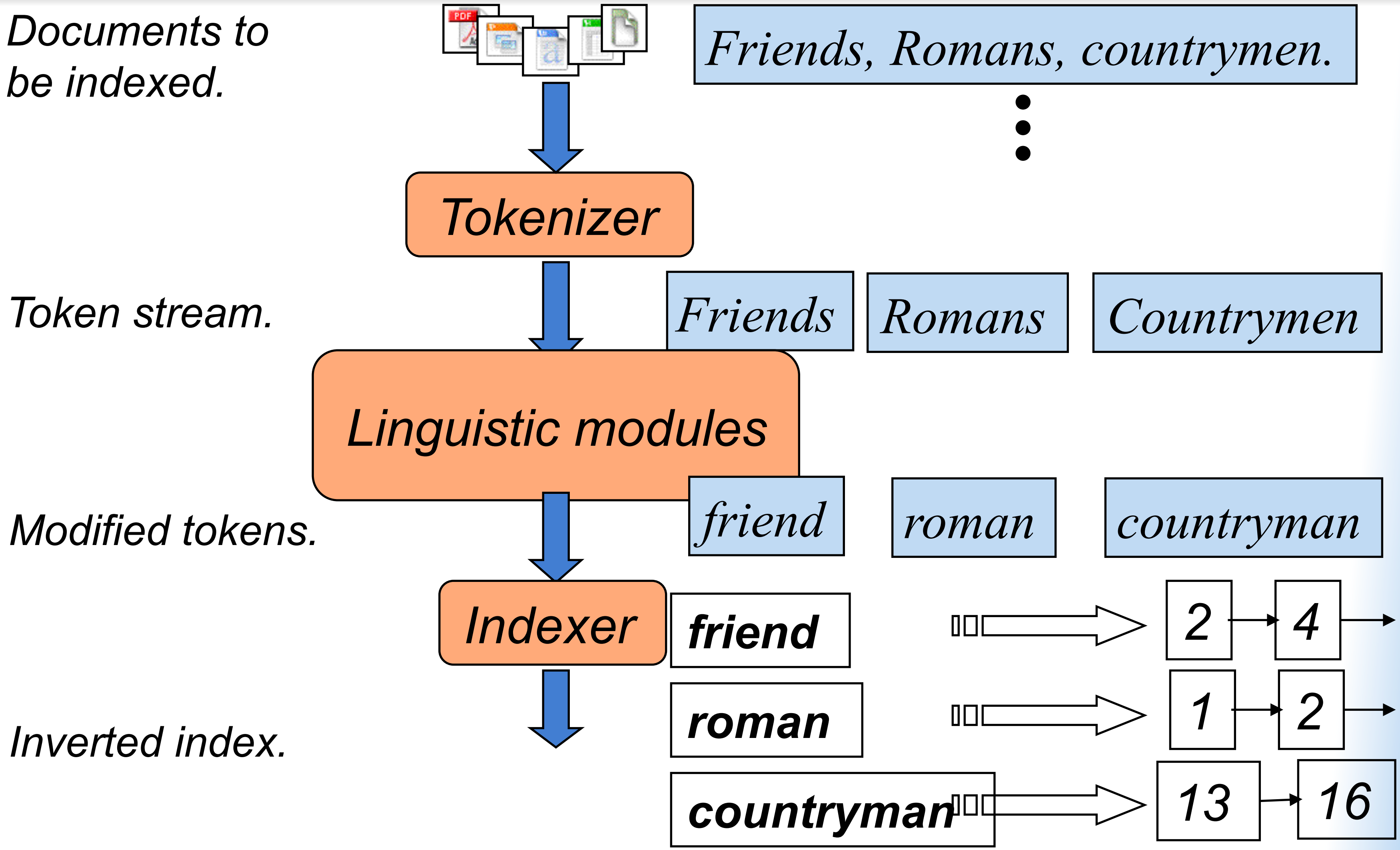
Multimedia Information Extraction and Retrieval

Indexing and Query Answering

Ralf Moeller
Hamburg Univ. of Technology

Recall basic indexing pipeline



Tokenization

- Input: "***Friends, Romans and Countrymen***"
- Output: Tokens
  - ***Friends***
  - ***Romans***
  - ***Countrymen***
- Each such token is now a candidate for an index entry, after further processing
  - Described below
- But what are valid tokens to emit?

