DATA20021

University of Helsinki, Department of Computer Science

Information Retrieval

Lecture 2: IR Jargon + Boolean Retrieval

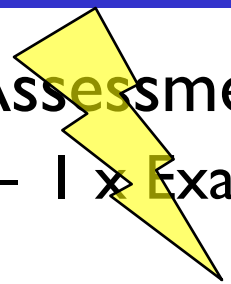
Simon J. Puglisi

puglisi@cs.helsinki.fi

Spring 2020


NewsFlash!

Assessment Structure

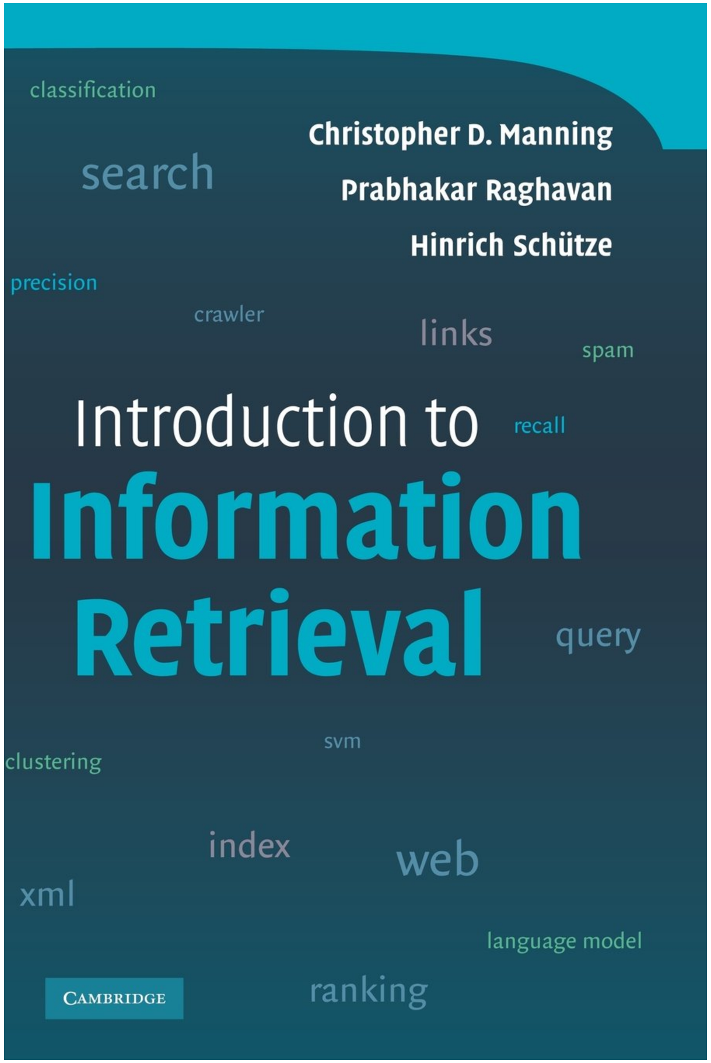


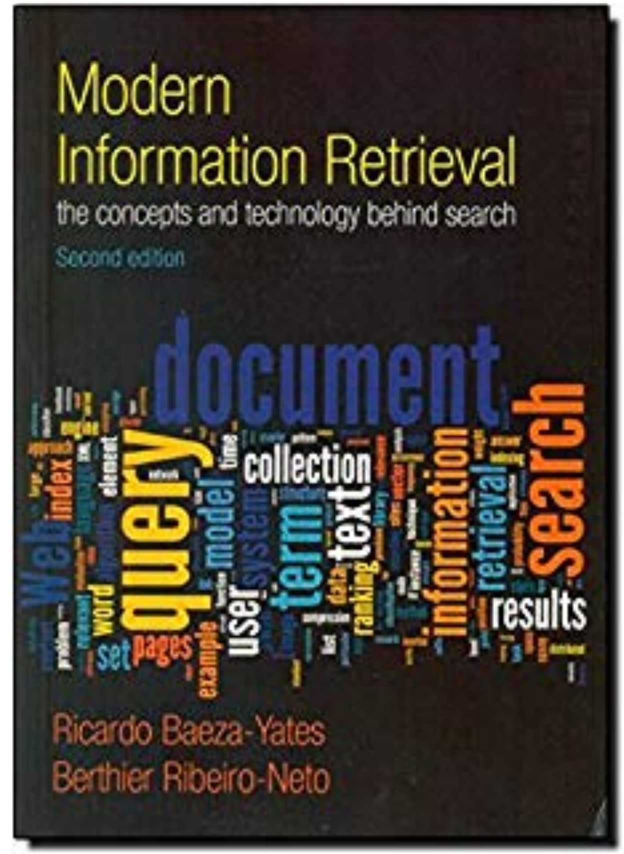
- Assessment:
  - 1 x Exam

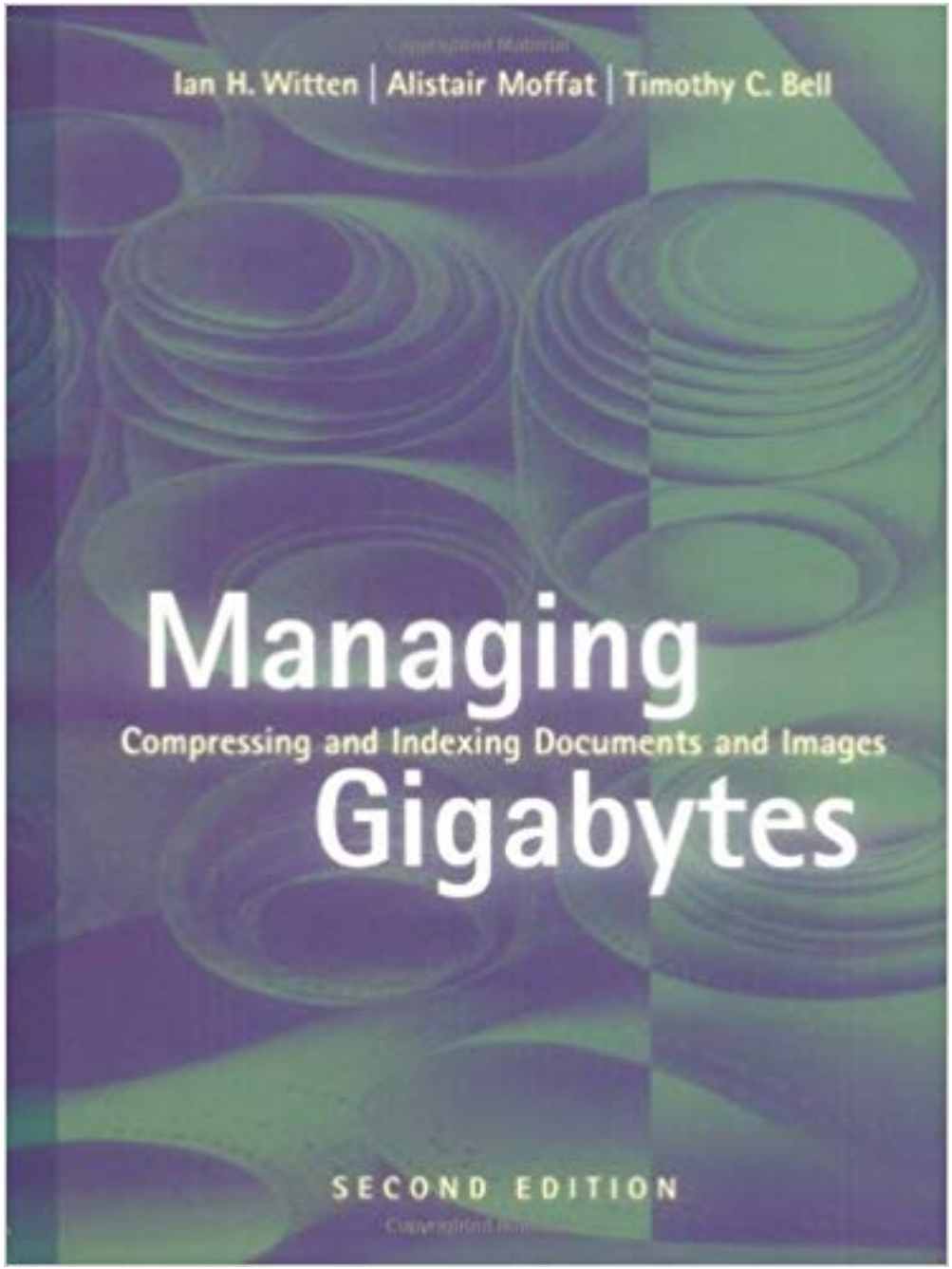
- Assessment:
  - Simon will give you 3 or 4 exercise sets (1 per week)
    - Submit solutions anytime before the exam to gain up to 10 points (out of 60) – recommend you do them week to week
    - The first exercise set available this evening (will email you)
  - Dorota will give you one longer assignment
    - Available in the 2nd half of the course, worth 10 points (out of 60)
  - Exam is worth 40 points out of 60

- 10 (exercises) + 10 (assignment) + 40 (exam) = 60

Books,…


classification
Christopher D. Manning
search
Prabhakar Raghavan
Hinrich Schütze
precision
crawler
links
spam
Introduction to
recall
Information
Retrieval
query
svm
clustering
index
web
xml
language model
CAMBRIDGE
ranking


Modern
Information Retrieval
the concepts and technology behind search
Second edition
document
collection
query
search
text
term
information
retrieval
results
Web
index
model
system
user
pages
set
Ricardo Baeza-Yates
Berthier Ribeiro-Neto


Ian H. Witten | Alistair Moffat | Timothy C. Bell
Managing
Compressing and Indexing Documents and Images
Gigabytes
SECOND EDITION

What you will get out of this course…

- A view under the hood of Search Engines
- A view of the frontiers of Information Retrieval

What you will get out of this course…

- A view under the hood of Search Engines
- A view of the frontiers of Information Retrieval
- Who cares?

