

Part 2: Boolean Retrieval



Francesco Ricci

Most of these slides comes from the course:
Information Retrieval and Web Search,
Christopher Manning and Prabhakar Raghavan

Content

- Term document matrix
- Information needs and evaluation of IR
- Inverted index
- Processing Boolean queries
- The merge algorithm
- Query optimization
- Skip pointers
- Dictionary data structures
 - Hash tables
 - Binary trees

Term-document incidence

	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

Incidence vectors

- So we have a 0/1 vector for each term
- To answer query:
 - ***Brutus, Caesar*** and NOT ***Calpurnia***
 - take the vectors for
 - ***Brutus*** 110100
 - ***Caesar*** 110111
 - ***Calpurnia*** (complemented) 101111
 - Bitwise *AND*
 - 110100 *AND* 110111 *AND* 101111 = 100100

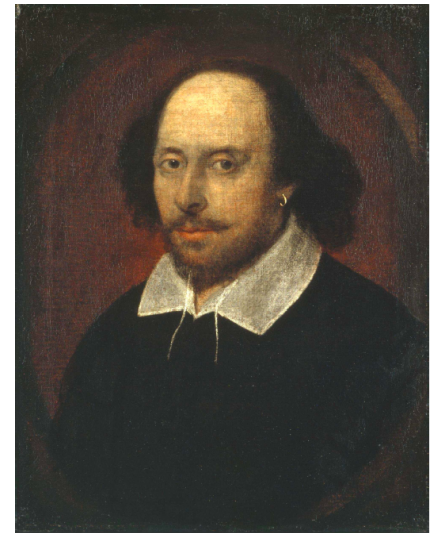
Answers to query

□ Antony and Cleopatra, Act III, Scene ii

Agrippa [Aside to DOMITIUS ENOBARBUS]: Why, Enobarbus,
When Antony found Julius **Caesar** dead,
He cried almost to roaring; and he wept
When at Philippi he found **Brutus** slain.

□ Hamlet, Act III, Scene ii

Lord Polonius: I did enact Julius **Caesar** I was killed i' the
Capitol; **Brutus** killed me.



<http://www.rhymezone.com/shakespeare/>

Basic assumptions of IR

- **Collection:** fixed set of documents
- **Goal:** retrieve documents with information that is relevant to the user's information need and helps the user complete a task
- Using the **Boolean Retrieval Model** means that the information need must be translated into a **Boolean expression:**
 - terms combined with AND, OR, and NOT operators
- We want to support **ad hoc retrieval:** provide documents relevant to an arbitrary user information need.

How good are the retrieved docs?

- *Precision* : Fraction of retrieved docs that are relevant to user's information need
- *Recall* : Fraction of relevant docs in collection that are retrieved
- More precise definitions and measurements to follow in another lecture on evaluation.

Relevance and Retrieved documents

Information need

Ex: male

relevant

not relevant

TP

FP

retrieved

FN

TN

not retrieved

Documents

Query and system

$$\begin{aligned} \text{Precision } P &= \text{tp}/(\text{tp} + \text{fp}) \\ &= \text{tp}/\text{retrieved} \end{aligned}$$

$$\begin{aligned} \text{Recall } R &= \text{tp}/(\text{tp} + \text{fn}) \\ &= \text{tp}/\text{relevant} \end{aligned}$$

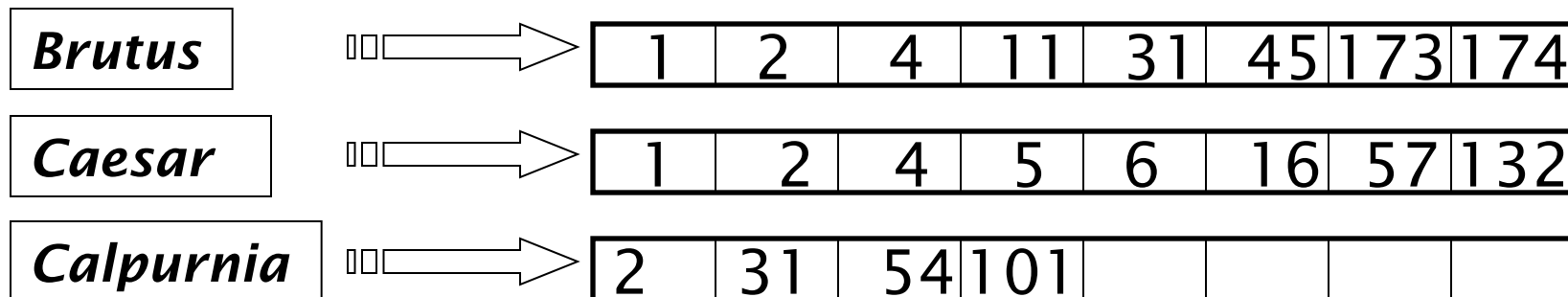
Bigger collections

- Consider a more realistic case
- $N = 1$ 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
- 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
- What's a better representation?
 - We only record the positions of the 1's.