Tokenization

- Issues in tokenization:
  - ***Finland's capital*** →
    Finland? Finlands? Finland's?
  - ***Hewlett-Packard*** →
    Hewlett and ***Packard*** as two tokens?
    - ***State-of-the-art***: break up hyphenated sequence.
    - ***co-education*** ?
    - ***the hold-him-back-and-drag-him-away-maneuver*** ?
    - It's effective to get the user to put in possible hyphens
  - ***San Francisco***: one token or two? How do you decide it is one token?

Numbers

- ***3/12/91*** ***Mar. 12, 1991***
- ***55 B.C.***
- ***B-52***
- ***My PGP key is 324a3df234cb23e***
- ***100.2.86.144***
  - Often, don't index as text.
    - But often very useful: think about things like looking up error codes/stacktraces on the web
    - (One answer is using n-grams: later)
  - Will often index "meta-data" separately
    - Creation date, format, etc.

Tokenization: Language issues

- ***L'ensemble*** → one token or two?
  - ***L*** ? ***L'*** ? ***Le*** ?
  - Want ***l'ensemble*** to match with ***un ensemble***
- German noun compounds are not segmented
  - Lebensversicherungsgesellschaftsangestellter
  - 'life insurance company employee'

Normalization

- Need to "normalize" terms in indexed text as well as query terms into the same form
  - We want to match ***U.S.A.*** and ***USA***
- We most commonly implicitly define equivalence classes of terms
  - e.g., by deleting periods in a term
- Alternative is to do asymmetric expansion:
  - Enter: ***window*** Search: ***window, windows***
  - Enter: ***windows*** Search: ***Windows, windows***
  - Enter: ***Windows*** Search: ***Windows***
- Potentially more powerful, but less efficient

Normalization: other languages

- Accents: ***résumé*** vs. ***resume***.
- Most important criterion:
  - How are your users like to write their queries for these words?
- Even in languages that standardly have accents, users often may not type them
- German: Tuebingen vs. Tübingen
  - Should be equivalent

Case folding

- Reduce all letters to lower case
  - exception: upper case (in mid-sentence?)
    - e.g., ***General Motors***
    - ***Fed*** vs. ***fed***
    - ***SAIL*** vs. ***sail***

  - Often best to lowercase everything, since users will use lowercase regardless of 'correct' capitalization…

Stop words

- With a stop list, you exclude from dictionary entirely the commonest words. Intuition:
  - They have little semantic content: *the, a, and, to, be*
  - They take a lot of space: ~30% of postings for top 30
- But the trend is away from doing this:
  - Good compression techniques means the space for including stopwords in a system is very small
  - Good query optimization techniques mean you pay little at query time for including stop words.
  - You need them for:
    - Phrase queries: “King of Denmark”
    - Various song titles, etc.: “Let it be”, “To be or not to be”
    - “Relational” queries: “flights to London”

Thesauri

- Handle synonyms and homonyms
  - Hand-constructed equivalence classes
    - e.g., ***car*** = ***automobile***
    - ***color*** = ***colour***
- Rewrite to form equivalence classes
- Index such equivalences
  - When the document contains ***automobile***, index it under ***car*** as well (usually, also vice-versa)
- Or expand query?
  - When the query contains ***automobile***, look under ***car*** as well

Lemmatization

- Reduce inflectional/variant forms to base form
- E.g.,
  - *am, are, is* → *be*
  - *car, cars, car's, cars'* → *car*
- *the boy's cars are different colors* → *the boy car be different color*
- Lemmatization implies doing "proper" reduction to dictionary headword form

Simpler Form: Stemming

- Reduce terms to their "roots" before indexing
- "Stemming" suggests crude affix chopping
  - language dependent
  - e.g., ***automate(s), automatic, automation*** all reduced to ***automat***.

for example compressed and compression are both accepted as equivalent to compress*.*