What you will get out of this course…

- A view under the hood of Search Engines
- A view of the frontiers of Information Retrieval
- Who cares?
  - Search Engines are one of the wonders of the modern world – as computer scientists you should know how they work
  - Search as a tool is an enabling technology all over CS/IT/DS
  - Many challenging and interesting open problems in IR
  - Web companies are big employers and pay well

The Next Three Weeks

16.1: Introduction to Indexing
- Boolean Retrieval model
- Inverted Indexes

21.1: Index Compression
- variable-byte coding
- (Partitioned) Elias-Fano coding (used inside Google)
- (maybe) BitFunnel (a recent thing out of Microsoft)

23.1: Index Construction
- preprocessing documents prior to search (stemming, et c.)
- building the index efficiently

28.1: Web Crawling
- large scale web search engine architecture

30.1: Query Processing
- scoring and ranking search results
- Vector-Space model

Outline

- What is Information Retrieval?
- Boolean Retrieval model
- The ubiquitous *Inverted Index*

Definition of *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).

– Manning et al, 2008

Another definition of *information retrieval*

Information retrieval deals with the representation, storage, organization of, and access to information items. The representation and organization of items should provide the user with easy access to the information in which [they] are interested.

– Baeza-Yates & Ribeiro-Neto, 1999

And another…

Information retrieval (IR) is the study of systems that manage information items with the aim of helping users satisfy information needs.

It therefore deals with the way information items should be represented, stored, and recalled in response to different information requests.

Challenging Peculiarities

- IR is challenging for two main reasons:
  1. Information needs can be hard to specify
     - What is the [building near the park by the bank in East Melbourne] called?
  2. Relevance is largely subjective
     - I meant Michael [Jordan] not the River [Jordan]
     - That [Burberry + Jacket] is so last season.

Ambiguity of Language

- Language is our key means of communication
  - With it we can express a limitless range of ideas and meanings: language is *generative*.

- However, this broadness leads to complications in IR:
  - Polysemy: a word has more than one meaning, e.g. “jaguar” (or “present” or “plane”)
  - Synonymy: multiple different words express the same content, e.g. do you “save” a file or “write” it? Or “store” it?

Synonymy

- Furnas et al. (1982) ran a series of experiments to examine the vocabulary mismatch problem, requiring participants to
  - Specify descriptions of verbal objects
  - Prepare instructions for the carrying out of various tasks.
  - Create keyword descriptions of cooking recipes
  - Categorise objects hierarchically

- They found the chance of two people using the same main content word to describe a subject is only 10-20%

Challenging Peculiarities

- IR is challenging for two main reasons:
  1. Information needs can be hard to specify
     - What is the [building near the park by the bank in East Melbourne] called?
  2. Relevance is largely subjective
     - I meant Michael [Jordan] not the River [Jordan]
     - That [Burberry + Jacket] is so last season.

Answers versus Relevance in IR

- We could define an *answer* to a query as a document that matches the query according to some formal criteria: if it contains all the keywords (for instance)
- However, this does not mean the document is a helpful response for that particular information need…
- What we really want are documents that contain the information we are seeking
  - *We want relevant documents.*

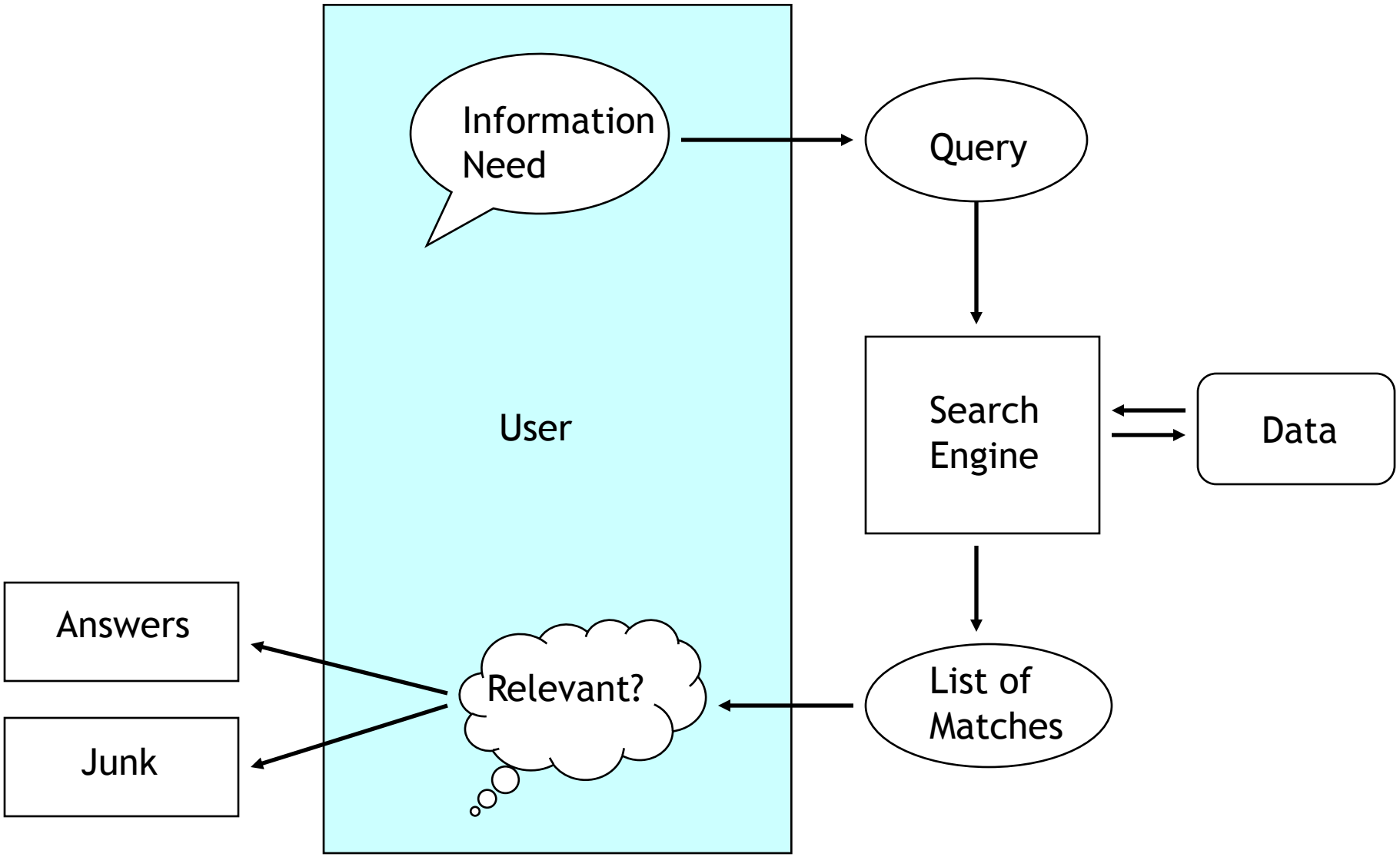
Relevance

- Relevance can be defined as follows:

  A document is relevant if it contains knowledge that helps the user to resolve their information need

- This definition implies answers do not need not contain *all* the required information – a document that helps is also relevant

Search Process as viewed by an IR researcher…



And also…

- IR is challenging because of the enormous amounts of data/information involved:
  - 16 trillion pages on the web (as of 2016)
  - ~5 billion of these are indexed (maybe)
  - Terabytes (Petabytes?) of information
  - It changes all the time
  - Users want answers fast (…*good* answers fast)
  - Clever, highly efficient algorithms and data structures required

Flavours of IR System

- Web Search (we all use this all the time)
- Site-based searching (Enterprise Search)
  - wikipedia.org, www.helsinki.fi, www.amazon.com, …
  - Many organisations with a web presence provide some kind of site-search at their homepage.
- File System Search, Help Utilities
  - Facilities like OSX spotlight and Microsoft Word online help are search engines searching over a small data collection
- Searching with intermediaries
  - Phone help-desks rely on search engines to access databases of problems and solutions - operators must interpret problems into queries to a search engine.

(Desirable) Characteristics of IR Systems

- Robust in the face of rich, complex data
  - Data on the web is ugly: poor authorship, malformed HTML, multiple files formats, binary data
- Tolerant of searcher input error
  - Queries by most users are fairly primitive, contain spelling errors – just about anything is a valid query
- Able to produce useful output
  - This is an informal criteria as results are subjective

Measuring Performance in IR Systems

- *Efficiency* is an important measure of performance: how fast can answers be determined, and with what resources (memory, disk, …)

- *Quality of answers* is also important. It is easy to fetch bad answers

IR Effectiveness

- An IR system is said to be *effective* if it is good at finding relevant documents in response to queries
- Effectiveness is relative. Some queries are difficult to resolve, others may have many relevant documents
- Effectiveness is measured by counting the number of relevant documents retrieved across a set of queries

Retrieval Effectiveness

- Many (many) methods for measuring retrieval effectiveness have been proposed
  - See later lectures by Dorota

- Two key measures are:

$$recall = \frac{(\#relevant\ \ docs\ \ retrieved)}{(\#relevant\ \ docs)}$$

$$precision = \frac{(\#relevant\ \ docs\ \ retrieved)}{(\#retrieved\ \ docs)}$$

Inverted Indexes,…

Unstructured data in 1650: Shakespeare

Unstructured data in 1650

- Which plays of Shakespeare contain the words BRUTUS and CAESAR, but NOT CALPURNIA?
- We could `grep` all of Shakespeare's plays for BRUTUS and CAESAR,

```
grep 'brutus\|caesar' Othello.txt
```

then `grep` the matching files again and throw out the ones containing CALPURNIA.

