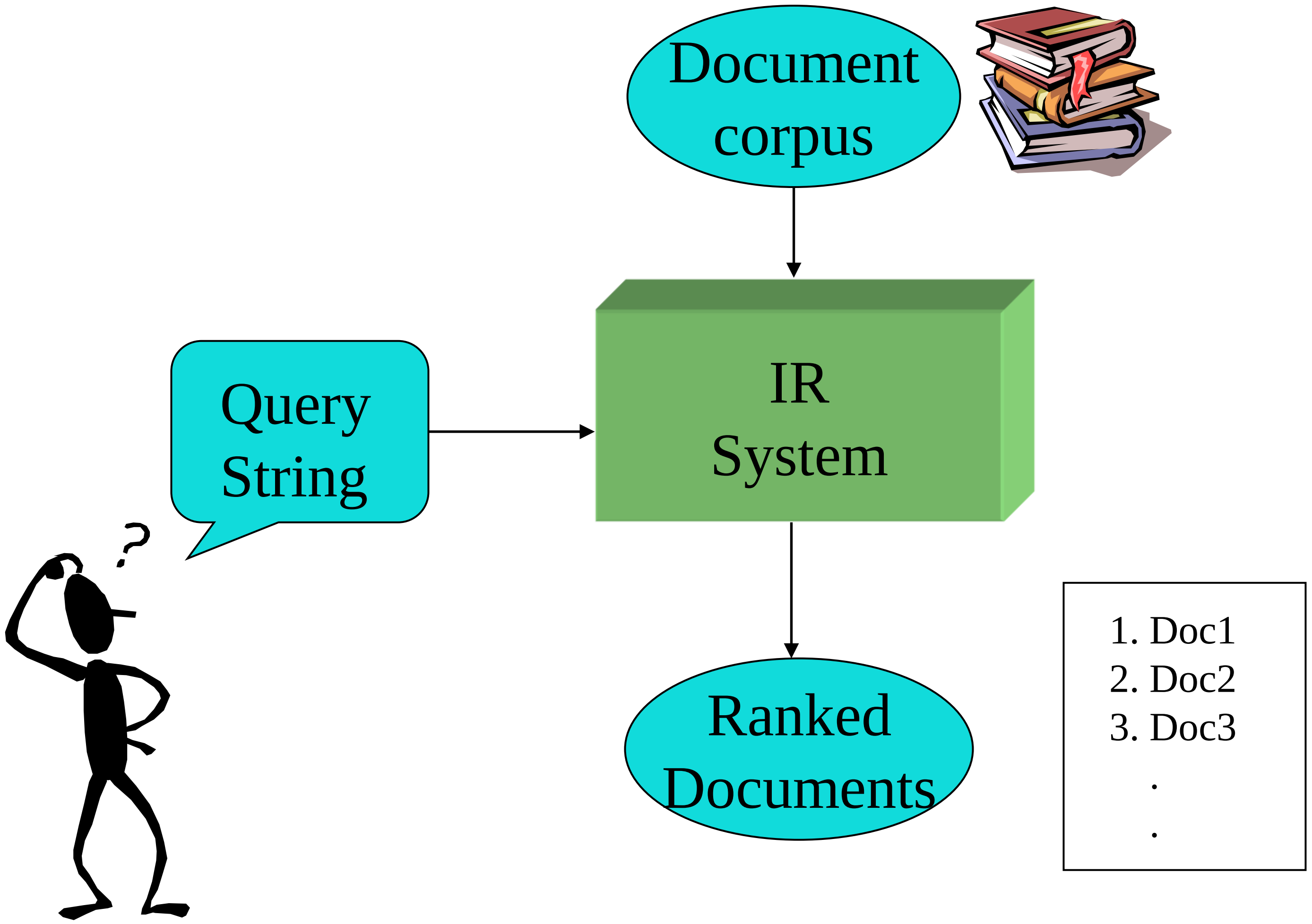
Information Retrieval

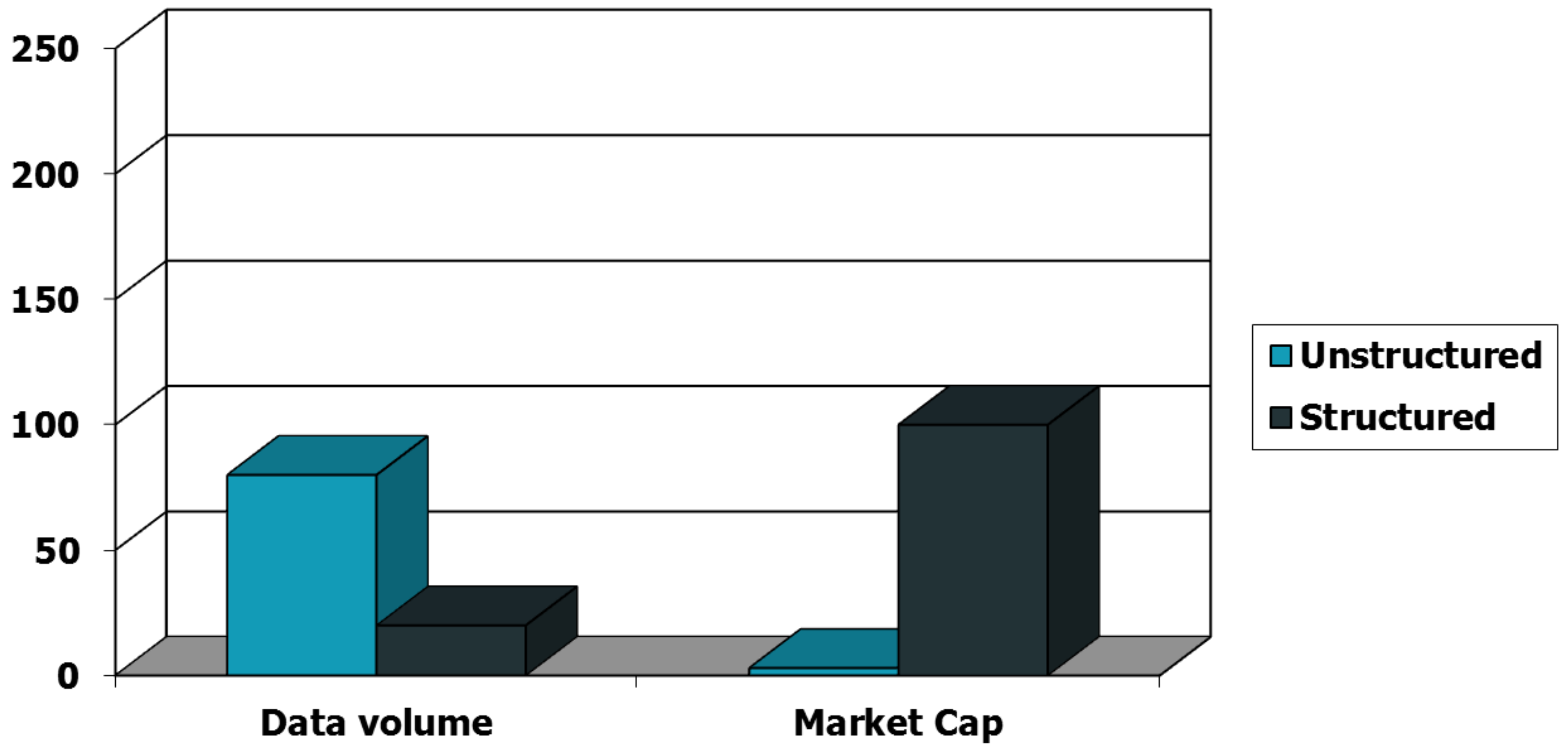
- Information Retrieval (IR) is finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers).

  – These days we frequently think first of web search, but there are many other cases:
    - E-mail search
    - Searching your laptop
    - Corporate knowledge bases
    - Legal information retrieval

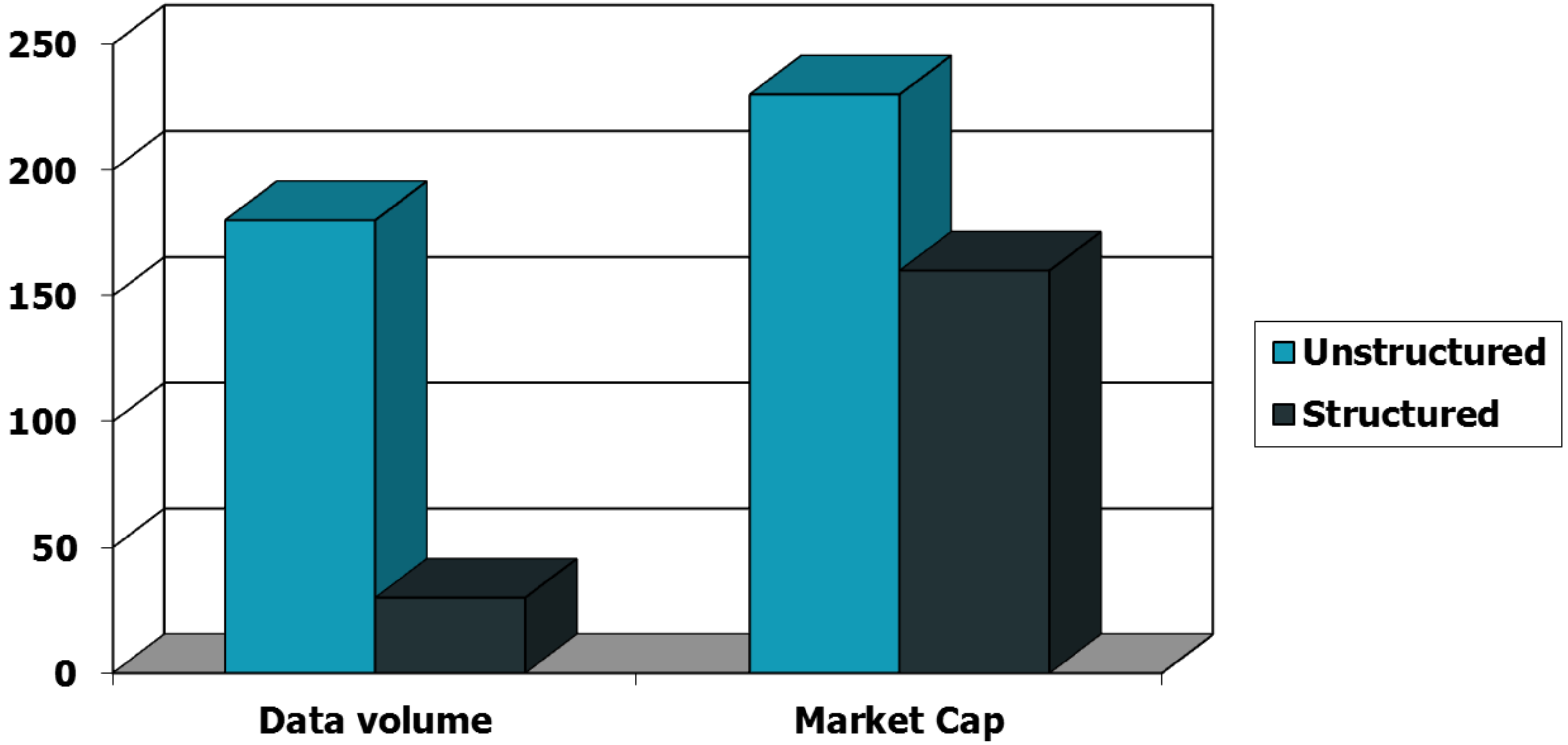
IR System


Document corpus
Query String
IR System
Ranked Documents
1. Doc1
2. Doc2
3. Doc3
.
.

Unstructured (text) vs. structured (database) data in the mid-nineties



Unstructured (text) vs. structured (database) data today



Relevance

Relevance is a subjective judgment and may include:

- Being on the proper subject.
- Being timely (recent information).
- Being authoritative (from a trusted source).
- Satisfying the goals of the user and his/her intended use of the information (*information need*).

❖ Simplest notion of relevance is that the query string appears verbatim in the document.

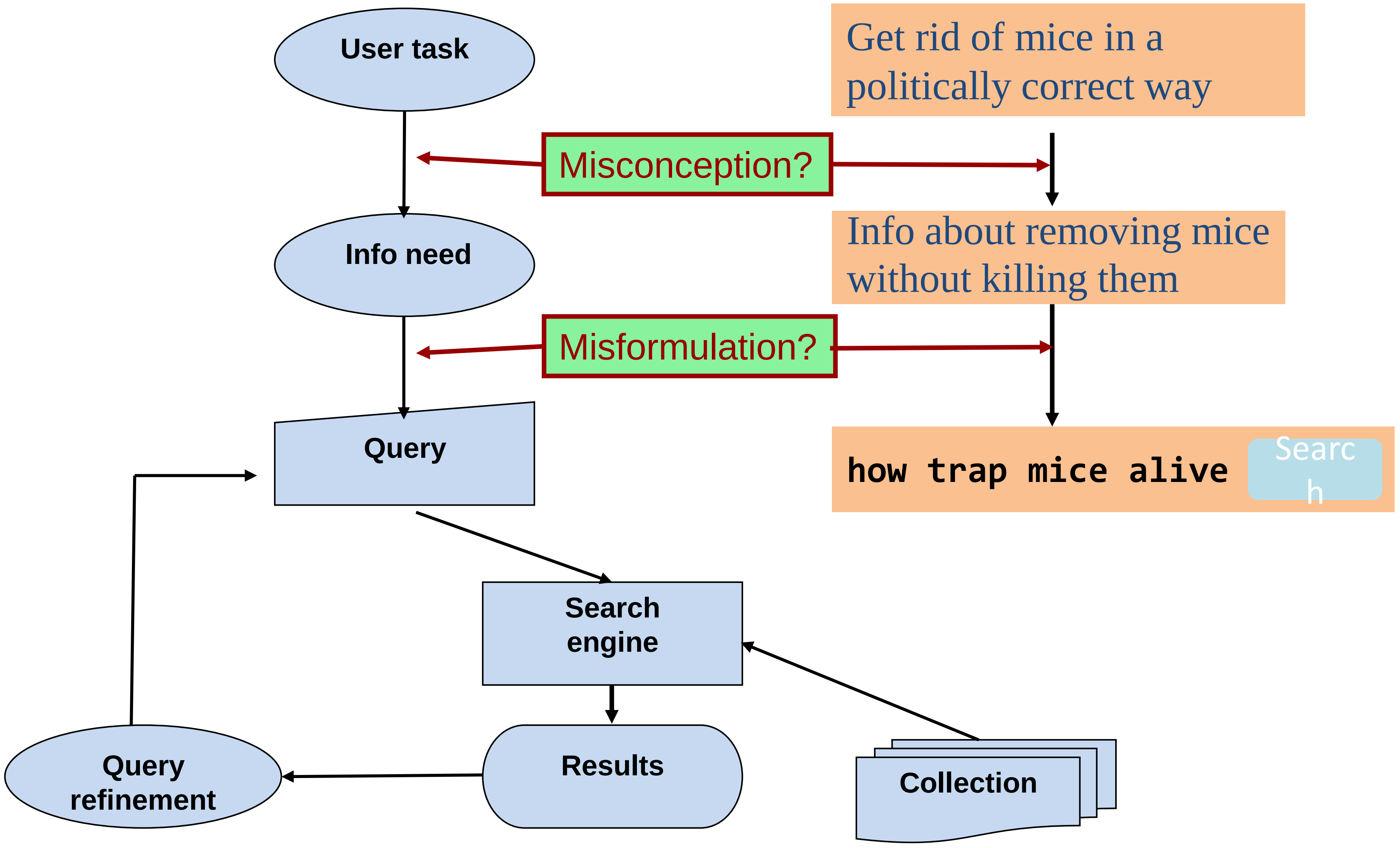
❖ Slightly less strict notion is that the words in the query appear frequently in the document, in any order (*bag of words*).

