DD2476: Search Engines and Information Retrieval Systems

Johan Boye*
KTH

Lecture 2

* Many slides inspired by Manning, Raghavan and Schütze

Indexing pipeline

Documents	Adobe · Adobe® PostScript® 3™ · You Tube · <?xml?>
Byte stream	F r i e n d s , r o m a n s , a n d c o u n t r y m e n
Token stream	**Friends romans countrymen**
Term stream	**friend roman countryman**
Inverted index	**friend** → 1 → 10 → 12 → 16 → 82 **roman** → 6 → 10 → 23 **countryman** → 9 → 10

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.
  - **Latin-1** (ISO-8859-1), 8-bit, ASCII + 128 extra chars
  - **Unicode** (109 000 code points)
  - **UTF-8** (variable-length encoding of Unicode)

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**

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

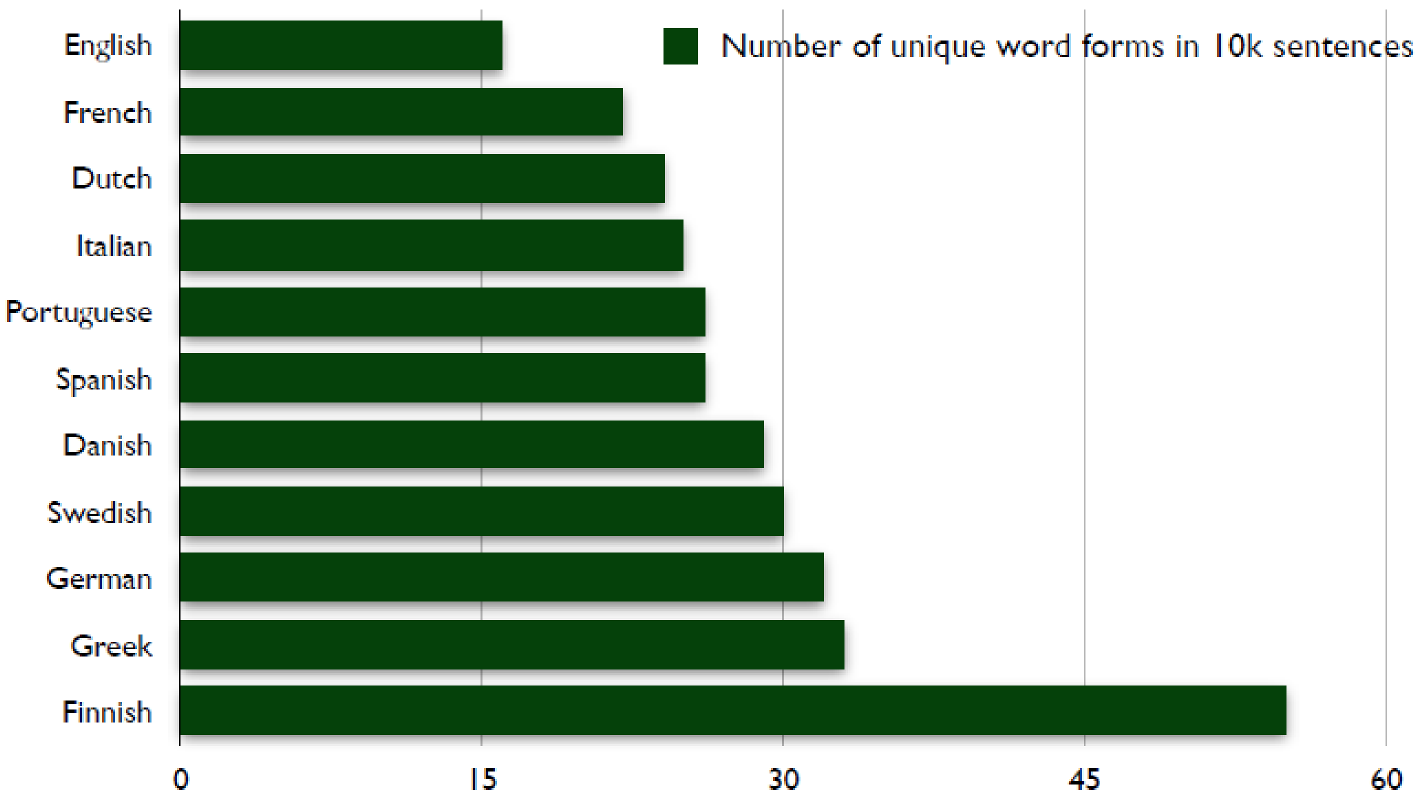
- Words are built from smaller meaningful units called **morphemes**.
- A morpheme belongs to one of two classes:
  - **stem:** the core meaning-bearing unit
  - **affix:** small units glued to the stem to signal various grammatical functions
- An affix can in its turn be classified as a
  - **prefix** (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”*

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

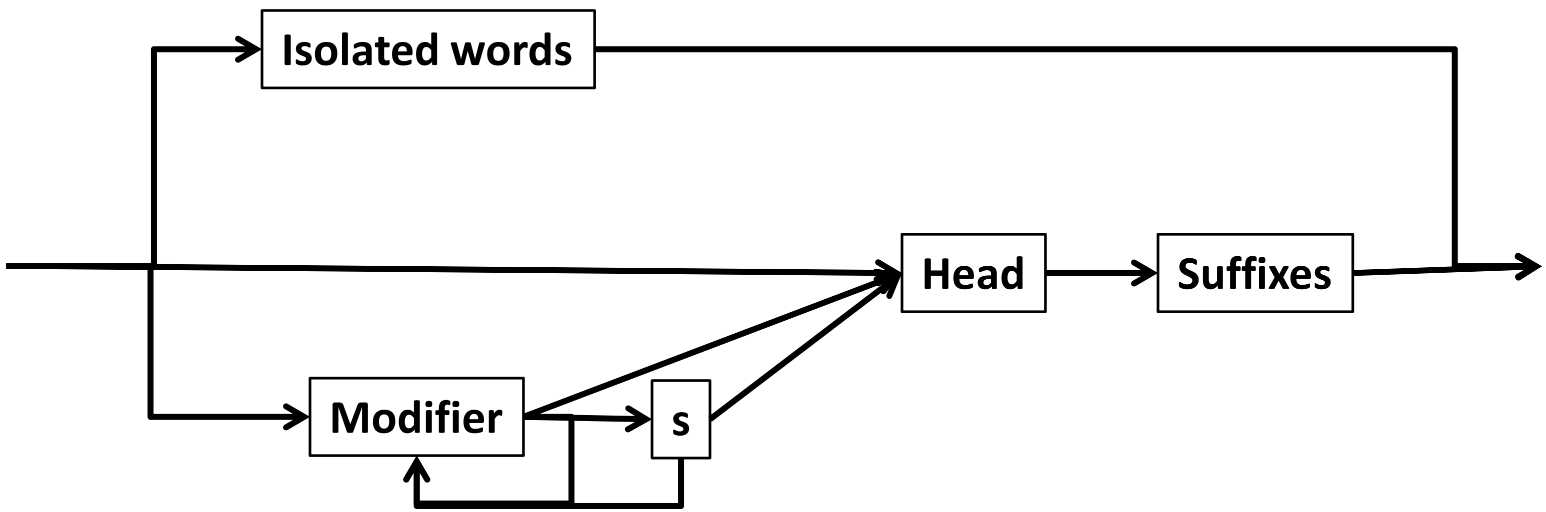
- Don’t do morphological or syntactic analysis, just **chop off the suffixes**
  - No need to know that ”foxes” is plural of ”fox”
- Much **less expensive** than lemmatization, but **can be very wrong** sometimes
  - stocks → stock, stockings → stock
- Stemming usually improves **recall** but lowers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