→

for exampl compress and compress ar both accept as equival to compress

Porter's Algorithm

- Common algorithm for stemming English
  - Results suggest at least as good as other stemming options
- Conventions + 5 phases of reductions
  - phases applied sequentially
  - each phase consists of a set of commands
  - sample convention: *Of the rules in a compound command, select the one that applies to the longest suffix.*

Porter’s Algorithm

- $[C](VC)^m[V]$
  - m indicates repetition, C = consonant, V = vowel
  - X denotes a sequence of Xs
- Examples:
  - m=0 TR, EE, TREE, Y, BY
  - m=1 TROUBLE, OATS, TREES, IVY
  - m=2 TROUBLES, PRIVATE, OATEN
- Conditions:
  - *S - the stem ends with S (and similarly for the other letters).
  - *v* - the stem contains a vowel.
  - *d - the stem ends with a double consonant (e.g. -TT, -SS).
  - *o - the stem ends cvc, where the second c is not W, X or Y (e.g. -WIL, -HOP).

Porter's Algorithm

Step 1a

SSES -> SS	*caresses*	*->*	*caress*
IES -> I	*ponies*	*->*	*poni*
	ties	*->*	*ti*
SS -> SS	*caress*	*->*	*caress*
S ->	*cats*	*->*	*cat*

Step 1b

(m>0) EED -> EE	*feed*	*->*	*feed*
	agreed	*->*	*agree*
*(*v*) ED ->*	*plastered*	*->*	*plaster*
	bled	*->*	*bled*
*(*v*) ING ->*	*motoring*	*->*	*motor*
	sing	*->*	*sing*

Porter's Algorithm

If the second or third of the rules in Step 1b is successful, the following is done:

```
    AT -> ATE                       conflat(ed)   ->  conflate
    BL -> BLE                       troubl(ed)    ->  trouble
    IZ -> IZE                       siz(ed)       ->  size
    (*d and not (*L or *S or *Z))
       -> single letter
                                    hopp(ing)     ->  hop
                                    fall(ing)     ->  fall
                                    hiss(ing)     ->  hiss
                                    fizz(ed)      ->  fizz
    (m=1 and *o) -> E               fail(ing)     ->  fail
                                    fil(ing)      ->  file
Step 1c
    (*v*) Y -> I                    happy         ->  happi
                                    sky           ->  sky
```

Porter's Algorithm

```
Step 2
    (m>0) ATIONAL ->   ATE           relational      ->  relate
    (m>0) TIONAL  ->   TION          conditional     ->  condition
                                     rational        ->  rational
    (m>0) ENCI    ->   ENCE          valenci         ->  valence
    (m>0) ANCI    ->   ANCE          hesitanci       ->  hesitance
    (m>0) IZER    ->   IZE           digitizer       ->  digitize
    (m>0) ABLI    ->   ABLE          conformabli     ->  conformable
    (m>0) ALLI    ->   AL            radicalli       ->  radical
    (m>0) ENTLI   ->   ENT           differentli     ->  different
    (m>0) ELI     ->   E             vileli         - >  vile
    (m>0) OUSLI   ->   OUS           analogousli     ->  analogous
    (m>0) IZATION ->   IZE           vietnamization  ->  vietnamize
    (m>0) ATION   ->   ATE           predication     ->  predicate
    (m>0) ATOR    ->   ATE           operator        ->  operate
    (m>0) ALISM   ->   AL            feudalism       ->  feudal
    (m>0) IVENESS ->   IVE           decisiveness    ->  decisive
    (m>0) FULNESS ->   FUL           hopefulness     ->  hopeful
    (m>0) OUSNESS ->   OUS           callousness     ->  callous
    (m>0) ALITI   ->   AL            formaliti       ->  formal
    (m>0) IVITI   ->   IVE           sensitiviti     ->  sensitive
    (m>0) BILITI  ->   BLE           sensibiliti     ->  sensible
```