Basic assumptions of Information Retrieval

- Collection: A set of documents
  - Assume it is a static collection for the moment

- Goal: Retrieve documents with information that is relevant to the user's information need and helps the user complete a task

The classic search model



How good are the retrieved docs?

- *Precision* : Fraction of retrieved docs that are relevant to the user's information need
- *Recall* : Fraction of relevant docs in collection that are retrieved

  - More precise definitions and measurements to follow later

Unstructured data in 1620

- Which plays of Shakespeare contain the words ***Brutus*** *AND* ***Caesar*** but *NOT* ***Calpurnia***?
- One could `grep` all of Shakespeare's plays for ***Brutus*** and ***Caesar,*** then strip out lines containing ***Calpurnia***?
- Why is that not the answer?
  - Slow (for large corpora)
  - *NOT* ***Calpurnia*** is non-trivial
  - Other operations (e.g., find the word ***Romans*** near ***countrymen***) not feasible
  - Ranked retrieval (best documents to return)
    - Later lectures

Term-document incidence matrices

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	1	1	0	0	0	1
Brutus	1	1	0	1	0	0
Caesar	1	1	0	1	1	1
Calpurnia	0	1	0	0	0	0
Cleopatra	1	0	0	0	0	0
mercy	1	0	1	1	1	1
worser	1	0	1	1	1	0

Brutus *AND* ***Caesar*** *BUT NOT* ***Calpurnia***

1 if play contains word, 0 otherwise

Bigger collections

- Consider *N* = 1 million documents, each with about 1000 words.
- Avg 6 bytes/word including spaces/punctuation
  - 6GB of data in the documents.
- Say there are *M* = 500K *distinct* terms among these.

Can’t build the matrix

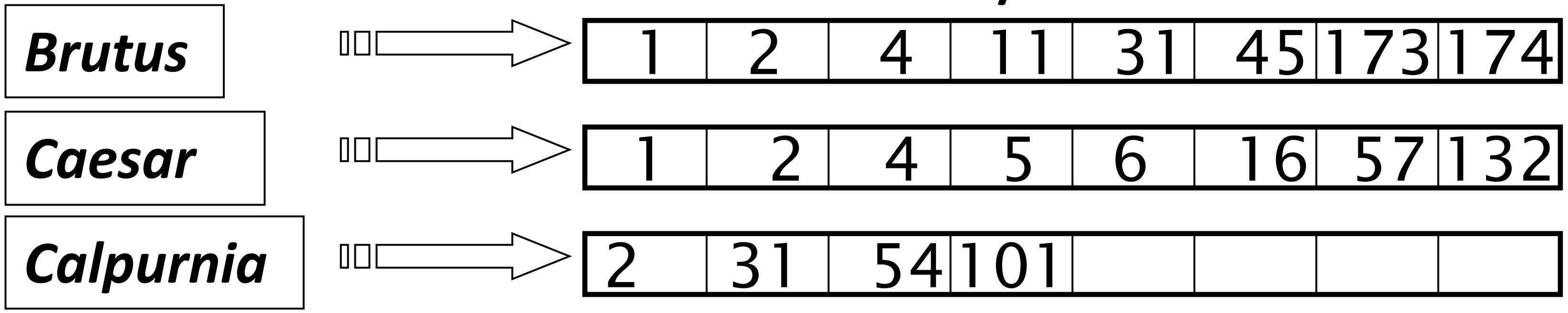
- 500K x 1M matrix has half-a-trillion 0’s and 1’s.
- But it has no more than one billion 1’s.
  - matrix is extremely sparse.

Why?

- What’s a better representation?
  - We only record the 1 positions.

Inverted index

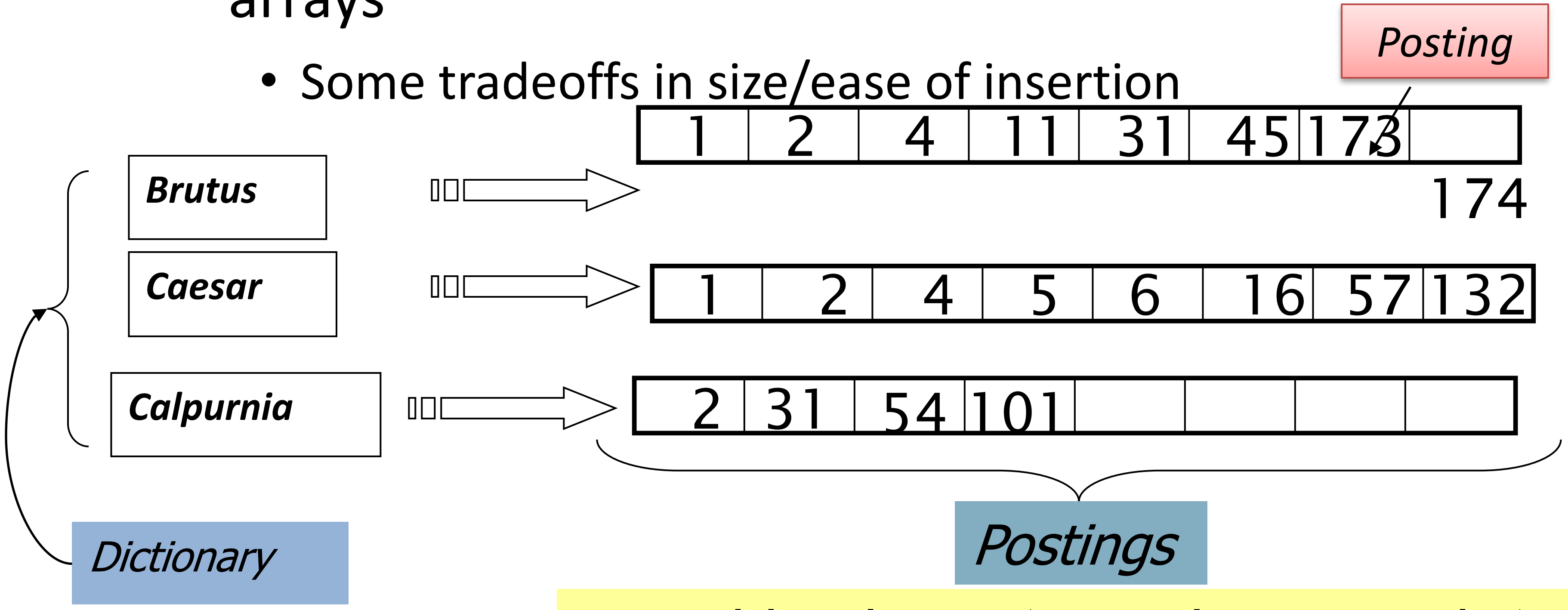
- For each term *t*, we must store a list of all documents that contain *t*.
  - Identify each doc by a **docID**, a document serial number
- Can we used fixed-size arrays for this?



What happens if the word ***Caesar*** is added to document 14?

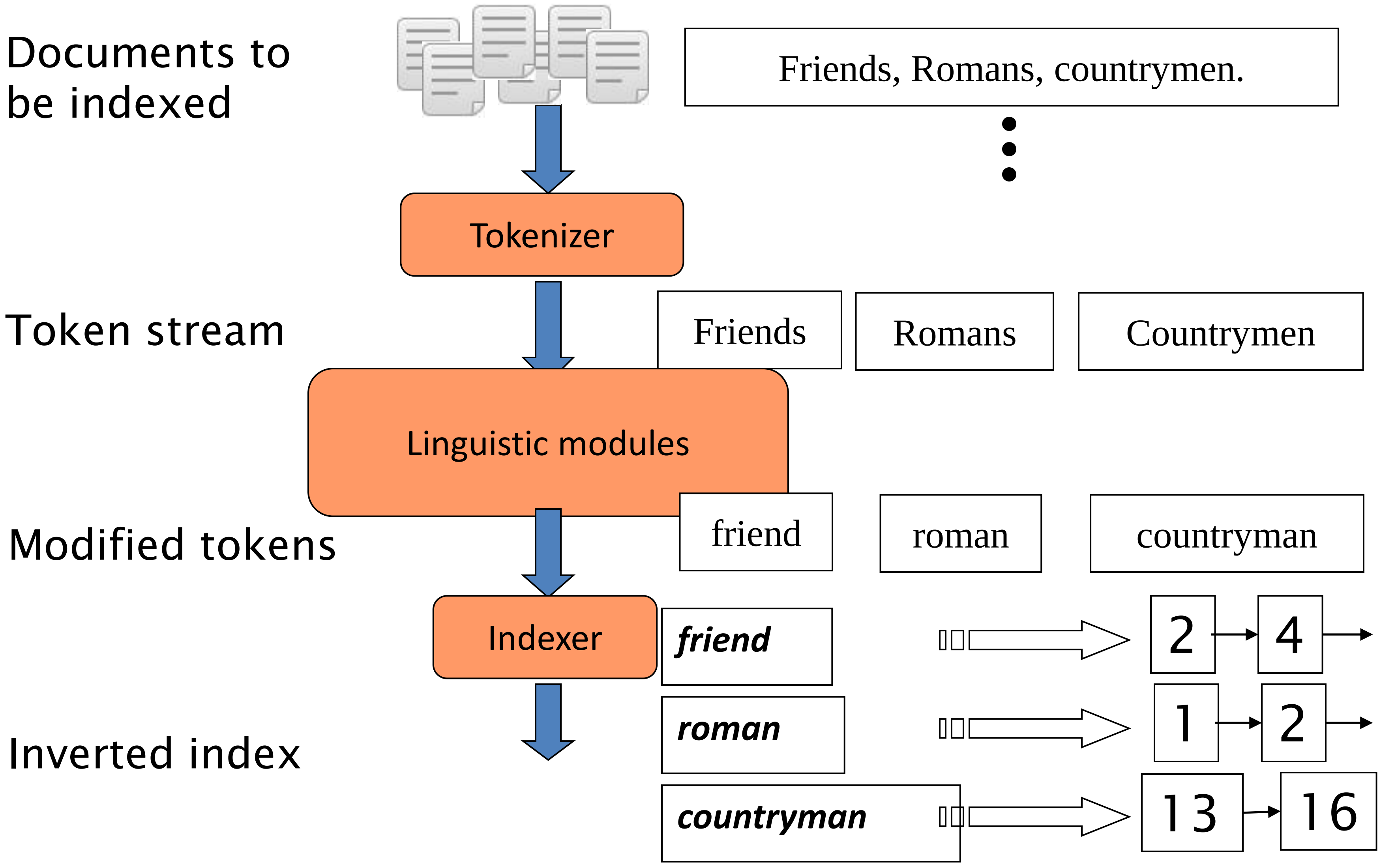
Inverted Index

- We need variable-size postings lists
  - On disk, a continuous run of postings is normal and best
  - In memory, can use linked lists or variable length arrays
    - Some tradeoffs in size/ease of insertion



Sorted by docID (more later on why).

Inverted index construction



Initial stages of text processing

- Tokenization
  - Cut character sequence into word tokens
    - Deal with ***"John's"***, ***a state-of-the-art solution***
- Normalization
  - Map text and query term to same form
    - You want ***U.S.A.*** and ***USA*** to match
- Stemming
  - We may wish different forms of a root to match
    - ***authorize***, ***authorization***
- Stop words
  - We may omit very common words (or not)
    - ***the, a, to, of***

Indexer steps: Token sequence

- Sequence of (Modified token, Document ID) pairs.

Doc 1

I did enact Julius Caesar I was killed i' the Capitol; Brutus killed me.