- Why is `grep` not the solution?
  - Slow (for large collections)
  - "NOT CALPURNIA" is non-trivial (well… it almost is)
  - Other operations (e.g. find the word ROMANS "near" COUNTRYMAN) not necessarily feasible

Term-document incidence matrix

	Anthony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth	…
ANTHONY	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	
…							

- Entry is 1 if term occurs. Eg: CALPURNIA occurs in *Julius Caesar*.
- Entry is 0 if term doesn't occur. Eg: CALPURNIA doesn't occur in *The Tempest*.

Incidence vectors

- So we have a 0/1 (binary) vector for each term.
- To answer the query BRUTUS AND CAESAR AND NOT CALPURNIA:
  - Take the vectors for BRUTUS, CAESAR, and CALPURNIA
  - Complement the vector of CALPURNIA
  - Do a (bitwise) AND on the three vectors
  - 110100 AND 110111 AND 101111 = 100100

0/1 Vector for Brutus

	Anthony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth	…
Anthony	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	
…							

Answers to Query

Anthony 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.

Bigger collections

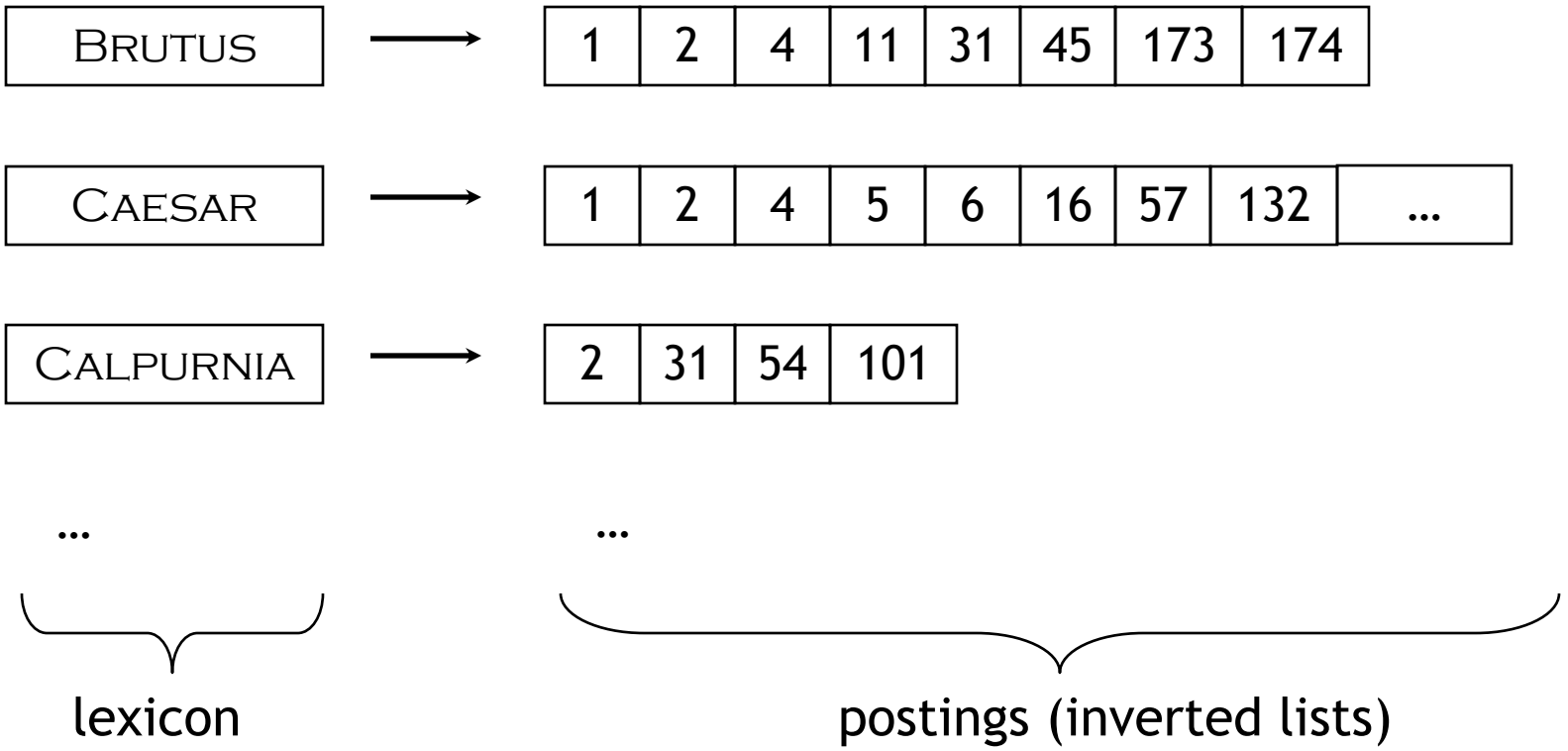
- Consider N = 10^7 documents, each with about 1000 words
- On average 6 bytes per word, including spaces and punctuation → size of document collection is about 6 GB
- Assume there are M = 500,000 distinct terms in the collection

Can’t build the incidence matrix…

- M = 500,000 × 10^7 = 5 *trillion* 0s and 1s.
- But the matrix has no more than one billion 1s.
  - Matrix is extremely sparse (mostly 0s).
- What is a better representation?
  - We only record the positions of the 1s.

Inverted Index

- For each term *t*, we store a list of all documents that contain *t*:



Inverted Index Construction

1. Collect the documents to be indexed:

Friends, Romans, contrymen.	So let it be with Caesar.	…

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

Friends	Romans	countrymen	So	…

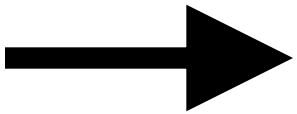
3. Do linguistic processing, producing a list of normalized words, which are the indexing terms:

friend	roman	countryman	so	…

4. Index the documents that each term occurs in by creating an inverted index, consisting of an lexicon and a set of postings lists (or inverted lists)

Tokenization and preprocessing

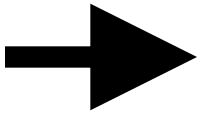
- **Doc 1.** I did enact Julius Caesar: I was killed i' the Capitol; Brutus killed me.
- **Doc 2.** So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious:



- **Doc 1.** I did enact julius caesar I was killed in the capitol brutus killed me
- **Doc 2.** so let it be with caesar the noble brutus hath told you caesar was ambitious

Generate postings

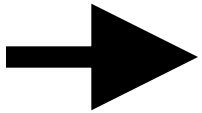
- **Doc 1.** I did enact julius caesar I was killed in the capitol brutus killed me
- **Doc 2.** so let it be with caesar the noble brutus hath told you caesar was ambitious



term	docID
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
in	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

Sort postings

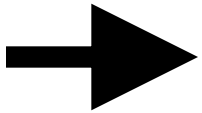
term	docID
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
in	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2



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
in	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