Porter's Algorithm

```
Step 3
    (m>0) ICATE ->   IC                triplicate     ->  triplic
    (m>0) ATIVE ->                     formative      ->  form
    (m>0) ALIZE ->   AL                formalize      ->  formal
    (m>0) ICITI ->   IC                electriciti    ->  electric
    (m>0) ICAL  ->   IC                electrical     ->  electric
    (m>0) FUL   ->                     hopeful        ->  hope
    (m>0) NESS  ->                     goodness       ->  good
Step 4
    (m>1) AL    ->                     revival        ->  reviv
    (m>1) ANCE  ->                     allowance      ->  allow
    (m>1) ENCE  ->                     inference      ->  infer
    (m>1) ER    ->                     airliner       ->  airlin
    (m>1) IC    ->                     gyroscopic     ->  gyroscop
    (m>1) ABLE  ->                     adjustable     ->  adjust
    (m>1) IBLE  ->                     defensible     ->  defens
    (m>1) ANT   ->                     irritant       ->  irrit
    (m>1) EMENT ->                     replacement    ->  replac
    (m>1) MENT  ->                     adjustment     ->  adjust
    (m>1) ENT   ->                     dependent      ->  depend
    (m>1 and (*S or *T)) ION ->        adoption       ->  adopt
    (m>1) OU    ->                     homologou      ->  homolog
    (m>1) ISM   ->                     communism      ->  commun
    (m>1) ATE   ->                     activate       ->  activ
    (m>1) ITI   ->                     angulariti     ->  angular
    (m>1) OUS   ->                     homologous     ->  homolog
    (m>1) IVE   ->                     effective      ->  effect
    (m>1) IZE   ->                     bowdlerize     ->  bowdler
```

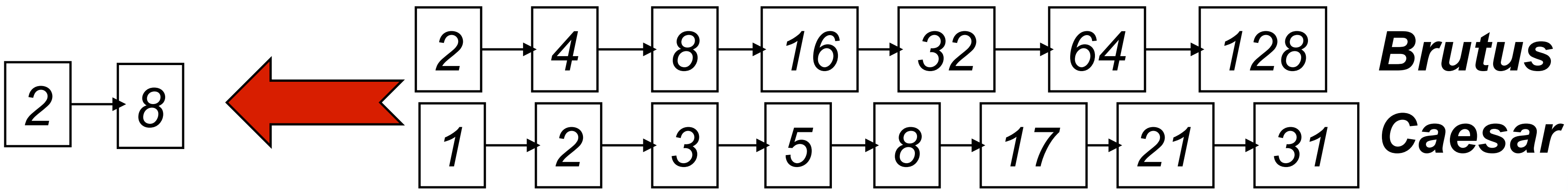
Porter's Algorithm

```
Step 5a

    (m>1) E      ->                    probate         ->   probat
                                       rate            ->   rate
    (m=1 and not *o) E ->              cease           ->   ceas

Step 5b

    (m > 1 and *d and *L) -> single letter
                                       controll        ->   control
                                       roll            ->   roll
```

Faster postings merges: Skip pointers

Recall basic merge

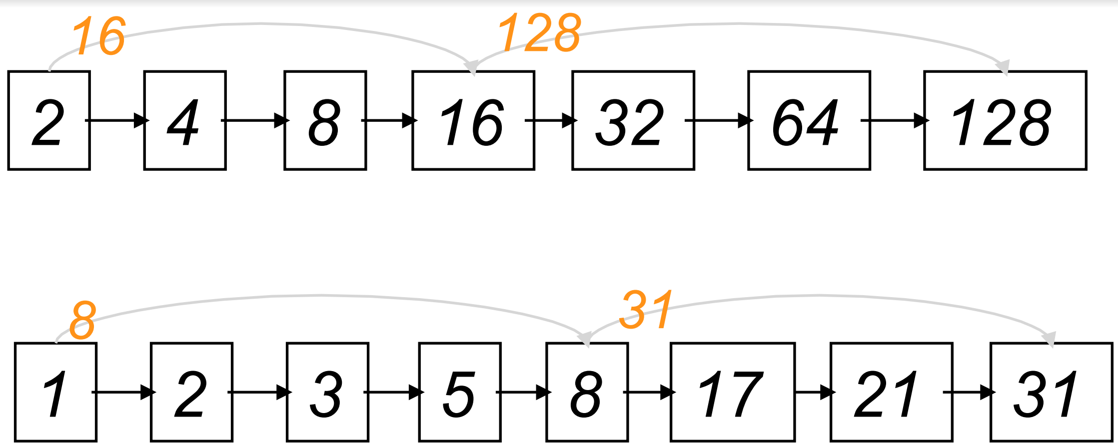
- Walk through the two postings simultaneously, in time linear in the total number of postings entries