Doc 2

So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious

Term	docID
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

Indexer steps: Sort

- Sort by terms
  - And then docID

Core indexing step

Term	docID
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

Term	docID
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

Indexer steps: Dictionary & Postings

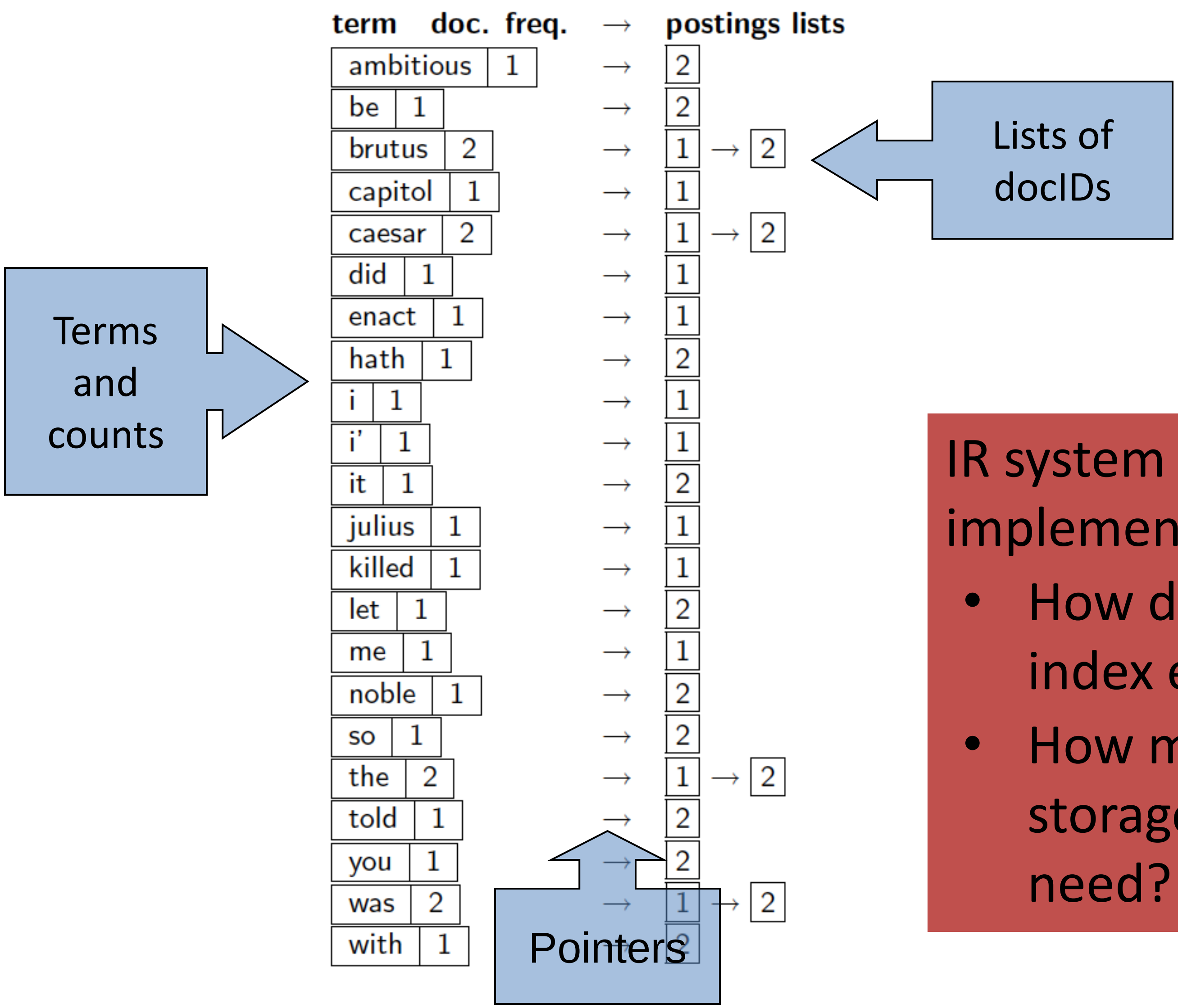
- Multiple term entries in a single document are merged.
- Split into Dictionary and Postings
- Doc. frequency information is added.

Why frequency?
Will discuss later.

Term	docID
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

term	doc. freq.	→	postings lists
ambitious	1	→	2
be	1	→	2
brutus	2	→	1 → 2
capitol	1	→	1
caesar	2	→	1 → 2
did	1	→	1
enact	1	→	1
hath	1	→	2
i	1	→	1
i'	1	→	1
it	1	→	2
julius	1	→	1
killed	1	→	1
let	1	→	2
me	1	→	1
noble	1	→	2
so	1	→	2
the	2	→	1 → 2
told	1	→	2
you	1	→	2
was	2	→	1 → 2
with	1	→	2

Where do we pay in storage?



term	doc. freq.	→	postings lists
ambitious	1	→	2
be	1	→	2
brutus	2	→	1 → 2
capitol	1	→	1
caesar	2	→	1 → 2
did	1	→	1
enact	1	→	1
hath	1	→	2
i	1	→	1
i'	1	→	1
it	1	→	2
julius	1	→	1
killed	1	→	1
let	1	→	2
me	1	→	1
noble	1	→	2
so	1	→	2
the	2	→	1 → 2
told	1	→	2
you	1	→	2
was	2	→	1 → 2
with	1	→	2

IR system implementation

- How do we index efficiently?
- How much storage do we need?

The index we just built

- How do we process a query?
  - Later - what kinds of queries can we process?

Our focus

Query processing: AND

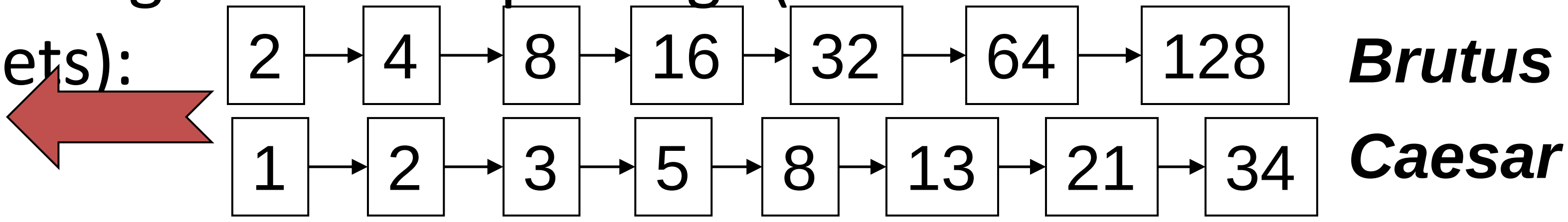
- Consider processing the query:

  Brutus *AND* ***Caesar***

  – Locate ***Brutus*** in the Dictionary;
    - Retrieve its postings.

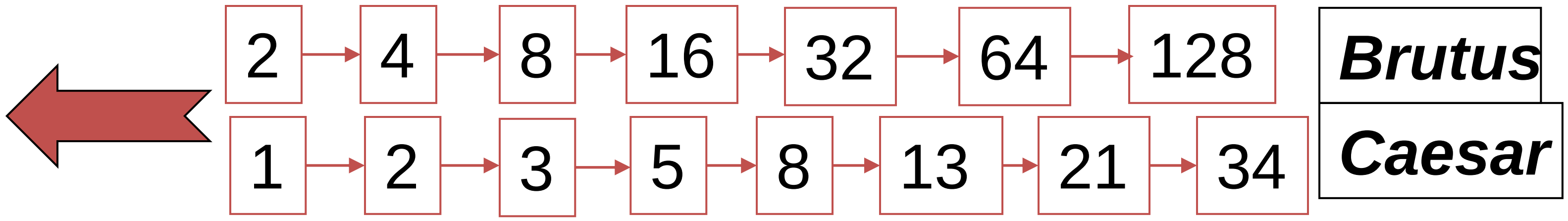
  – Locate ***Caesar*** in the Dictionary;
    - Retrieve its postings.

  – "Merge" the two postings (intersect the document sets):



The merge

- Walk through the two postings simultaneously, in time linear in the total number of postings entries



If the list lengths are *x* and *y*, the merge takes O(*x*+*y*) operations.
Crucial: postings sorted by docID.

Intersecting two postings lists (a "merge" algorithm)

```
INTERSECT($p_1, p_2$)
answer $\leftarrow \langle\ \rangle$
while $p_1 \neq$ NIL and $p_2 \neq$ NIL
do if $docID(p_1) = docID(p_2)$
      then ADD($answer, docID(p_1)$)
           $p_1 \leftarrow next(p_1)$
           $p_2 \leftarrow next(p_2)$
      else if $docID(p_1) < docID(p_2)$
              then $p_1 \leftarrow next(p_1)$
              else $p_2 \leftarrow next(p_2)$
return answer
```

Merging

What about an arbitrary Boolean formula?

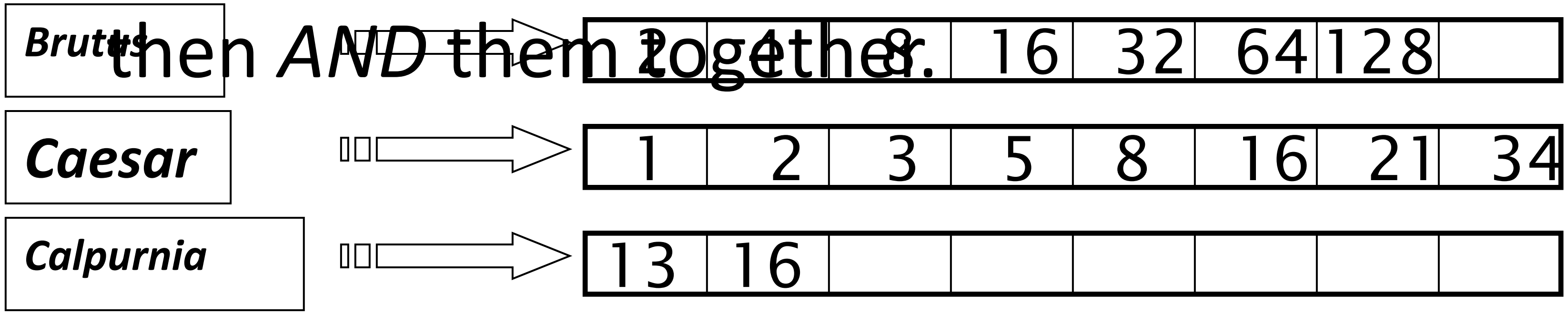
(Brutus *OR* ***Caesar)*** *AND NOT*

(Antony *OR* ***Cleopatra)***

- Can we always merge in “linear” time?
  – Linear in what?
- Can we do better?

Query optimization

- What is the best order for query processing?
- Consider a query that is an *AND* of *n* terms.
- For each of the *n* terms, get its postings, then *AND* them together.

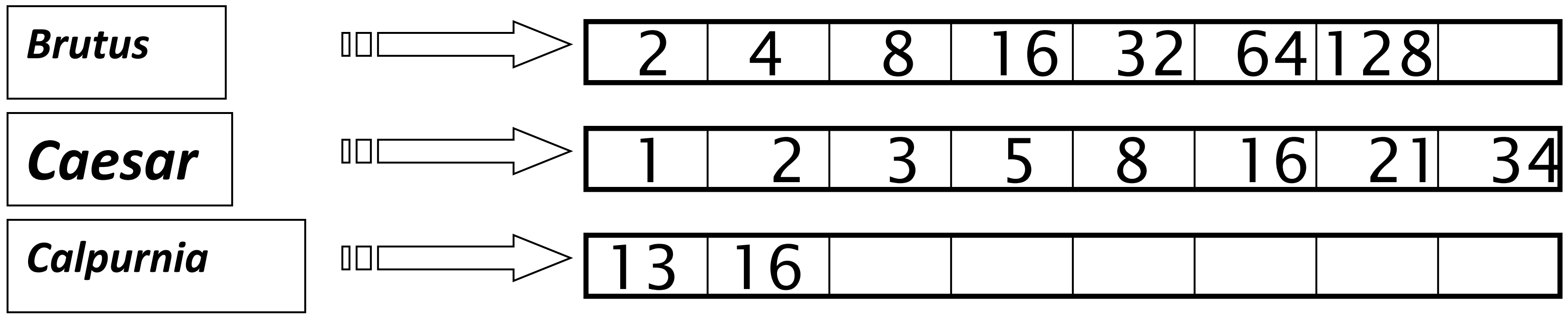


Query: ***Brutus*** *AND* ***Calpurnia*** *AND* ***Caesar***

Query optimization example

- Process in order of increasing freq:
  - *start with smallest set, then keep cutting further*.

This is why we kept document freq. in dictionary



Execute the query as (***Calpurnia*** *AND* ***Brutus)*** *AND* ***Caesar***.

More general optimization

- e.g., *(**madding** OR **crowd**) AND (**ignoble** OR **strife**)*
- Get doc. freq.'s for all terms.
- Estimate the size of each *OR* by the sum of its doc. freq.'s (conservative).
- Process in increasing order of *OR* sizes.

Query processing exercises

- Exercise: If the query is ***friends*** *AND* ***romans*** *AND (NOT* ***countrymen****)*, how could we use the freq of ***countrymen***?
- Exercise: Extend the merge to an arbitrary Boolean query. Can we always guarantee execution in time linear in the total postings size?
- Hint: Begin with the case of a Boolean *formula* query: in this, each query term appears only once in the query.

Introduction to

Information Retrieval

Faster postings list intersection via skip pointers

Brutus –→ 1 → 2 → 4 → 11 → 31 → 45 → 173 → 174 =m
Calpurnia –→ 2 → 31 → 54 → 101=n
Intersection =⇒ 2 → 31

Using intersection of merge algorithm this is the final posting list

. If the list lengths are m and n , the intersection takes $O(m+n)$ operations.
Can we do better than this? That is, empirically, can we usually process postings list intersection in sublinear time?
We can, if the index isn't changing too fast.

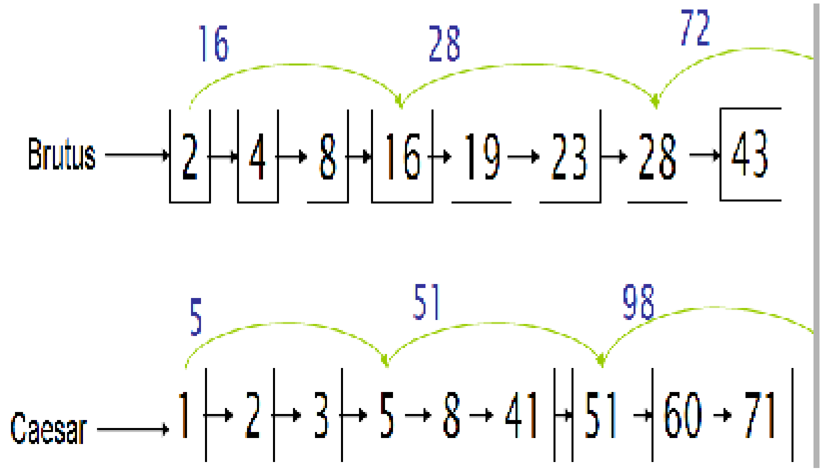


Fig: 1

Postings lists with skip pointers.The postings intersection can use a skip pointer when the end point is still less than the item on the other list.

```
INTERSECTWITHSKIPS($p_1, p_2$)
 1  $answer \leftarrow \langle\ \rangle$
 2  while $p_1 \neq$ NIL and $p_2 \neq$ NIL
 3  do if $docID(p_1) = docID(p_2)$
 4        then ADD($answer, docID(p_1)$)
 5             $p_1 \leftarrow next(p_1)$
 6             $p_2 \leftarrow next(p_2)$
 7        else if $docID(p_1) < docID(p_2)$
 8               then if $hasSkip(p_1)$ and $(docID(skip(p_1)) \leq docID(p_2))$
 9                      then while $hasSkip(p_1)$ and $(docID(skip(p_1)) \leq docID(p_2))$
10                             do $p_1 \leftarrow skip(p_1)$
11                      else $p_1 \leftarrow next(p_1)$
12               else if $hasSkip(p_2)$ and $(docID(skip(p_2)) \leq docID(p_1))$
13                      then while $hasSkip(p_2)$ and $(docID(skip(p_2)) \leq docID(p_1))$
14                             do $p_2 \leftarrow skip(p_2)$
15                      else $p_2 \leftarrow next(p_2)$
16  return $answer$
```

Suppose we've stepped through the lists in the figure until we have matched 8 on each list and moved it to the results list.

