps between two DocID will be small
- We can use *variable byte codes* to use less storage
- Still, recovering a DocID now is more complex.
- Most importantly, access to the list must be sequential: to recover a DocID we need to read all the previous ones.
- But the algorithms for union and intersection access the list sequentially.

# VARIABLE BYTE CODES

## ENCODE NUMBERS IN A VARIABLE NUMBER OF BYTES



One byte usually encodes  $2^8 = 256$  different values

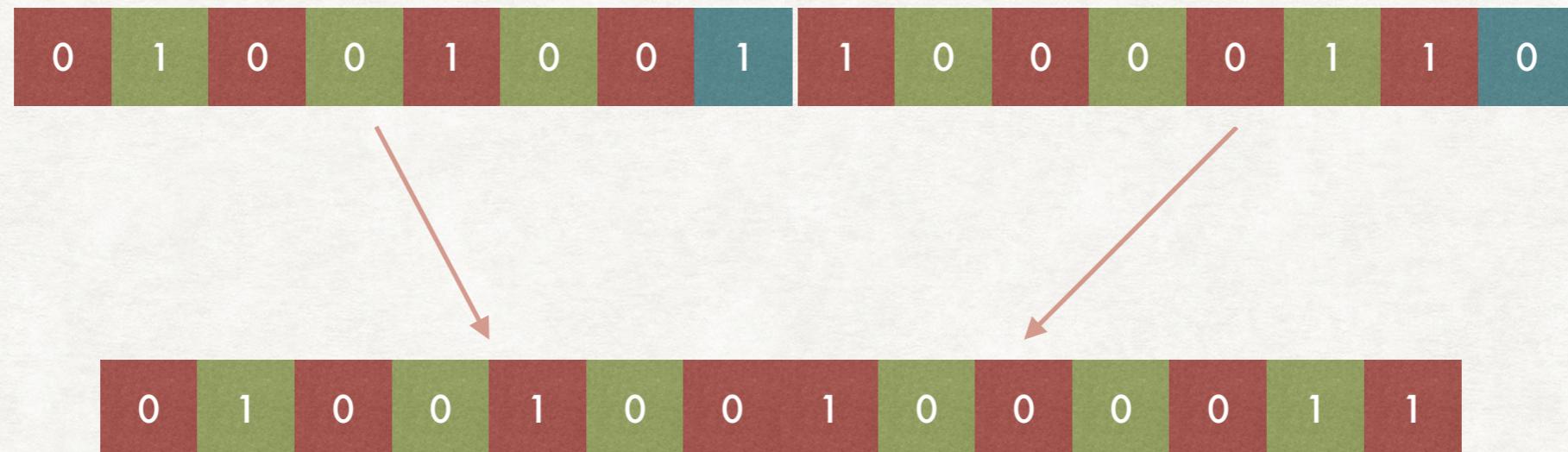


We encode  $2^7 = 128$  different values in the first seven bits  
the last bit is a *continuation bit*.

If it is 0 then we have completed reading the number  
otherwise we must continue to read the next byte

# VARIABLE BYTE CODES

## ENCODE NUMBERS IN A VARIABLE NUMBER OF BYTES



$$2^12 + 2^9 + 2^6 + 2^1 + 2^0 = 4675$$

| # of bytes | Max value |
|------------|-----------|
| 1          | 127       |
| 2          | 16383     |
| 3          | 2097152   |
| 4          | 268435456 |

The most frequent terms will have small gaps in their postings lists.

Hence, we can store the size of most gaps in only a few bytes

# UPDATING THE INDEX

# STORING POSTINGS

## HOW THE POSTINGS LISTS ARE ORGANISED ON DISK

- How are the postings stored on-disk?
- One file per postings list can lead to too many files for a filesystem to manage efficiently
- One single large file containing all the postings can be better (here we select this solution)
- In reality we can have a combination of both, with multiple large files each storing part of the postings