Inverted index

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

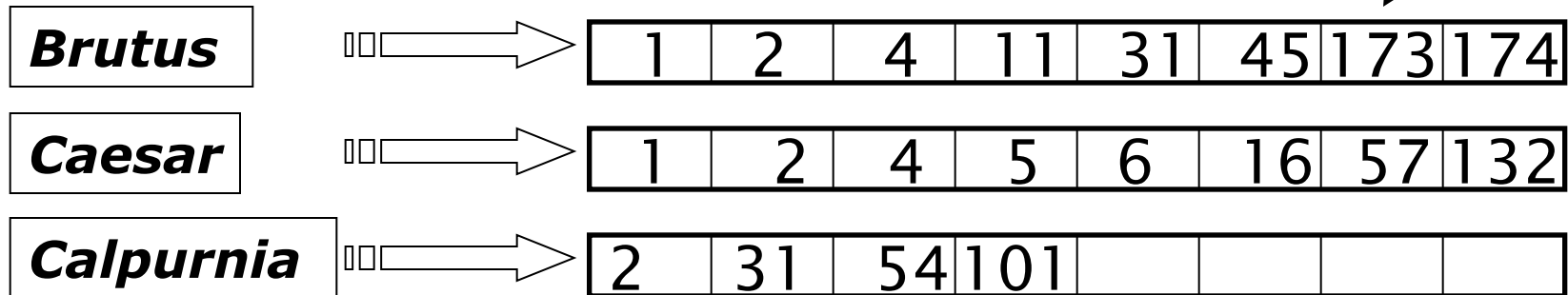


What happens if the word *Caesar* is added to document 14?

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

Posting

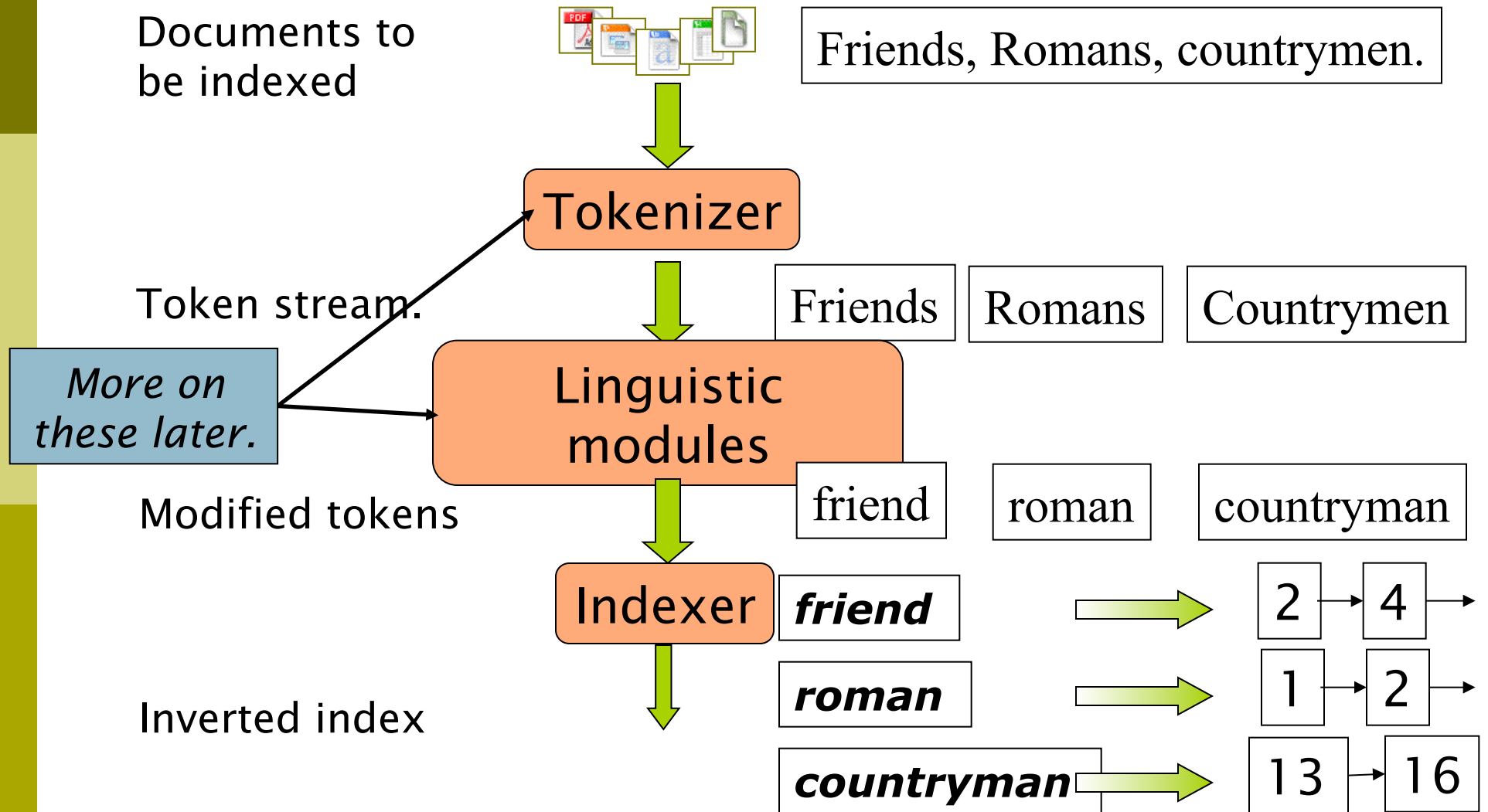


Dictionary

Postings

Sorted by docID (more later on why)

Inverted index construction



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

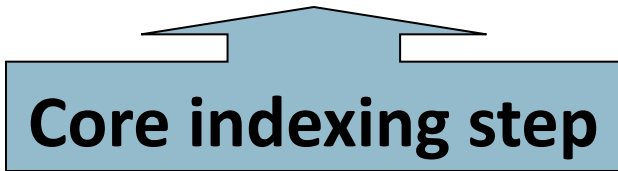
So let it be with
Caesar. The noble
Brutus hath told you
Caesar was ambitious