We advance both pointers, giving us 16 on the upper list and 41 on the lower list.
The smallest item is then the element 16 on the top list. Rather than simply advancing the upper pointer, we first check the skip list pointer and note that 28 is also less than 41.

Hence we can follow the skip list pointer, and then we advance the upper pointer to 28 . We thus avoid stepping to 19 and 23 on the upper list.

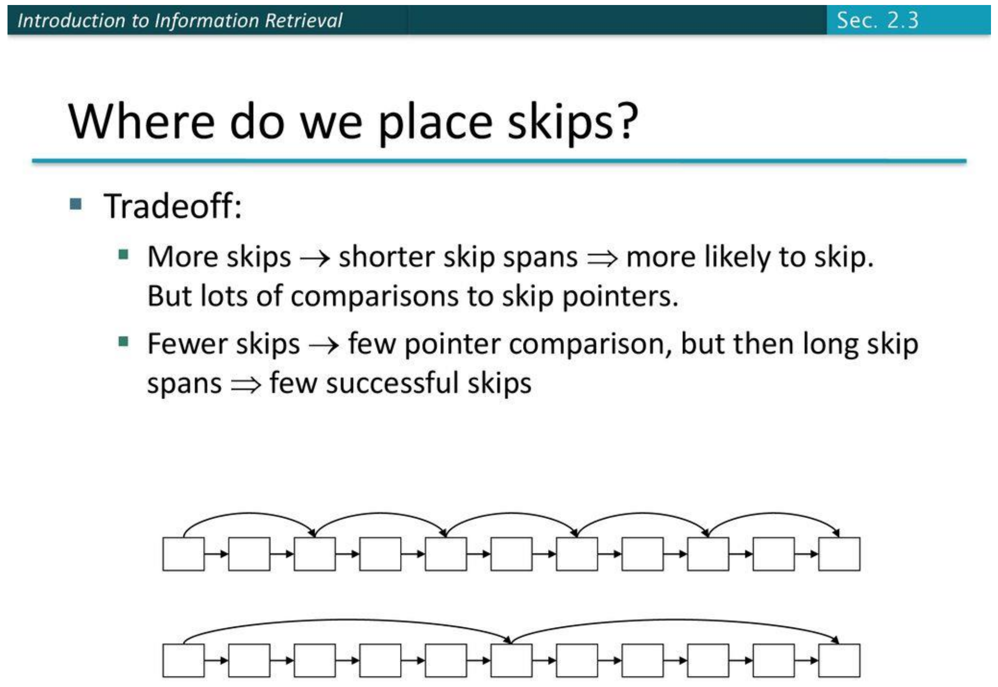
A number of variant versions of postings list intersection with skip pointers is possible depending on when exactly you check the skip pointer.

One version is shown in Figure 1 . Skip pointers will only be available for the original postings lists.
For an intermediate result in a complex query, the call hasSkip(p) will always return false.
Finally, note that the presence of skip pointers only helps for AND queries, not for OR queries.

A simple heuristic for placing skips, which has been found to work well in practice, is that for a postings list of length $\underline{P}$, use $\underline{\sqrt{P}}$ evenly-spaced skip pointers. This heuristic can be improved upon; it ignores any details of the distribution of query terms.

Where do we place skips? There is a tradeoff.

More skips → shorter skip span → more likely to skip. But it also means lots of comparisons to skip pointers, and lots of space storing skip pointers.
Fewer skips → few pointer comparisons → but then long skip spans fewer opportunities to skip.



Where do we place skips?

- Tradeoff:
  - More skips $\rightarrow$ shorter skip spans $\Rightarrow$ more likely to skip. But lots of comparisons to skip pointers.
  - Fewer skips $\rightarrow$ few pointer comparison, but then long skip spans $\Rightarrow$ few successful skips

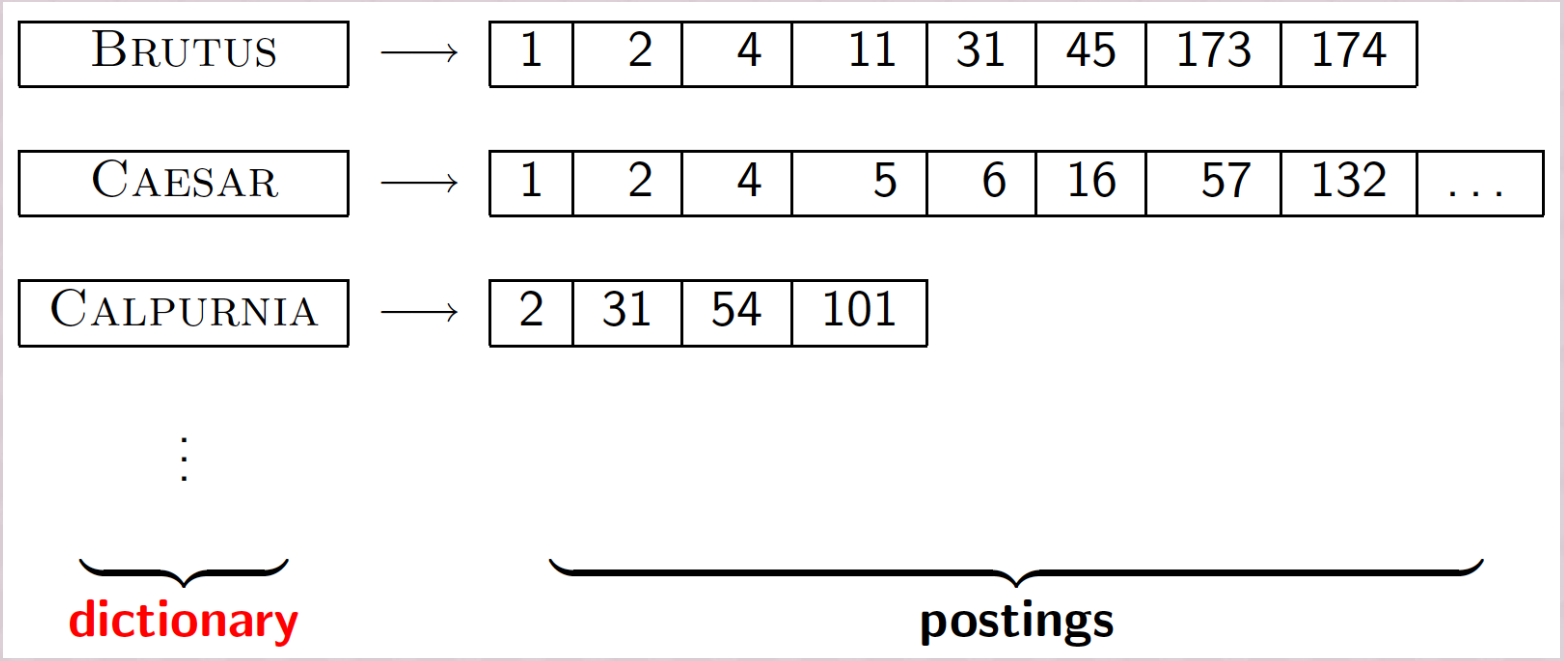
F.A.Q

1)
We have a two-word query. For one term the postings list consists of the following 16 entries:
[4,6,10,12,14,16,18,20,22,32,47,81,120,122,157,180]
and for the other it is the one entry postings list:
[47].
Work out how many comparisons would be done to intersect the two postings lists with the following two strategies. Briefly justify your answers:
a. Using standard postings lists
b. Using postings lists stored with skip pointers, with a skip length of $\sqrt{P}$.

2) Why are skip pointers not useful for queries of the form x OR y?

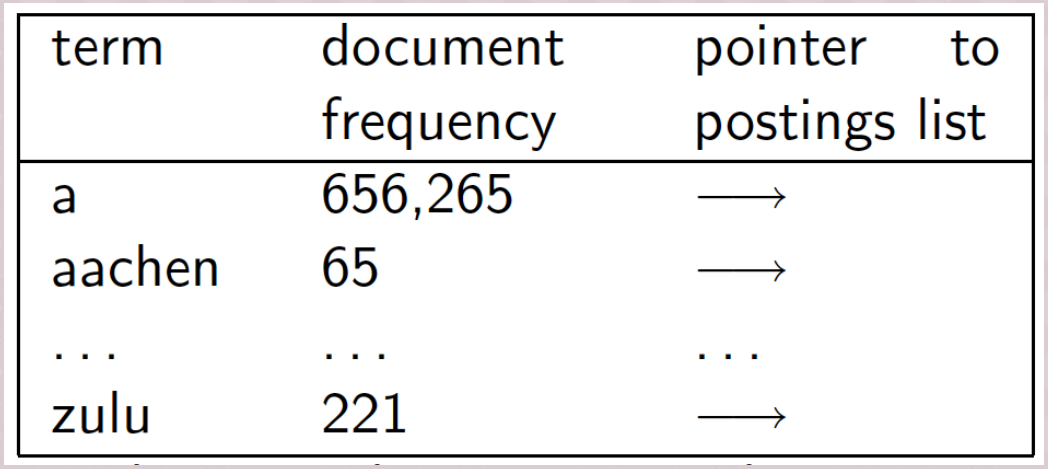
DICTIONARY DATA STRUCTURES FOR INVERTED INDEXES

- The dictionary data structure stores the term vocabulary, document frequency, pointers to each postings list … in what data structure?


BRUTUS → 1 2 4 11 31 45 173 174
CAESAR → 1 2 4 5 6 16 57 132 ...
CALPURNIA → 2 31 54 101
⋮
dictionary
postings

A NAÏVE DICTIONARY

- An array of struct:



term	document frequency	pointer to postings list
a	656,265	$\longrightarrow$
aachen	65	$\longrightarrow$
...	...	...
zulu	221	$\longrightarrow$

char[20]	int	Postings *
20 bytes	4/8 bytes	4/8 bytes

- How do we store a dictionary in memory efficiently?
- How do we quickly look up elements at query time?

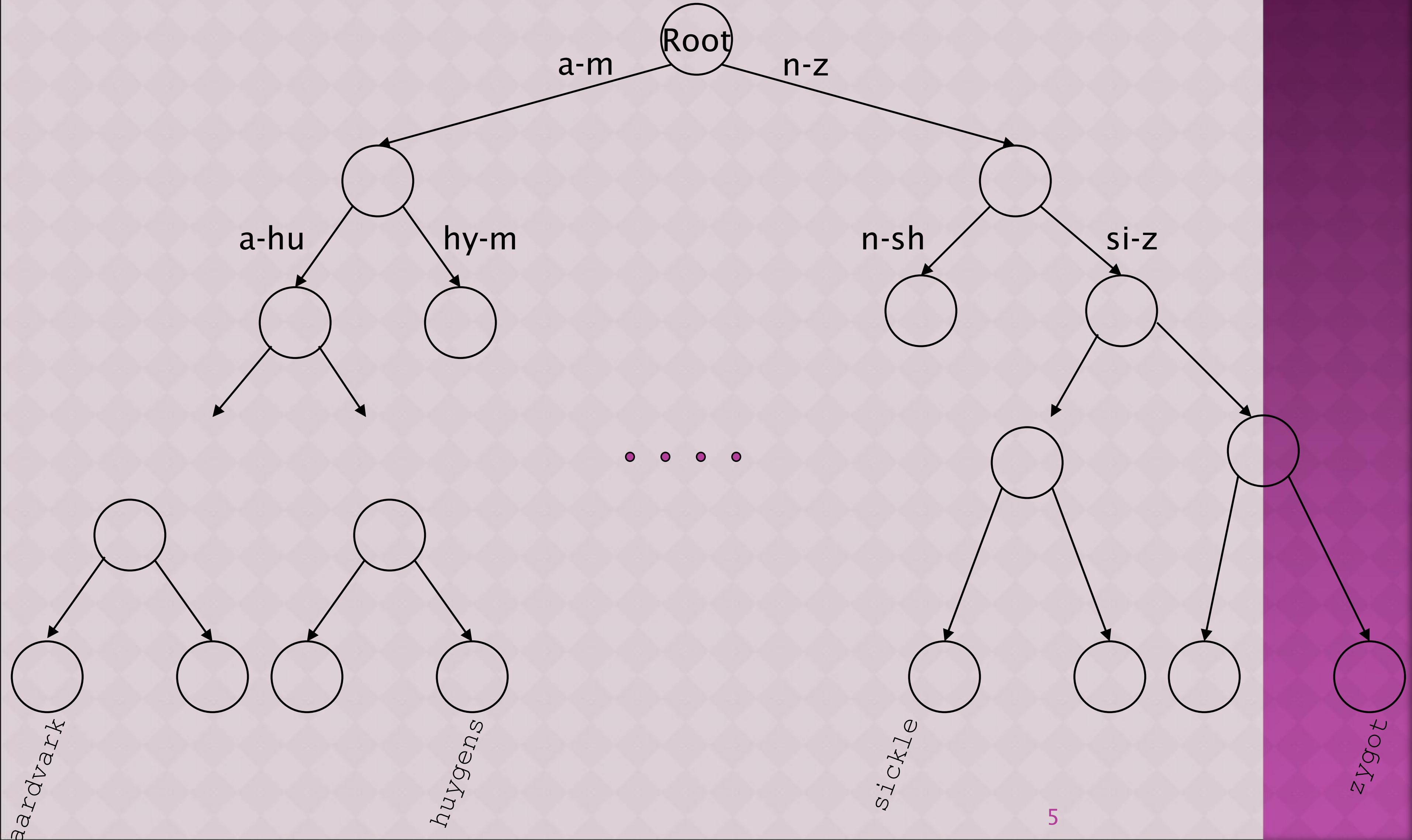
DICTIONARY DATA STRUCTURES

- Two main choices:
  - Hashtables
  - Trees
- Some IR systems use hashtables, some trees