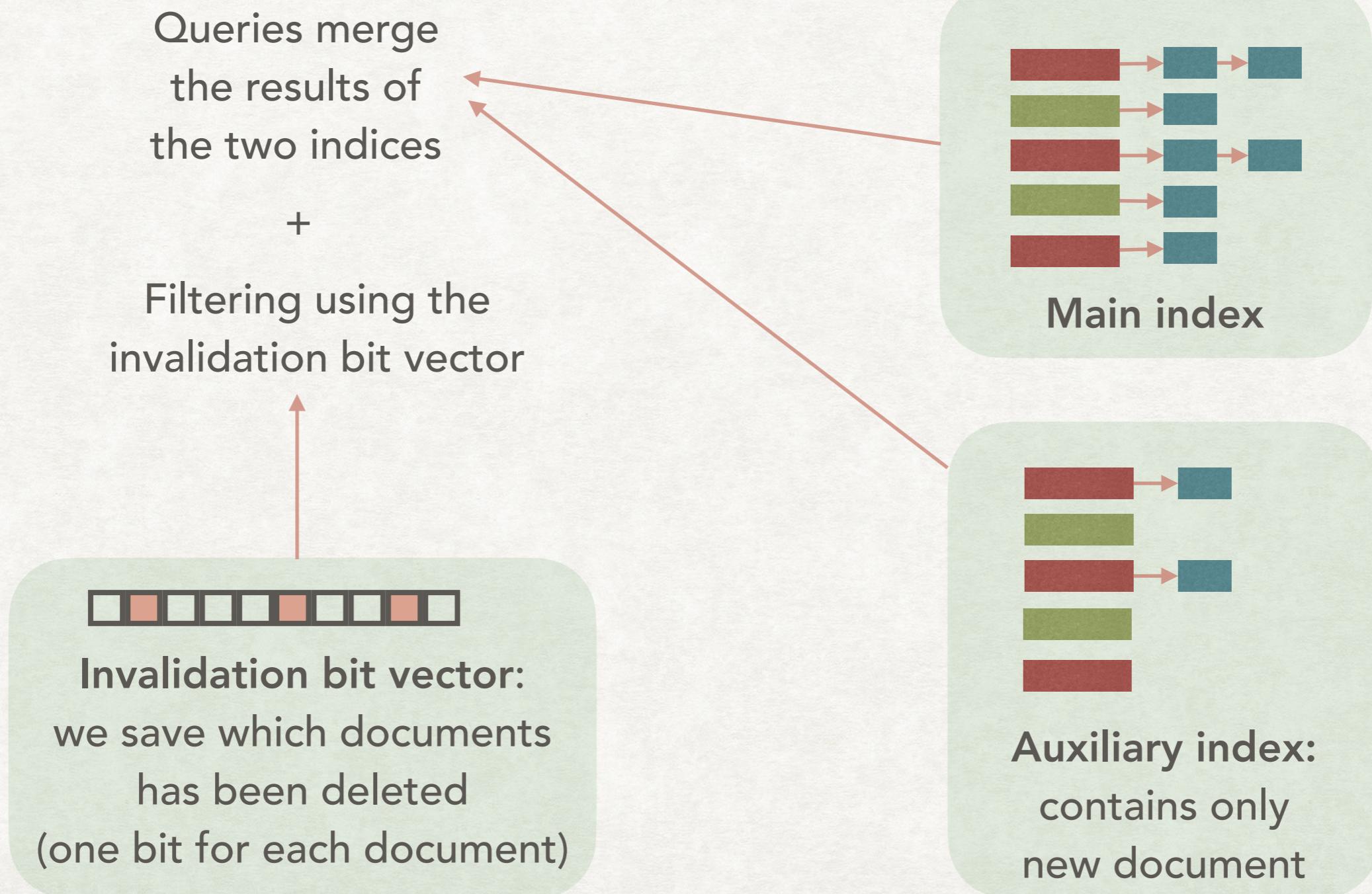
# DYNAMIC INDEXING

## FOR COLLECTIONS THAT CHANGE WITH TIME

- How can we insert new documents (or delete old ones) in an inverted index?
- We can rebuild the index:
  - Not very efficient.
  - Only useful when the number of changes is small.
  - To keep the system online while reindexing we need to keep the old index until new one is ready.