Create lists, count document frequency

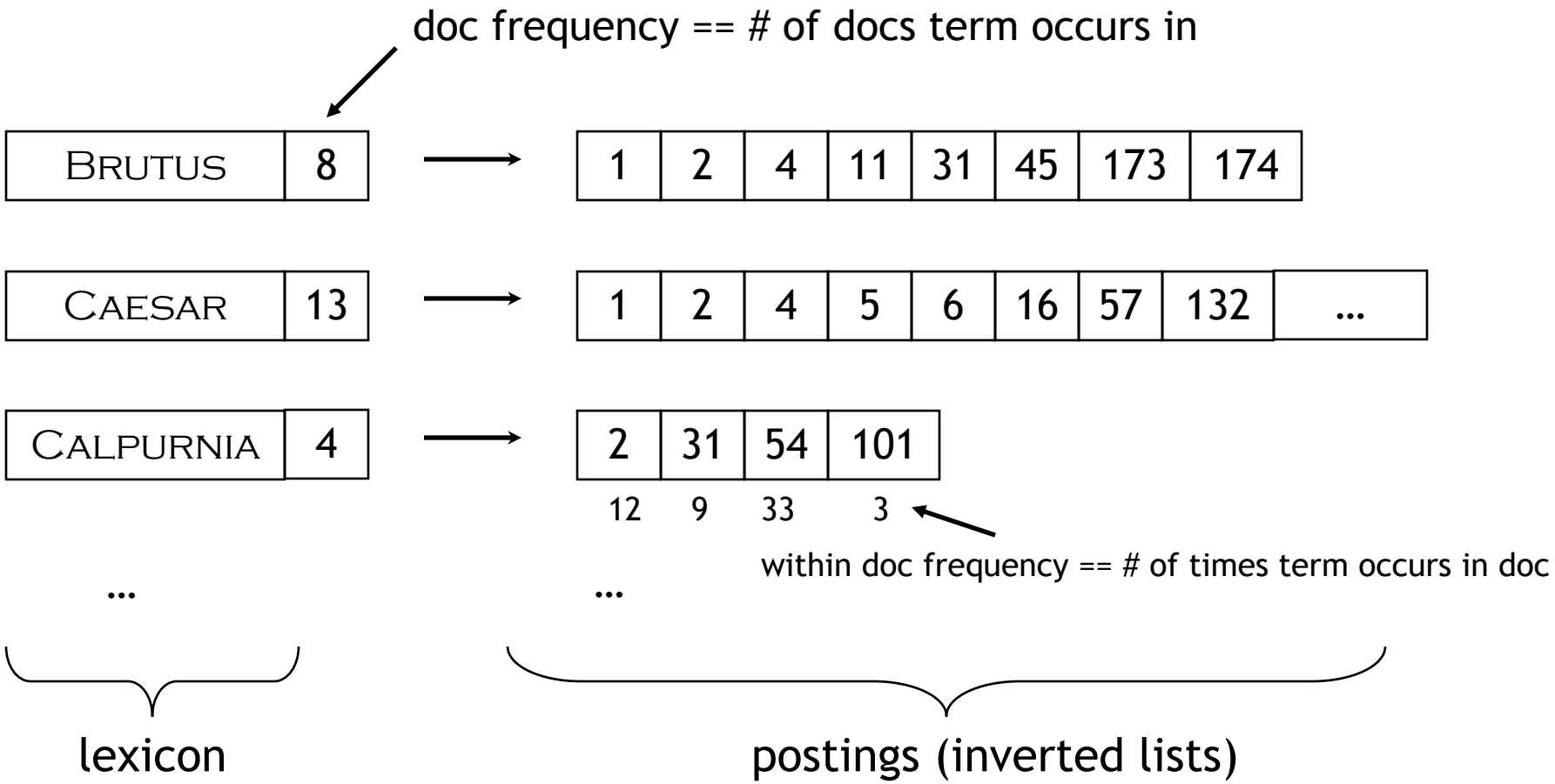
term	docID
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
in	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2



term	doc freq	inv list
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	1
I	2	1
in	1	1
it	1	2
julius	1	1
killed	2	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

Split the result into a lexicon and a postings file

- For each term *t*, we store a list of all documents that contain *t*:



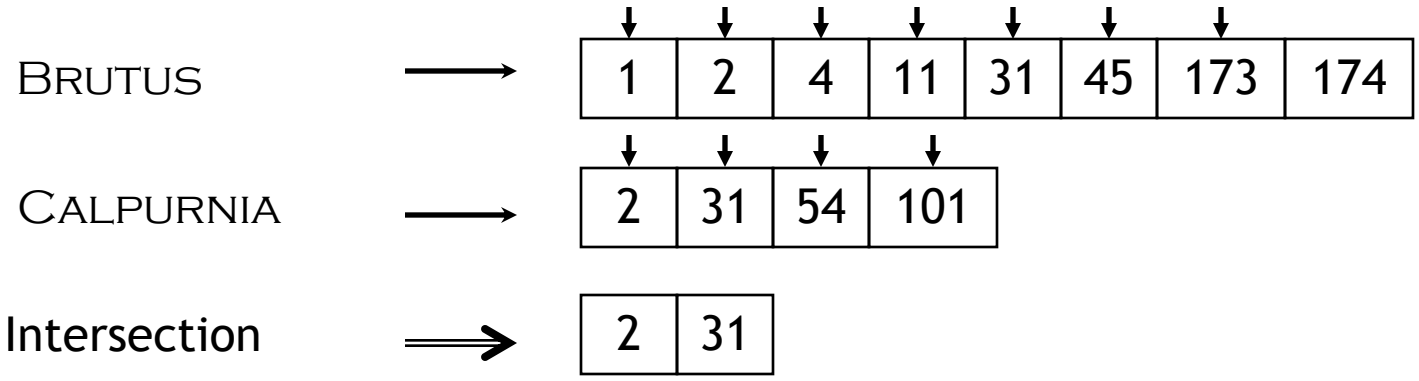
Processing Boolean Queries, …

Simple Conjunctive Query (two terms)

- Consider the query: BRUTUS AND CALPURNIA

- To find all matching documents using inverted index:
  1. Locate BRUTUS in the lexicon
  2. Retrieve its inverted list from file
  3. Locate CALPURNIA in the lexicon
  4. Retrieve its inverted list from file
  5. Intersect the two inverted lists
  6. Return intersection to user

Simple Conjunctive Query (two terms)



- This is linear in the length of the postings lists: O(p+q).
- This only works if postings lists are sorted.

Simple Conjunctive Query (two terms)

```
INTERSECT(p, q)
1   answer ← {}
2   while p ≠ null and q ≠ null do
3         if docID(p) = docID(q) then
4               Add(answer , docID(p))
5               p ← next(p)
6               q ← next(q)
7         else if docID(p) < docID(q) then
8               p ← next(p)
9         else
10              q ← next(q)
11  end while
12  return answer
```

Boolean Queries

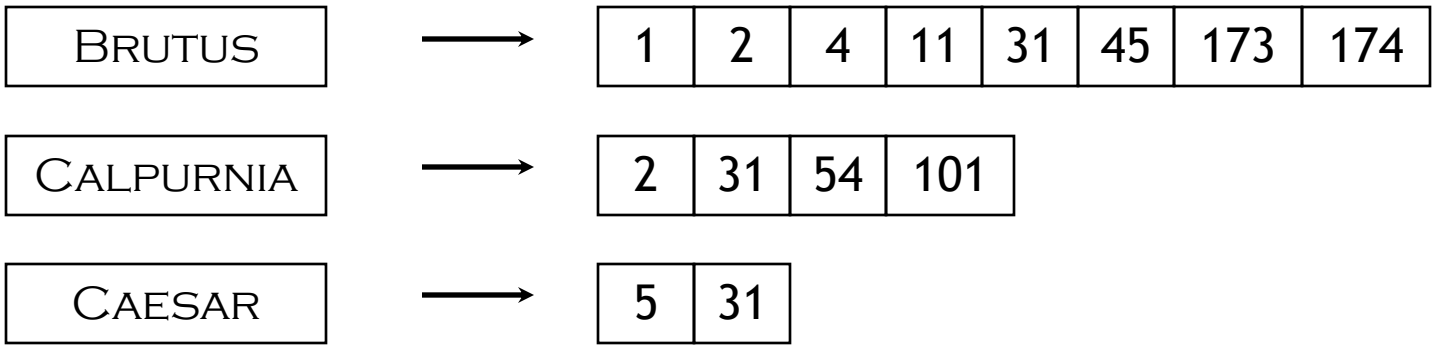
- The Boolean retrieval model can answer any query that is a Boolean expression.
  - Boolean queries use AND, OR and NOT to join query terms.
  - Views each document as a set of terms.
  - Is precise: Document matches condition or not.
- Primary commercial retrieval tool for 3 decades
- Many professional searchers (e.g., lawyers) still like Boolean queries
  - You know exactly what you are getting
- Many search systems you use are also Boolean: email, Flamma (?), etc.
- Google lets you use Boolean operators (“must contain …”)

Query Optimization

- What is the best order for query processing?
- Consider a query that is an AND of n terms, $n > 2$
- For each of the terms, get its postings list, then AND them together
- Example query: BRUTUS AND CALPURNIA AND CAESAR

Query Optimization

- Example query: BRUTUS AND CALPURNIA AND CAESAR
- Simple and effective optimization: Process in order of increasing frequency
- Start with the shortest postings list, then keep cutting further
- In this example, first CAESAR, then CALPURNIA, then BRUTUS



