DD2477: Search Engines and Information Retrieval Systems

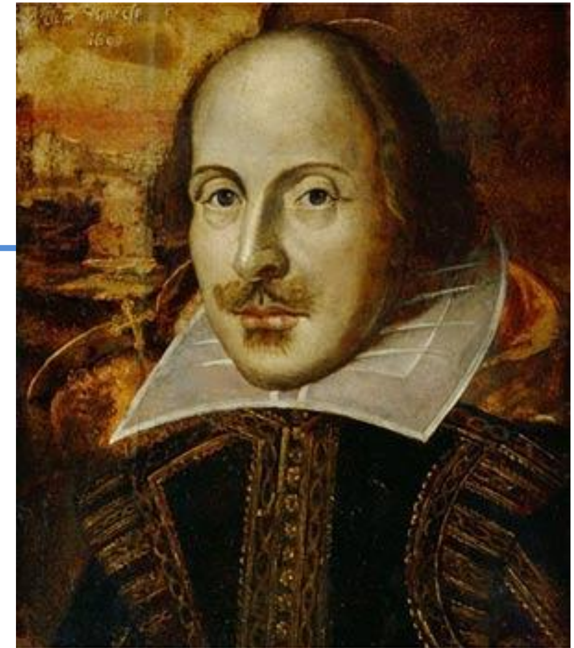
Johan Boye*
KTH

Lecture 2

* Many slides inspired by Manning, Raghavan and Schütze

The Boolean search model

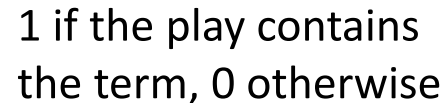
A first IR example



- Which plays by Shakespeare contain the words ”**Brutus**” AND ”**Caesar**” but NOT ”**Calpurnia**”?
- Suggestion:
  - Search for ”**Brutus**” in all plays → resulting set R1
  - Search for ”**Caesar**” i R1 → resulting set R2
  - Search in R2 and return plays that do **not** contain ”**Calpurnia**”
- Is there something wrong with this solution?

Term-document matrix

	Antony and Cleopatra	Julius Caesar	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	0	1	1	1
citizen	1	1	0	0	1	0



Boolean search

- The terms can be represented by **bit vectors**:
  - Brutus: 110100, Caesar: 110111, Calpurnia: 010000, NOT Calpurnia 101111
  - Bitwise AND: 110100 & 110111 & 101111 = 100100
  - The answer is the first and fourth column of the matrix: **Antony and Cleopatra** and **Hamlet**

Boolean search: Advantages

- Simple model to understand and implement
- A Boolean query has a (a mathematically) precise meaning
- Works well for **expert users** working with a **well-defined** document collection (e.g.librarians)

Boolean search: Problems

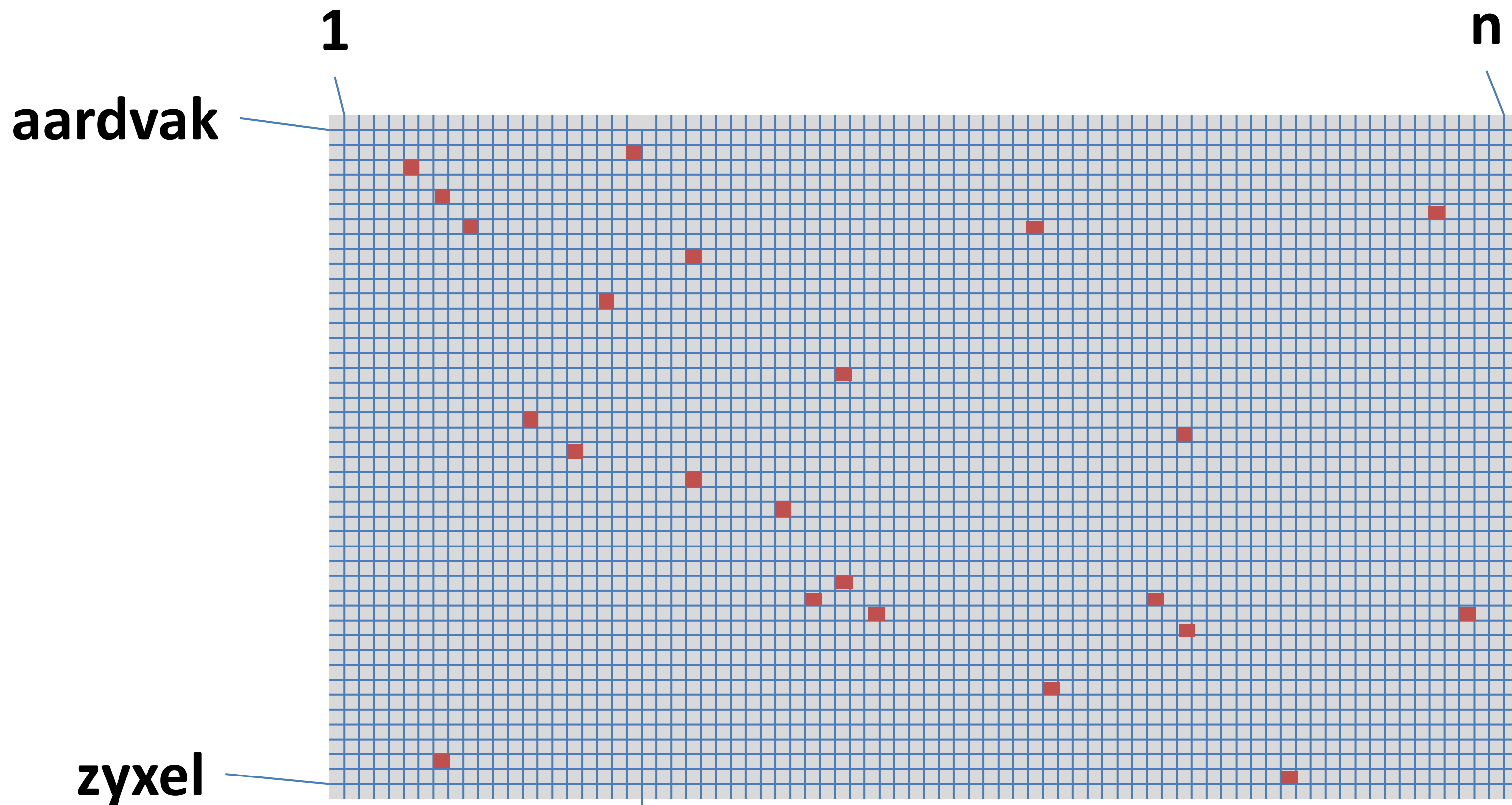
- Many users have difficulties formulating search queries
- Often returns **too many** or **too few** results
  - ”zyxel P-660h” → 192 000 results
  - ”zyxel P-660h” ”no card found” → 0 results
- Requires skill to formulate a search query returning a managable number of results
- **No rankning** of results
- All terms equally important

Boolean search in Google

- `marathon -sparta`
  - looks for documents containing ”marathon” but NOT ”sparta”
- The ”Advanced Search” menu offers more possibilities

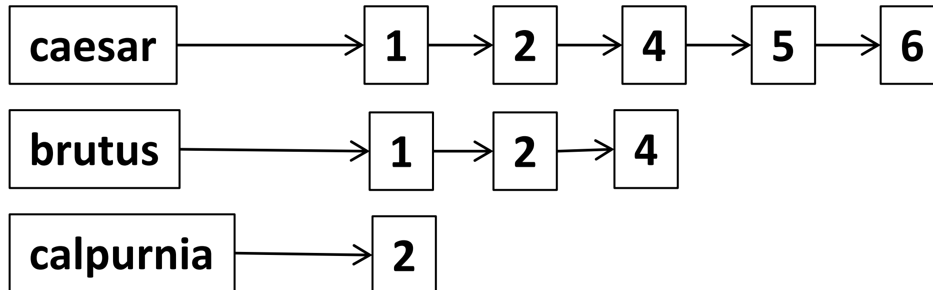
The index

- Conceptually: the **term-document matrix**



Practical indexing

- We need a **sparse matrix representation.**
- In the computer assignments we use:
  - a **hashtable** for the dictionary
  - **arraylists** for the rows
- Rows are called **postings lists.**



Quiz

We want to build a term-document matrix from these documents:

```
Sartre:  to be is to do
Kant:    to do is to be
Sinatra: do be do be do
Hamlet:  to be or not to be
ABBA:    I do I do I do I do I do
```

How many columns will the matrix have? How many rows?