# AUXILIARY INDEX

## A MORE EFFICIENT SOLUTION



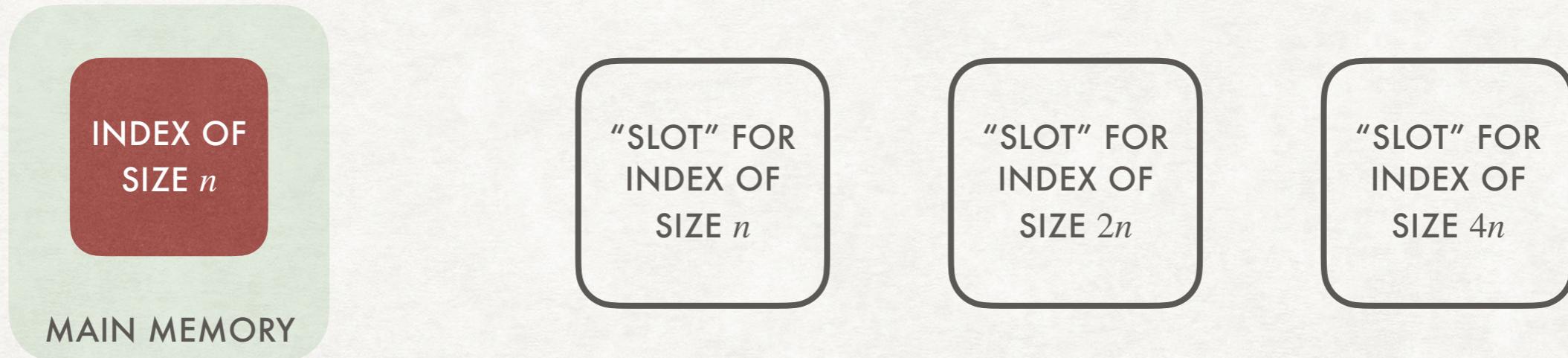
# AUXILIARY INDEX

## CAN WE DO BETTER?

- When the auxiliary index becomes too big we need to merge it with the main index.
- We can improve the efficiency by keeping  $\log_2(T/n)$  auxiliary indices (each one of size double the previous one) where  $T$  is the total number for postings and  $n$  the size of the smaller auxiliary index.
- This increase the complexity of all algorithms used to answer queries, so it is a trade-off.