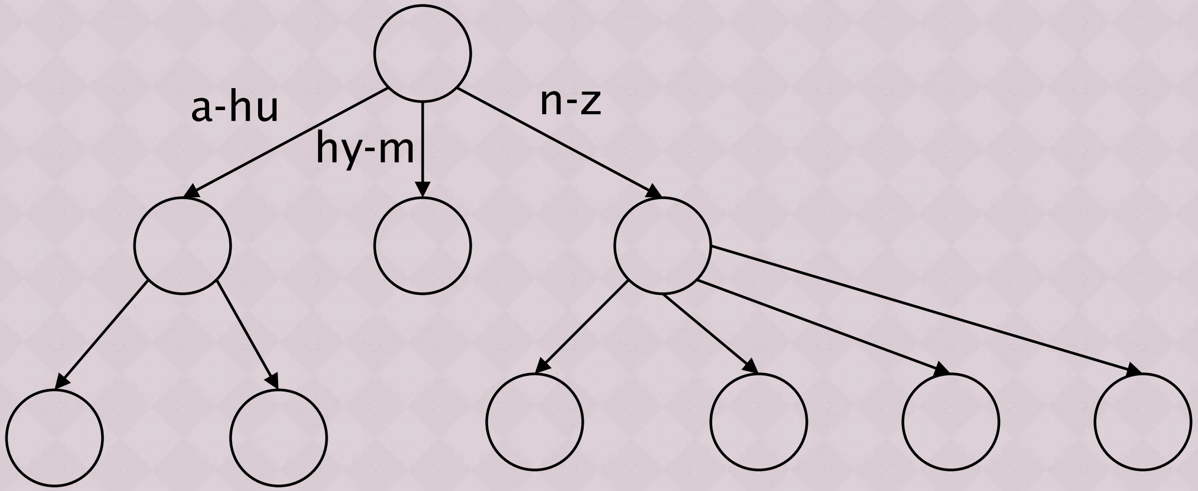
HASHTABLES

- Each vocabulary term is hashed to an integer
  - (We assume you've seen hashtables before)
- Pros:
  - Lookup is faster than for a tree: O(1)
- Cons:
  - No easy way to find minor variants:
    - judgment/judgement
  - No prefix search [tolerant retrieval]
  - If vocabulary keeps growing, need to occasionally do the expensive operation of rehashing *everything*

TREE: BINARY TREE



TREE: B-TREE



- Definition: Every internal nodel has a number of children in the interval [*a*,*b*] where *a*, *b* are appropriate natural numbers, e.g., [2,4].

TREES

- Simplest: binary tree
- More usual: B-trees
- Trees require a standard ordering of characters and hence strings … but we typically have one
- Pros:
  - Solves the prefix problem (terms starting with *hyp*)
- Cons:
  - Slower: O(log *M*) [and this requires *balanced* tree]
  - Rebalancing binary trees is expensive
    - But B-trees mitigate the rebalancing problem

WILD-CARD QUERIES

WILD-CARD QUERIES: *

- ***mon****: find all docs containing any word beginning with “mon”.
- Easy with binary tree (or B-tree) lexicon: retrieve all words in range: ***mon* ≤ *w* < *moo***
- ****mon*:** find words ending in “mon”: harder
  - Maintain an additional B-tree for terms *backwards*.

  Can retrieve all words in range: ***nom* ≤ *w* < *non***.

Exercise: from this, how can we enumerate all terms meeting the wild-card query ***pro*cent*** ?

QUERY PROCESSING

- At this point, we have an enumeration of all terms in the dictionary that match the wild-card query.
- We still have to look up the postings for each enumerated term.
- E.g., consider the query:

  se*ate *AND* ***fil*er***

  This may result in the execution of many Boolean *AND* queries.

B-TREES HANDLE *'S AT THE END OF A QUERY TERM

- How can we handle *'s in the middle of query term?
  - ***co*tion***
- We could look up ***co**** AND ***tion*** in a B-tree and intersect the two term sets
  - Expensive
- The solution: transform wild-card queries so that the *'s occur at the end
- This gives rise to the **Permuterm** Index.

PERMUTERM INDEX

- For term ***hello***, index under:
  - ***hello$, ello$h, llohe, lohel, o$hell, $hello***

  where $ is a special symbol.
- Queries:
  - X   lookup on **X$**   X*   lookup on   **$X***
  - *X   lookup on **X$***   *X*   lookup on   **X***
  - X*Y lookup on **Y$X***   X*Y*Z   ??? Exercise!

Query = ***hel*o***
X=*hel*, Y=*o*
Lookup ***o$hel****

PERMUTERM QUERY PROCESSING

- Rotate query wild-card to the right
- Now use B-tree lookup as before.
- *Permuterm problem: ≈ quadruples lexicon size*

Empirical observation for English.

BIGRAM (*K*-GRAM) INDEXES

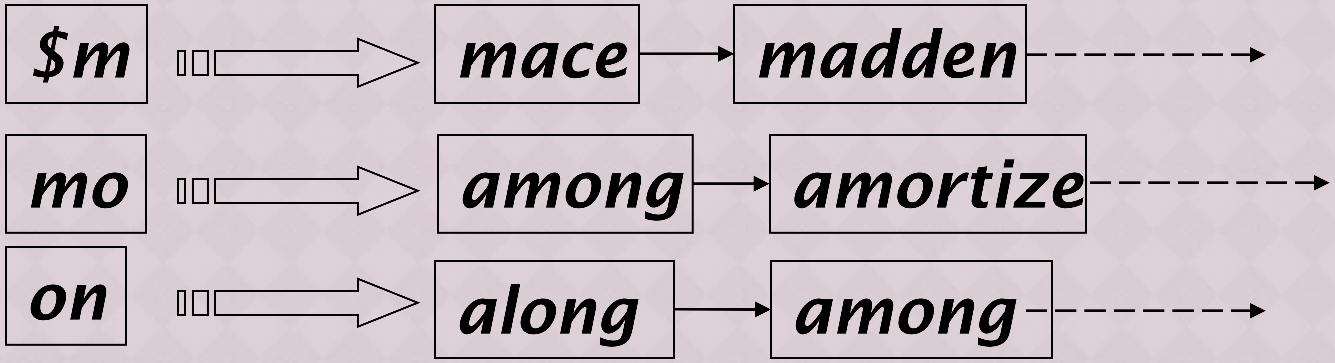
- Enumerate all *k*-grams (sequence of *k* chars) occurring in any term
- *e.g.*, from text "***April is the cruelest month***" we get the 2-grams (*bigrams*)

  a,ap,pr,ri,il,l,i,is,s,t,th,he,e,c,cr,ru,ue,el,le,es,st,t, m,mo,on,nt,h

  - $ is a special word boundary symbol
- Maintain a *second* inverted index *from bigrams to dictionary terms* that match each bigram.

BIGRAM INDEX EXAMPLE

- The *k*-gram index finds *terms* based on a query consisting of *k*-grams (here *k*=2).



PROCESSING WILD-CARDS

- Query ***mon**** can now be run as
  - *$m AND mo AND on*
- Gets terms that match AND version of our wildcard query.
- But we'd enumerate ***moon***.
- Must post-filter these terms against query.
- Surviving enumerated terms are then looked up in the term-document inverted index.
- Fast, space efficient (compared to permuterm).

PROCESSING WILD-CARD QUERIES

- As before, we must execute a Boolean query for each enumerated, filtered term.
- Wild-cards can result in expensive query execution (very large disjunctions…)
  - pyth* AND prog*
- If you encourage "laziness" people will respond!

- Which web search engines allow wildcard queries?

Type your search terms, use '*' if you need to.
E.g., Alex* will match Alexander.

SPELLING CORRECTION

SPELL CORRECTION

- Two principal uses
  - Correcting document(s) being indexed
  - Correcting user queries to retrieve "right" answers
- Two main flavors:
  - Isolated word
    - Check each word on its own for misspelling
    - Will not catch typos resulting in correctly spelled words
    - e.g., ***from*** $\rightarrow$ ***form***
  - Context-sensitive
    - Look at surrounding words,
    - e.g., ***I flew form Heathrow to Narita.***

DOCUMENT CORRECTION

- Especially needed for OCR'ed documents
  - Correction algorithms are tuned for this: rn/m
  - Can use domain-specific knowledge
    - E.g., OCR can confuse O and D more often than it would confuse O and I (adjacent on the QWERTY keyboard, so more likely interchanged in typing).
- But also: web pages and even printed material have typos
- Goal: the dictionary contains fewer misspellings
- But often we don't change the documents and instead fix the query-document mapping

QUERY MIS-SPELLINGS

- Our principal focus here
  - E.g., the query ***Alanis Morisett***
- We can either
  - Retrieve documents indexed by the correct spelling, OR
  - Return several suggested alternative queries with the correct spelling
    - *Did you mean … ?*

ISOLATED WORD CORRECTION

- Fundamental premise – there is a lexicon from which the correct spellings come
- Two basic choices for this
  - A standard lexicon such as
    - Webster's English Dictionary
    - An "industry-specific" lexicon – hand-maintained
  - The lexicon of the indexed corpus
    - E.g., all words on the web
    - All names, acronyms etc.
    - (Including the mis-spellings)

ISOLATED WORD CORRECTION

- Given a lexicon and a character sequence Q, return the words in the lexicon closest to Q
- What's "closest"?
- We'll study several alternatives
  - Edit distance (Levenshtein distance)
  - Weighted edit distance
  - *n*-gram overlap

EDIT DISTANCE

- Given two strings S_1 and S_2, the minimum number of operations to convert one to the other
- Operations are typically character-level
  - Insert, Delete, Replace, (Transposition)
- E.g., the edit distance from ***dof*** to ***dog*** is 1
  - From ***cat*** to ***act*** is 2 (Just 1 with transpose.)
  - from ***cat*** to ***dog*** is 3.
- Generally found by dynamic programming.
- See http://www.merriampark.com/ld.htm for a nice example plus an applet.

WEIGHTED EDIT DISTANCE

- As above, but the weight of an operation depends on the character(s) involved
  - Meant to capture OCR or keyboard errors
    Example: ***m*** more likely to be mis-typed as ***n*** than as ***q***
  - Therefore, replacing ***m*** by ***n*** is a smaller edit distance than by ***q***
  - This may be formulated as a probability model
- Requires weight matrix as input
- Modify dynamic programming to handle weights

USING EDIT DISTANCES

- Given query, first enumerate all character sequences within a preset (weighted) edit distance (e.g., 2)
- Intersect this set with list of "correct" words
- Show terms you found to user as suggestions
- Alternatively,
  - We can look up all possible corrections in our inverted index and return all docs … slow
  - We can run with a single most likely correction
- The alternatives disempower the user, but save a round of interaction with the user

EDIT DISTANCE TO ALL DICTIONARY TERMS?

- Given a (mis-spelled) query – do we compute its edit distance to every dictionary term?
  - Expensive and slow
  - Alternative?
- How do we cut the set of candidate dictionary terms?
- One possibility is to use *n*-gram overlap for this
- This can also be used by itself for spelling correction.

N-GRAM OVERLAP

- Enumerate all the *n*-grams in the query string as well as in the lexicon
- Use the *n*-gram index (recall wild-card search) to retrieve all lexicon terms matching any of the query *n*-grams
- Threshold by number of matching *n*-grams
    - Variants – weight by keyboard layout, etc.

EXAMPLE WITH TRIGRAMS

- Suppose the text is ***november***
  - Trigrams are *nov, ove, vem, emb, mbe, ber*.
- The query is ***december***
  - Trigrams are *dec, ece, cem, emb, mbe, ber*.
- So 3 trigrams overlap (of 6 in each term)
- How can we turn this into a normalized measure of overlap?

ONE OPTION – JACCARD COEFFICIENT

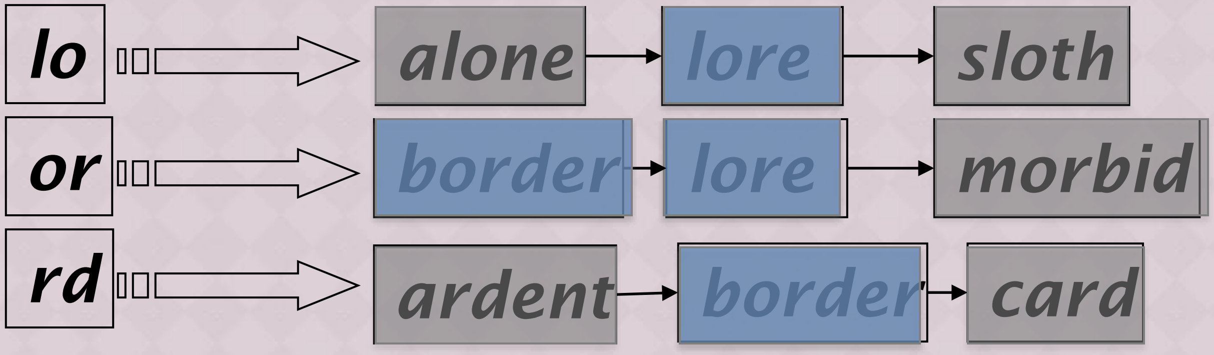
- A commonly-used measure of overlap
- Let *X* and *Y* be two sets; then the J.C. is

$$|X \cap Y| / |X \cup Y|$$

- Equals 1 when *X* and *Y* have the same elements and zero when they are disjoint
- *X* and *Y* don't have to be of the same size
- Always assigns a number between 0 and 1
  - Now threshold to decide if you have a match
  - E.g., if J.C. > 0.8, declare a match

MATCHING TRIGRAMS

- Consider the query ***lord*** – we wish to identify words matching 2 of its 3 bigrams (***lo***, ***or***, ***rd***)



Standard postings "merge" will enumerate …

Adapt this to using Jaccard (or another) measure.

CONTEXT-SENSITIVE SPELL CORRECTION

- Text: ***I flew <u>from</u> Heathrow to Narita*.**
- Consider the phrase query "***flew <u>form</u> Heathrow***"
- We'd like to respond

  Did you mean "***flew from Heathrow***"?

because no docs matched the query phrase.