Quiz

	Sartre	Kant	Sinatra	Hamlet	ABBA
to	1	1	0	1	0
be	1	1	1	1	0
is	1	1	0	0	0
do	1	1	1	0	1
or	0	0	0	1	0
not	0	0	0	1	0
I	0	0	0	0	1

Quiz

We implement the term-document matrix using **postings lists**.

```
Sartre:  to be is to do
Kant:    to do is to be
Sinatra: do be do be do
Hamlet:  to be or not to be
ABBA:    I do I do I do I do I do
```

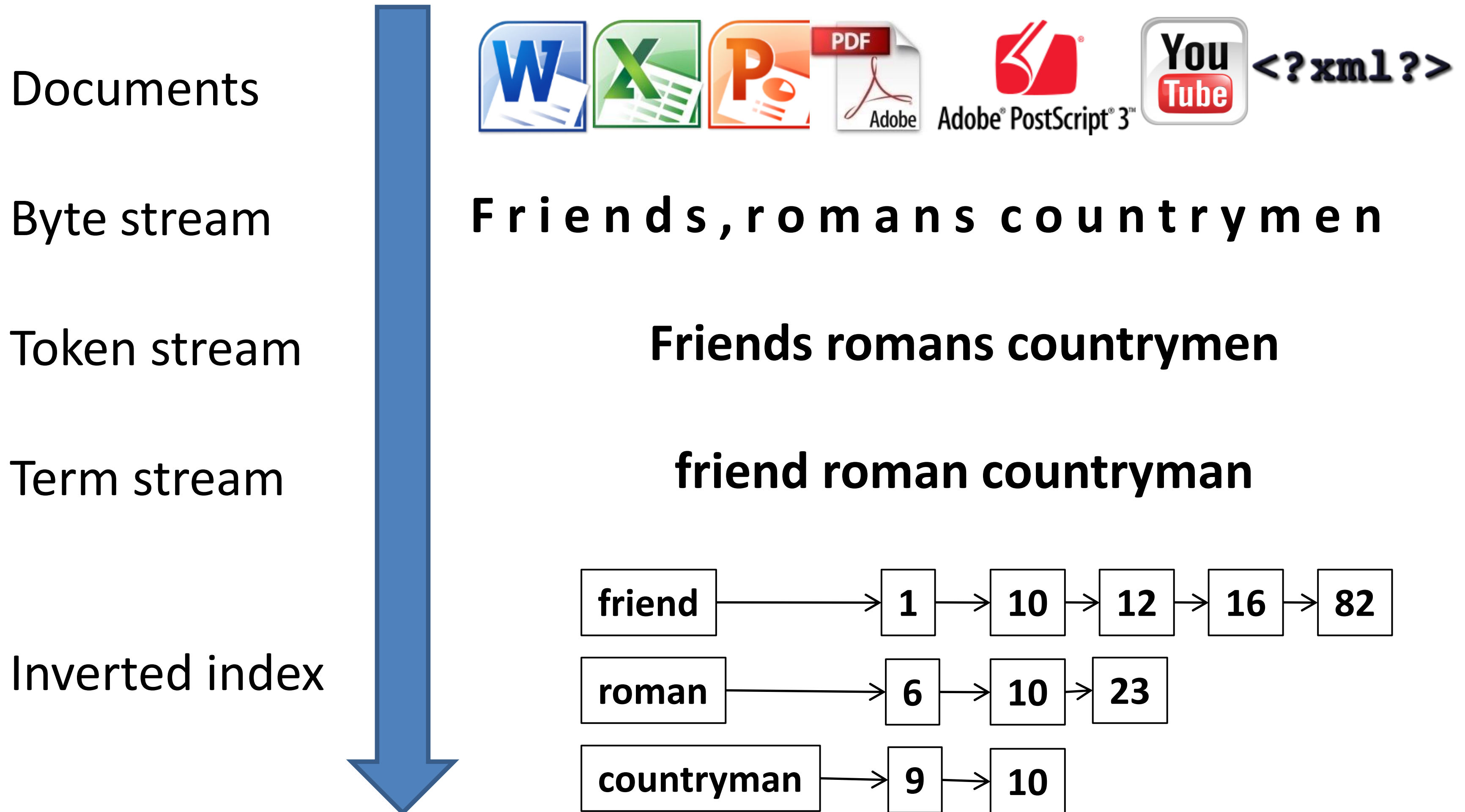
Which word(s) will have the longest postings list? How long is it?

Quiz

- “be” and “do” have the longest postings lists

Tokenization

Indexing pipeline



Basic text processing

- Text comes in many **different formats** (html, text, Word, Excel, PDF, PostScript, …), **languages** and **character sets**
- It might need to be
  - separated from images and other non-textual content
  - stripped of markup in HTML or XML

Character formats

- Text encodings
  - **ASCII** (de-facto standard from 1968), 7-bit (=128 chars, 94 printable). Most common on the www until Dec 2007. Now used in < 0.1% of websites*
  - **Latin-1** (ISO-8859-1), 8-bit, ASCII + 128 extra chars
  - **Unicode** (109 000 code points)
  - **UTF-8** (variable-length encoding of Unicode) Used in >96% of known websites*

*https://w3techs.com/technologies/overview/character_encoding

Tokenization

How many tokens are there in this text?

- Look, harry@hp.com, that’s Harry’s mail address at Hewlett-Packard. Boy, does that guy know Microsoft Word! He’s really working with the state-of-the-art in computers. And yesterday he told me my IP number is 131.67.238.92. :-)

Tokenization

- A token is a **meaningful minimal unit** of text.
- Usually, **spaces** and **punctuation** delimit tokens
- Is that always the case?
  - **San Francisco, Richard III, et cetera, …**
  - **J.P. Morgan & co**
  - **http://www.kth.se, jboye@nada.kth.se**
  - **:-)**
- The exact definition is application-dependent:
  - Sometimes it's important to include punctuation among the tokens (e.g. language modeling)
  - Sometimes it's better not to (e.g. search engines)

Some tricky tokenization issues

- Apostrophes
  - **Finland’s → Finland’s? Finlands? Finland? Finland s?**
  - **don’t → don’t ? don t ? do not ? don t?**
- Hyphens
  - **state-of-the-art → state-of-the-art? state of the art?**
  - **Hewlett-Packard**
  - **the *San Francisco-Los Angeles* flight**
- Numbers
  - Can contain spaces or punctuation: **123 456.7** or **123,456.7** or **123 456,7**
  - **+46 (8) 790 60 00**
  - **131.169.25.10**
  - My PGP key is **324a3df234cb23e**

So how do we do it?

- In assignment 1.1:
  - In the general case, assume that space and punctuation (except apostrophes and hyphens) separate tokens
  - **Specify special cases with regular expressions**

Sub-word tokenization

Sometimes it can be useful to tokenize into subwords...

- ... because words can be syntactically related...
  - ”cat” and ”cats”, ”äpple” and ”äpplenas”
- ... or sematically related
  - ”pianostämning” and ”pianostämmare”
  - tokenizing ”piano” + ”stäm” + ”ning” / ”mare” could be useful

One such tokenization method is ”Byte-Pair Encoding” (next video).

Text normalization

Normalization

- After tokenization, we sometimes need to “normalize” tokens
  - Abbreviations: **U.S., US → U.S.**
  - Case folding: **Window, window → window**
  - Diacritica: **a, å, ä, à, á, â → a, c, ç, č → c, n, ñ → n, l, ł, → l, …**
  - Umlaut: **Tübingen → Tuebingen**, **Österreich → Oesterriech**
- Need for normalization is highly dependent on application
  - Is it always a good idea to lowercase Apple and Windows?
  - Should we remove diacritica?
  - When should we regard run and runs as the same word?

Morphemes

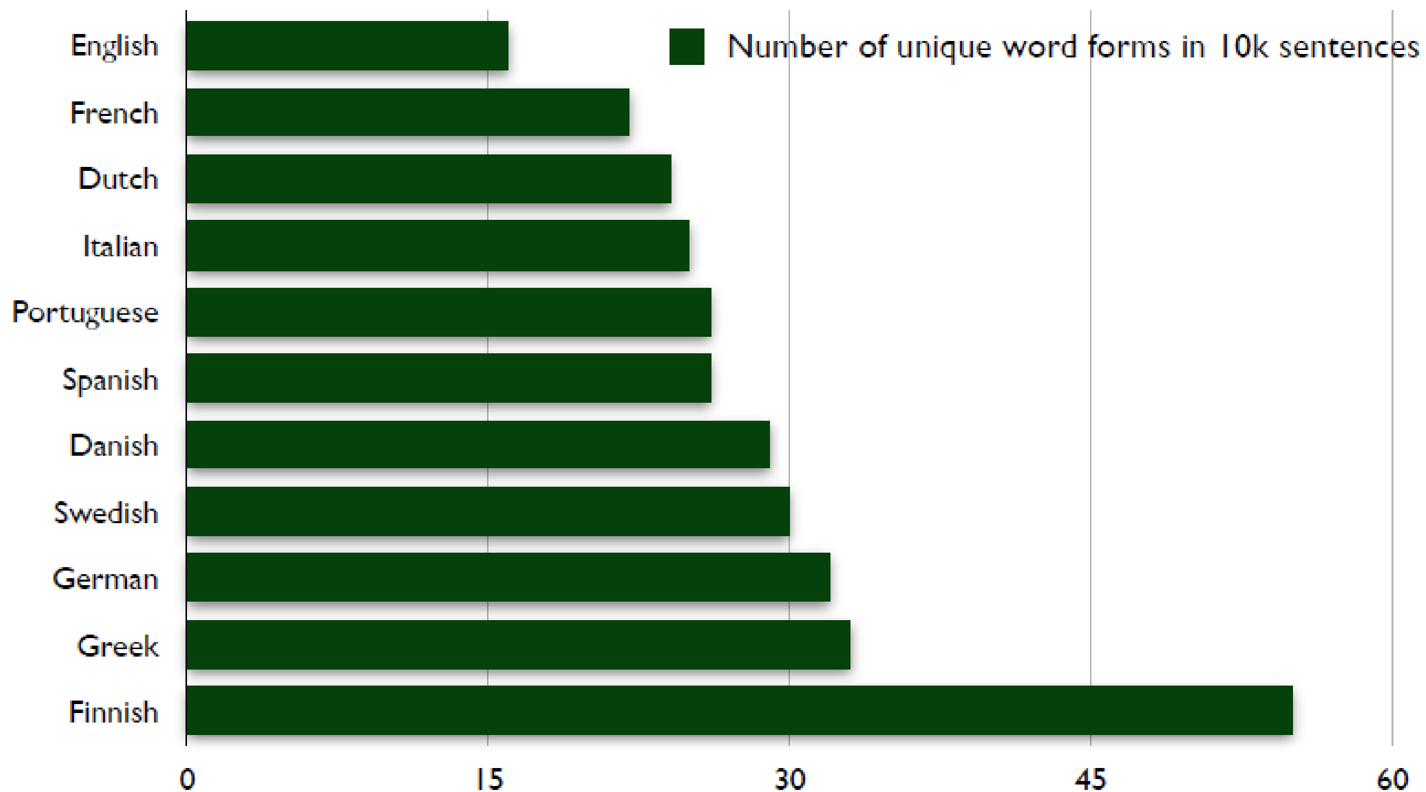
- Words are built from smaller meaningful units called **morphemes**.
- A morpheme belongs to one of two classes:
  - **stem:** the core meaning-bearing unit
  - **affix:** small units glued to the stem to signal various grammatical functions
- An affix can in its turn be classified as a
  - **prefix** (un-)
  - **suffix** (-s, -ed, -ly)
  - **infix** (*Swedish* korru-m-pera)
  - **circumfix** (*German* ge-sag-t)

Word formation

- Words can be **inflected** to signal grammatical information:
  - play, plays, played, playing
  - cat, cats, cat’s, cats’
- Words can also be **derived** from other words:
  - friend → friendly → friendliness → unfriendliness
- Words can be **compound**:
  - smart + phone → smartphone
  - anti + missile → anti-missile
- Clitics
  - Le + hôtel → **L’**hôtel, Ce + est → **c’**est
  - She is → she**’s**, She has → she**’s**

Language variation

- English morphology is exceptionally simple!