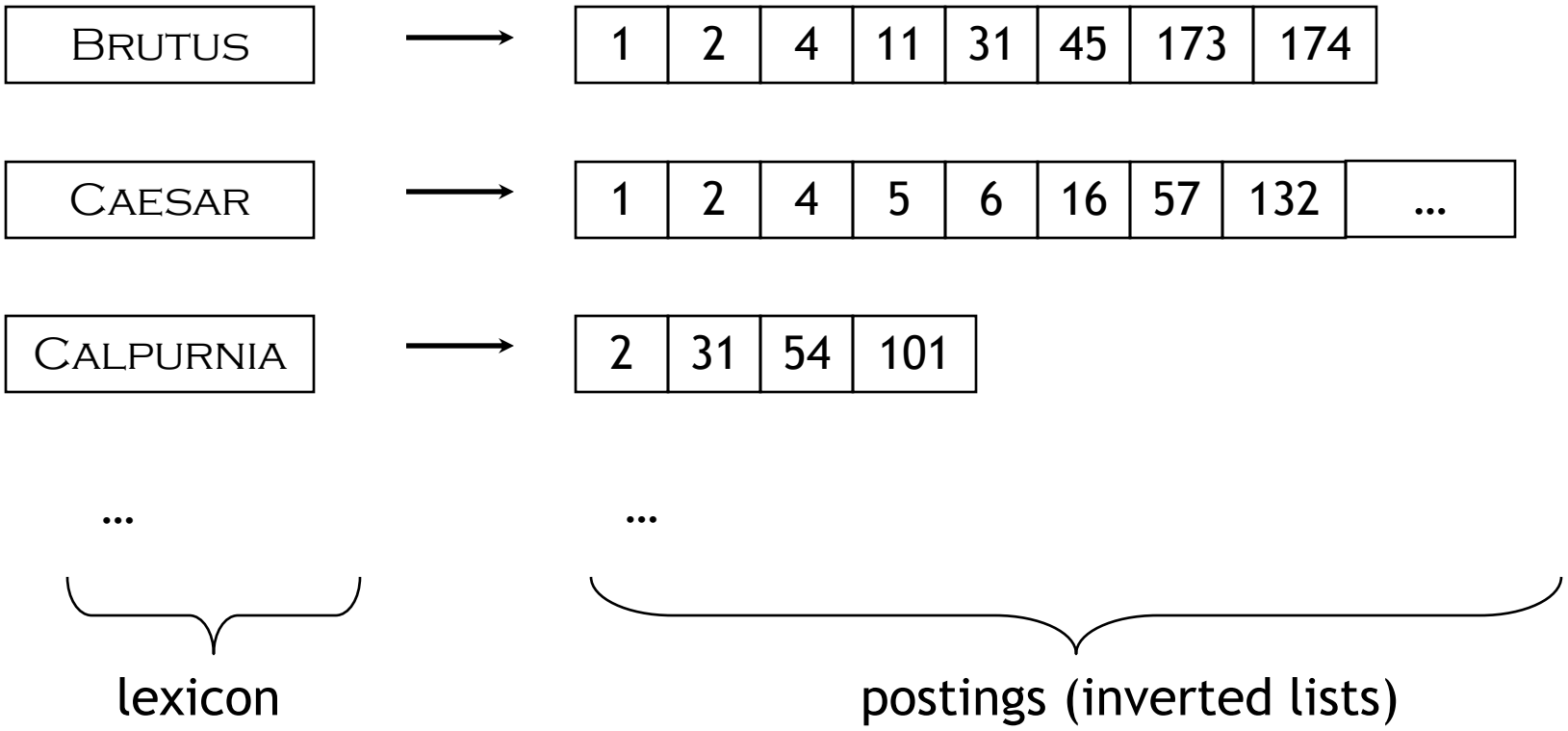
Optimized intersection for conjunctive queries

```
INTERSECT({t_1, . . . , t_n})
1    terms ← SORTBYINCREASINGFREQUENCY({t_1, . . . , t_n})
2    result ← postings(first(terms))
3    terms ← rest(terms)
4    while terms ≠ null and result ≠ null do
5         result ← INTERSECT(result, postings(first(terms)))
6         terms ← rest(terms)
7    end while
8    return result
```

The Lexicon,…

Our friend the Inverted Index

- For each term *t*, we store a list of all documents that contain *t*:



Lexicon

- The lexicon is the data structure for storing the term vocabulary
- Term vocabulary: the data
- Lexicon: the data structure for storing the term vocabulary

Lexicon as an array of fixed width entries

- For each term, *t*, we need to store a couple of items:
  - document frequency (# of documents containing *t*)
  - pointer to postings list for *t*
  - . . .
- Assume for the time being that we can store this information in a fixed-length entry
- Assume that we store these entries in an array

Lexicon as an array of fixed width entries

	term	document frequency	pointer to postings list
	a	656,265	→
	aardvark	65	→
	…	…	…
	zulu	221	→
space needed	20 bytes	4 bytes	4 bytes

Lexicon as an array of fixed width entries

	term	document frequency	pointer to postings list
	a	656,265	→
	aardvark	65	→
	…	…	…
	zulu	221	→
space needed	20 bytes	4 bytes	4 bytes

- How do we look up a word in this array at query time?

Lexicon as an array of fixed width entries

	term	document frequency	pointer to postings list
	a	656,265	→
	aardvark	65	→
	…	…	…
	zulu	221	→
space needed	20 bytes	4 bytes	4 bytes

- How do we look up a word in this array at query time?
  - Binary search – it's just a sorted array
  - O(|t|log|L|) time, where |t| term length and |L| vocab. size

Faster Lexicons

- Our two main options are *hash tables* and *tries*
  - Best choice depends on a few factors (TBD)
- Both these options allow us to lookup t in O(|t|) time (or very close to it)
- Both assume (somewhat) that the lexicon fits in RAM
  - If this isn't the case (and a large part of the lexicon has to reside on disk) we have other options like B-trees

Hash Tables

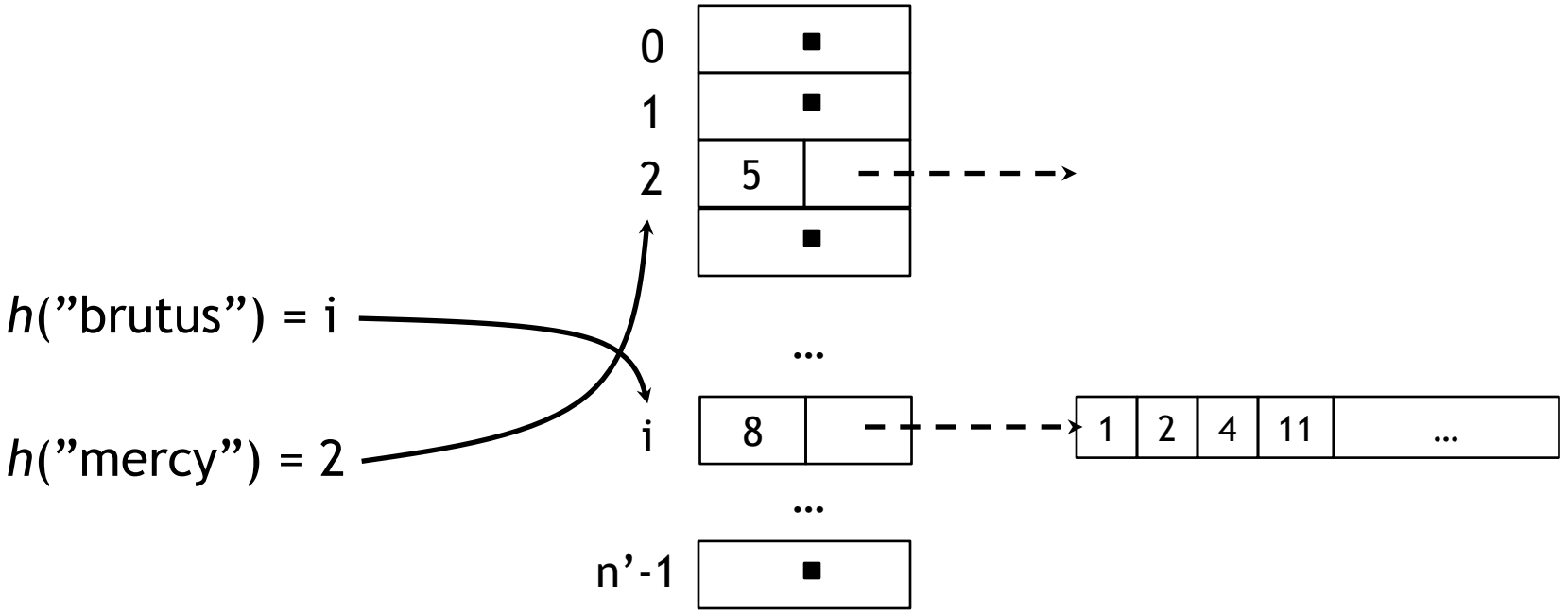
- A *hash table* H[0..n'-1] is (essentially) an array of fixed size n'
- A *hash function* h(t) maps a (string) term t to an integer in a desired range 0..n'-1
- Here is the hash function Java uses for String objects:

```
public int hashCode() {
    int h = hash;
    if (h == 0 && value.length > 0) {
        char val[] = value;

        for (int i = 0; i < value.length; i++) {
            h = 31 * h + val[i];
        }
        hash = h;
    }
    return h;
}
```

Hash Tables

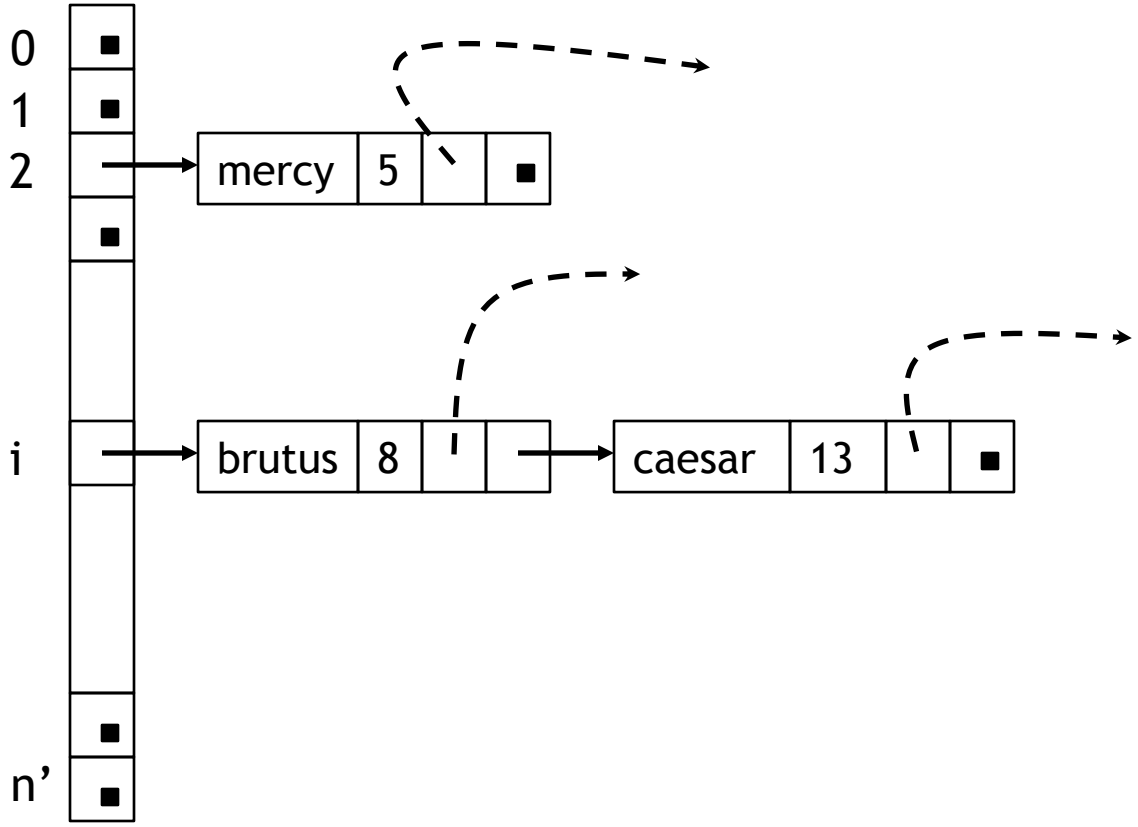
- A *hash table* H[0..n'-1] is (essentially) an array of fixed size n'
- A *hash function* h(t) maps a (string) term t to an integer in a desired range 0..n'-1
- A hash table uses a hash function to map a term into an index (entry) in its array: at H[h(t)] we put a pointer to the postings list for term t (and any other data we need, like document frequency)