CONTEXT-SENSITIVE CORRECTION

- Need surrounding context to catch this.
- First idea: retrieve dictionary terms close (in weighted edit distance) to each query term
- Now try all possible resulting phrases with one word “fixed” at a time
  - *flew from heathrow*
  - *fled form heathrow*
  - *flea form heathrow*
- **Hit-based spelling correction:** Suggest the alternative that has lots of hits.

EXERCISE

- Suppose that for ***“flew <u>form</u> Heathrow”*** we have 7 alternatives for flew, 19 for form and 3 for heathrow.

How many “corrected” phrases will we enumerate in this scheme?

ANOTHER APPROACH

- Break phrase query into a conjunction of biwords (Lecture 2).
- Look for biwords that need only one term corrected.
- Enumerate only phrases containing "common" biwords.

GENERAL ISSUES IN SPELL CORRECTION

- We enumerate multiple alternatives for "Did you mean?"
- Need to figure out which to present to the user
  - The alternative hitting most docs
  - Query log analysis
- More generally, rank alternatives probabilistically

$$\text{argmax}_{corr}\ P(corr \mid query)$$

  - From Bayes rule, this is equivalent to

$$\text{argmax}_{corr}\ \underbrace{P(query \mid corr)}_{\text{Noisy channel}} * \underbrace{P(corr)}_{\text{Language model}}$$

SOUNDEX

- Class of heuristics to expand a query into phonetic equivalents
  - Language specific – mainly for names
  - E.g., ***chebyshev*** $\rightarrow$ ***tchebycheff***
- Invented for the U.S. census … in 1918

SOUNDEX – TYPICAL ALGORITHM

- Turn every token to be indexed into a 4-character reduced form
- Do the same with query terms
- Build and search an index on the reduced forms
  - (when the query calls for a soundex match)

SOUNDEX – TYPICAL ALGORITHM

1. Retain the first letter of the word.
2. Change all occurrences of the following letters to '0' (zero):
   'A', E', 'I', 'O', 'U', 'H', 'W', 'Y'.
3. Change letters to digits as follows:

- B, F, P, V $\rightarrow$ 1
- C, G, J, K, Q, S, X, Z $\rightarrow$ 2
- D,T $\rightarrow$ 3
- L $\rightarrow$ 4
- M, N $\rightarrow$ 5
- R $\rightarrow$ 6

SOUNDEX CONTINUED

4. Remove all pairs of consecutive digits.
5. Remove all zeros from the resulting string.
6. Pad the resulting string with trailing zeros and return the first four positions, which will be of the form <uppercase letter> <digit> <digit> <digit>.

E.g., ***Herman*** becomes H655.

Will ***hermann*** generate the same code?

SOUNDEX

- Soundex is the classic algorithm, provided by most databases (Oracle, Microsoft, …)
- How useful is soundex?
- Not very – for information retrieval
- Okay for “high recall” tasks (e.g., Interpol), though biased to names of certain nationalities
- Zobel and Dart (1996) show that other algorithms for phonetic matching perform much better in the context of IR

WHAT QUERIES CAN WE PROCESS?

- We have
  - Positional inverted index with skip pointers
  - Wild-card index
  - Spell-correction
  - Soundex
- Queries such as

(SPELL(moriset) /3 toron*to) OR *SOUNDEX(**chaikofski)***

EXERCISE

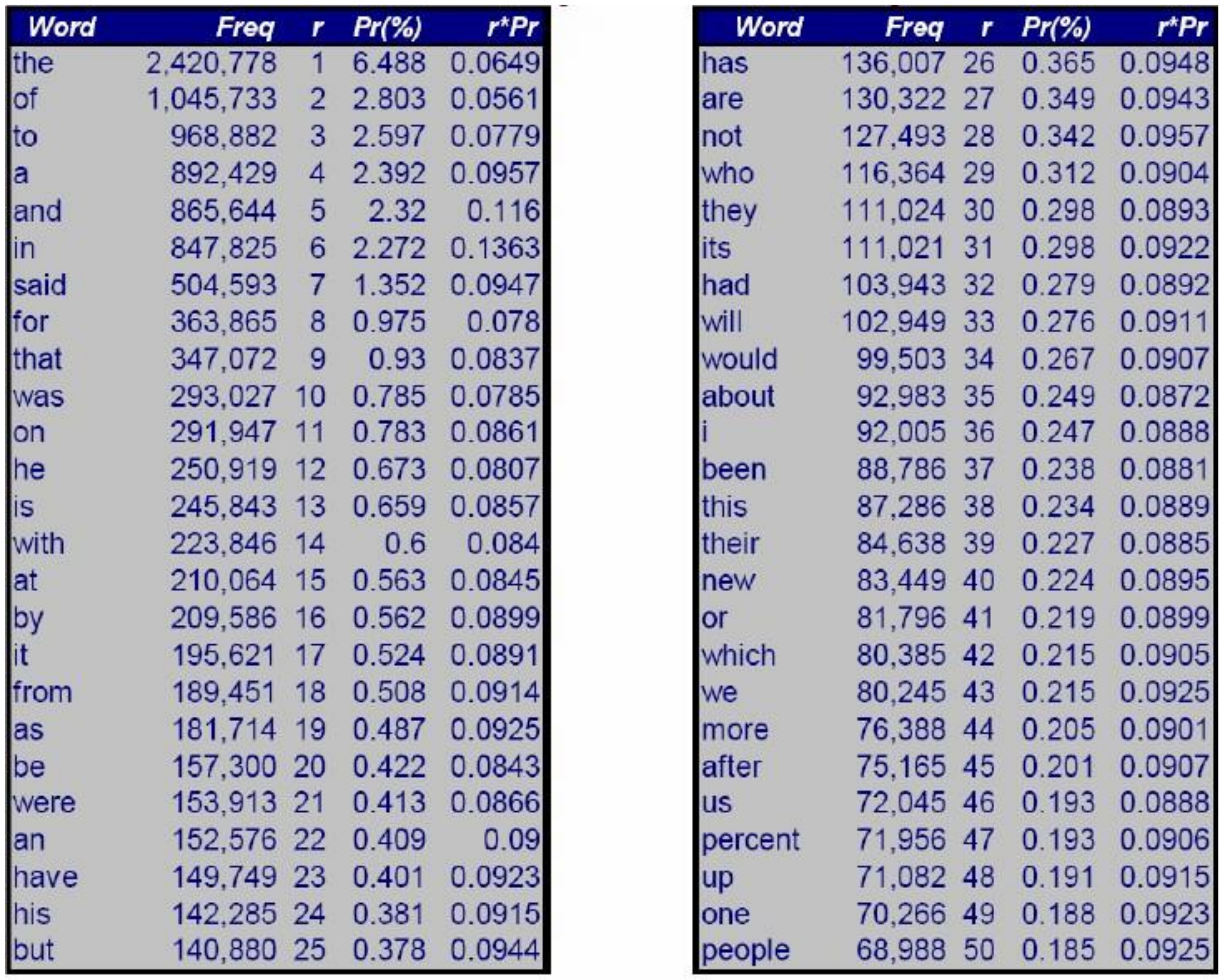
- Draw yourself a diagram showing the various indexes in a search engine incorporating all the functionality we have talked about
- Identify some of the key design choices in the index pipeline:
  - Does stemming happen before the Soundex index?
  - What about *n*-grams?
- Given a query, how would you parse and dispatch sub-queries to the various indexes?

Zipf's and Heap's law.

Zipf's law.

Zipf's law is a law about the frequency distribution of words in a language (or in a collection that is large enough so that it is representative of the language). To illustrate Zipf's law let us suppose we have a collection and let there be **V unique words** in the collection (the vocabulary).
For each word in the collection we need to compute the **freq(word)** = how many times word occurs in the collection. Then we rank the words in descending by their frequency (most frequent word has rank 1, next frequent word has rank 2, ...)

The slides provide an example, which we reproduce here:



Word	Freq	r	Pr(%)	r*Pr
the	2,420,778	1	6.488	0.0649
of	1,045,733	2	2.803	0.0561
to	968,882	3	2.597	0.0779
a	892,429	4	2.392	0.0957
and	865,644	5	2.32	0.116
in	847,825	6	2.272	0.1363
said	504,593	7	1.352	0.0947
for	363,865	8	0.975	0.078
that	347,072	9	0.93	0.0837
was	293,027	10	0.785	0.0785
on	291,947	11	0.783	0.0861
he	250,919	12	0.673	0.0807
is	245,843	13	0.659	0.0857
with	223,846	14	0.6	0.084
at	210,064	15	0.563	0.0845
by	209,586	16	0.562	0.0899
it	195,621	17	0.524	0.0891
from	189,451	18	0.508	0.0914
as	181,714	19	0.487	0.0925
be	157,300	20	0.422	0.0843
were	153,913	21	0.413	0.0866
an	152,576	22	0.409	0.09
have	149,749	23	0.401	0.0923
his	142,285	24	0.381	0.0915
but	140,880	25	0.378	0.0944
has	136,007	26	0.365	0.0948
are	130,322	27	0.349	0.0943
not	127,493	28	0.342	0.0957
who	116,364	29	0.312	0.0904
they	111,024	30	0.298	0.0893
its	111,021	31	0.298	0.0922
had	103,943	32	0.279	0.0892
will	102,949	33	0.276	0.0911
would	99,503	34	0.267	0.0907
about	92,983	35	0.249	0.0872
i	92,005	36	0.247	0.0888
been	88,786	37	0.238	0.0881
this	87,286	38	0.234	0.0889
their	84,638	39	0.227	0.0885
new	83,449	40	0.224	0.0895
or	81,796	41	0.219	0.0899
which	80,385	42	0.215	0.0905
we	80,245	43	0.215	0.0925
more	76,388	44	0.205	0.0901
after	75,165	45	0.201	0.0907
us	72,045	46	0.193	0.0888
percent	71,956	47	0.193	0.0906
up	71,082	48	0.191	0.0915
one	70,266	49	0.188	0.0923
people	68,988	50	0.185	0.0925

Top 50 words from 84,678 Associated Press 1989 articles

Let r be the rank of word, Prob(r) be the probability of a word at rank r. We do not care about the names of the words, we care only about their ranks and frequencies. By definition Prob(r) = freq(r) / N where
freq(r) = the number of times the word at rank r appears in the collection and
N = total number of words in the collection (not number of unique words).

Then Zipf's law states that

r * Prob(r) = A,

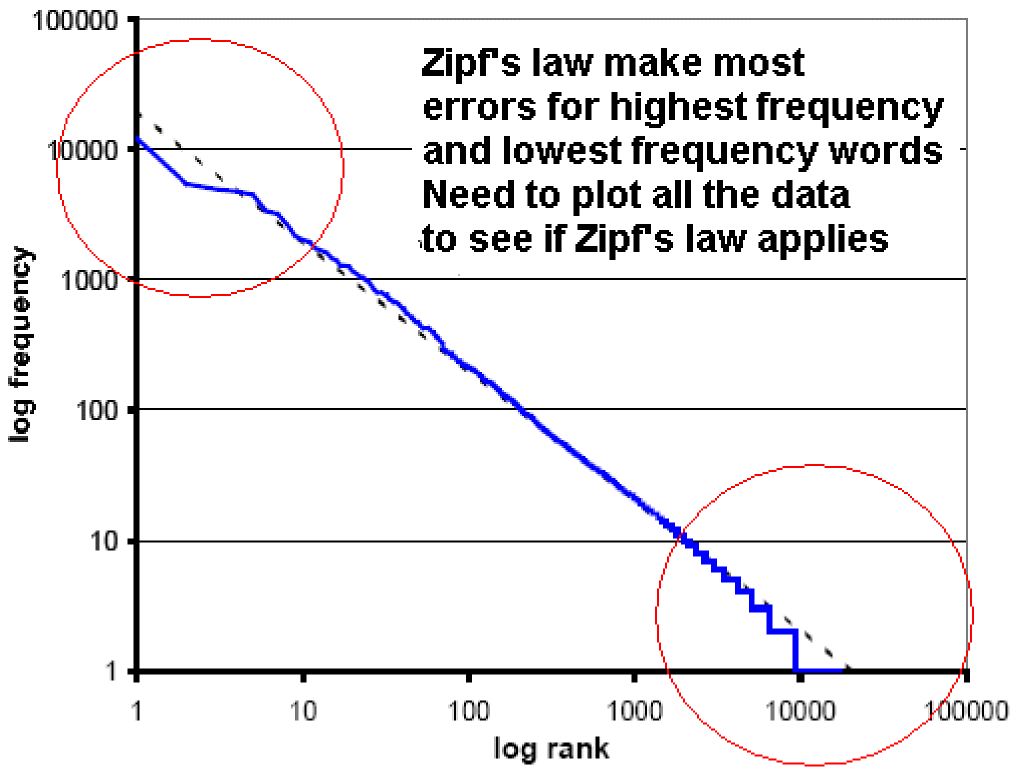
where A is a constant which should empirically be determined from the data. In most cases **A = 0.1**. Zipf's law is not an exact law, but a statistical law and therefore does not hold exactly but only on average (for most words).

Taking into account that Prob(r) = freq(r) / N we can rewrite Zipf's law as

r * freq(r) = A * N

To establish that Zip's law holds we need to compute freq(r), which involves computing the frequency of each word and then ranking the words. Then we need to compute r * freq(r) and see if r * freq(r) is approximately a constant. This does not mean that for all words r * freq(r) has to be exactly the same, but it has to be close to the same number for most words. The simplest way to show that Zipf's law holds is to plot the data. Remember that looking at most frequent and least frequent words only is misleading. For those types of words Zipf's law has the highest errors.

Instead of plotting r vs. freq(r), it is better to **plot log(r) on the x-axis and log(freq(r)) on the y axis**. **If Zipf's law holds we should see a line with slope -1** (this means if A is the point where the line crosses the x-axis and B is the point where the line crosses the y-axis and O is the origin of the coordinate system then OA = OB).
Another, equivalent way is to **plot log(r)** on the x-axis and **log(Prob(r))** on the y axis.



Zipf curve for the unigrams extracted from a 250,000 word tokens corpus

Source: Extension of Zipf's law to words and Phrases by Ha, Garcia, Smith

Notice the slope of the line is -1.