Language variation

Parler

The verb *parler* "to speak", in French orthography and IPA transcription

	Indicative				Subjunctive		Conditional	Imperative
	Present	**Simple past**	**Imperfect**	**Simple future**	**Present**	**Imperfect**	**Present**	**Present**
je	parl-e /paʁl/	parl-ai /paʁle/	parl-ais /paʁlɛ/	parl-erai /paʁləʁe/	parl-e /paʁl/	parl-asse /paʁlas/	parl-erais /paʁləʁɛ/	
tu	parl-es /paʁl/	parl-as /paʁla/	parl-ais /paʁlɛ/	parl-eras /paʁləʁa/	parl-es /paʁl/	parl-asses /paʁlas/	parl-erais /paʁləʁɛ/	parl-e /paʁl/
il	parl-e /paʁl/	parl-a /paʁla/	parl-ait /paʁlɛ/	parl-era /paʁləʁa/	parl-e /paʁl/	parl-ât /paʁlɑ/	parl-erait /paʁləʁɛ/	
nous	parl-ons /paʁlɔ̃/	parl-âmes /paʁlɑm/	parl-ions /paʁljɔ̃/	parl-erons /paʁləʁɔ̃/	parl-ions /paʁljɔ̃/	parl-assions /paʁlasjɔ̃/	parl-erions /paʁləʁjɔ̃/	parl-ons /paʁlɔ̃/
vous	parl-ez /paʁle/	parl-âtes /paʁlɑt/	parl-iez /paʁlje/	parl-erez /paʁləʁe/	parl-iez /paʁlje/	parl-assiez /paʁlasje/	parl-eriez /paʁləʁje/	parl-ez /paʁle/
ils	parl-ent /paʁl/	parl-èrent /paʁlɛːʁ/	parl-aient /paʁlɛ/	parl-eront /paʁləʁɔ̃/	parl-ent /paʁl/	parl-assent /paʁlas/	parl-eraient /paʁləʁɛ/	

Some non-English words

- *German:* ***Lebensversicherungsgesellschaftsangestellter***
  - ”Life insurance company employee”
- *Greenlandic*: ***iglukpisuktunga***
  - *iglu = house, kpi = build, suk = (I) want, tu = myself, nga = me*
- *Finnish*: ***järjestelmättömyydellänsäkäänköhän***
  - *”not even with its lack of order”*

Tokenization using byte-pair encoding

Idea:

- First learn (once) the vocabulary (set of token types) directly from a large corpus
- Then tokenize files/sentences using the learned vocabulary

Tokenization using byte-pair encoding

Method:

1. Initial vocabulary is the set of all bytes (a,b,c,..., A,B,C,...)
2. Then choose the two symbols that are most frequently adjacent in the training corpus (e.g. ’th’)
3. Add a new symbol ’th’ to the vocabulary
4. Replace all adjacent ’t’ ’h’ by ’th’
5. Repeat from 2 until *k* merges have been done. (typically 25,000)

Tokenization using byte-pair encoding

t h e _ t h i n _ t h i e f _ t h r e a t e n e d _ t h e _ t h u g

t h → th

th e _ th i n _ th i e f _ th r e a t e n e d _ th e _ th u g

th e → the

the _ th i n _ th i e f _ th r e a t e n e d _ the _ th u g

th i → thi

the _ thi n _ thi e f _ th r e a t e n e d _ the _ th u g

Tokenization using byte-pair encoding

- Greedy, language-independent method
- Note that the algorithms work on bytes, not chars
  - many non-English chars have >1 byte, e.g. åäö
  - these tend to be merged early in the process
- When used for tokenization, merges should be applied in the order they were learned
- But greedy left-to-right maximal matching works pretty well too

More on text normalization

Lemmatization

- Map **inflected form** to its **lemma** (=base form)
- ”The boys’ cars are different colours” → ”The boy car be different color”
- Requires language-specific linguistic analysis
  - part-of-speech tagging
  - morphological analysis
- Particularly useful in morphologically rich languages, like Finnish, Turkish, Hungarian

Stemming

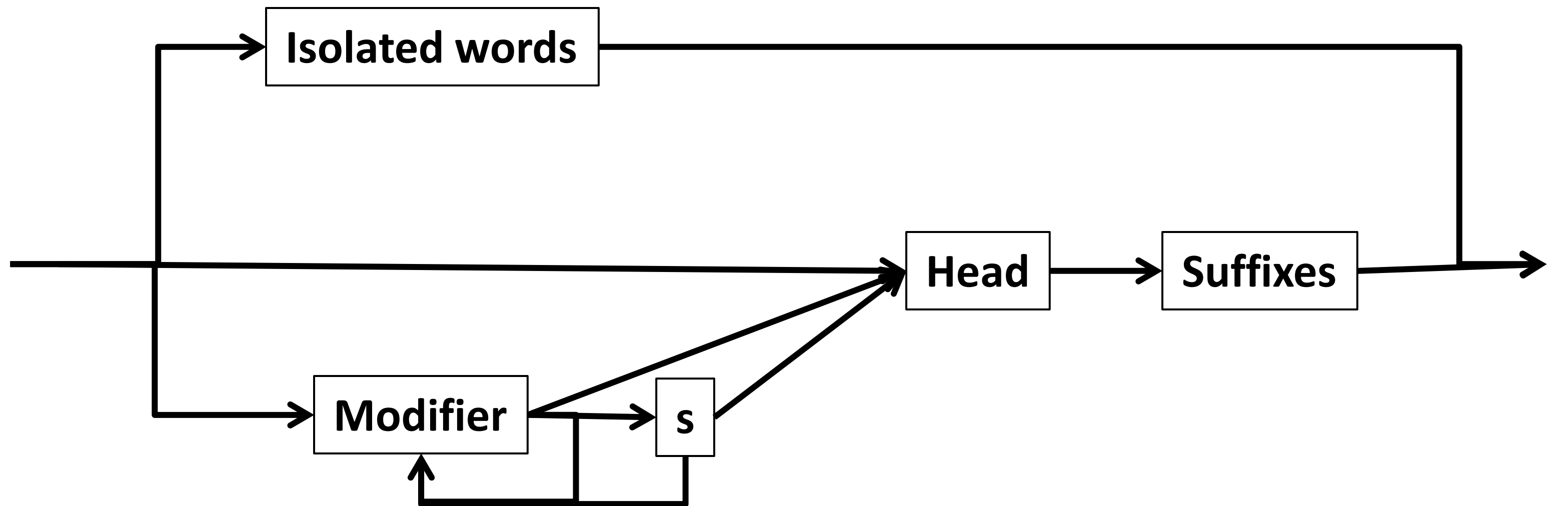
- Don’t do morphological or syntactic analysis, just **chop off the suffixes**
  - No need to know that ”foxes” is plural of ”fox”
- Much **less expensive** than lemmatization, but **can be very wrong** sometimes
  - stocks → stock, stockings → stock
- Stemming usually improves **recall** but lowers **precision**

Porter's algorithm

- Rule-based stemming for English
  - ATIONAL → ATE
  - SSES → SS
  - ING → ε
- Some context-sensitivity
- (W>1) EMENT → ε
  - REPLACEMENT → REPLAC
  - CEMENT → CEMENT

Compound splitting

Can be achieved with finite-state techniques.