Language-specific issues

- Chinese and Japanese have no spaces between words:
  - 莎拉波娃现在居住在美国东南部的佛罗里达。
  - Not always guaranteed a unique tokenization
- Japanese have several alphabets
  - **Katakana** and **Hiragana** (syllabic)
  - **Kanji** (Chinese characters)
  - **Romaji** (Western characters)
  - All of these may be intermingled in the same sentence

Chinese tokenization

我喜欢新西兰花

Chinese tokenization

我喜欢新西兰花

我 | 喜欢 | 新西兰 | 花
"I like New Zealand flowers"

我 | 喜欢 | 新 | 西兰花
"I like fresh broccoli"

Chinese tokenization

The greedy matching algorithm:

1. Put a pointer in the beginning of the string
2. Find the longest prefix of the string that matches a word in the dictionary
3. Move the pointer over that prefix
4. Go to 2

我喜欢新西兰花

我 | 喜欢 | 新西兰 | 花
"I like New Zealand flowers"

我 | 喜欢 | 新 | 西兰花
"I like fresh broccoli"

Greedy matching

`Thecatinthehat` $\rightarrow$ **`The cat in the hat`**

`Thetabledownthere` $\rightarrow$ **`?`**

- Wouldn’t work so well for English
- But works very well for Chinese

Sum-up

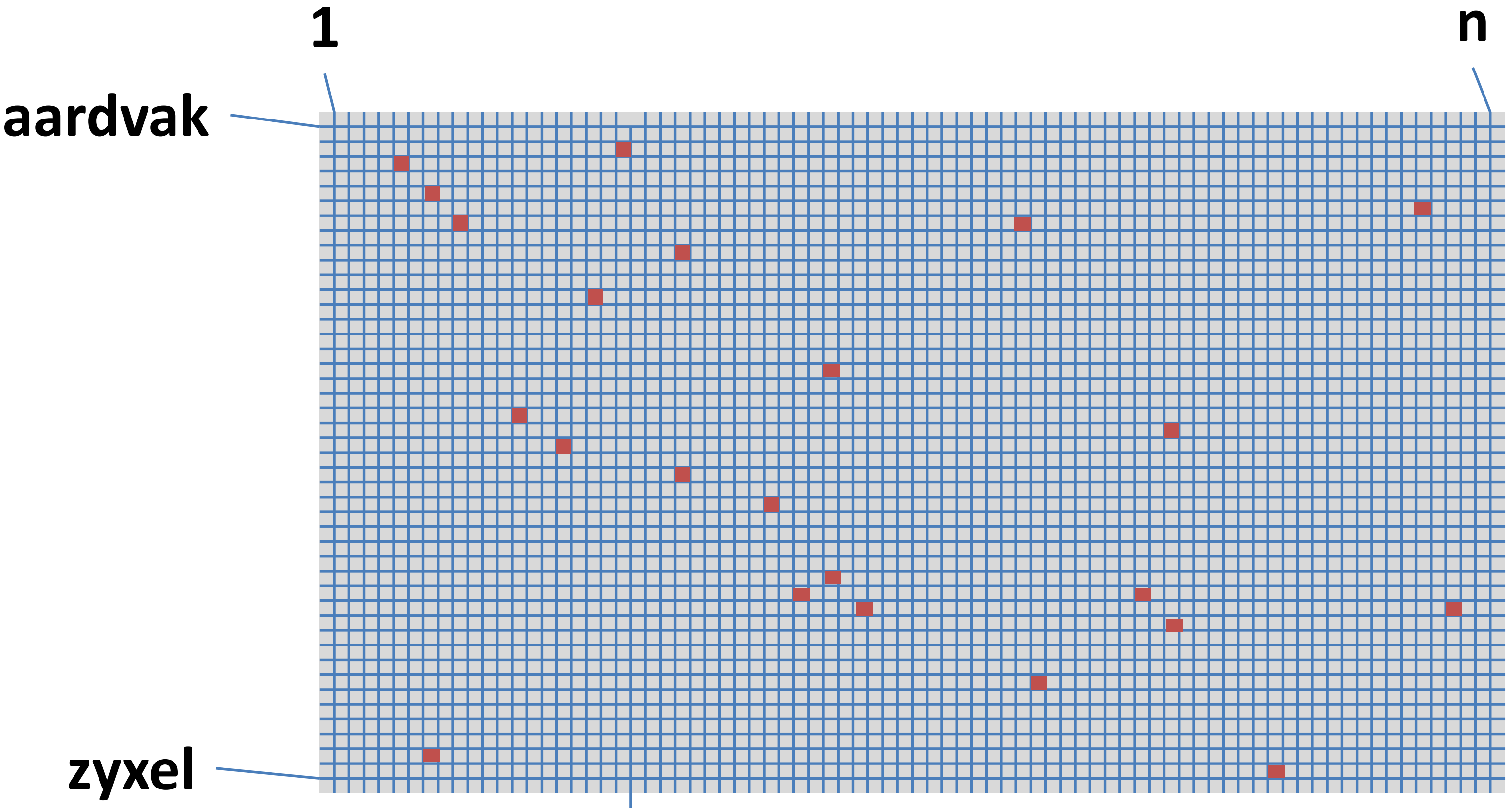
- **Reading**, **tokenizing** and **normalizing** contents of documents
  - **File types** and **character encodings**
  - Tokenization issues: **punctuation, compound words, word order, stop words**
  - Normalization issues: **diacritica, case folding, lemmatization, stemming**
- We’re ready for **indexing**

Indexing and search

- **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**



One-word queries

denmark

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

Multi-word queries

copenhagen denmark

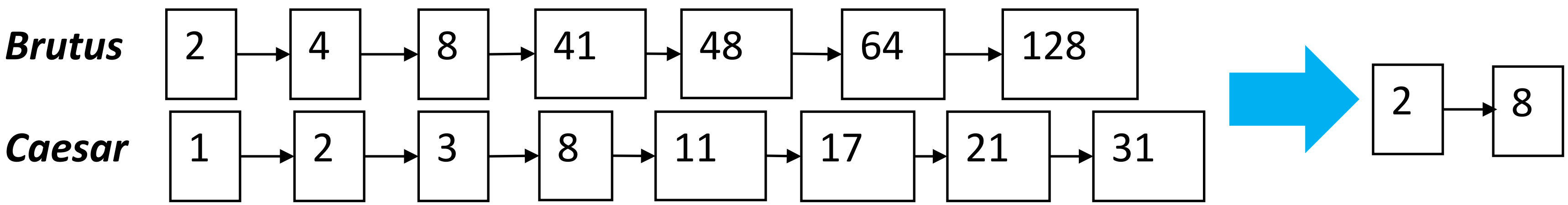
- **Intersection** query (Task 1.3)
- **Phrase** query (Task 1.4)
- **Union** query (Assignment 2)