# LOGARITHMIC MERGING

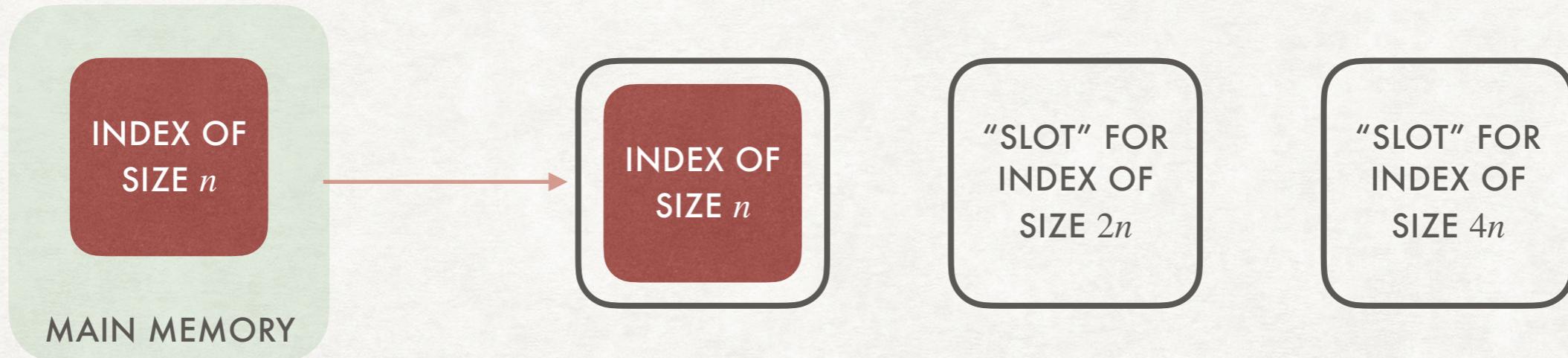
## MAIN IDEAS



The smaller index is kept in memory

# LOGARITHMIC MERGING

## MAIN IDEAS

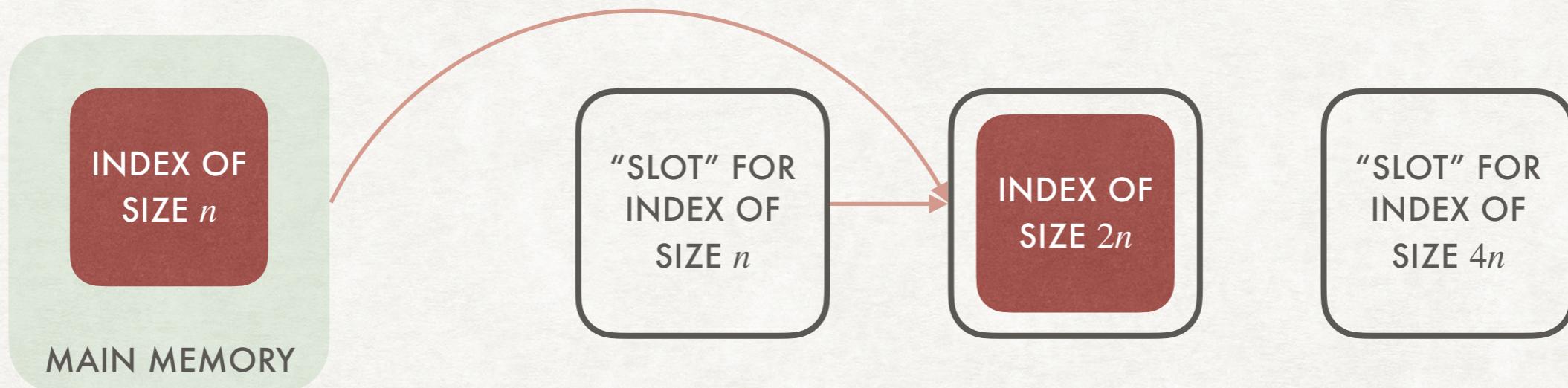


When full it is copied to disk.

A new (empty) index of size  $n$  is created in memory

# LOGARITHMIC MERGING

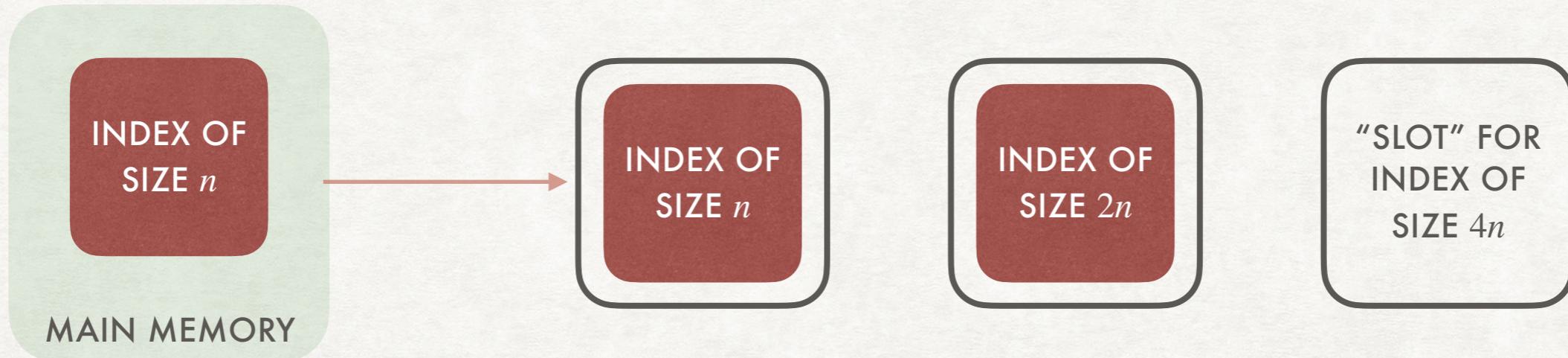
## MAIN IDEAS



When it is full, since there is already an index of size  $n$  saved on disk, it is merged with it to form an index of size  $2n$ .  
A new (empty) index of size  $n$  is created in memory