Compound splitting

- In Swedish: **försäkringsbolag** (insurance company)
  - **bolag** is the head
  - **försäkring** is a modifier
  - the **s** is an infix
- This process can be recursive:
  - försäkringsbolagslagen (the insurance company law)
  - **en** is a suffix indicating definite form
  - **lag** is the head
  - the **s** is an infix
  - **försäkringsbolag** is the modifier

Stop words

- Can we exclude the most common words?
  - In English: **the**, **a**, **and**, **to**, **for**, **be**, …
  - Little semantic content
  - ~30% of postings for top 30 words
- However:
  - ”**Let it be**”, ”**To be or not to be**”, ”**The Who**”
  - ”**King of Denmark**”
  - ”**Flights to London**” vs ”**Flights from London**”
  - Trend is to keep stop words: compression techniques means that space requirements are small

Sum-up

- **Reading**, **tokenizing** and **normalizing** contents of documents
  - **File types** and **character encodings**
  - Tokenization issues: **punctuation, compound words, word order, stop words**
  - Normalization issues: **diacritica, case folding, lemmatization, stemming**
- We're ready for **indexing**

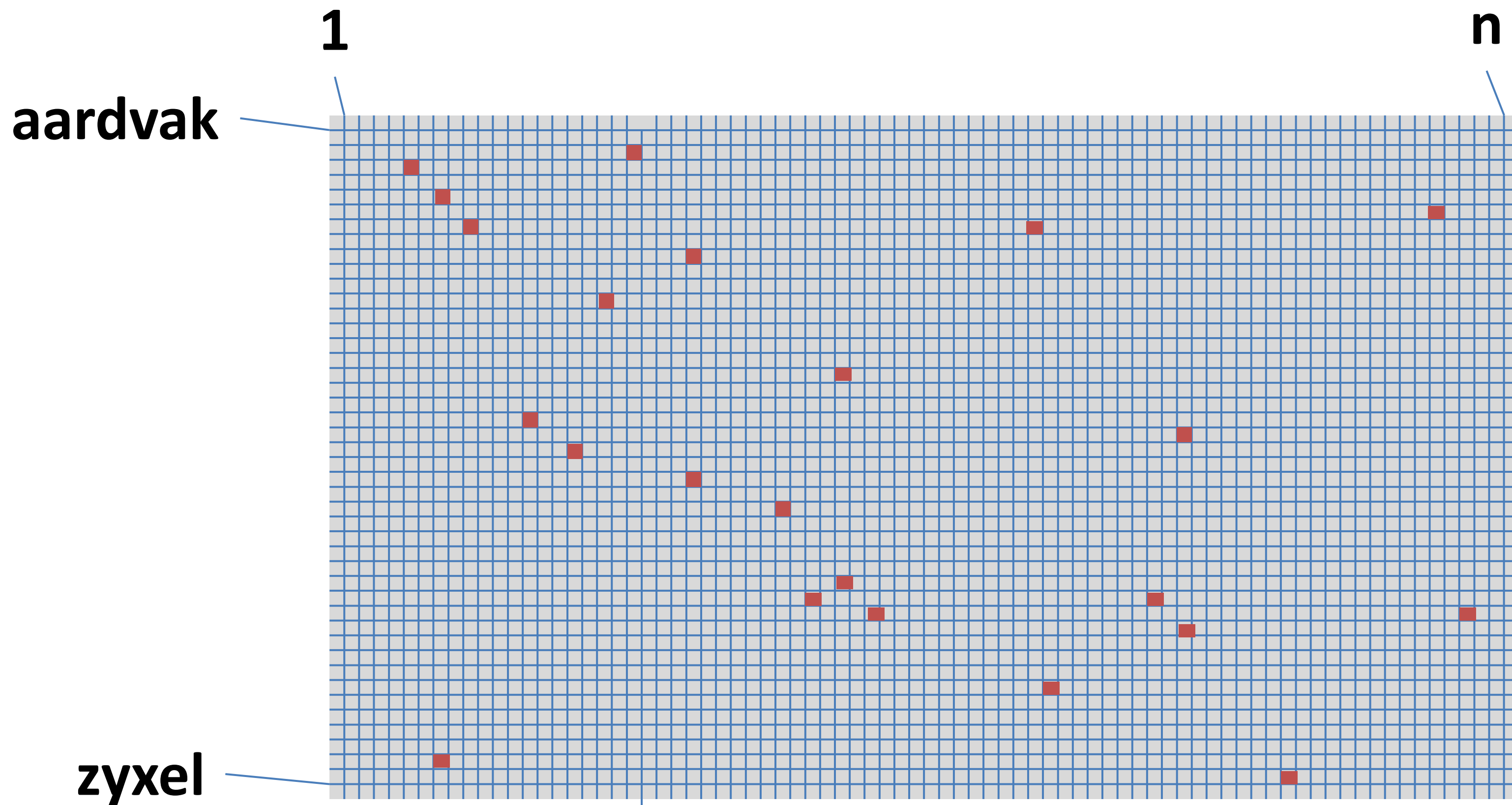
Indexing and search

Indexing and search

- **Recap:**
  - We want to quickly find the **most relevant documents** satisfying our **information need**.
  - The user gives a **search query**.
  - The engine searches through the **index**, retrieves the **matching** documents, and possibly **ranks** them.

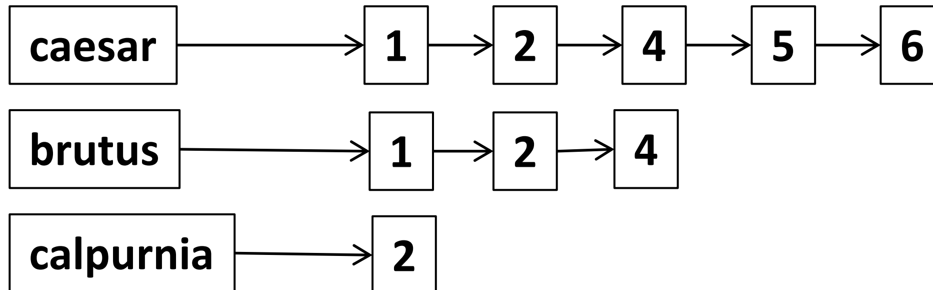
The index

- Conceptually: the **term-document matrix**



Practical indexing

- We need a **sparse matrix representation.**
- In the computer assignments we use:
  - a **hashtable** for the dictionary
  - **arraylists** for the rows
- Rows are called **postings lists.**



One-word queries

denmark

- Return all the documents in which ’**denmark**’ appears. (Task 1.2)

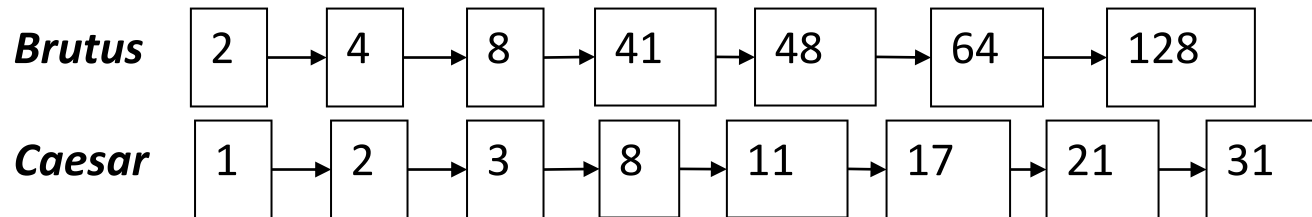
Multi-word queries

copenhagen denmark

- **Intersection** query (Task 1.3)
- **Phrase** query (Task 1.4)
- **Union** query (Assignment 2)