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



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

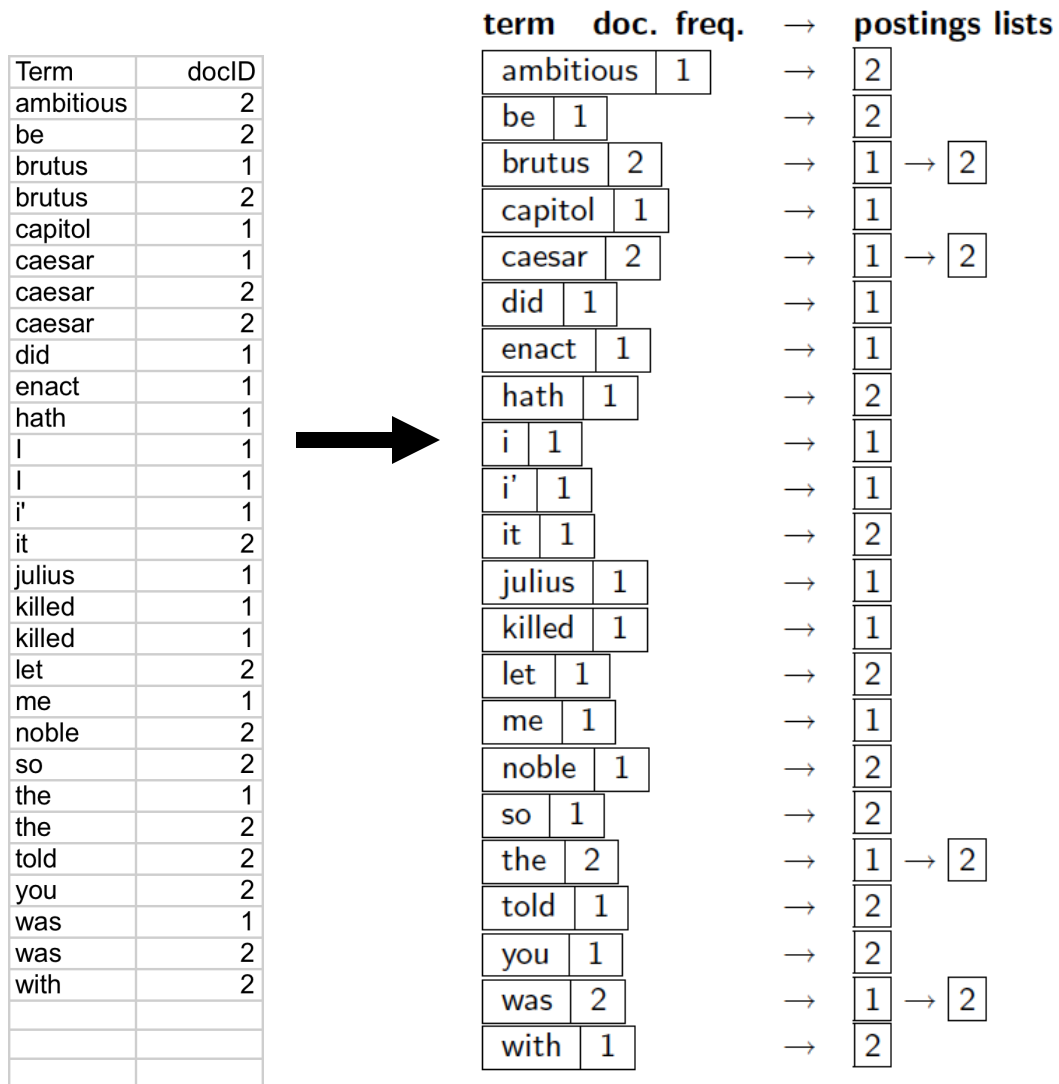


Term	docID
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

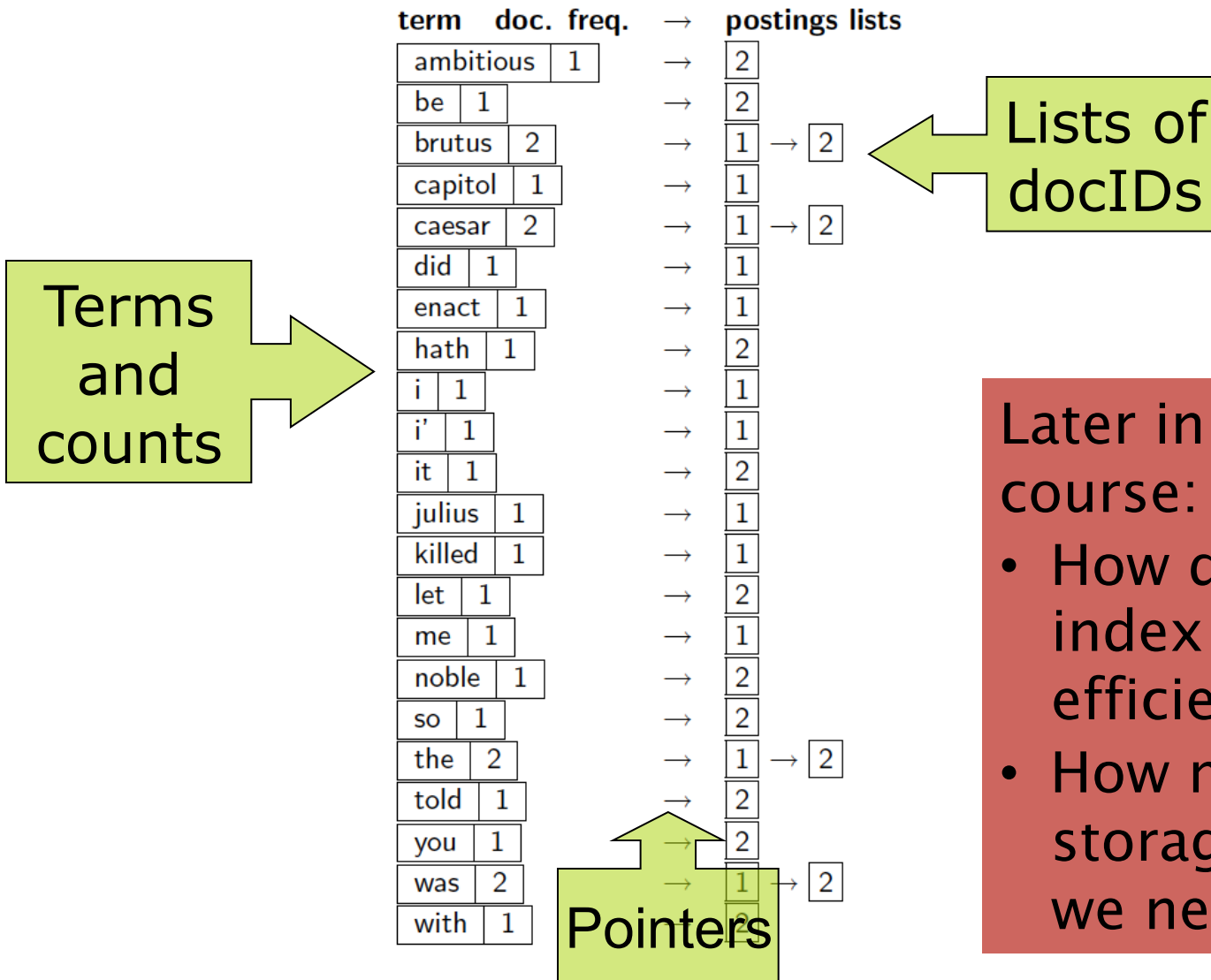
Indexer steps: Dictionary & Postings

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

Why frequency?
Will discuss later



Where do we pay in storage?

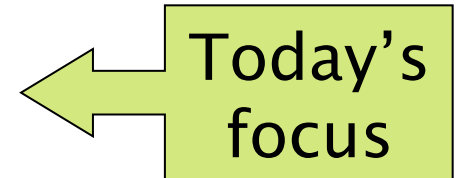


Later in the course:

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

The index we just built

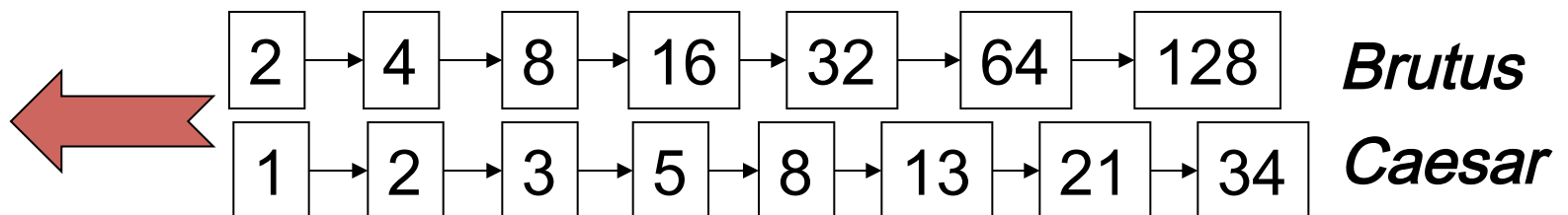
- How do we process a query?
- Later - what kinds of queries can we process?