Practical indexing

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

Intersection

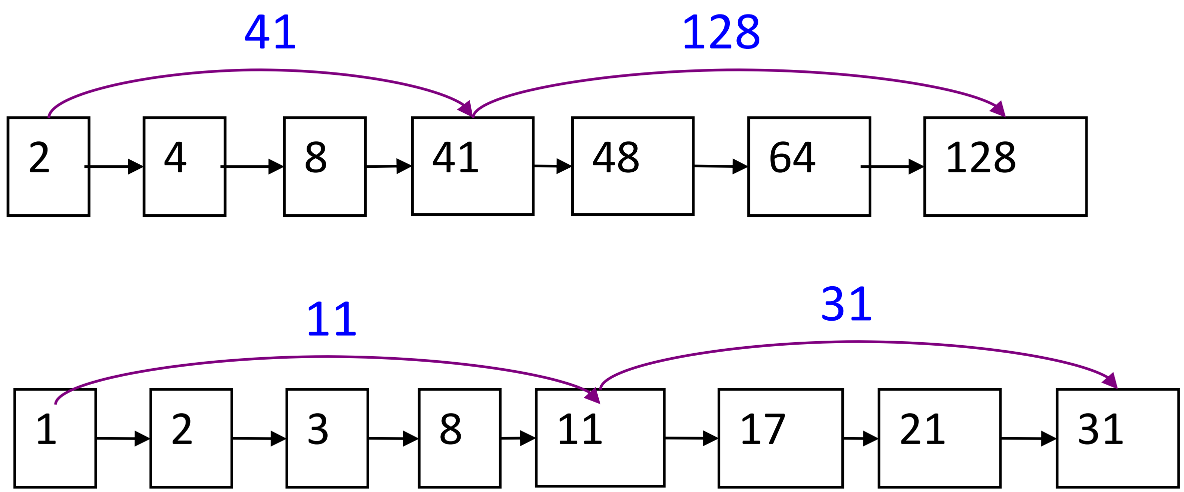
- Walk through two postings lists simultaneously



- 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

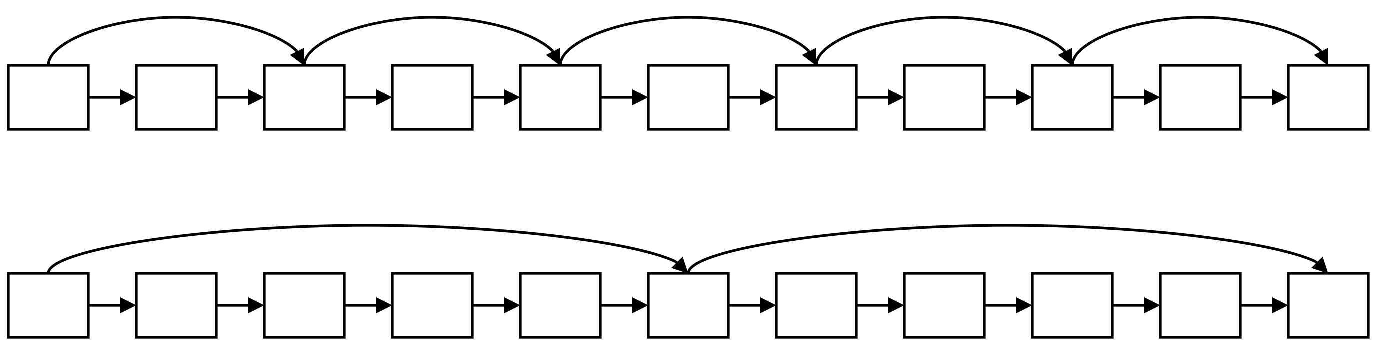
- Add skip pointers at indexing time



- 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



Phrase queries

- E.g. **”Donald Trump”**
- Should not match ”President Trump”
  - 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;