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

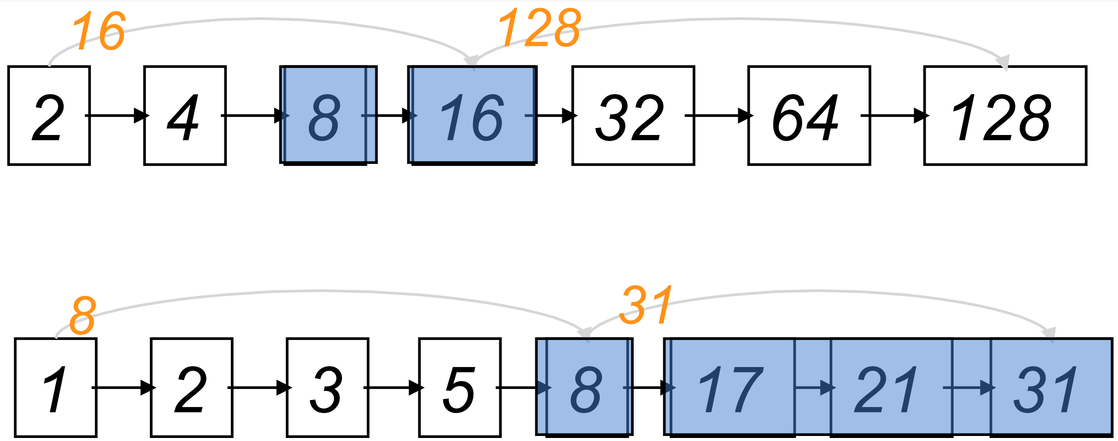
Can we do better?
Yes, if index isn't changing too fast.

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



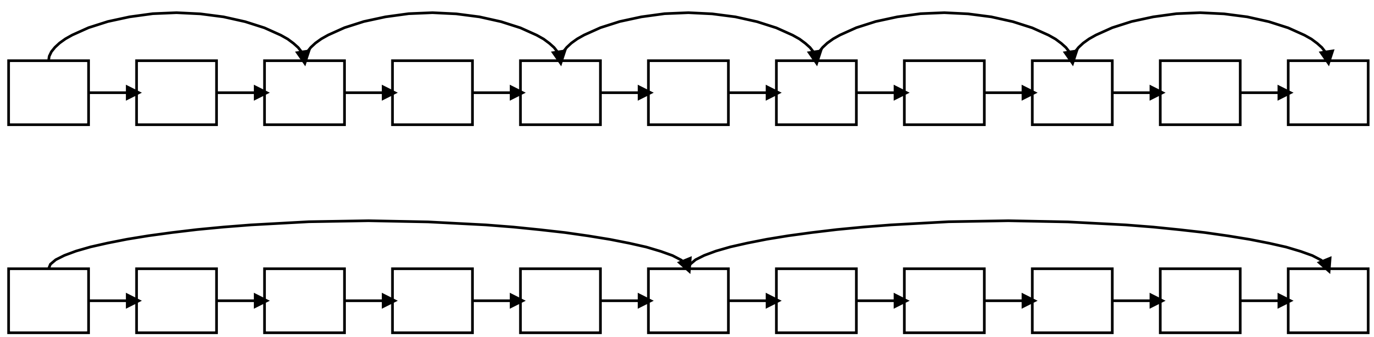
*Suppose we've stepped through the lists until we process **8** on each list.*

*When we get to **16** on the top list, we see that its successor is **32**.*

*But the skip successor of **8** on the lower list is **31**, so we can skip ahead past the intervening postings.*

Where do we place skips?