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

How we can merge?



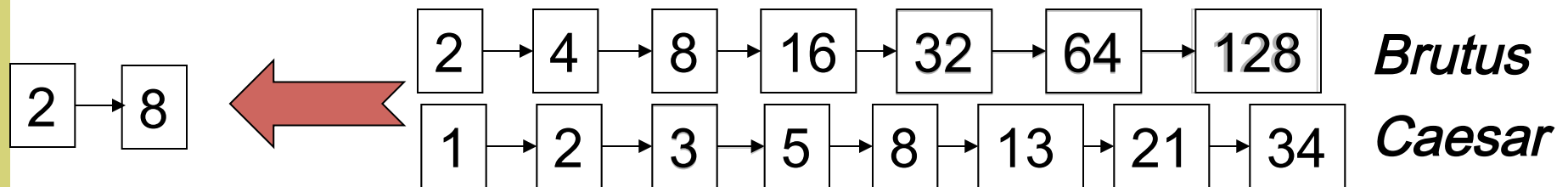
The idea

```
brutus  nn 02 nn 04 nn nn nn nn nn nn nn nn nn nn nn nn 16
cesar   01 02 nn nn 05 nn nn 08 nn nn nn nn 13 nn nn nn
position 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16
```

- If we have the incidence vectors we scan in parallel the entries of the two vectors – starting from the first position (*here I wrote "0x" instead of 1 and "nn" instead of 0*)
- Try to replicate this idea but imagine that in these two arrays you removed the "nn" entries ...
- Advance to the next entry in the smallest position, i.e., to the smallest docID.