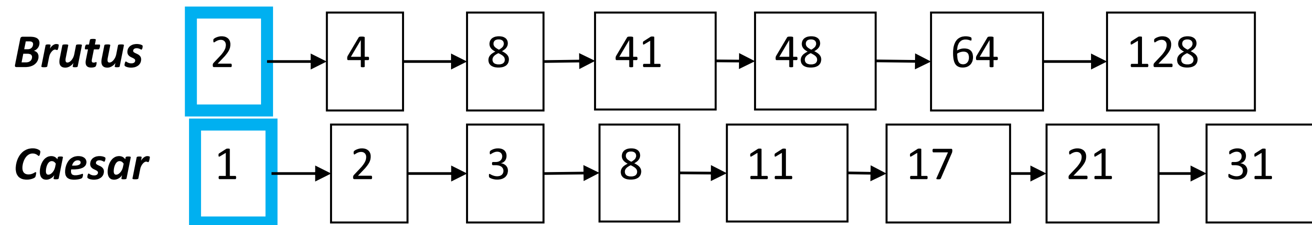
Intersection

- Walk through two postings lists simultaneously



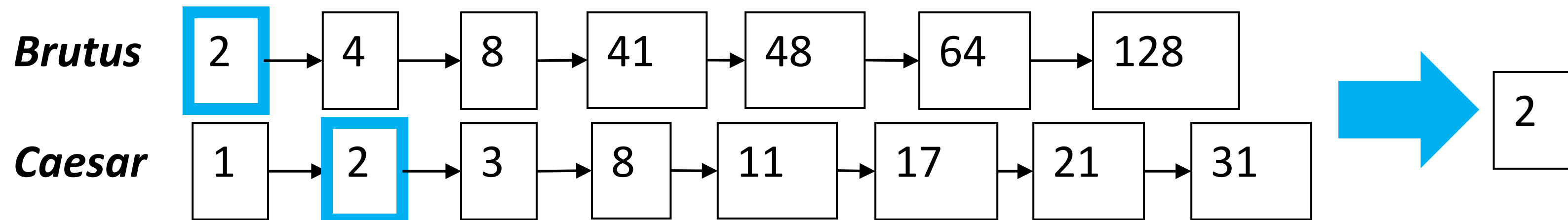
Intersection

- Walk through two postings lists simultaneously



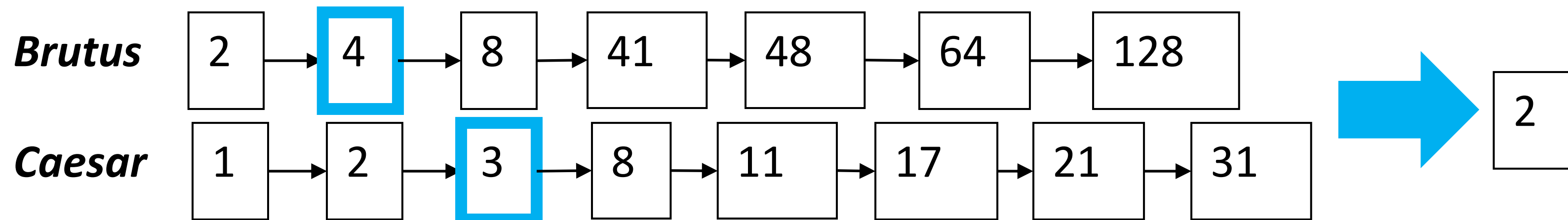
Intersection

- Walk through two postings lists simultaneously



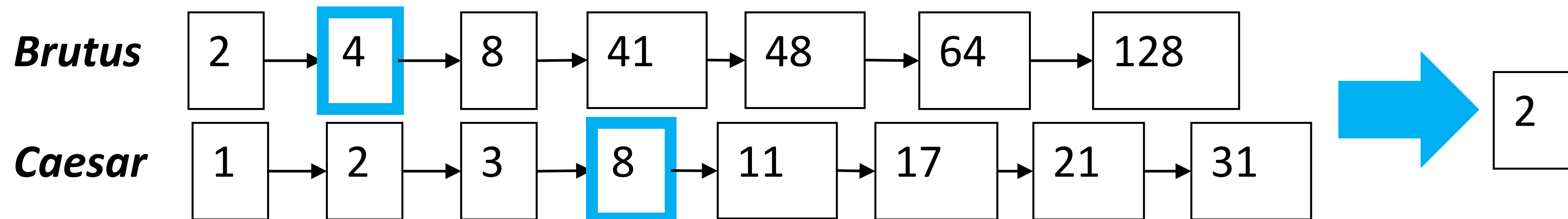
Intersection

- Walk through two postings lists simultaneously



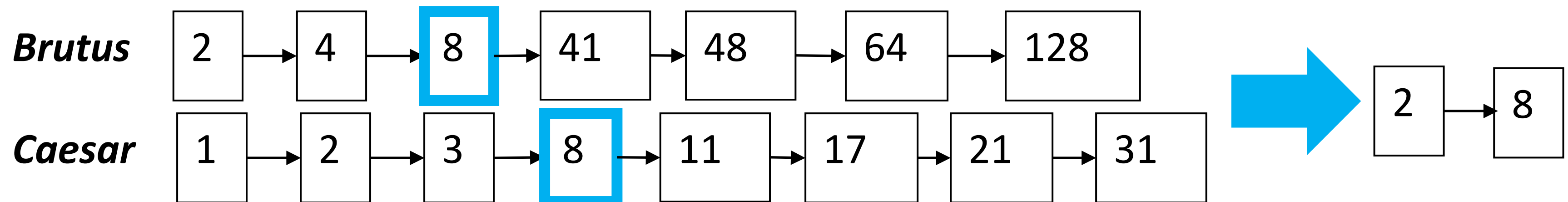
Intersection

- Walk through two postings lists simultaneously



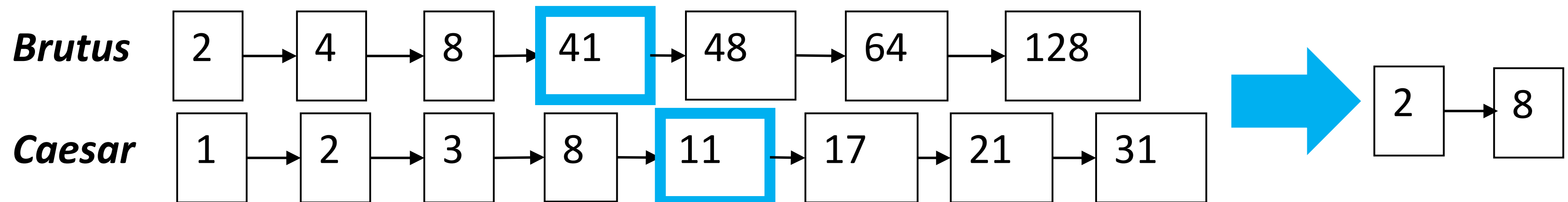
Intersection

- Walk through two postings lists simultaneously



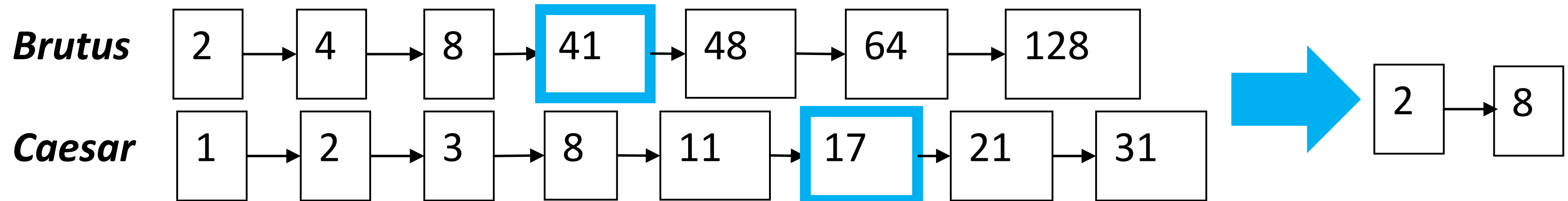
Intersection

- Walk through two postings lists simultaneously



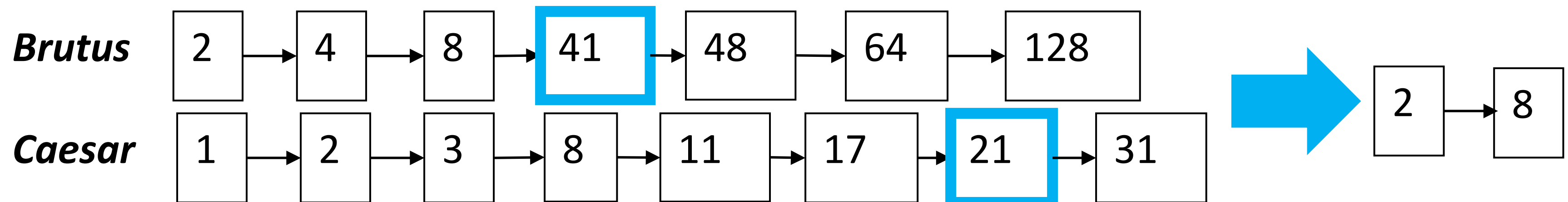
Intersection

- Walk through two postings lists simultaneously



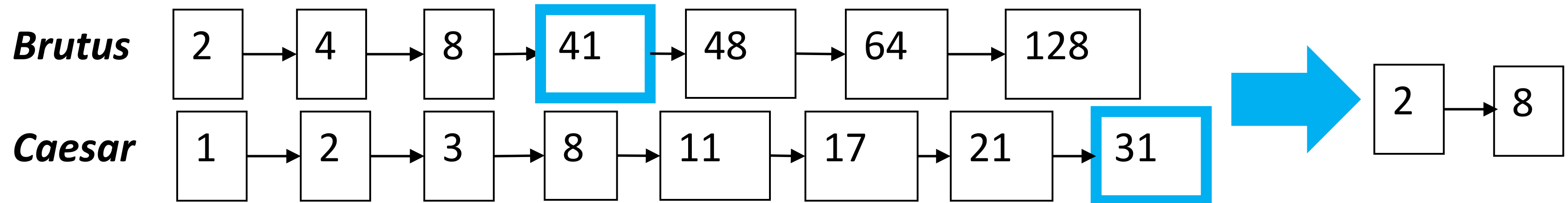
Intersection

- Walk through two postings lists simultaneously



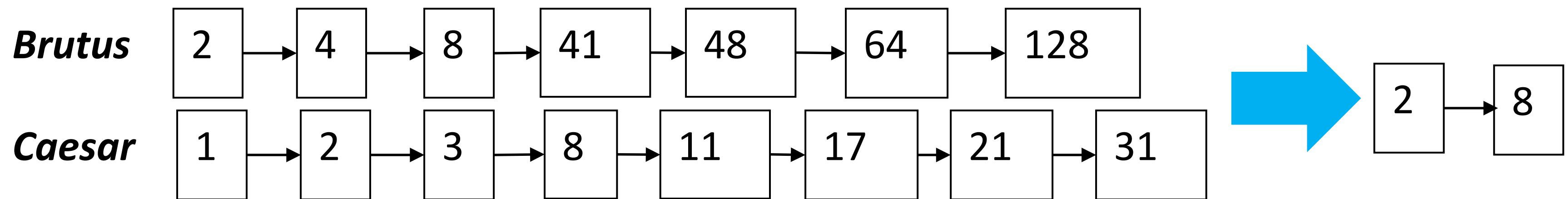
Intersection

- Walk through two postings lists simultaneously



Intersection

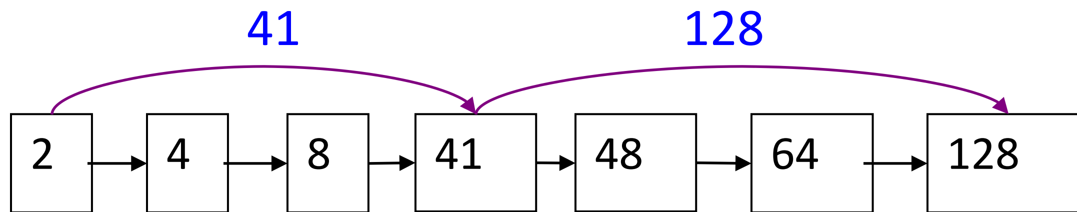
- Walk through two postings lists simultaneously

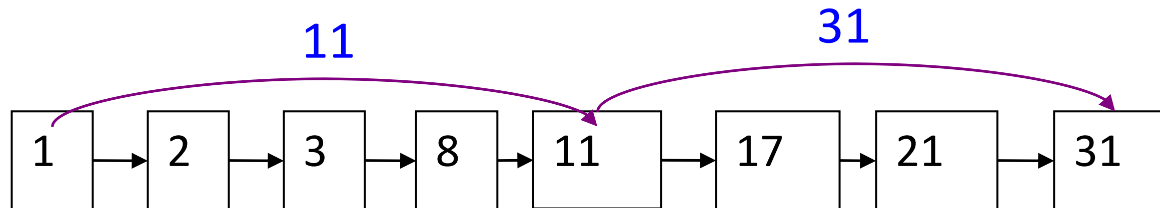


- Runs in O(n+m), where n,m are the lengths of the lists
- We can do better (if index isn't changing too fast)