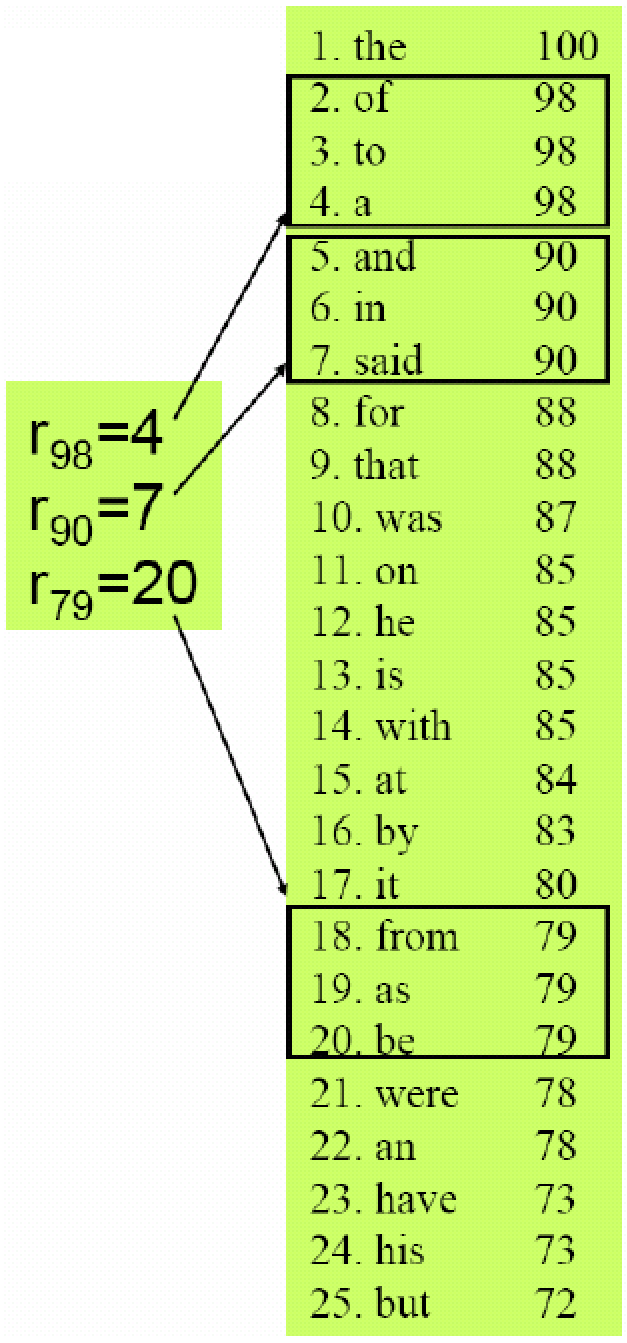
We can use Zipf's law to calculate the number of words that appear n times in the collection.

Let M**axRank(n)** = among all words that appear n times let MaxRank(n) be the maximum of the ranks of those words. For example, if n = 90, the words that appear n = 90 times are “and”, “in”, “said” with ranks 5, 6, 7. Then MaxRank(90) = max(5, 6, 7) = 7

Another example: MaxRank(79) = max(18, 19, 20) = 20

Notice that the number of words that appear n times is
NumberWordsOccur(n) =
MaxRank(n) – MaxRank(n + 1).

For example, the number of words that appear 79 times is MaxRank(79) – MaxRank(80) =
max(18, 19, 20) – max(17) = 20 – 17 = 3
We can look at the picture on the right side to see that exactly 3 words appear 79 times.



1. the	100
2. of	98
3. to	98
4. a	98
5. and	90
6. in	90
7. said	90
8. for	88
9. that	88
10. was	87
11. on	85
12. he	85
13. is	85
14. with	85
15. at	84
16. by	83
17. it	80
18. from	79
19. as	79
20. be	79
21. were	78
22. an	78
23. have	73
24. his	73
25. but	72

We know from Zipf's law that for the frequency and the rank are related.

If r = MaxRank(n), this means that the rank is r and the frequency is n; So r * freq(r) = A * N means

MaxRank(n) * n = A*N, which implies

MaxRank(n) = A * N / n

Applying this formula twice we obtain

NumberWordsOccur(n) =
MaxRank(n) – MaxRank(n + 1) =
A*N / n – A * N / (n + 1) =
A* N (1/n – 1/(n + 1) = A * N / [n * (n + 1)]
So,
NumberWordsOccur(n) = A * N / [n * (n + 1)] is the number of words that occur n times.

We need to connect A * N to the number of unique words in the collection. This is easy because V, which is the number of unique words is simply the rank of the last word in the ranked list of words. We need to assume(quite reasonably) that the least occurring word occurs

only once.
So, Zipf's law applied to the least frequent word gives (here: r = V, and freq(r) = 1)
r * freq(r) = A * N, V * 1 = A * N,
A * N = V

Therefore
NumberWordsOccur(n) = V / [n * (n + 1)]
is the number of words that occur n times.
where V is the number of unique words in the collection

Application of the formula **NumberWordsOccur(n) = V / [n * (n + 1)].**
What fraction of all unique words appear only once?

We need “number of words that occur once” / “number of unique words” =
= NumberWordsOccur(1) / V = [V / [1 * (1 + 1)]] / V = 1 / [1 * (1 + 1)] = ½

Heap's law.

Heap's law states that the number of unique words V in a collection with N words is approximately **Sqrt[N].** The more general form of this law is

$$V = KN^{\beta} \ (0 < \beta < 1)$$

Typically

– $K \approx 10\text{–}100$

– $\beta \approx 0.4\text{–}0.6$ (approx. square-root of n)

Alpha and beta and usually found by fitting the data.

Power laws

Zipf's and Heap's law belong to a class of laws called power laws.
A power law is one that has the form

$$y = k * x^{c}$$

k and c are constants that have to be fit from the data.
If we are to write Zipf's law as power low, we notice that
y = freq(r), x = r, k = A * N, c = -1

If we are to write Heap's law as a power law we observe that y = V, x = N, k = K (from Heap's law), c = beta

Power laws have the useful property that if one takes the log of both sides of
one obtains a line. See picture.

$$y = kx^c$$ **start here**

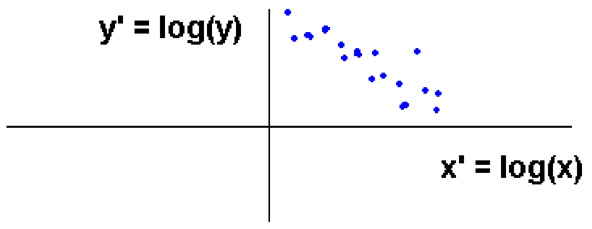
$$\log(y) = \log(kx^c)$$ **take the log of both sides**

$$\log(y) = \log(k) + c\log(x)$$ **simplify the right side**

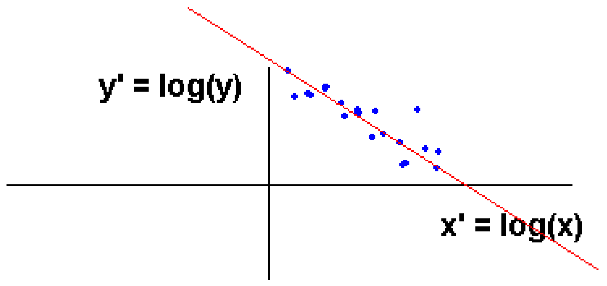
Let y' = log(y)
x' = log(x)

Transform the data by taking logs of the data

y' = m + c * x', where m = log(k) **This is a line**



plot the transformed data



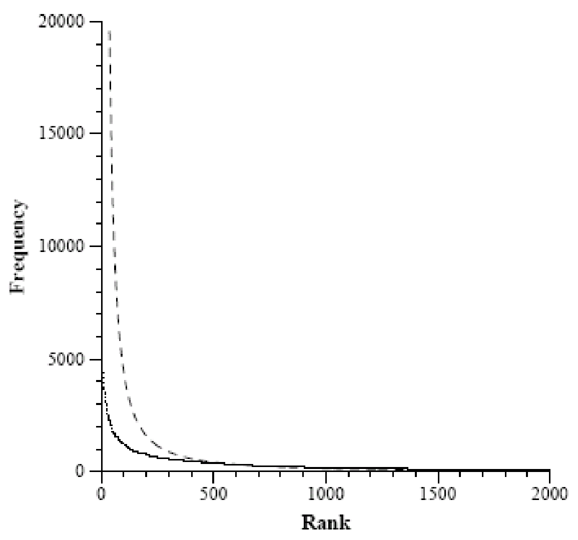
fit a line and find c and m

Use that m = log(k) to find that

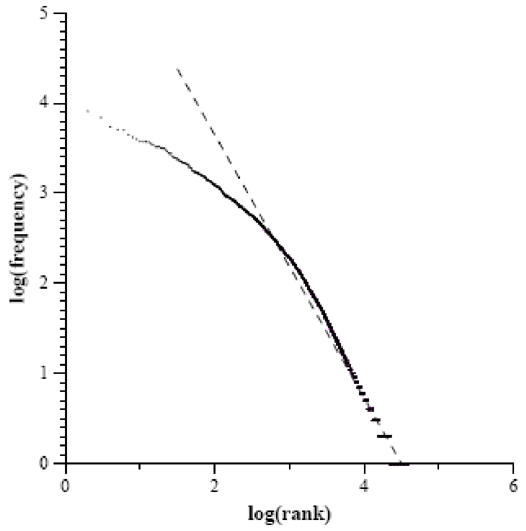
k = exp(m) **[watch the base of your log here]**

Note: when you do the line fitting, do not use x and y but use x' = log(x) and y' = log(y) in the formulas for the least squares line fitting.

Fitting Zipf's law:



Plot of term frequency vs. rank.

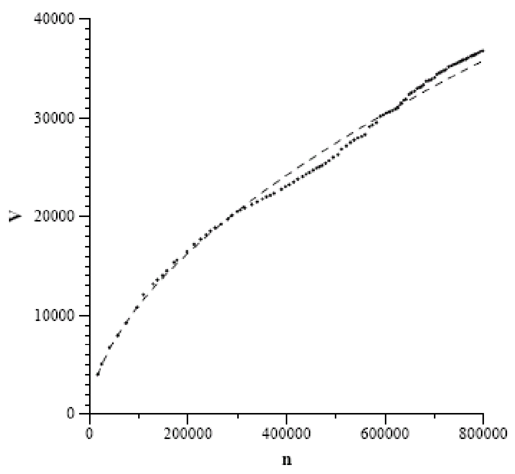


Log-log plot of term frequency vs. rank.

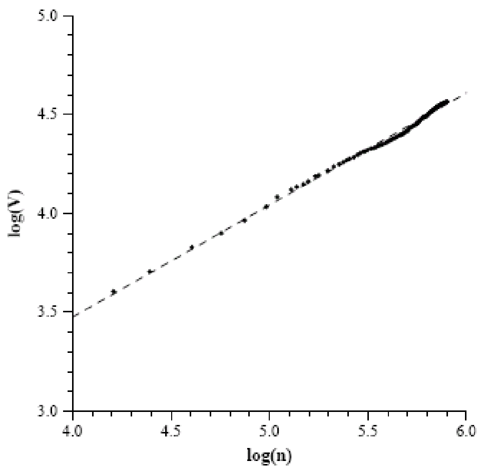
Source: Modeling Web Data by French

If you fit data that obeys Zipf's law you should get c close to -1.

Fitting Heap's law:



Plot of unique vocabulary terms vs. total terms.



Log-log plot of unique vocabulary terms vs. total terms.

n: number of total words encountered
V = V(n) = number of unique words encountered, is a function of n

(Source: Modeling Web Data by French)

If you fit data that obeys Heap's you should get c = slope of the line close to 0.5

COMPRESSION

- Now, we will consider compressing the space for the dictionary and postings
  - Basic Boolean index only
  - No study of positional indexes, etc.

- We will consider compression schemes
  - Dictionary compression
  - Postings list compression

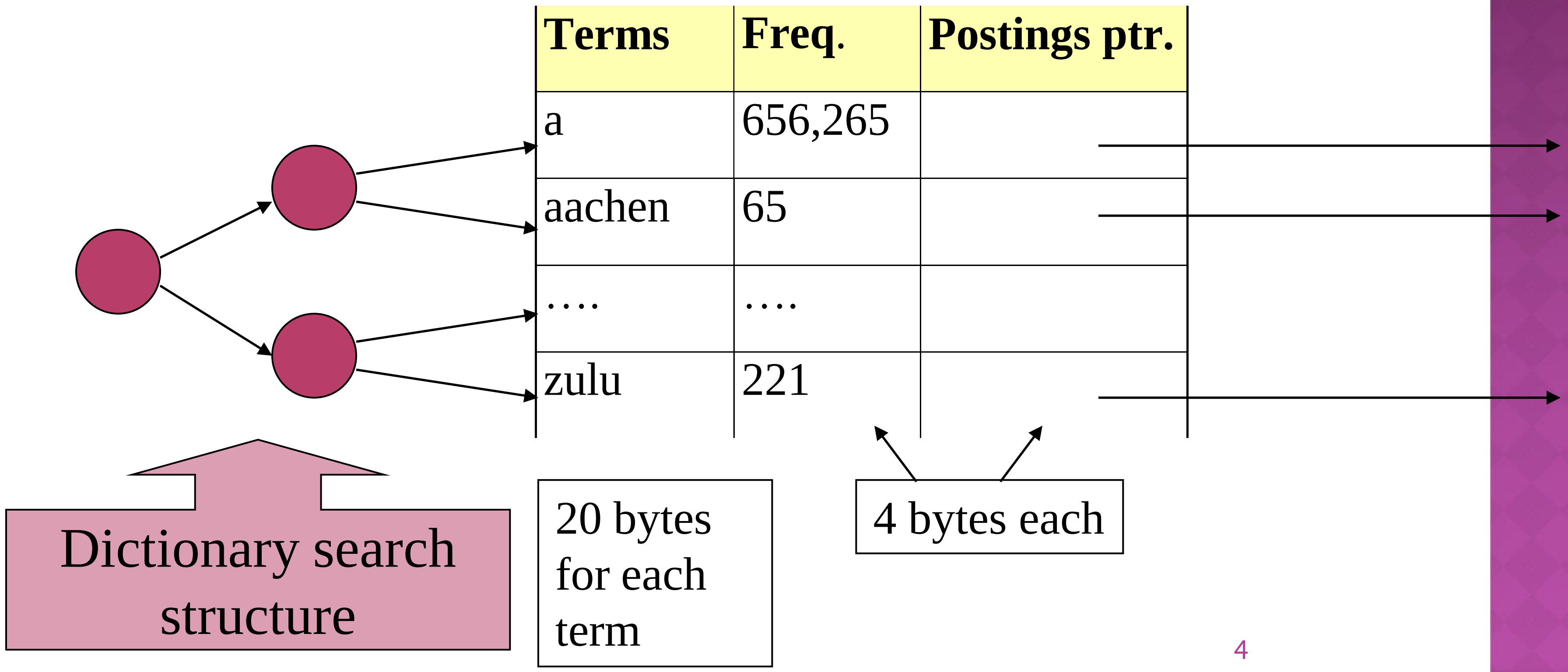
DICTIONARY COMPRESSION

WHY COMPRESS THE DICTIONARY?