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( $p_1, p_2$ )
1   $answer \leftarrow \langle \rangle$ 
2  while  $p_1 \neq \text{NIL}$  and  $p_2 \neq \text{NIL}$ 
3  do if  $docID(p_1) = docID(p_2)$ 
4      then  $\text{ADD}(answer, docID(p_1))$ 
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  $p_1 \leftarrow next(p_1)$ 
9          else  $p_2 \leftarrow next(p_2)$ 
10 return  $answer$ 
```

Boolean queries: Exact match

- The **Boolean retrieval model** is being able to ask a query that is a Boolean expression:
 - Boolean Queries are queries using *AND*, *OR* and *NOT* to join query terms
 - Views each document as a set of words
 - Is precise: document matches condition or not.
 - Perhaps the simplest model to build an IR system on
- Primary commercial retrieval tool for 3 decades
- Many search systems you still use are Boolean:
 - Email, library catalog, Mac OS X Spotlight.

Example: WestLaw <http://www.westlaw.com/>

- ❑ Largest commercial (paying subscribers) legal search service (started 1975; ranking added 1992)
- ❑ Tens of terabytes of data; 700,000 users
- ❑ Majority of users *still* use boolean queries
- ❑ Example query:
 - What is the statute of limitations in cases involving the federal tort claims act?
 - **LIMIT! /3 STATUTE ACTION /S FEDERAL /2 TORT /3 CLAIM**
 - ❑ /3 = within 3 words, /S = in same sentence

More general merges

- Exercise: Adapt the merge for the queries:

Brutus AND NOT Caesar

Brutus OR NOT Caesar

Can we still run through the merge in time $O(x+y)$?

What can we achieve?

Merging

What about an arbitrary Boolean formula?

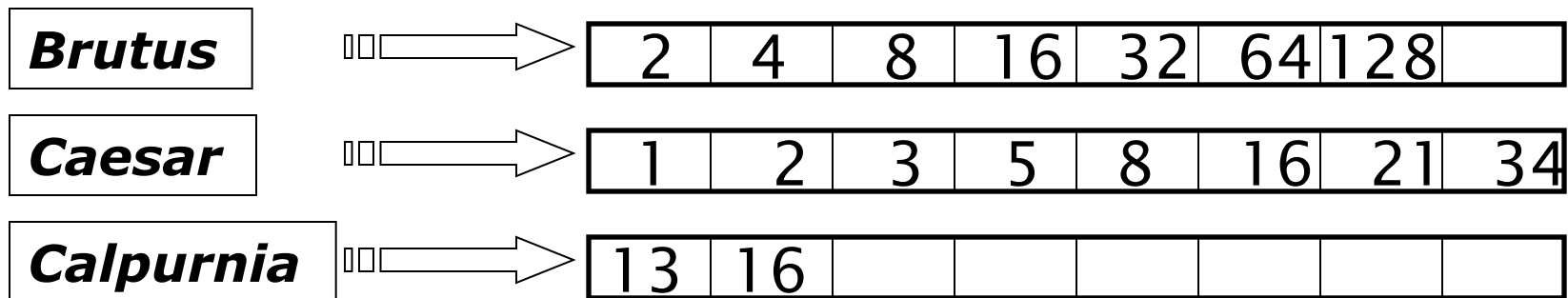
(Brutus OR Caesar) AND NOT

(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



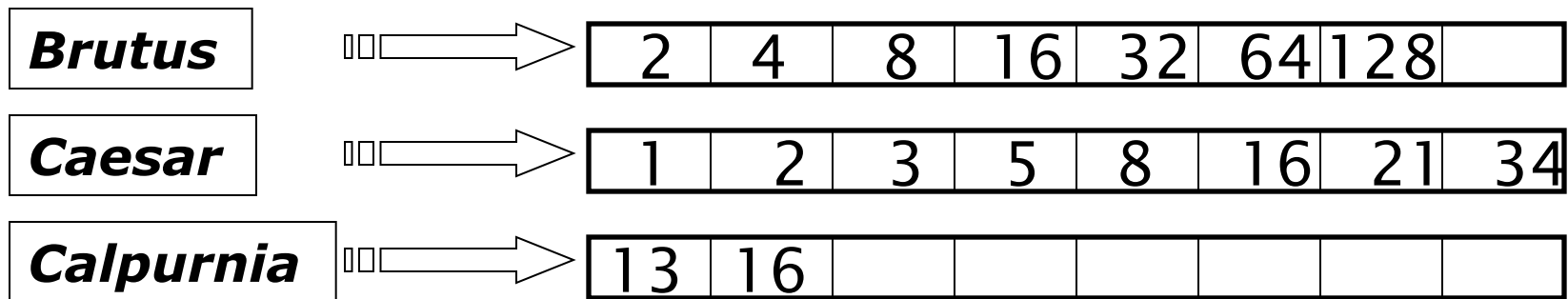
Query: **Brutus AND Calpurnia AND Caesar**

Query optimization example

- Process in order of increasing term freq, i.e., posting list length:
 - *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.

Algorithm for conjunctive queries

INTERSECT($\langle t_1, \dots, t_n \rangle$)