Hash Tables: collisions

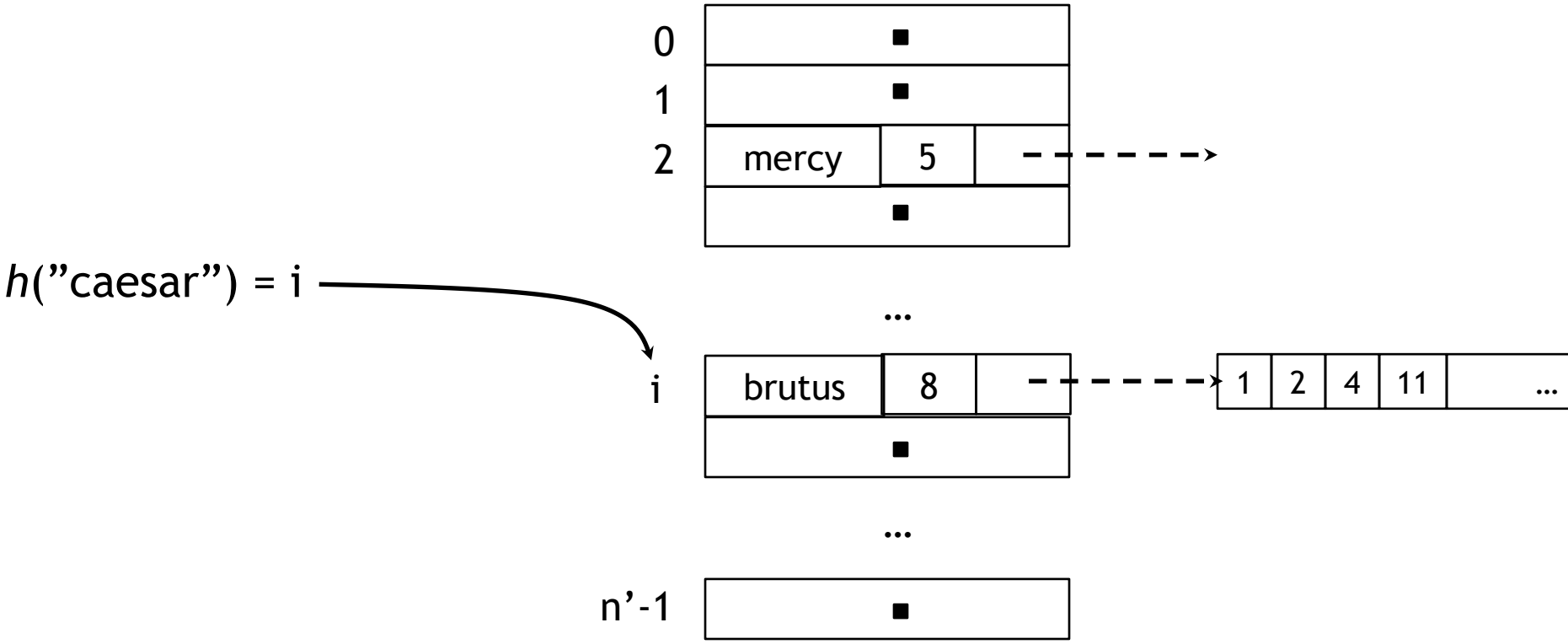
- Ideally, any two different terms $t_1 \neq t_2$ will get mapped to different integers by the hash function $h(t_1) \neq h(t_2)$
- When this is *not* the case (i.e. $t_1 \neq t_2$ but $h(t_1) == h(t_2)$) we say we have a *collision*.
  - Provided n' is big enough relative to |L|, the number of distinct terms we need to hash, a good hash function will produce relatively few collisions
- The need to resolve collisions is a problem with hash tables in general (not just when they're being used as a lexicon)
- We have a few options…

Collision Resolution via Chaining



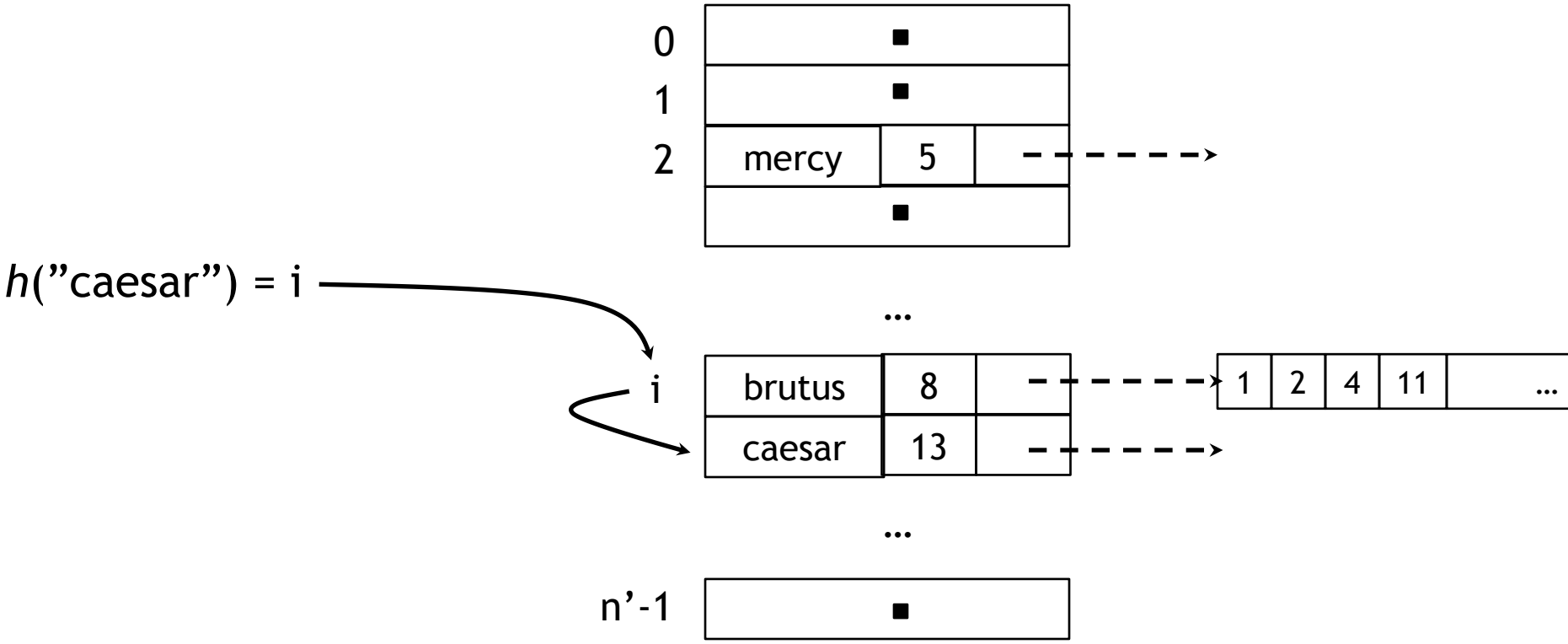
- |L|/n' is called the *load factor* of the hash table
  - No matter how good our hash function is, as |L|/n' → 1 (due to us putting more terms into the table) collisions will occur
  - Collisions cause chains to lengthen and lookup time to increase
  - Eventually must rehash everything to a new table with bigger n'