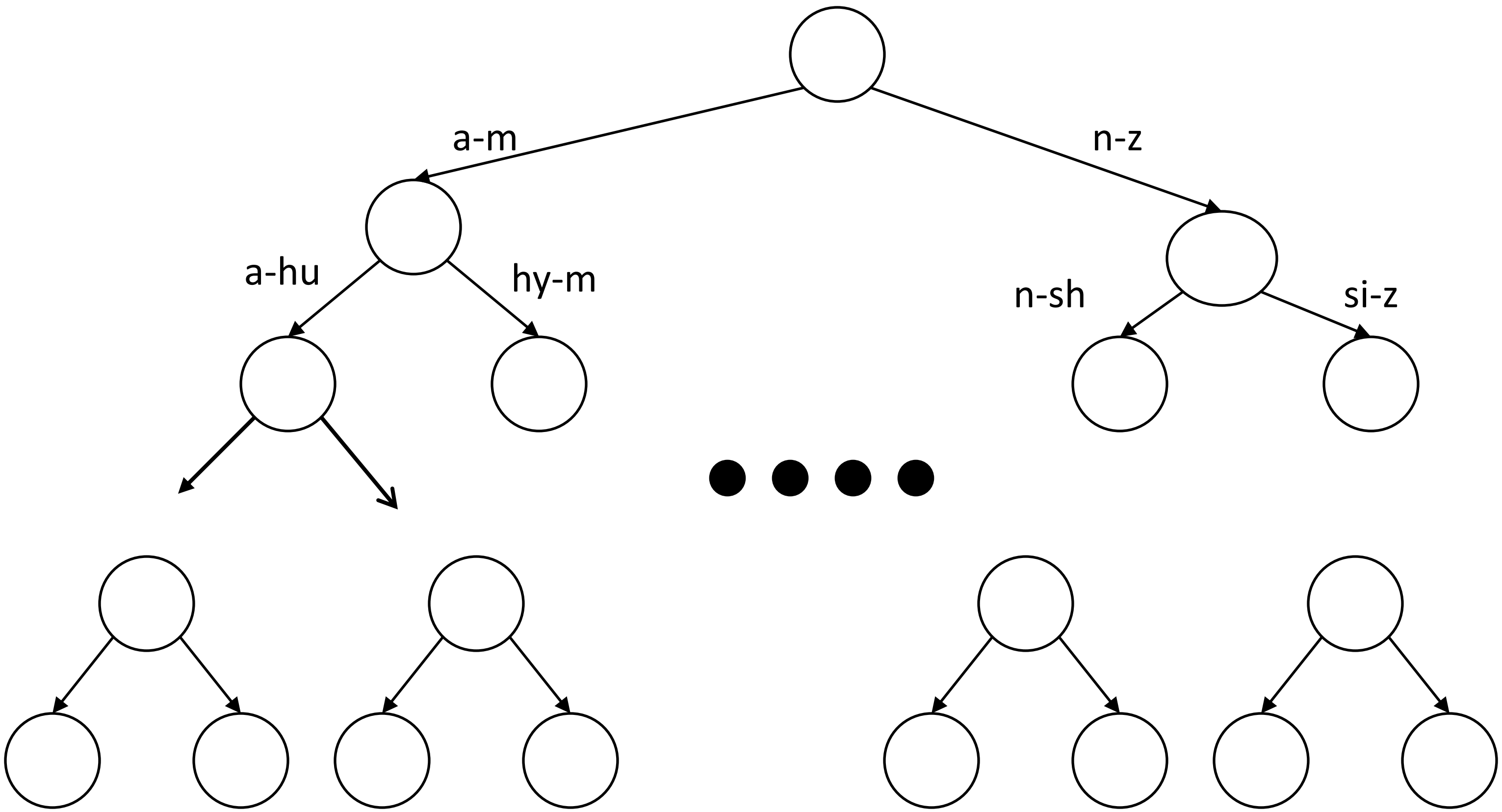
Positional index size

- 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

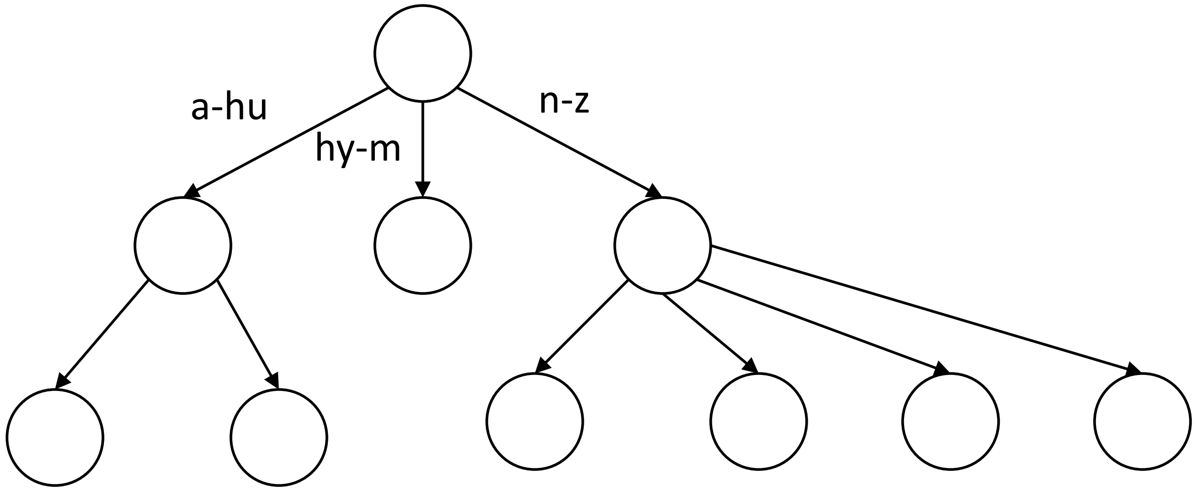
Dictionary structures

- How do we store terms in the dictionary?
- Hash tables:
  - **Lookup** in **constant time** O(1)
  - **No wildcard queries**
  - Occasionally we need to **rehash everything** as the vocabulary grows. This is **expensive**.
- Trees:
  - **Lookup** in **logarithmic time (**if tree is **balanced)**
  - Allows for **wildcard queries**
  - Requires standard (alphabetical) **order of terms**

Binary tree



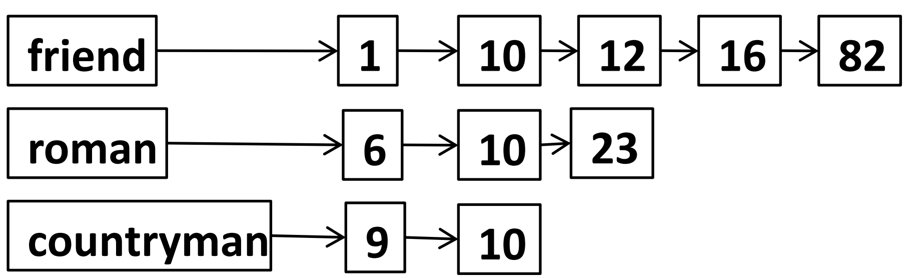
B-tree



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

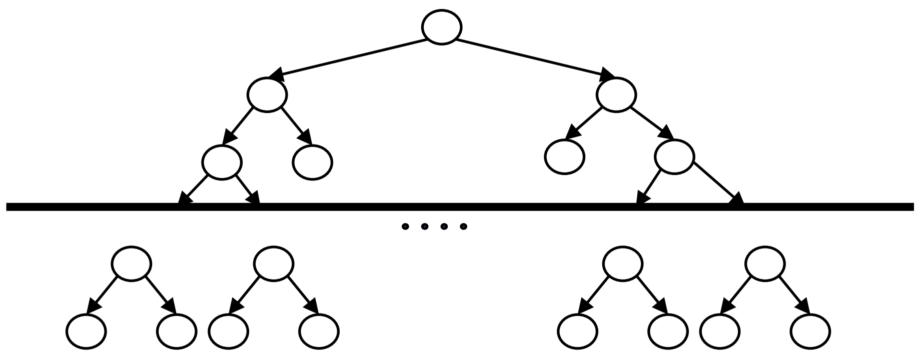
Large indexes (Task 1.7-1.8)

- 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.
- What if the index is **too large to fit** in main memory?
  - Dictionary in main memory, postings on disk
  - Dictionary (partially) on disk, postings on disk
  - Index on several disks (cluster)



Mixed MM and disk storage

- **Tree:** Nodes below a certain depth stored on disk



- **Hashtable:** All postings put on disk, hash keys in MM
- A **distributed hash table** allows keys and postings to be distributed of a large number of computers

Hardware basics

- Access to data in memory is ***much*** faster than access to data on disk.
- **Disk seeks**: No data is transferred from disk while the disk head is being positioned.
  - Therefore: Transferring **one large chunk of data** from disk to memory is faster than transferring many small chunks.
  - Disk I/O is **block-based**: Reading and writing of entire blocks (as opposed to smaller chunks).
  - Block sizes: 8KB to 256 KB.

Hardware assumptions

- In the book:

statistic	**value**
average seek time	5 ms = 5×10^{-3} s
transfer time per byte	0.02 µs = 2×10^{-8} s
processor's clock rate	10^{9} s^{-1} (1 GHz)
low-level operation (e.g., compare & swap a word)	0.01 µs = 10^{-8} s
size of main memory	several GB
size of disk space	1 TB or more

Basic indexing

- Term-document pairs are collected when documents are parsed

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	Doc #
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

Sorting step

- The list of term-doc pairs is sorted
- This must be done on disk for large lists
- Goal: Minimize the number of disk seeks

Term	Doc #
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

→

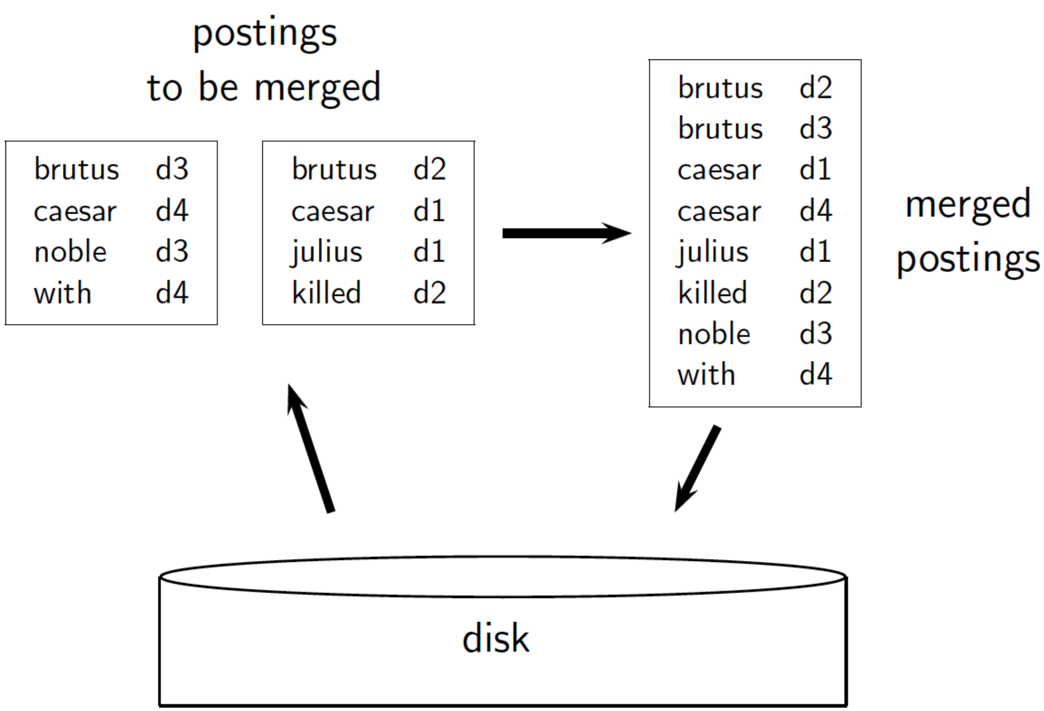
Term	Doc #
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

A bottleneck

- Say we want to sort 100,000,000 term-doc-pairs.
- A list can be sorted by $N \log_2 N$ comparison operations.
  - How much time does that take? (assume 10^{-8} s/operation)
- Suppose that each comparison additionally took 2 disk seeks
  - How much time? (assuming 5×10^{-3} /disk seek)

Blocked sort-based indexing

- (term-doc) records
  - Define a Block ~ 10M of such records
  - Accumulate postings for each block, sort, write to disk.
  - Then merge the blocks into one long sorted order.