# LOGARITHMIC MERGING

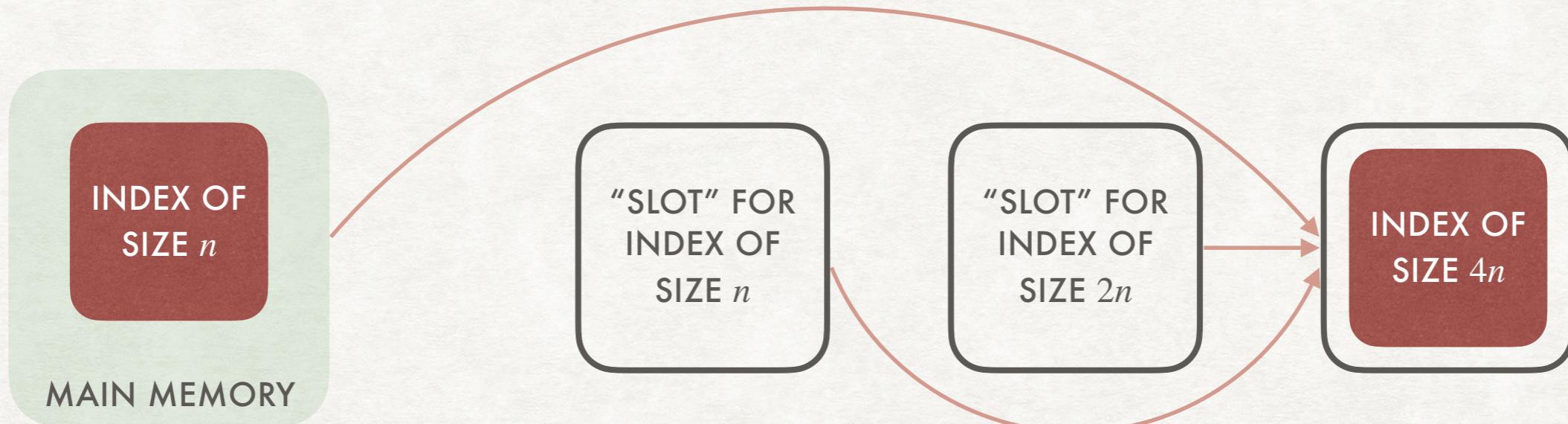
## MAIN IDEAS



When full it is copied to disk. No merge is necessary (the "slot" is free)  
A new (empty) index of size  $n$  is created in memory

# LOGARITHMIC MERGING

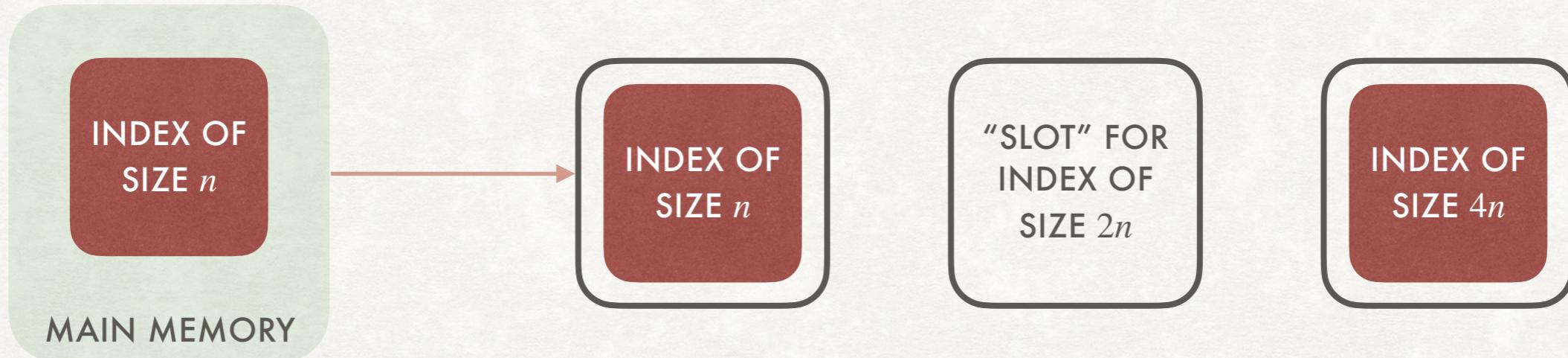
## MAIN IDEAS



When it is full, since there is already an index of size  $n$  saved on disk, it is merged with it to form an index of size  $2n$ . Since there is already an index of size  $2n$  saved on disk, it is merged with it to form an index of size  $4n$ . A new (empty) index of size  $n$  is created in memory.

# LOGARITHMIC MERGING

## MAIN IDEAS



When full it is copied to disk. No merge is necessary (the "slot" is free)  
A new (empty) index of size  $n$  is created in memory

Most merges are of small indices, even if sometimes a series of  
more merge operations are necessary.