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



Placing skips

- Simple heuristic: for postings of length *L*, use $\sqrt{L}$ evenly-spaced skip pointers.
- This ignores the distribution of query terms.
- 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
  - The cost of loading a bigger postings list outweighs the gain from quicker in memory merging

Phrase queries

Phrase queries

- Want to answer queries such as "***stanford university***" – as a phrase
- Thus the sentence *"I went to university at Stanford"* is not a match.
  - The concept of phrase queries has proven easily understood by users; about 10% of web queries are phrase queries
- No longer suffices to store only <*term* : *docs*> entries

A first attempt: Biword indexes

- Index every consecutive pair of terms in the text as a phrase
- For example the text “Friends, Romans, Countrymen” would generate the biwords
  - ***friends romans***
  - ***romans countrymen***
- Each of these biwords is now a dictionary term
- Two-word phrase query-processing is now immediate.

Longer phrase queries

- Longer phrases are processed as follows:
- ***stanford university palo alto*** can be broken into the Boolean query on biwords:

stanford university *AND* ***university palo*** *AND* ***palo alto***

Without the docs, we cannot verify that the docs matching the above Boolean query do contain the phrase.

Can have false positives!

Extended biwords

- Parse the indexed text and perform part-of-speech-tagging (POST).
- Bucket the terms into (say) Nouns (N) and articles/prepositions (X).
- Now deem any string of terms of the form NX*N to be an <u>extended biword</u>.
  - Each such extended biword is now made a term in the dictionary.
- Example: ***catcher in the rye***

  N X X N
- Query processing: parse it into N's and X's
  - Segment query into enhanced biwords
  - Look up index

Issues for biword indexes

- False positives, as noted before
- Index blowup due to bigger dictionary

- For extended biword index, parsing longer queries into conjunctions:
  - E.g., the query ***tangerine trees and marmalade skies*** is parsed into
  - ***tangerine trees*** *AND* ***trees and marmalade*** *AND* ***marmalade skies***

- No standard solution (for all biwords)

Solution 2: Positional indexes

- Store, for each ***term***, entries of the form:

  <number of docs containing ***term***;

  doc1: position1, position2 … ;

  doc2: position1, position2 … ;

  etc.>

Positional index example

*<**be**: 993427;*
1: 7, 18, 33, 72, 86, 231;
2: 3, 149;
4: 17, 191, 291, 430, 434;
5: 363, 367, …>

*Which of docs 1,2,4,5 could contain "**to be or not to be**"?*

- Can compress position values/ offsets
- Nevertheless, this expands postings storage *substantially*

Processing a phrase query

- Extract inverted index entries for each distinct term: ***to, be, or, not.***
- Merge 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

Proximity queries

- LIMIT! /3 STATUTE /3 FEDERAL /2 TORT
  Here, /*k* means "within *k* words of".
- Clearly, positional indexes can be used for such queries; biword indexes cannot.
- Exercise: Adapt the linear merge of postings to handle proximity queries. Can you make it work for any value of *k*?

Positional index size

- You can compress position values/offsets:
- Nevertheless, a positional index expands postings storage *substantially*
- Nevertheless, it is now standardly used because of the power and usefulness of phrase and proximity queries … whether used explicitly or implicitly in a ranking retrieval system.

Positional index size

- Need an entry for each occurrence, not just once per document
- Index size depends on average document size **Why?**
  - Average web page has <1000 terms
  - SEC filings, books, even some epic poems … easily 100,000 terms
- Consider a term with frequency 0.1%

Document size	*Postings*	*Positional postings*
1000	*1*	*1*
100,000	*1*	*100*

Rules of thumb

- A positional index 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

Wild-card queries: *

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

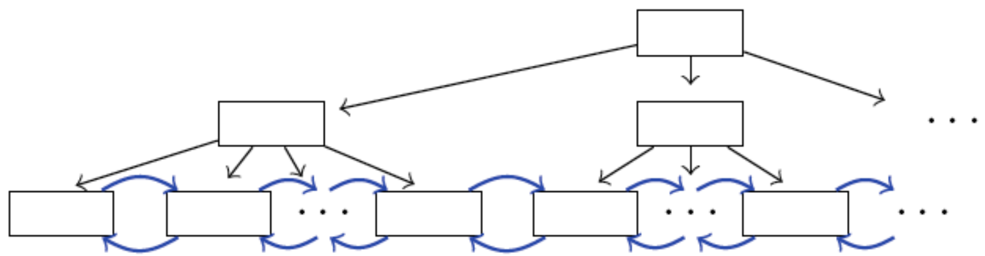
  Can retrieve all words in range: ***nom ≤ w < non.***