```
1  terms ← SORTBYINCREASINGFREQUENCY( $\langle t_1, \dots, t_n \rangle$ )
2  result ← postings(first(terms))
3  terms ← rest(terms)
4  while terms ≠ NIL and result ≠ NIL
5  do result ← INTERSECT(result, postings(first(terms)))
6     terms ← rest(terms)
7  return result
```

- The intermediate result is in memory
- The list is being intersected with is read from disk
- *The intermediate result is always shorter and shorter*

Exercise

- Recommend a query processing order for

*(tangerine OR trees) AND
(marmalade OR skies) AND
(kaleidoscope OR eyes)*

Term	Freq
eyes	213312
kaleidoscope	87009
marmalade	107913
skies	271658
tangerine	46653
trees	316812

What's ahead in IR? Beyond term search

- What about phrases?
 - ***Stanford University***
- Proximity: Find ***Gates NEAR Microsoft.***
 - Need index to capture position information in docs.
- Zones in documents: Find documents with (*author = ***Ullman****) AND (text contains ***automata***).

Evidence accumulation

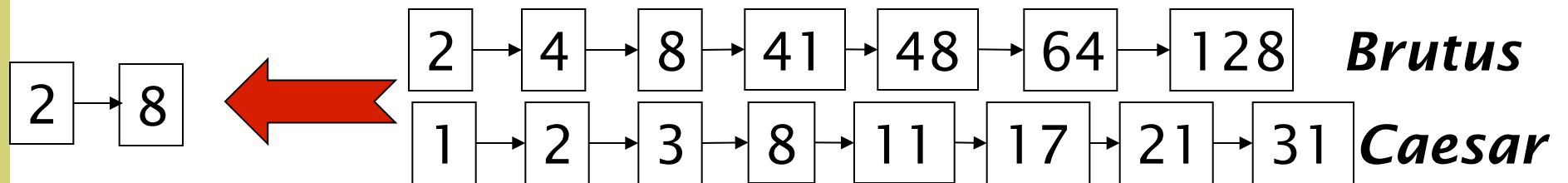
- 1 vs. 0 occurrence of a search term
 - 2 vs. 1 occurrence
 - 3 vs. 2 occurrences, etc.
 - Usually more seems better
- Need term frequency information in docs



FASTER POSTINGS MERGES:
SKIP POINTERS/SKIP LISTS