Skip pointers

- Add skip pointers at indexing time

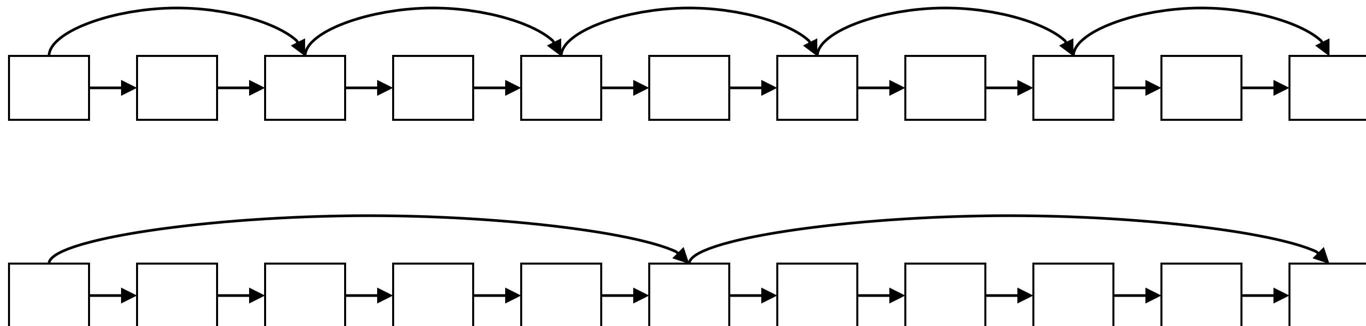




- By using skip pointers, we don’t have to compare 41 to 17 or 21

Skip pointers: Where?

- 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.
  - Heuristic: for length L, use $\sqrt{L}$ evenly spaced skip pointers



Positional indexes and phrase queries

Phrase queries

- E.g. **”Joe Biden”**
- Should not match ”President Biden”
  - The concept of phrase queries has proven easily understood by users; one of the few “advanced search” ideas that works
  - Many more queries are *implicit phrase queries*
- For this, it no longer suffices to store only <*term* : *docs*> entries

First attempt: Biword index

- "Friends, Romans, Countrymen" generates the **biwords**
  - ***friends romans***
  - ***romans countrymen***
- Each of these biwords is now a dictionary term
- Two-word phrase query-processing is now immediate.
- Longer phrases: **friends romans countrymen**
- Intersect **friends romans** and **romans countrymen**?

Biword index: disadvantages

- **False positives**
  - Requires post-processing to avoid
- **Index blowup** due to bigger dictionary
  - Infeasible for more than biwords, big even for them

Positional indexes

- For each term and doc, store the positions where (tokens of) the term appears

<***be***;
1: 7, 18, 33, 72, 86, 231;
2: 3, 149;
4: 17, 191, 291, 430, 434;
5: 363, 367, …>

- Intersection needs to deal with more than equality

Processing phrase queries

- Extract inverted index entries for each distinct term: ***to, be, or, not.***
- Intersect their *doc:position* lists to enumerate all positions with "***to be or not to be***".
  - ***to**:*
    - *2*:1,17,74,222,551; *4*:8,16,190,429,433; *7*:13,23,191; ...
  - ***be**:*
    - *1*:17,19; *4*:17,191,291,430,434; *5*:14,19,101; ...
- Same general method for proximity searches

Exercise

Which docs match the query "**fools rush in**" ?

fools: 2: 1,17,74,222;
4: 78,108,458;
7: 3,13,23,193;
in: 2: 3,37,76,444,851;
4: 10,20,110,470,500;
7: 5,15,25,195;

rush: 2: 2,75,194,321,702;
4: 9,69,149,429,569;
7: 14,404;

Positional index size

- Need an entry for each occurrence, not just once per document
- Consider a term with frequency 0.1%
  - Doc contain 1000 tokens → 1 occurrence
  - 100 000 tokens → 100 occurrences
- Rule of thumb: is 2–4 as large as a non-positional index
- Positional index size 35–50% of volume of original text
- Caveat: all of this holds for “English-like” languages

Large indexes

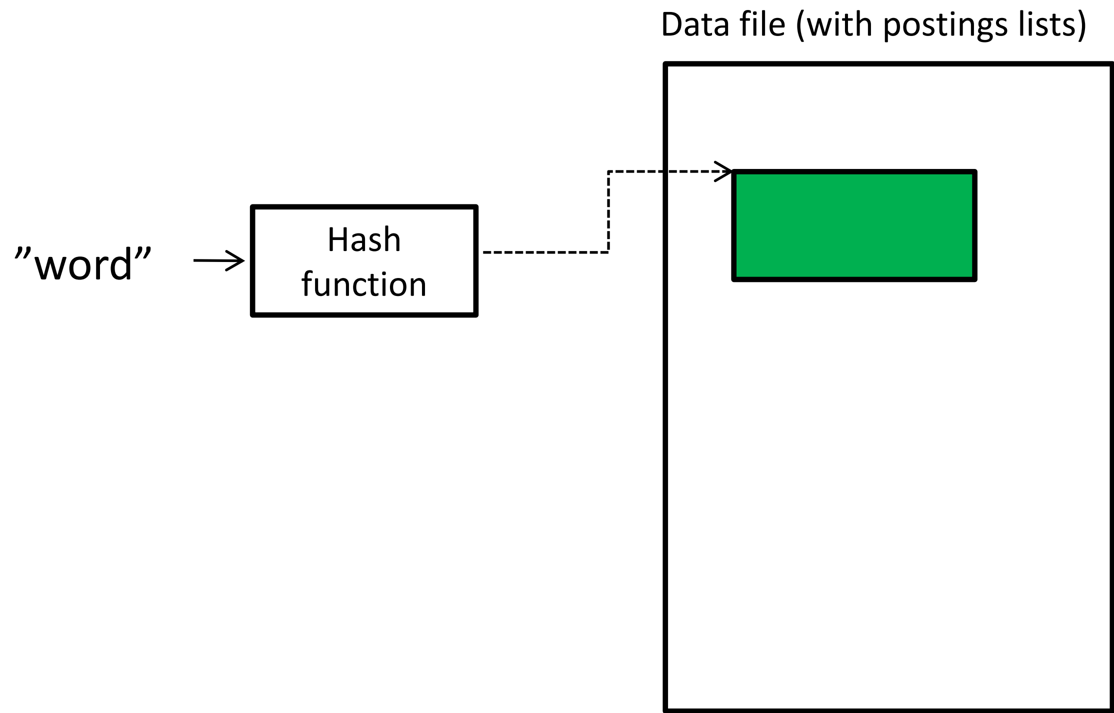
Large indexes (Task 1.7-1.8)

- The web is big:
  - 1998: **26 million** unique web pages
  - 2018: **130 trillion** (1.3×10^{14}) unique web pages!
  - about 4.26 billion of these are indexed.
- In real applications, the index is **too large to fit** in main memory.

Large indexes (Task 1.7-1.8)

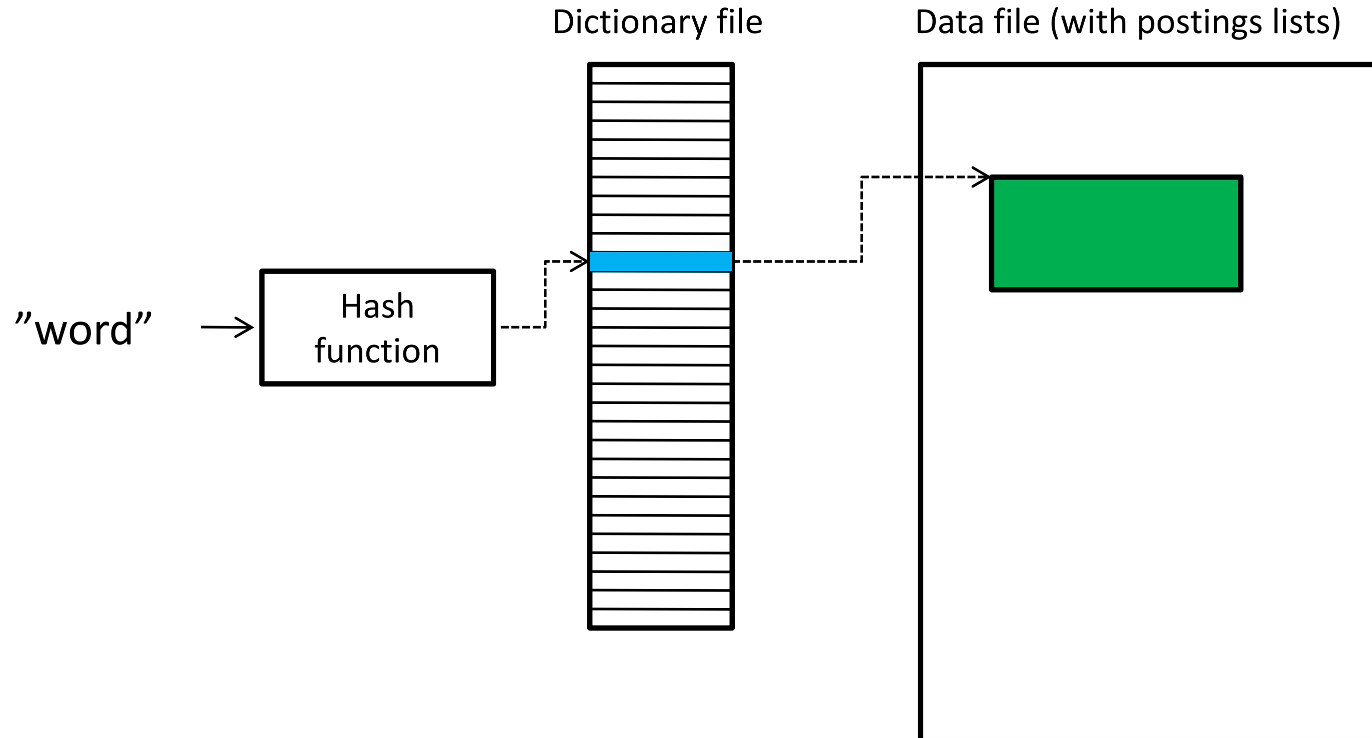
- Task 1.7 asks you to implement an index which is stored on disk
  - using any method (well, not quite...) for grade C
  - using a hash table with both dictionary and postings lists on disk for grade B
  - **first** construct the index in main memory, **then** write it to disk
  - if we have a lot of data, construct several sub-indexes, and then **merge** them

Hash tables on disk- what one would like to do



Why doesn’t this work?