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

B–tree

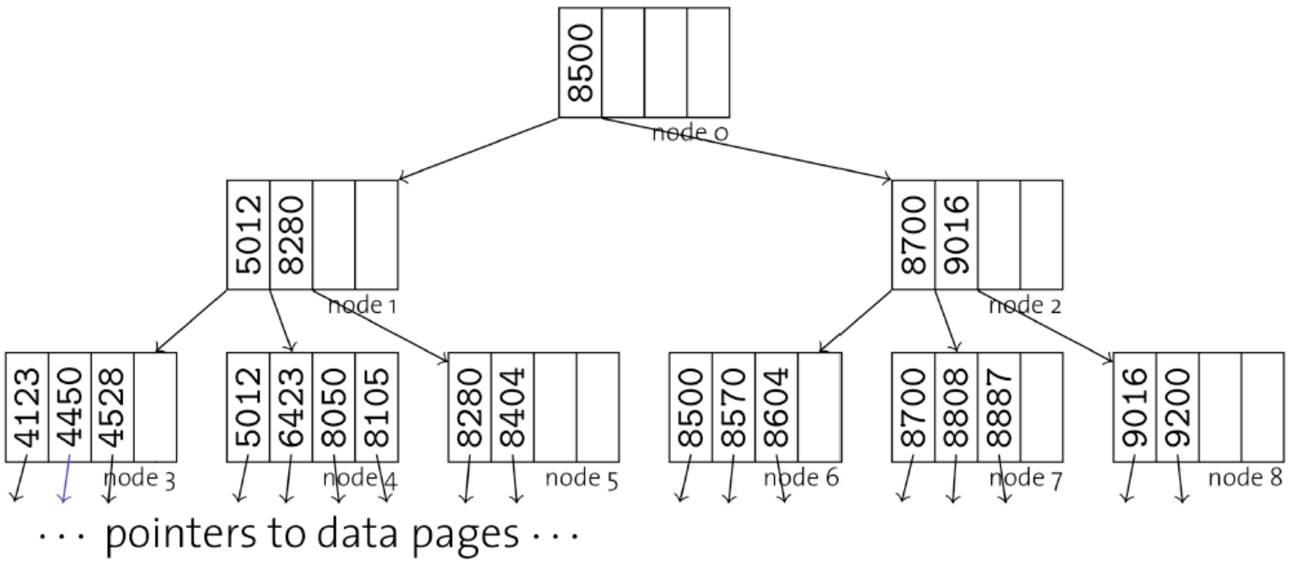
- Binary tree data structure
- Optimized for page–oriented storage of data on harddisks

- Original version: **B-tree**: R. Bayer and E. M. McCreight. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica*, vol. 1, no. 3, September 1972.
- leaf nodes are, generally, **not** in sequential order on disk,
- leaves are connected to form a **double-linked list**:[2]



B–tree: Central idea by example

Insert: Examples (Insert without Split)