- Search begins with the dictionary
- We want to keep it in memory
- Memory footprint: competition with other applications
- Embedded/mobile devices may have very little memory
- Even if the dictionary isn't in memory, we want it to be small for a fast search startup time
- So, compressing the dictionary is important

DICTIONARY STORAGE - FIRST CUT

- Array of fixed-width entries
  - ~400,000 terms; 28 bytes/term = 11.2 MB.



Terms	Freq.	Postings ptr.
a	656,265	
aachen	65	
….	….	
zulu	221	

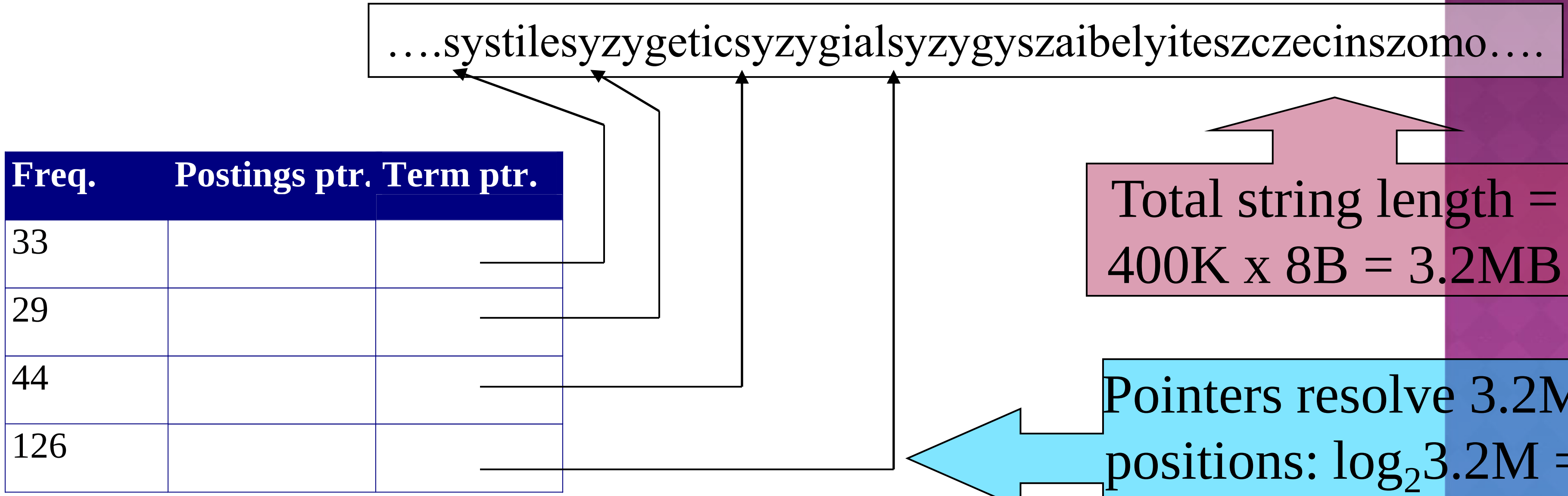
FIXED-WIDTH TERMS ARE WASTEFUL

- Most of the bytes in the **Term** column are wasted – we allot 20 bytes even for 1 letter terms.
  - And we still can't handle *terms with more than 20 chars*

- Written English averages ~4.5 characters/word.
- Ave. dictionary word in English: ~8 characters
  - How do we use ~8 characters per dictionary term?
- Short words dominate token counts but not type average.

COMPRESSING THE TERM LIST: APPROACH 1: DICTIONARY-AS-A-STRING

- Store dictionary as a (long) string of characters:
  - Pointer to next word shows end of current word
  - Hope to save up to 60% of dictionary space.



….systilesyzygeticsyzygialsyzygyszaibelyiteszczecinszomo….

Freq.	Postings ptr.	Term ptr.
33		
29		
44		
126		

Total string length = 400K x 8B = 3.2MB

Pointers resolve 3.2M positions: $\log_2 3.2M$ = 22bits = 3bytes

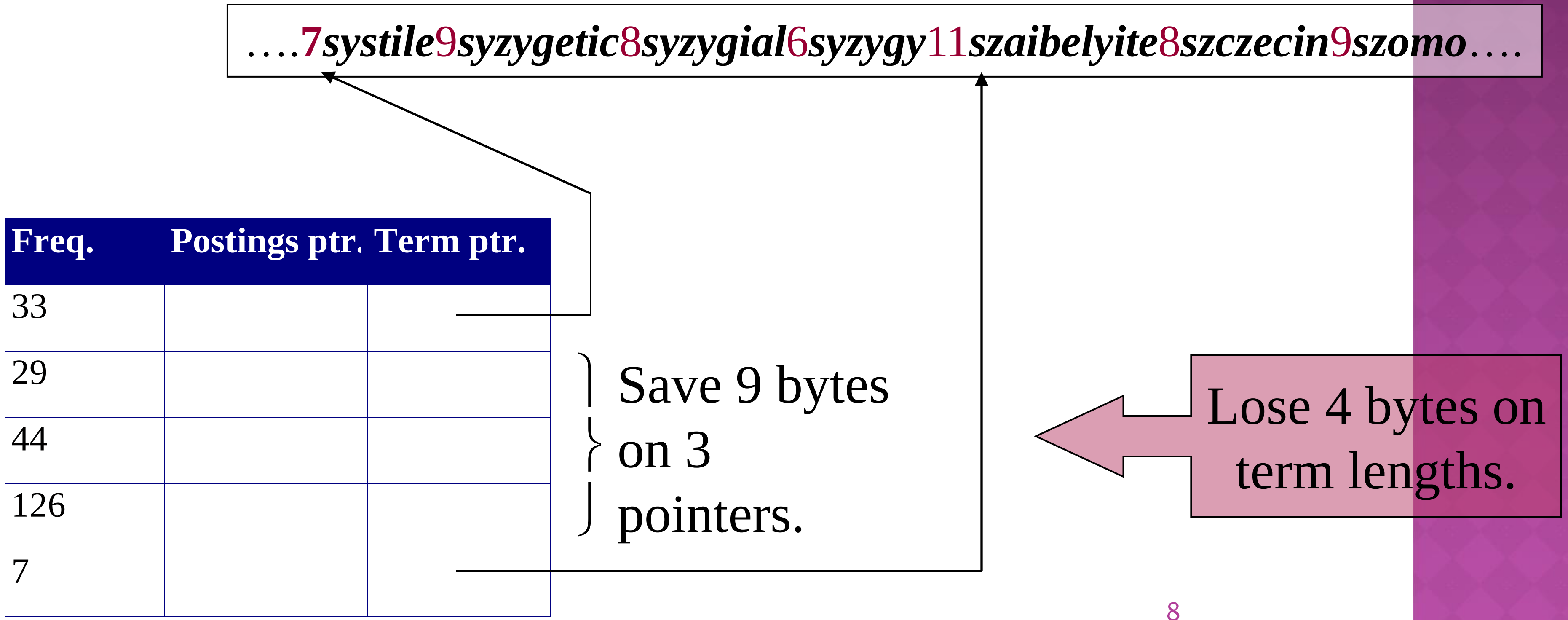
SPACE FOR DICTIONARY AS A STRING

- 4 bytes per term for Freq.
- 4 bytes per term for pointer to Postings.
- 3 bytes per term pointer
- Avg. 8 bytes per term in term string
- 400K terms x 19 ⇒ 7.6 MB (against 11.2MB for fixed width)

$\left.\begin{array}{l} \\ \\ \\ \end{array}\right\}$ Now avg. 11 bytes/term, not 20.

APPROACH 2: BLOCKING

- Store pointers to every *k*-th term string.
  - Example below: *k*=4.
- Need to store term lengths (1 extra byte)



….**7*systile*9*syzygetic*8*syzygial*6*syzygy*11*szaibelyite*8*szczecin*9*szomo***….

Freq.	Postings ptr.	Term ptr.
33		
29		
44		
126		
7		

Save 9 bytes on 3 pointers.

Lose 4 bytes on term lengths.

BLOCKING

- Group terms into blocks, each having k terms
- Store a term pointer only for first term of each block
- Store the length of each term as one additional byte at the beginning of each term
- Search for terms in the compressed dictionary
  - Locate the term’s block by binary search
  - Then locate term’s position within the block by linear search within the block
- By increasing block size k: tradeoff between better compression and speed of term lookup

NET SAVING

- Example for block size *k* = 4
- Where we used 3 bytes/pointer without blocking
  - 3 x 4 = 12 bytes,

now we use 3 + 4 = 7 bytes.

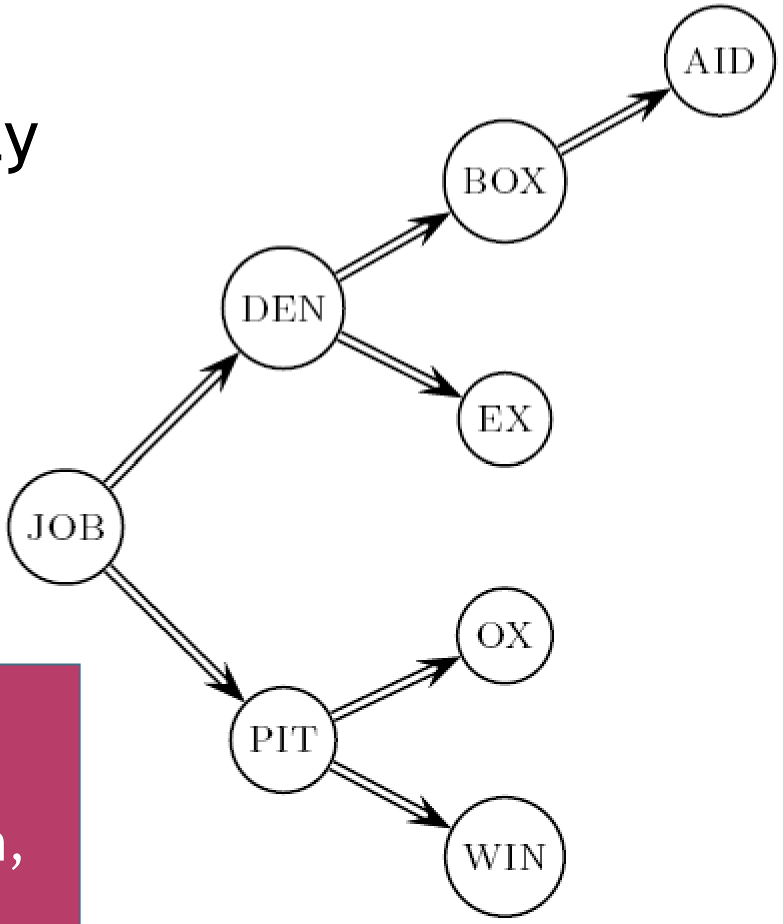
Saved another ~0.5MB. This reduces the size of the dictionary from 7.6 MB to 7.1 MB.
We can save more with larger *k*.

Why not go with larger *k*?

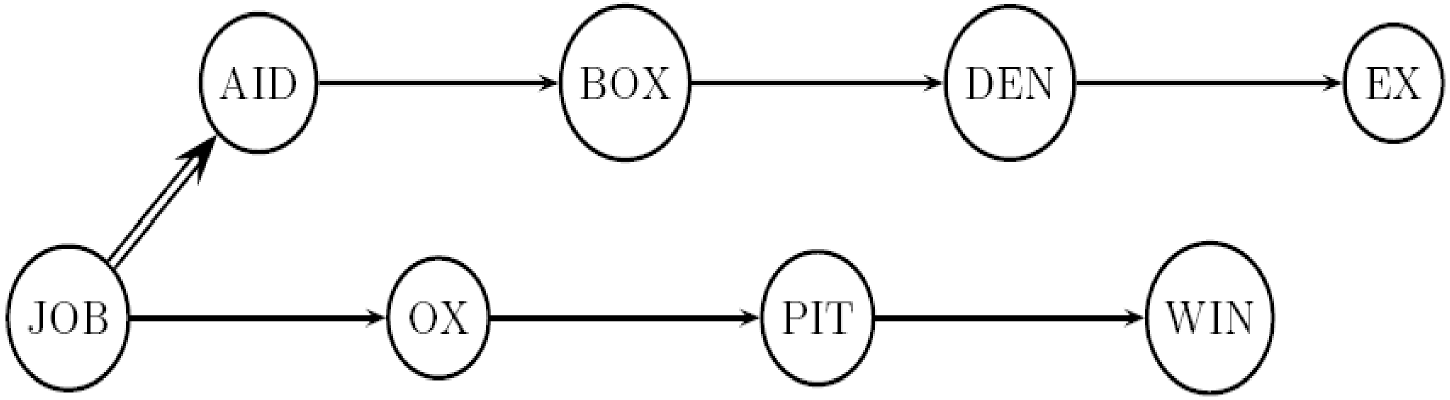
DICTIONARY SEARCH WITHOUT BLOCKING

- Assuming each dictionary term equally likely in query (not really so in practice!), average number of comparisons = (1+2·2+4·3+4)/8 ~2.6

Exercise: what if the frequencies of query terms were non-uniform but known, how would you structure the dictionary search tree?



DICTIONARY SEARCH WITH BLOCKING



- Binary search down to 4-term block;
  - Then linear search through terms in block.
- Blocks of 4 (binary tree), avg. = (1+2·2+2·3+2·4+5)/8 = 3 compares

EXERCISE

- Estimate the impact on search performance (and slowdown compared to *k*=1) with blocking, for block sizes of *k* = *4*, *8* and *16*.