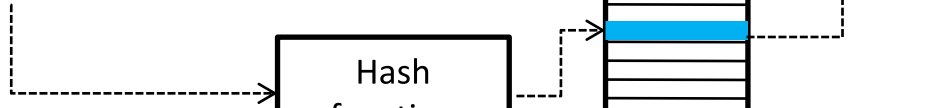
Hash tables on disk- what we will do

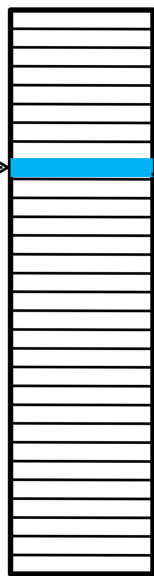


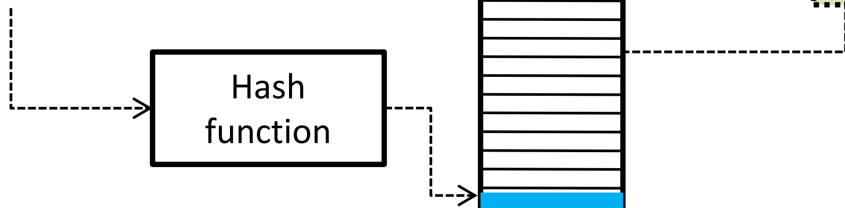
Hash table on disk

- Dictionary file:
  - with entries of a **fixed length**
  - entries contain pointer to the data file
- Data file
  - contains string representation of postings list
  - don't serialize the PostingsList objects! (waste of space)
- Hash function
  - inputs word (as a string)
  - outputs an integer [0…TABLESIZE-1] which is a pointer to the dictionary file.

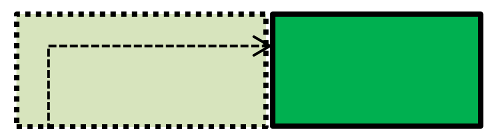
”**words** are useful”






”words **are** useful”
Hash function

Dictionary file

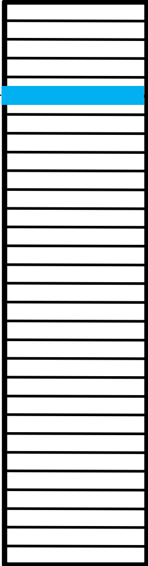

Data file (with postings lists)

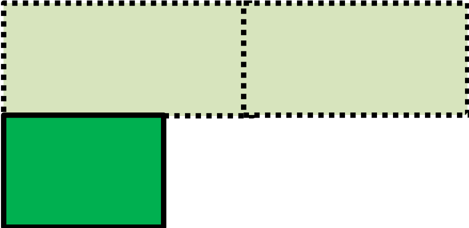
Dictionary file

Data file (with postings lists)

”words are **useful**”