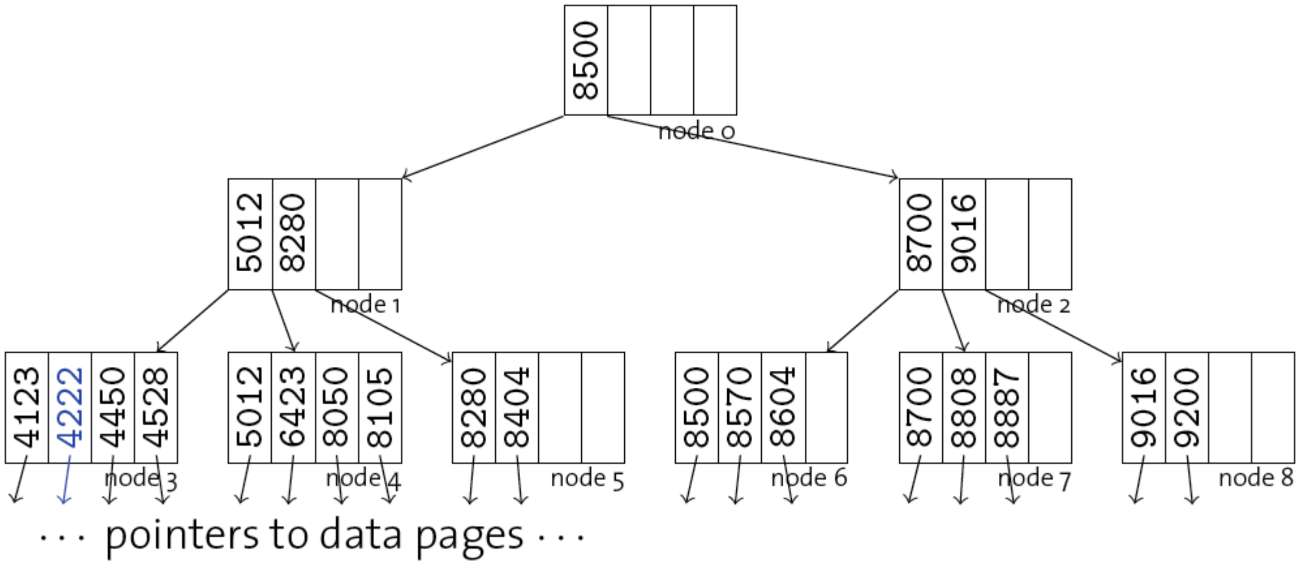
Insert new entry with key 4222.

→ Enough space in node 3, simply insert.

→ Keep entries **sorted within nodes**.

B-tree: Central idea by example

Insert: Examples (Insert without Split)



Insert new entry with key 4222.

⟶ Enough space in node 3, simply insert.

⟶ Keep entries **sorted within nodes**.

Query processing

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

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

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

B-trees handle *'s at the end of a query term

- How can we handle *'s in the middle of query term?
  - (Especially multiple *'s)
- The solution: transform every wild-card query so that the *'s occur at the end
- This gives rise to the Permuterm Index.

Permuterm index

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

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

  Exercise!

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

Permuterm query processing

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

Empirical observation for English.

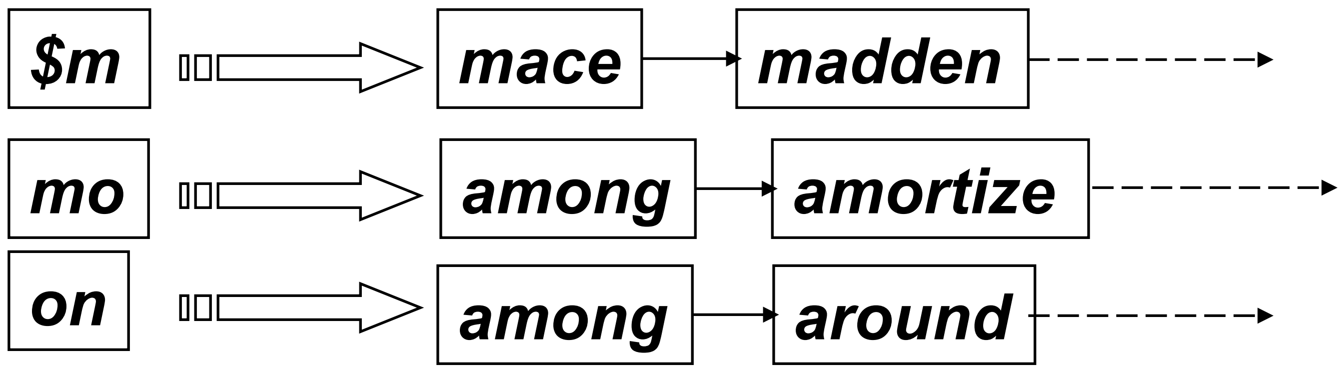
Bigram indexes

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

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

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

Bigram index example



Processing *n*-gram wild-cards

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