Recall basic merge

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

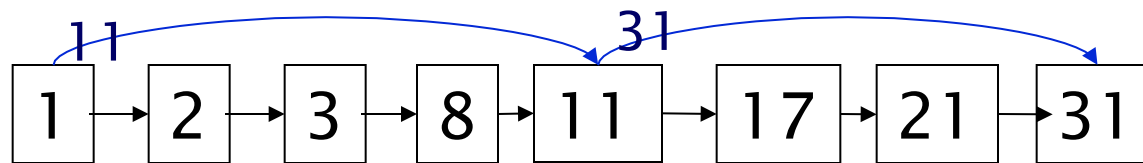
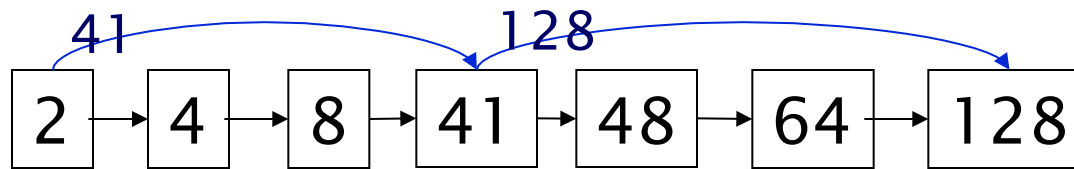


If the list lengths are m and n , the merge takes $O(m+n)$ operations.

Can we do better?

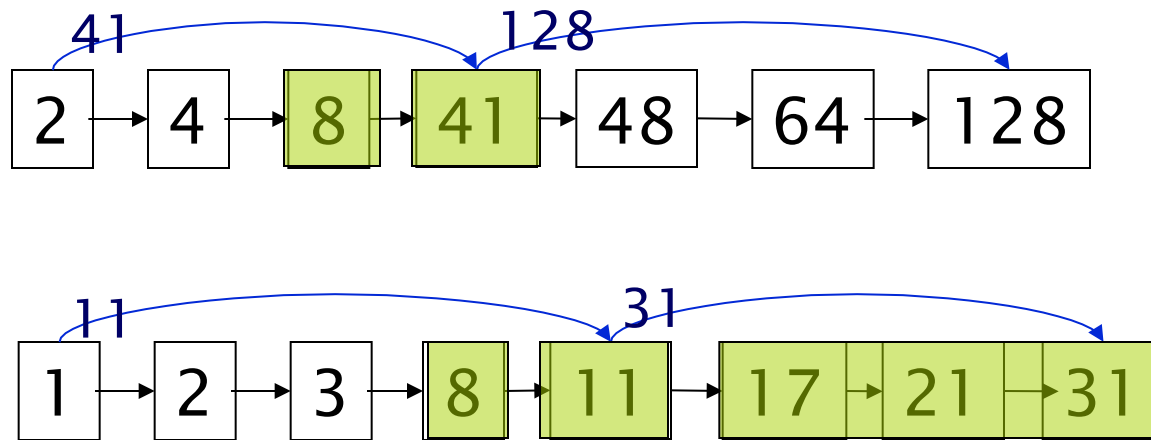
Yes (if index isn't changing too fast).

Augment postings with **skip pointers** (at indexing time)



- Why?
- To skip postings that will not figure in the search results.
- How?
- Where do we place skip pointers?

Query processing with skip pointers



Suppose we've stepped through the lists until we process **8** on each list. We match it and advance.

We then have **41** and **11** on the lower. **11** is smaller.

But instead to advance to **17** the skip successor of **11** on the lower list is **31**, and it is smaller than **41**, so we can skip ahead.

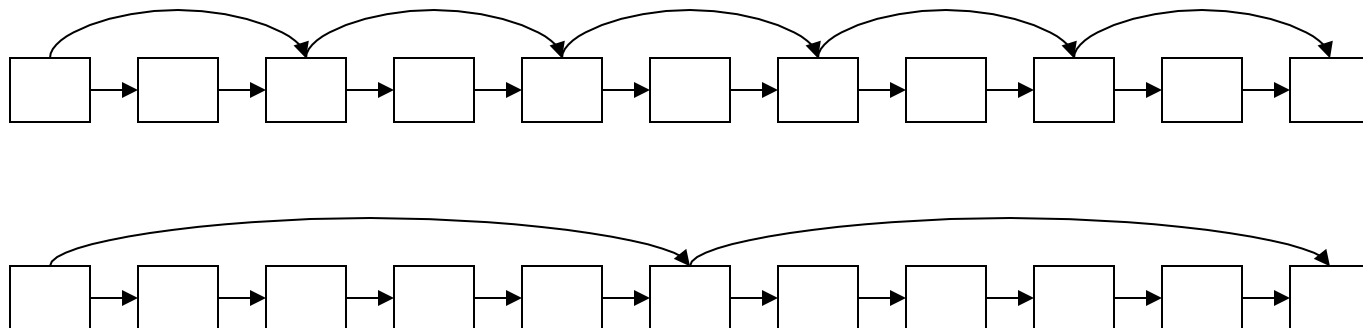
Intersect with skip pointers

INTERSECTWITHSKIPS(p_1, p_2)

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

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.

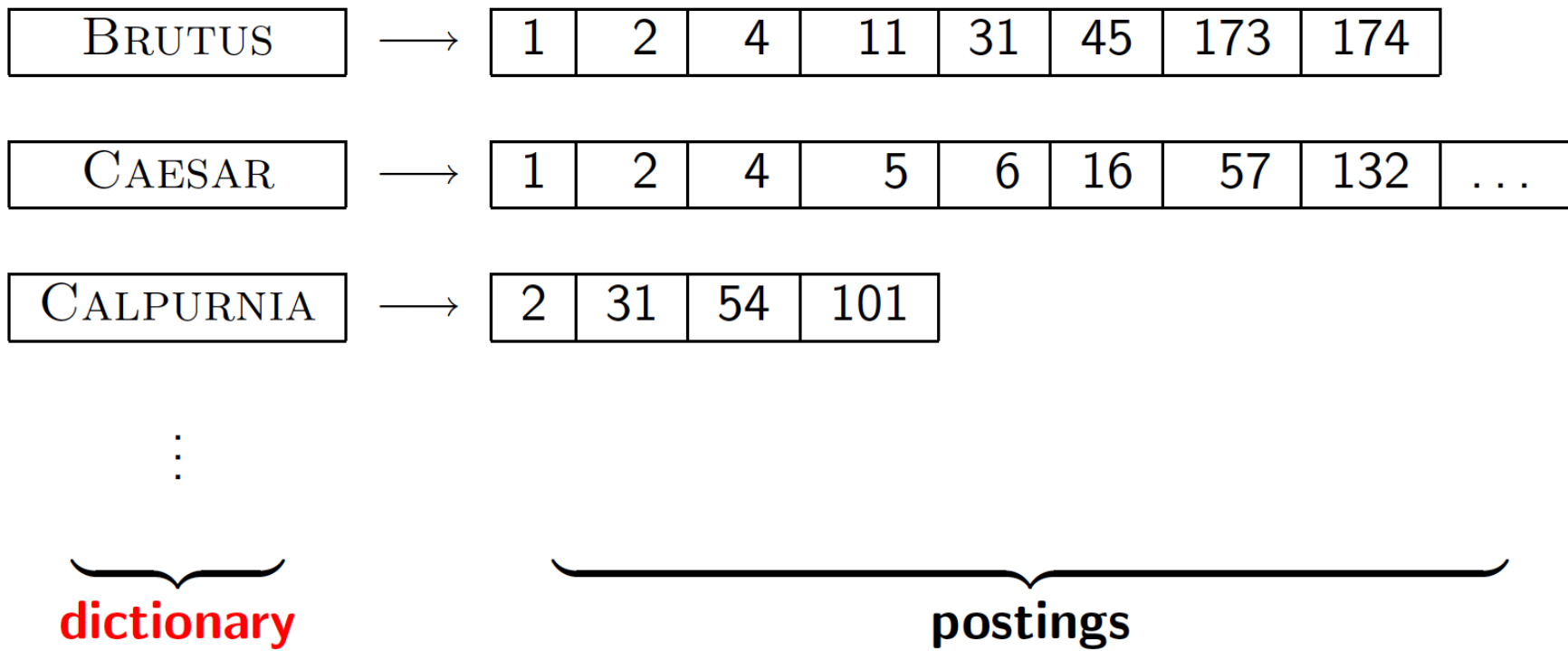


Placing skips

- Simple heuristic: for postings of length L , use \sqrt{L} evenly-spaced skip pointers