Collision Resolution via Linear Probing



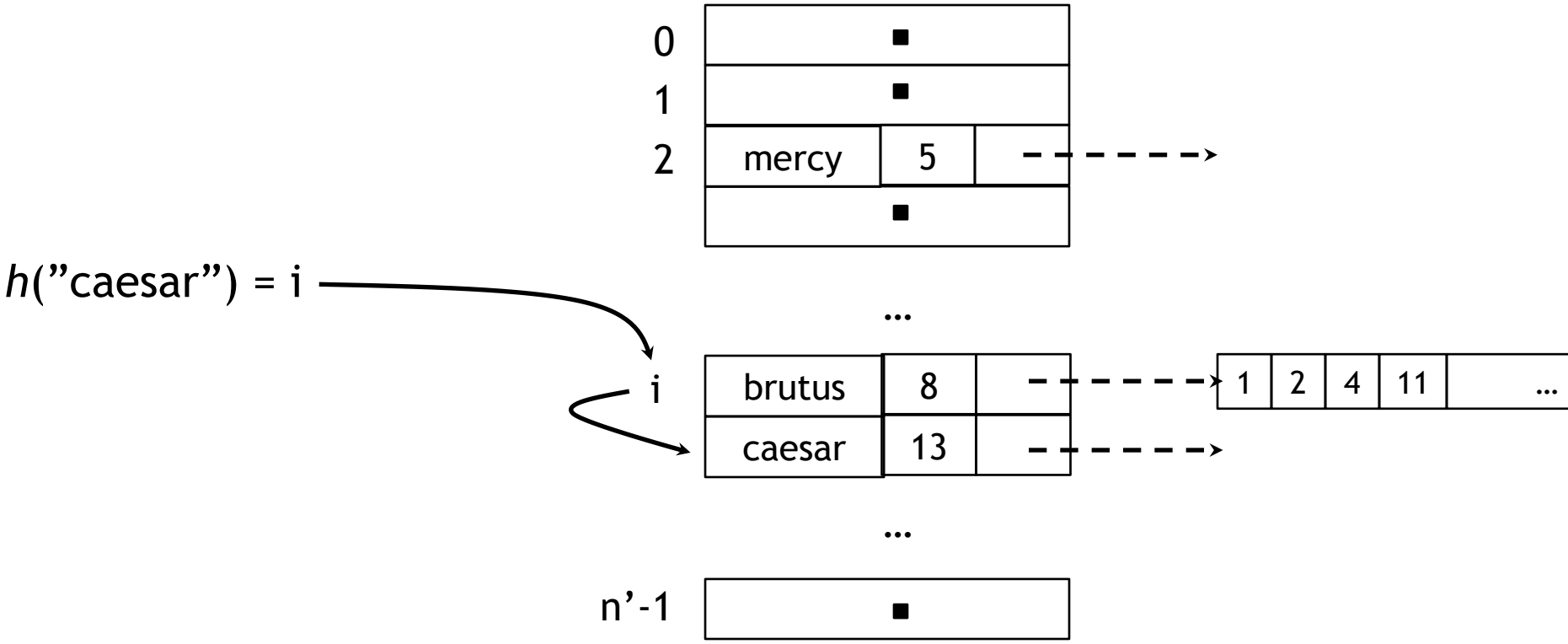
- If a collision occurs at index i, scan the table until free slot found

Collision Resolution via Linear Probing



- If a collision occurs at index i, scan the table until free slot found

Collision Resolution via Linear Probing



- If a collision occurs at index i, scan the table until free slot found
- As with chaining, |L|/n' → 1 means more collisions, more probing, and slower lookup times
  - Additional complication is that our terms are variable length strings
  - Chaining hash tables can tolerate load factors > 1, linear probing can't

Collision Avoidance via Perfect Hashing

- Because the lexicon is fixed (i.e. does not change) for some time after the collection is indexed (at least until the collection grows and the index needs to be rebuilt), we can preprocess the lexicon to derive what is called a *perfect hash function* for the particular set of strings it contains

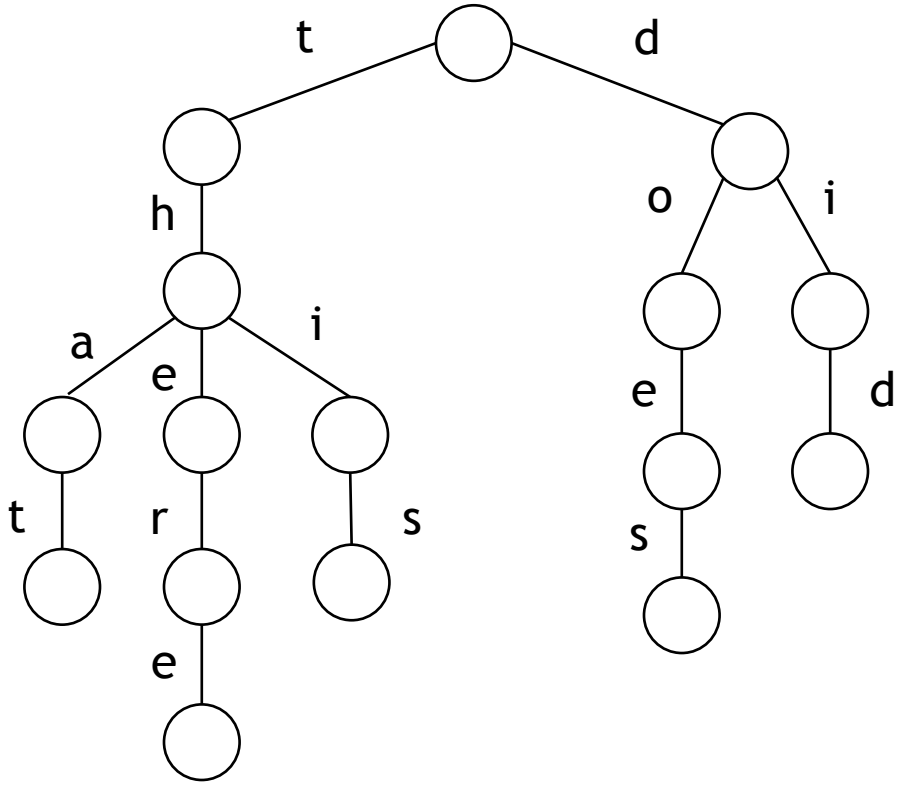
- A perfect hash function is a hash function that has no collisions
  - Works only for a specific set of terms, which must be given in advance
  - Takes some time to construct, and must be reconstructed if the set of terms changes

- We may return to perfect hash functions in more detail later in the course is there is time

Tries

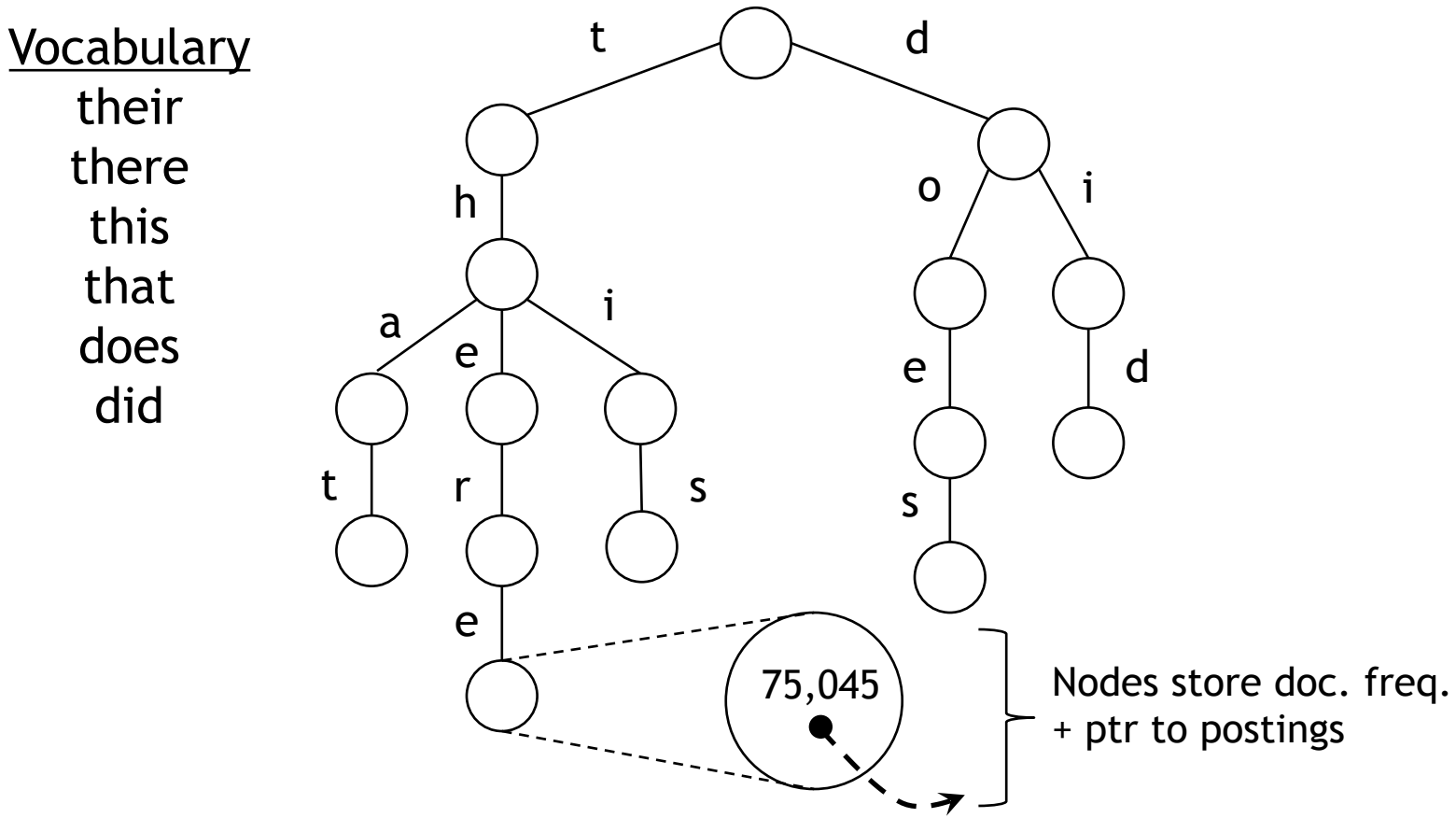
- A trie is a tree structure that stores a set of strings (this set is the terms in the vocabulary, in our case)