Sorting 10 blocks of 10M records

- First, read each block and sort within:
  - Quicksort takes $N \log_2 N$ expected steps
  - In our case (10M) $\log_2$ (10M) steps
- *Exercise: estimate total time to read each block from disk and and quicksort it.*
  - assuming transfer time 2 x 10^{-8} s per byte
- 10 times this estimate – gives us 10 sorted *runs* of 10M records each.

Blocked sort-based indexing

```
BSBIndexConstruction()
  n ← 0
  while (all documents have not been processed)
  do n ← n + 1
      block ← ParseNextBlock()
      BSBI-Invert(block)
      WriteBlockToDisk(block, f_n)
  MergeBlocks(f_1, ..., f_n; f_merged)
```

From BSBI to SPIMI

- BSBI requires that the dictionary can be kept in main memory
- Alternative approach: Construct several **separate** indexes and **merge** them
  - Generate separate dictionaries for each block
  - No need to keep dictionary in main memory
  - Accumulate postings directly in postings list (as in assignment 1).
- This is called **SPIMI** – Single-Pass In-Memory Index construction (Figure 4.4)

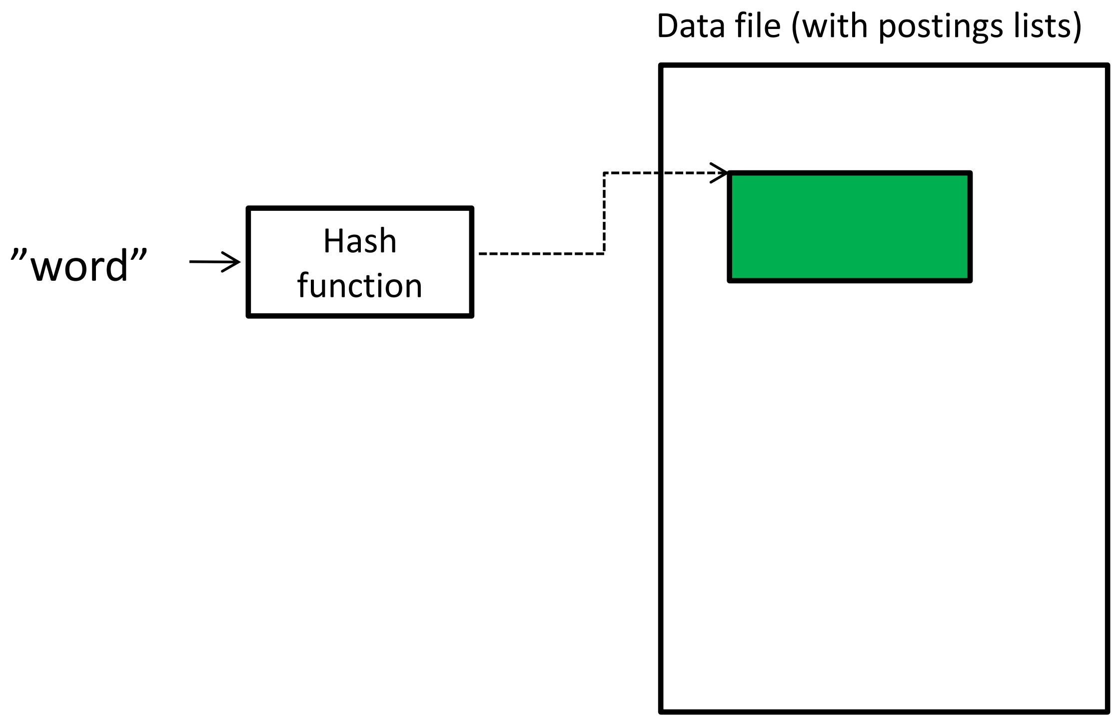
SPIMI-invert

```
SPIMI-Invert(token_stream)
output_file = NewFile()
dictionary = NewHash()
while (free memory available)
do token ← next(token_stream)
    if term(token) ∉ dictionary
      then postings_list = AddToDictionary(dictionary, term(token))
      else postings_list = GetPostingsList(dictionary, term(token))
    if full(postings_list)
      then postings_list = DoublePostingsList(dictionary, term(token))
    AddToPostingsList(postings_list, docID(token))
sorted_terms ← SortTerms(dictionary)
WriteBlockToDisk(sorted_terms, dictionary, output_file)
return output_file
```

Large indexes (Task 1.7-1.8)

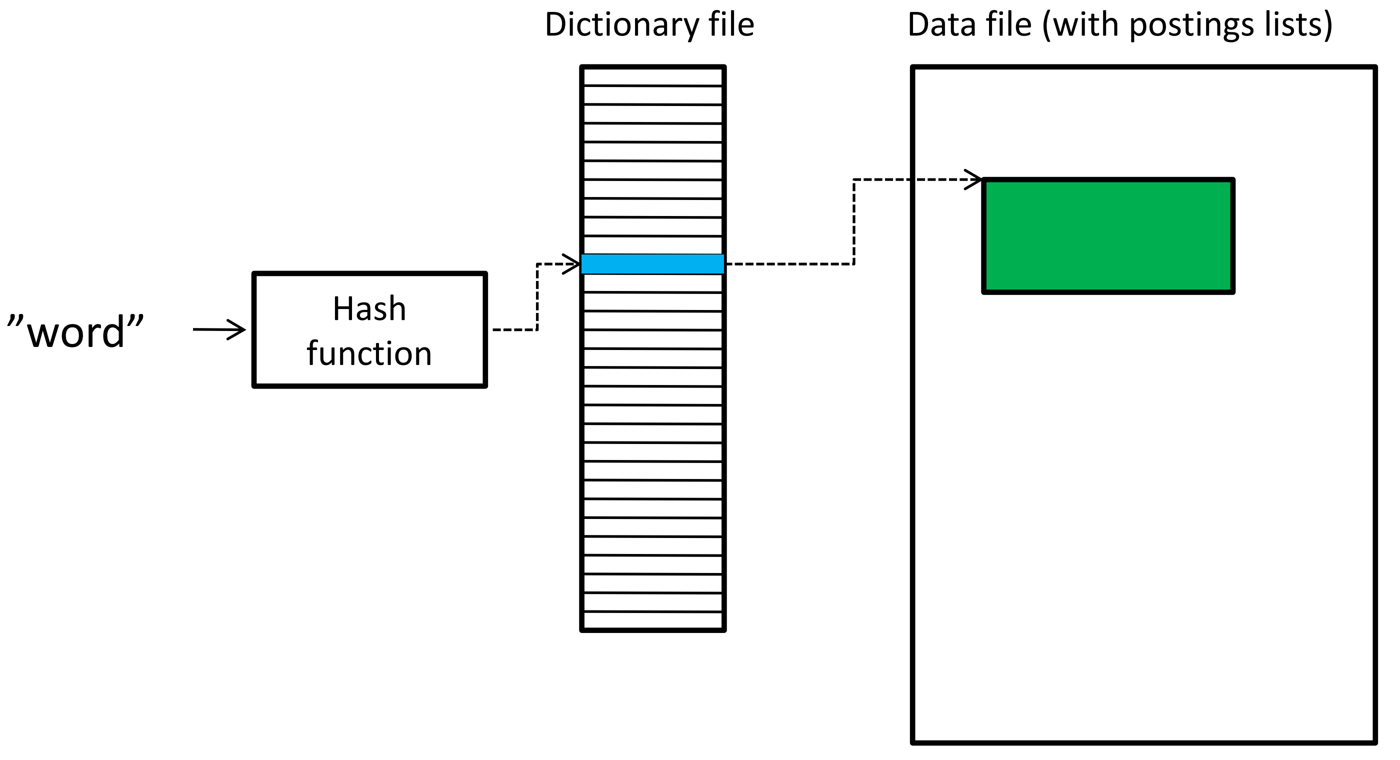
- Task 1.7 asks you to implement an index which is stored on disk
  - using any method (well, not quite…) for grade C
  - using a hash table with both dictionary and postings lists on disk for grade B
- Task 1.8 ask you to to a kind of SPIMI-invert, for grade A
  - construction of partial hash tables
  - merging of the partial hash tables into the final table