- This takes into account the distribution of query terms in a simple way
- Easy if the index is relatively static; harder if L keeps changing because of updates
- This definitely used to help; with modern hardware it may not (Bahle et al. 2002) unless you're memory-based
 - The I/O cost of loading a bigger postings list can outweigh the gains from quicker in memory merging!

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?**



A naïve dictionary

- An array of struct:

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

char[20]

20 bytes

int

4/8 bytes

Postings *

4/8 bytes

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

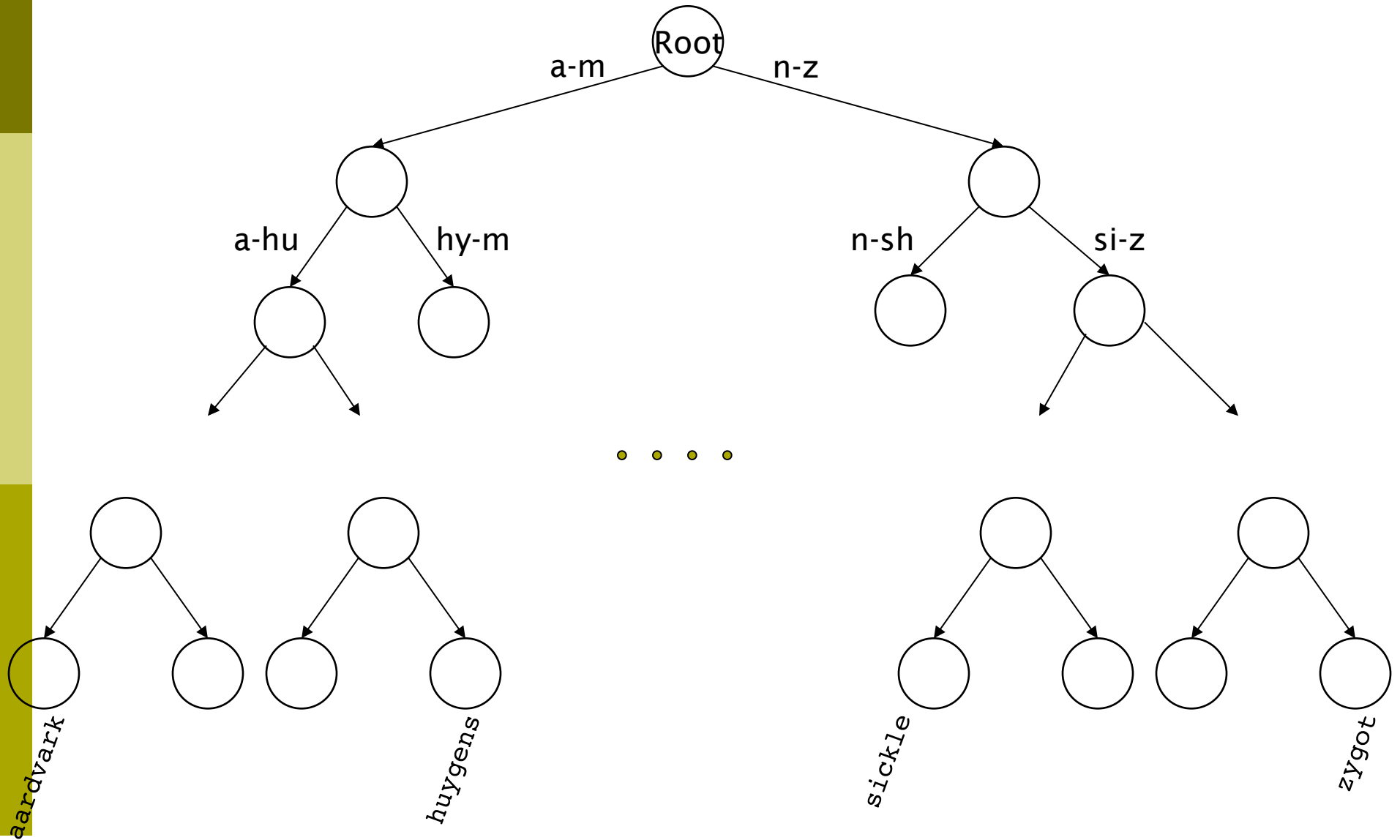
Dictionary data structures

- Two main choices:
 - Hash table
 - Tree
- Some IR systems use hashes, some trees

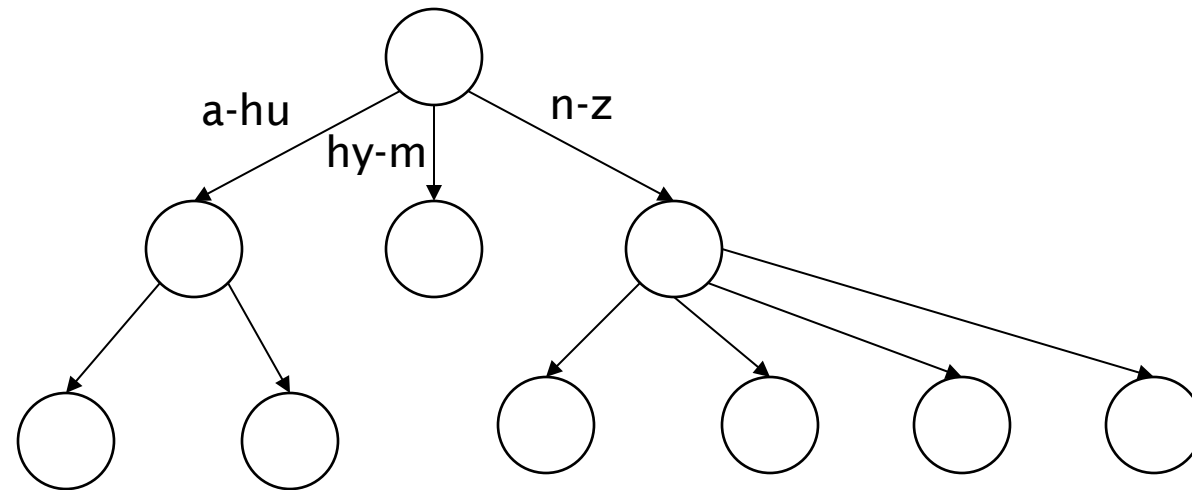
Hashes

- 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 node 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 have one – lexicographic
 - Unless we are dealing with Chinese (no unique ordering)
- *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.

Reading Material

- Chapter 1
- Section 2.3: Faster postings list intersection via skip pointers
- Section 3.1: Search structures for dictionaries