Vocabulary
their
there
this
that
does
did



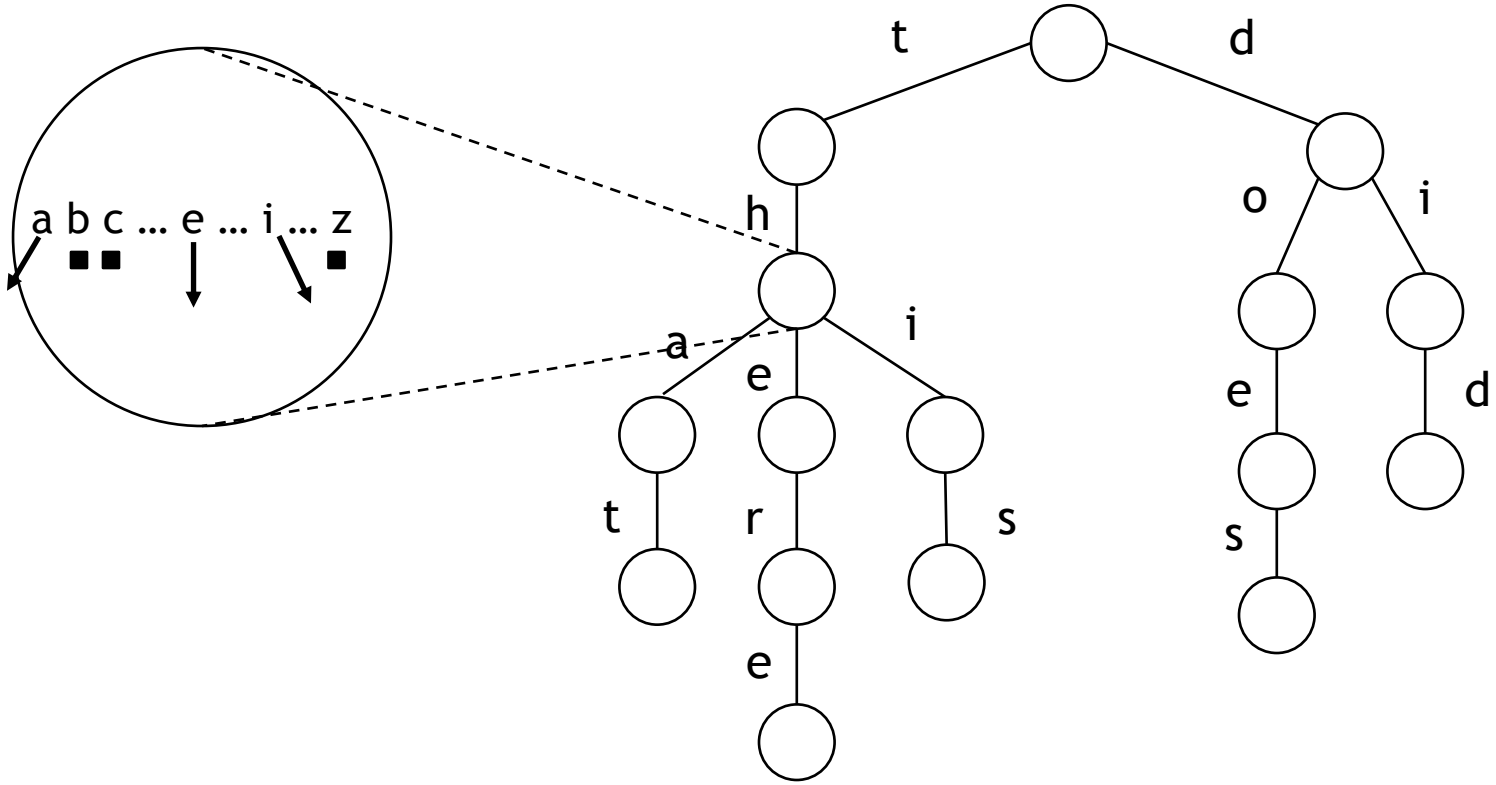
Tries: Lookups

- To lookup a string in the set, we try to match each successive character of the string to an edge label in the trie, starting from the root



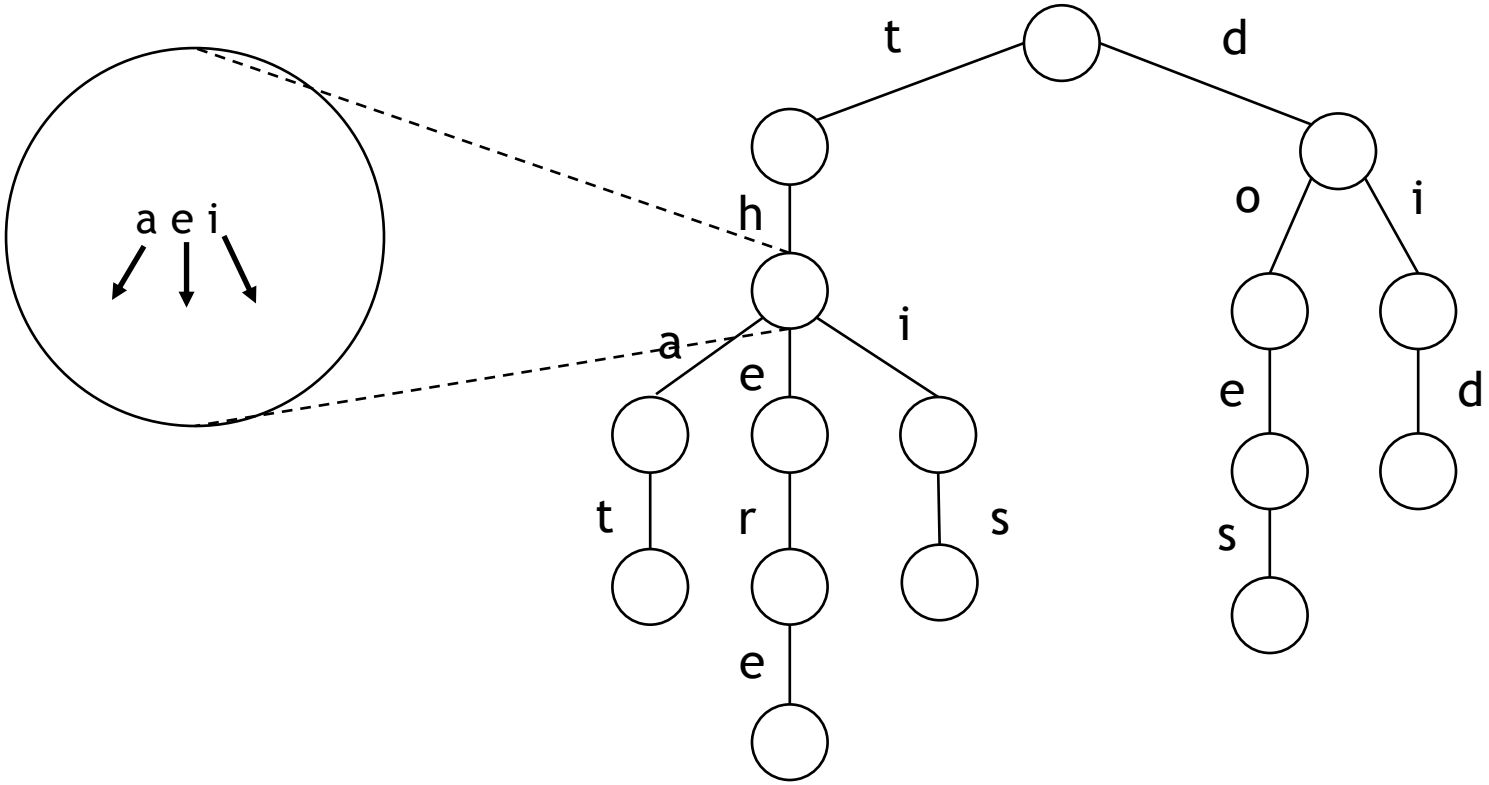
Tries: Branch structure

- As we walk down the trie during a lookup, at each node we encounter we need to make a decision about which outgoing branch to follow: this leads to a space-time trade-off



Tries: Branch structure

- As we walk down the trie during a lookup, at each node we encounter we need to make a decision about which outgoing branch to follow: this leads to a space-time trade-off



Nice Tries

- Tries support prefix matching much more naturally than hash tables do
- No need to rehash, but careful implementation needed to keep space usage down…
  - A key design decision is how to implement the search structure at each node
- Lots of fancy trie implementations out there…
  - We might look at one in more detail later in the course

Next Week

16.1: Introduction to Indexing
- Boolean Retrieval model
- Inverted Indexes

21.1: Index Compression
- variable-byte coding
- (Partitioned) Elias-Fano coding (used inside Google)
- (maybe) BitFunnel (a recent thing out of Microsoft)

23.1: Index Construction
- preprocessing documents prior to search (stemming, et c.)
- building the index efficiently

28.1: Web Crawling
- large scale web search engine architecture

30.1: Query Processing
- scoring and ranking search results
- Vector-Space model