Hash function

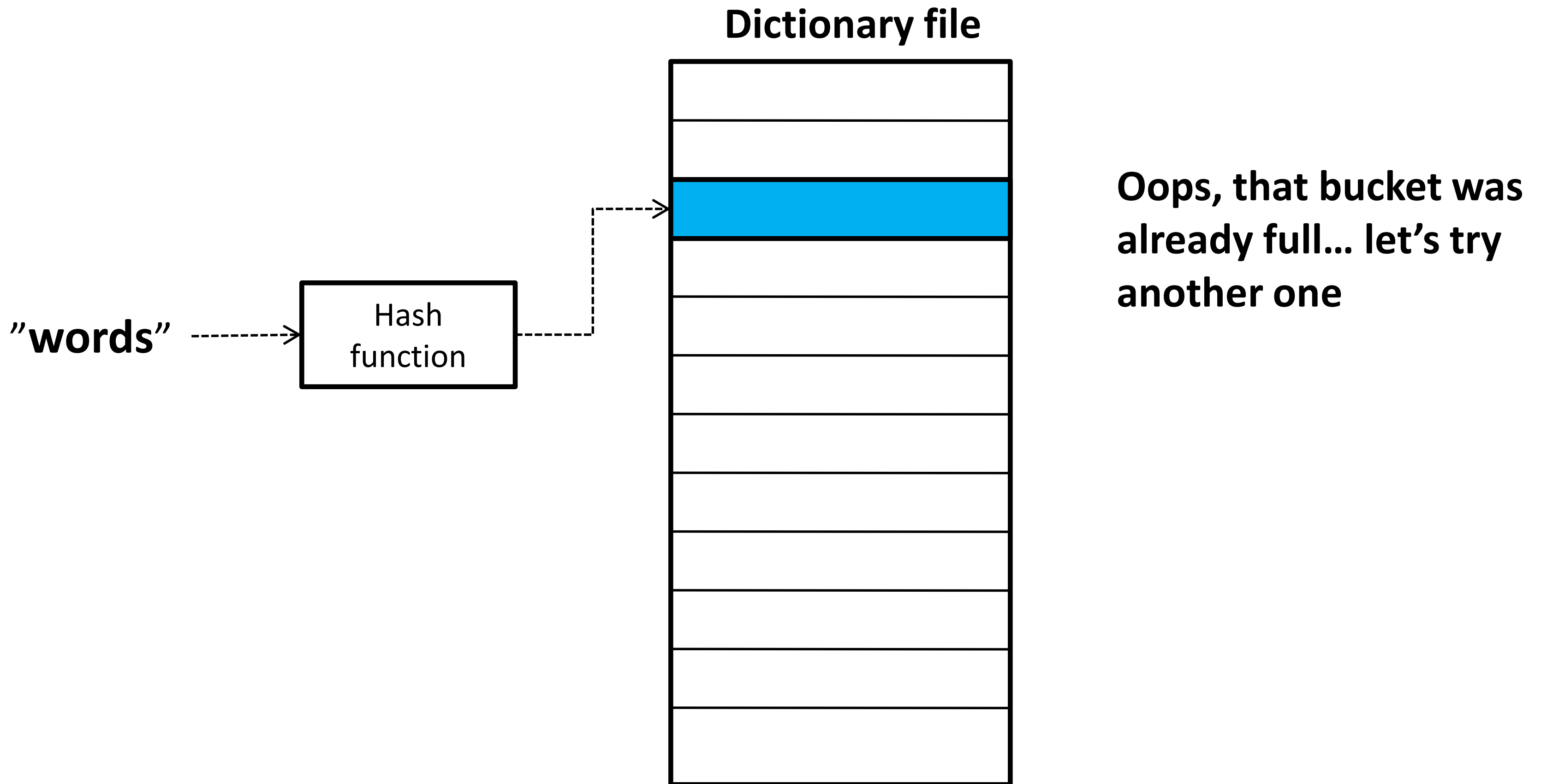




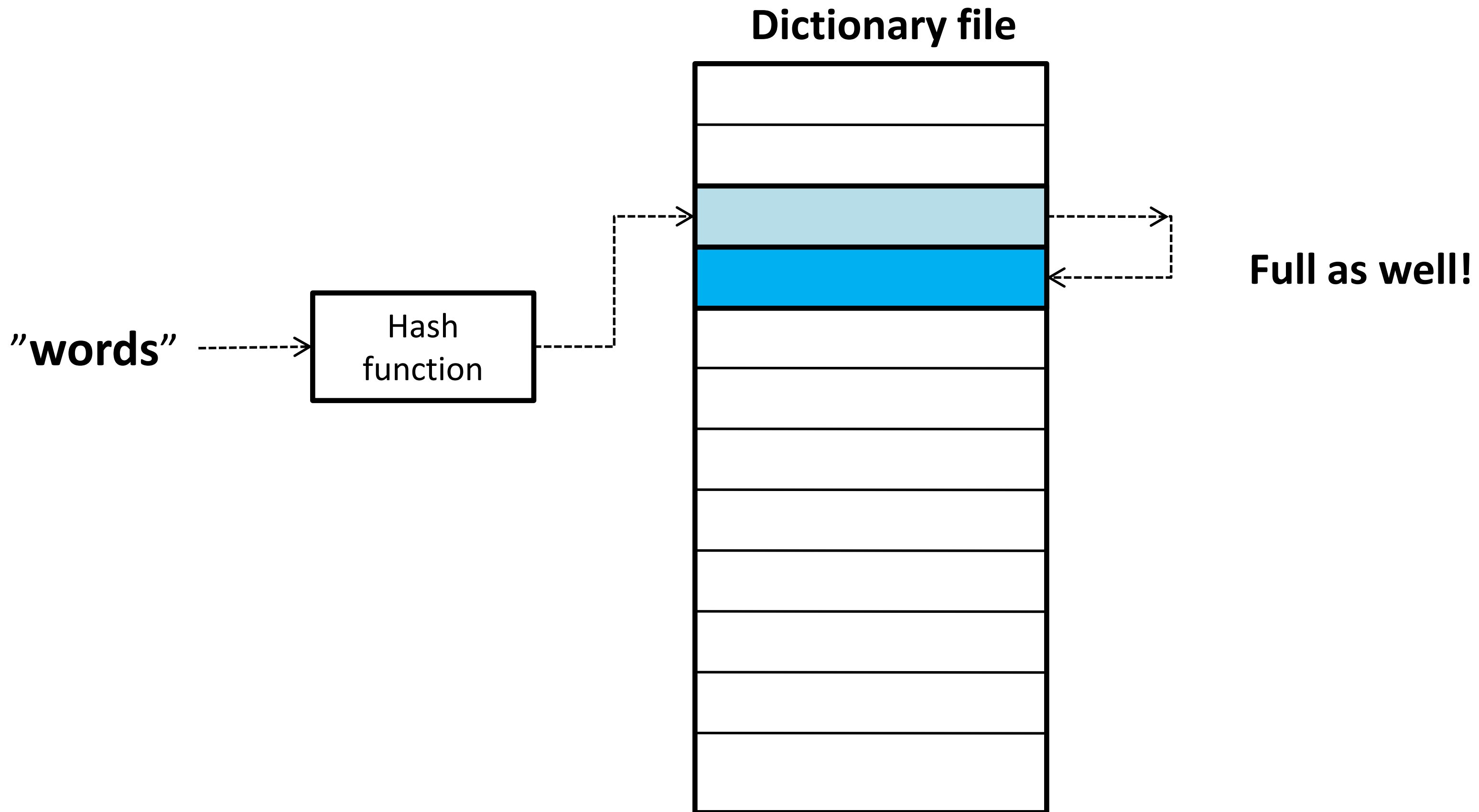
Hash table on disk

- Dictionary file
  - has a fixed size
  - will be mostly empty (load factor about 0.33)
- Data file grows dynamically
  - will be completely packed

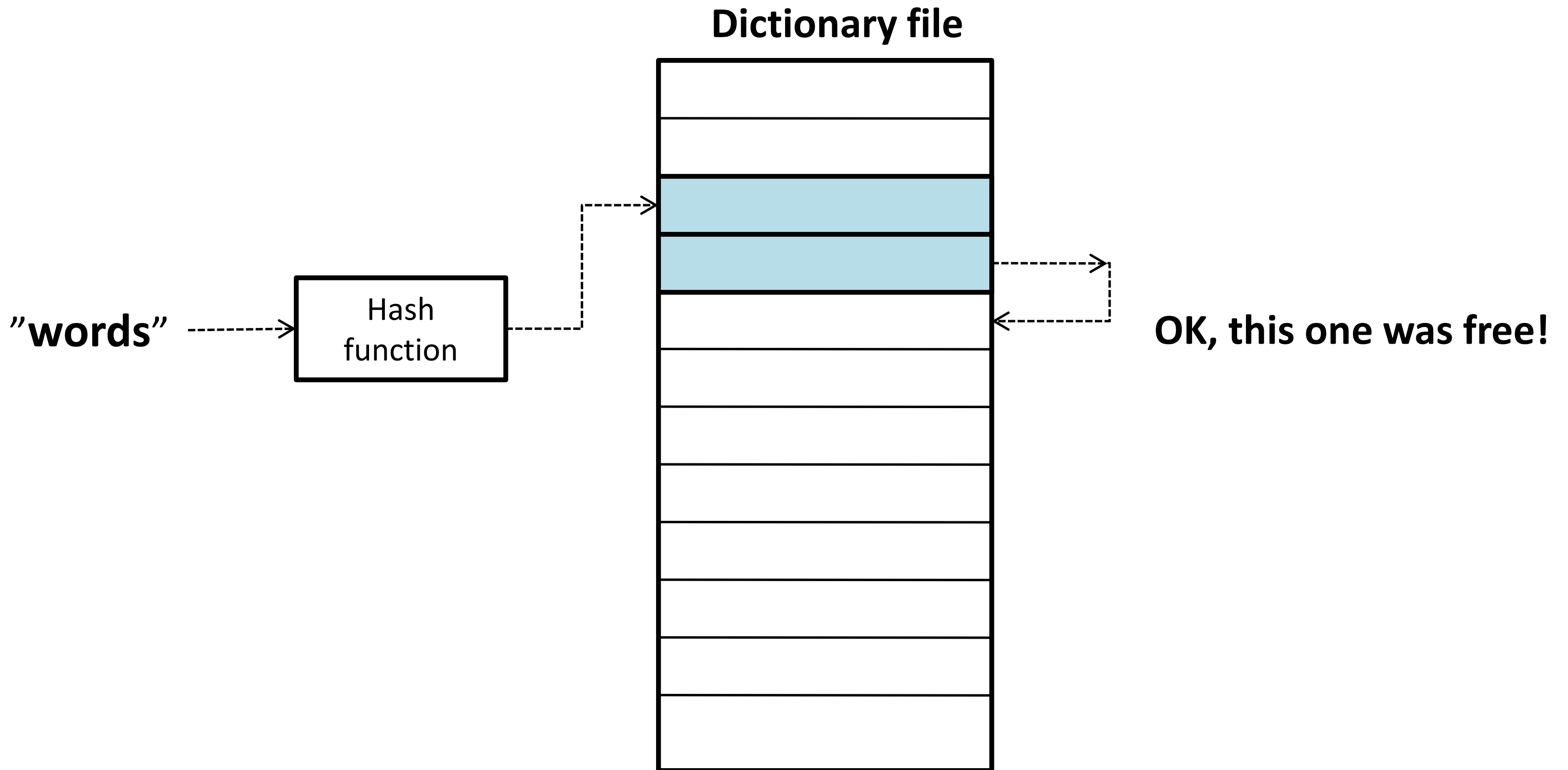
Hash collisions



Hash collisions



Hash collisions

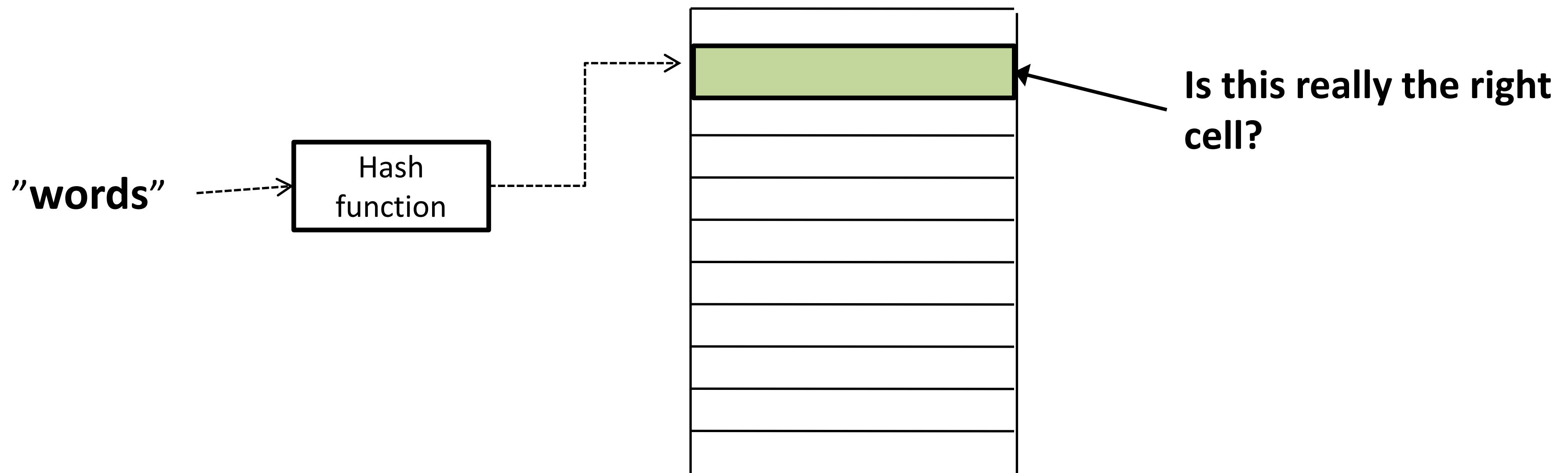


Hash function

- Have a look in the literature
  - or devise your own method
  - but be sure there aren’t too many collisions
  - about 1 collision/unique word is a reasonable target
  - (that means about 200,000 collisions)

Searching the disk index

- When the index is committed to disk, you can restart the search engine, and start searching without any start-up time.
- However, how do you detect hash collisions?



Merging indexes

- The solution in Task 1.7 still requires that the entire index can be kept in main memory
- Task 1.8 will ask you to construct a series of small indexes, which you will then **merge** in a background thread.
- You can then search the merged index just as in task 1.7

Dynamic indexing

- Document collections are rarely static.
  - Documents come in over time and need to be inserted.
  - Documents are deleted and modified.
- This means that the dictionary and postings lists have to be modified:
  - Postings updates for terms already in dictionary
  - New terms added to dictionary

Simplest approach

- Maintain "big" main index
- New docs go into "small" auxiliary index
- Search across both, merge results
- Deletions
  - Invalidation bit-vector for deleted docs
  - Filter docs output on a search result by this invalidation bit-vector
- Periodically, re-index into one main index

Assignment 1

- Tokenization (1.1)
- Basic indexing (1.2)
- Intersection search (1.3)
- Phrase search (1.4)
- Evaluation (1.5)
- Query construction (1.6)
- Large indexes on disk (1.7)
- Merging indexes (1.8)