Hash tables on disk- what one would like to do



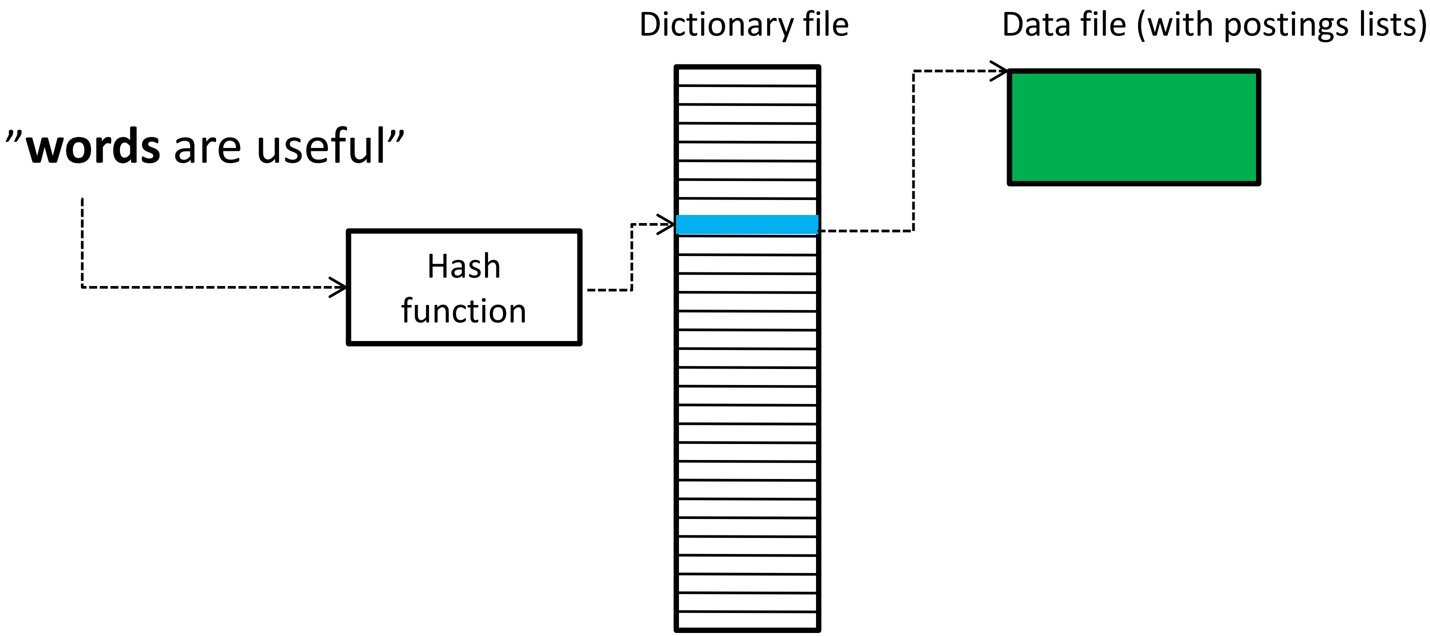
Why doesn’t this work?

Hash tables on disk- what we will do

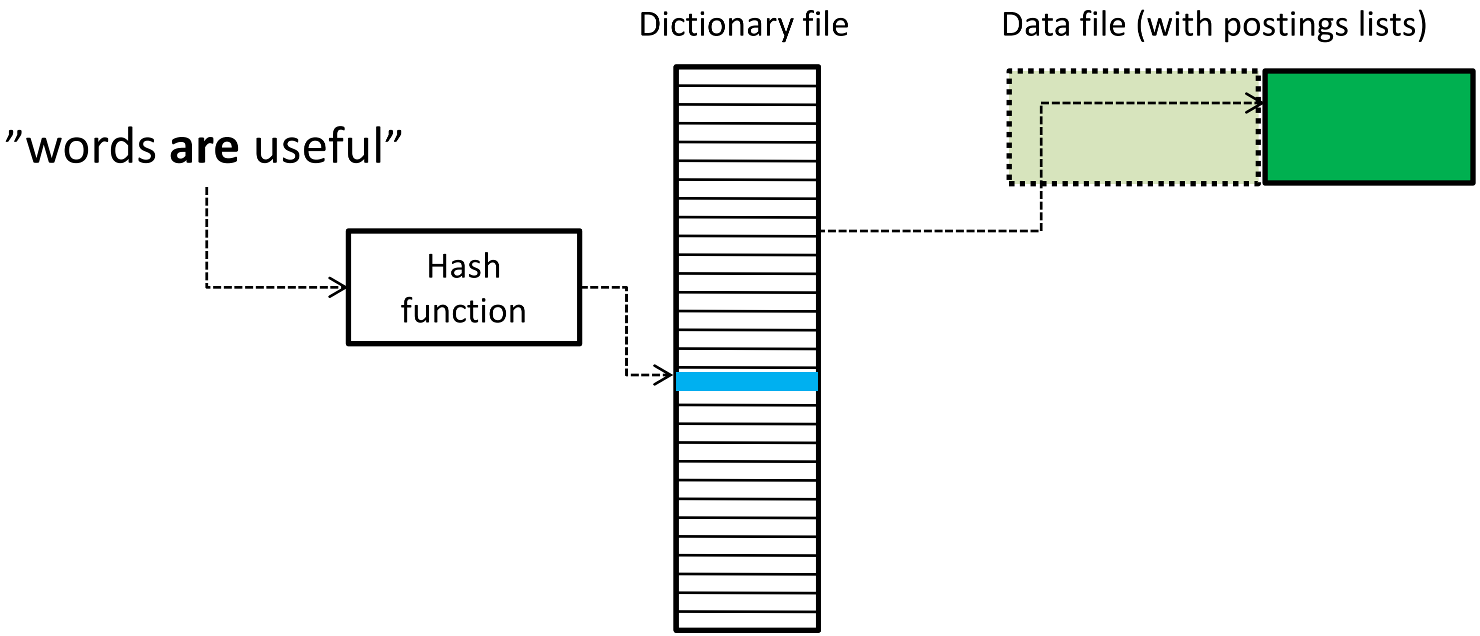


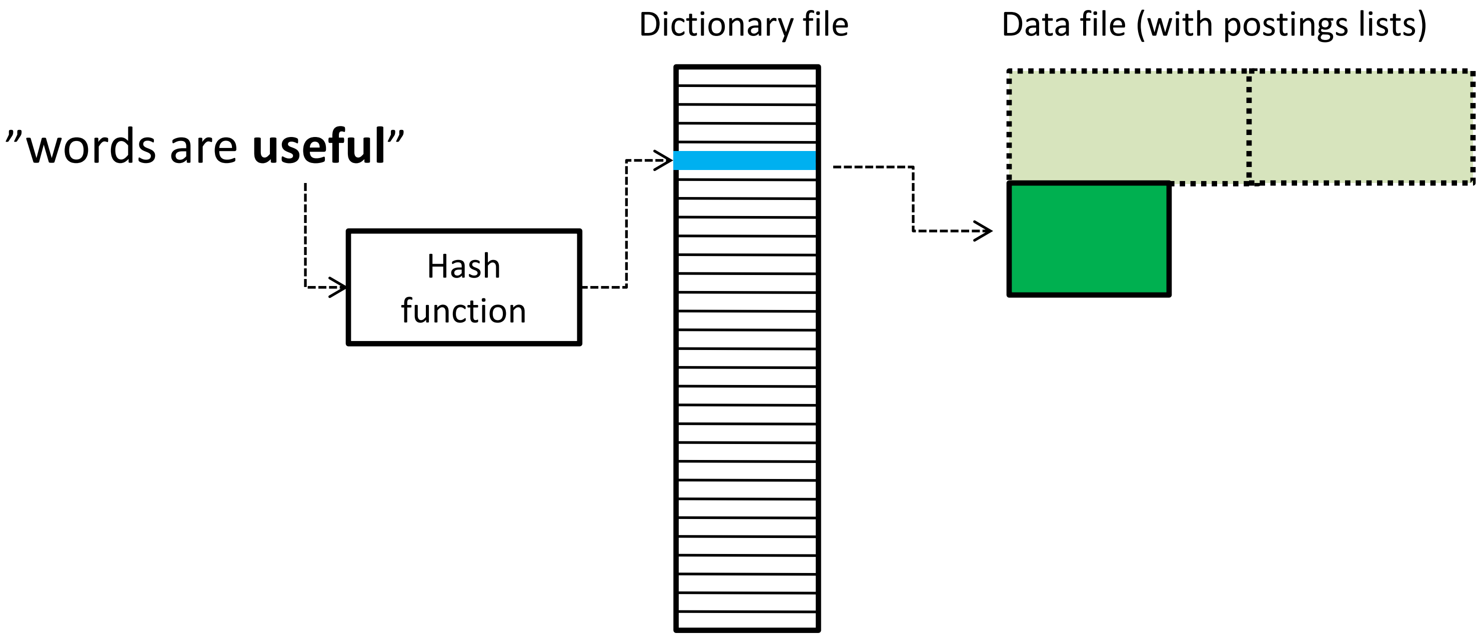
Hash table on disk

- 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.



Dictionary file
Data file (with postings lists)
"**words** are useful"
Hash function

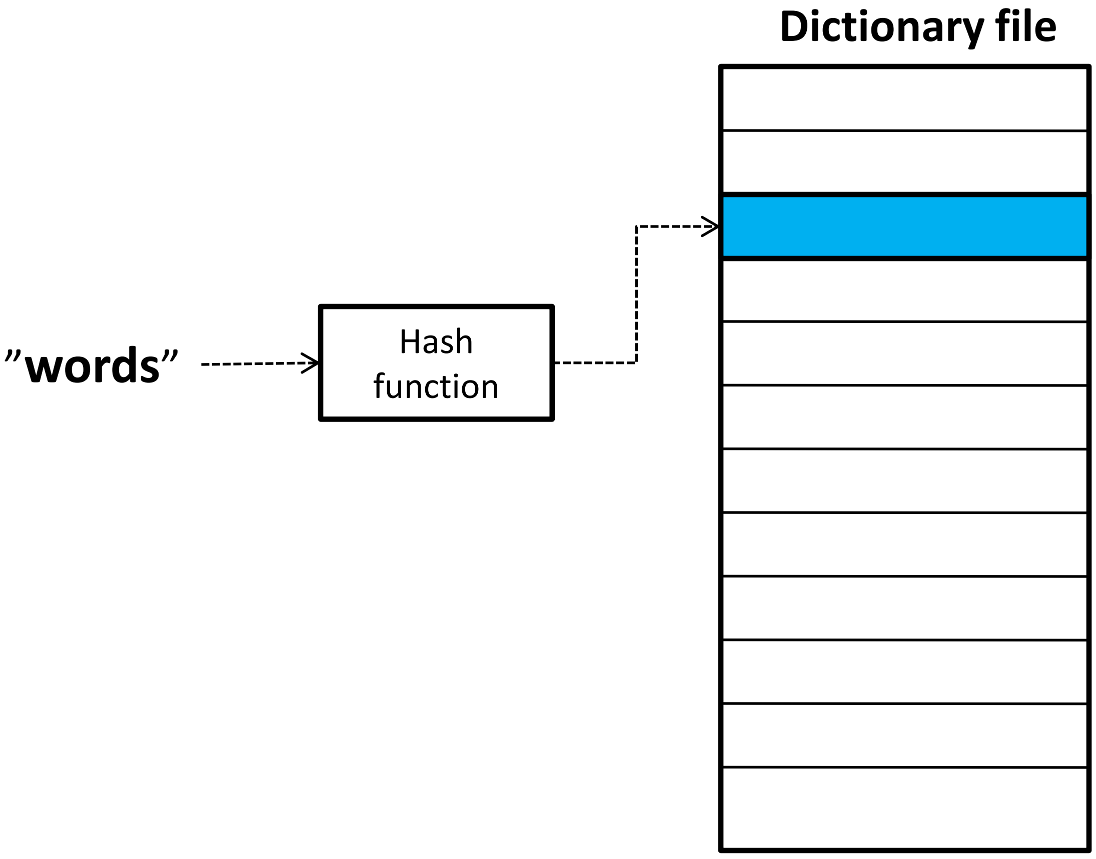

"words **are** useful"
Hash function
Dictionary file
Data file (with postings lists)


Dictionary file
Data file (with postings lists)
”words are **useful**”
Hash function

Hash table on disk

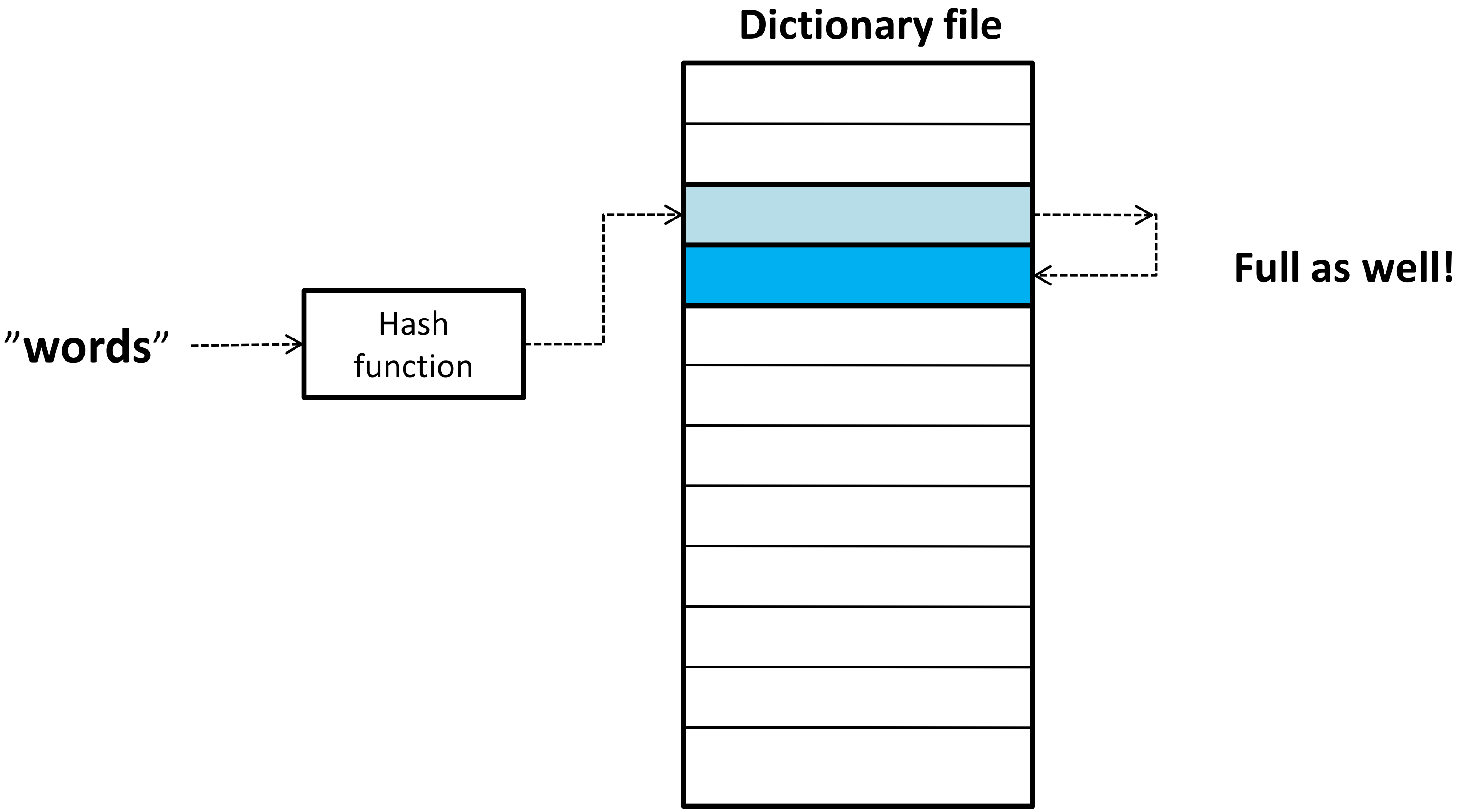
- Dictionary file
  - has a fixed size
  - will be mostly empty (load factor about 0.33)
- Data file grows dynamically
  - will be completely packed

Hash collisions

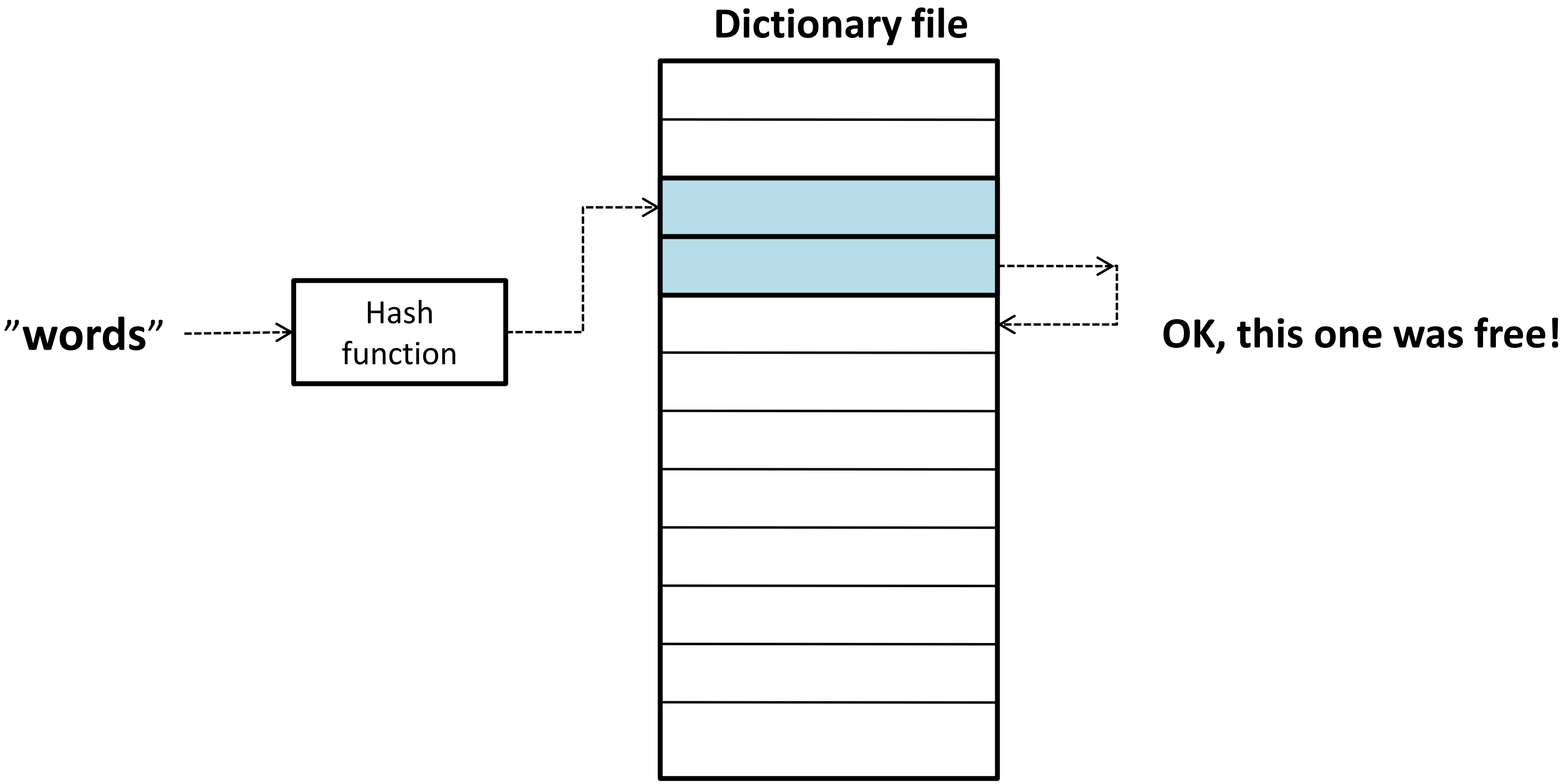


Oops, that bucket was already full… let’s try another one

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)

Dynamic indexing

- Up to now, we have assumed that collections are static.
- They rarely are:
  - 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

Logarithmic merge

- Maintain a series of indexes, each twice as large as the previous one.
- Keep smallest (Z_0) in memory
- Larger ones (I_0, I_1, …) on disk
- If Z_0 gets too big (> n), write to disk as I_0
  – or merge with I_0 (if I_0 already exists) as Z_1
- Either write merge Z_1 to disk as I_1 (if no I_1)
  – or merge with I_1 to form Z_2
- etc.

Dynamic indexing at search engines

- All the large search engines now do dynamic indexing
- Their indices have frequent incremental changes
  - News items, blogs, new topical web pages
- But (sometimes/typically) they also periodically reconstruct the index from scratch
  - Query processing is then switched to the new index, and the old index is then deleted

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)