Introduction to

Information Retrieval

Lectures 6: Dictionaries and tolerant retrieval

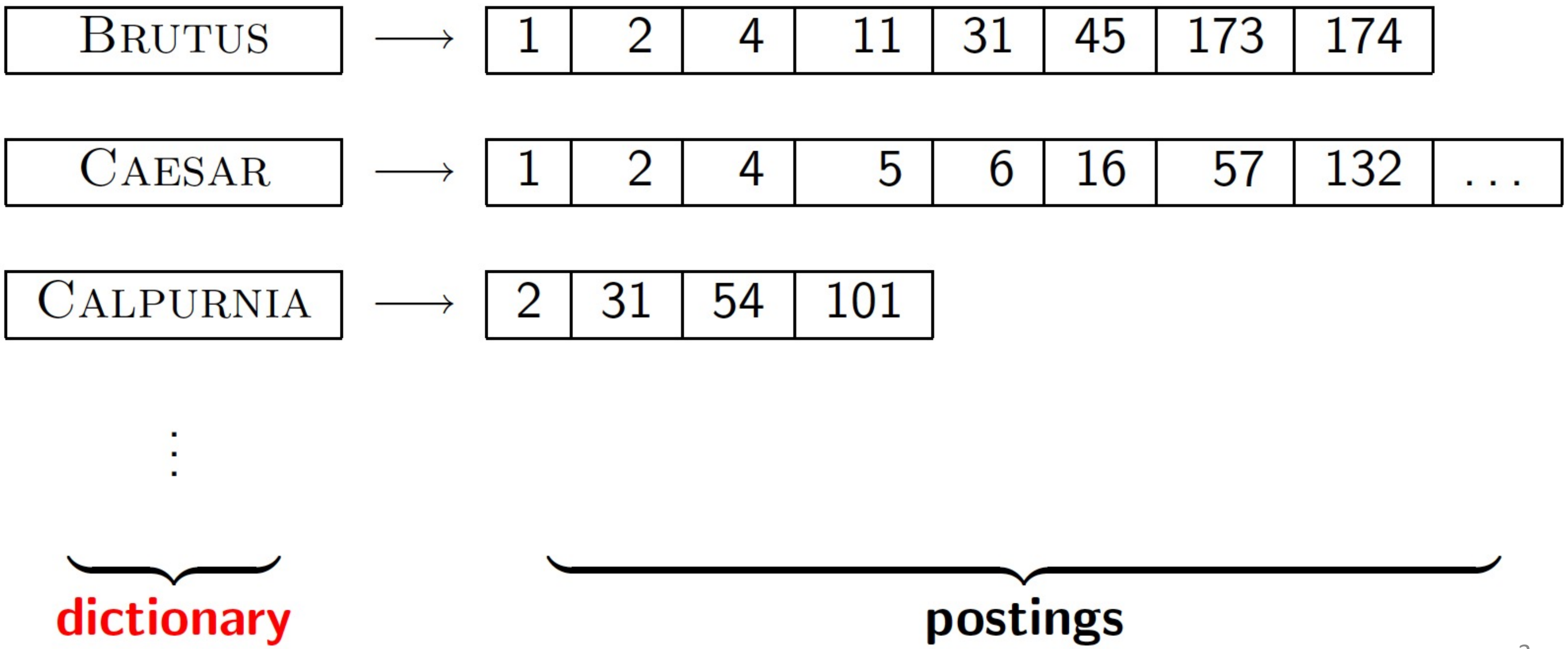
Dictionary

- Given a query and an inverted index, first tasks:
  - Determine whether each query term exists in the vocabulary
  - If yes, identify the pointers to the corresponding postings lists

- The vocabulary lookup operation uses a data structure called *dictionary*.

Dictionary data structures for inverted indexes

- The dictionary data structure stores the term vocabulary, document frequency, pointers to each postings list … in what data structure?



A naïve dictionary

- An array of struct:

term	document frequency	pointer to postings list
a	656,265	$\longrightarrow$
aachen	65	$\longrightarrow$
...	...	...
zulu	221	$\longrightarrow$
char[20]	int	Postings *
20 bytes	4/8 bytes	4/8 bytes

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

Dictionary data structures

- Two main choices:
  - Hashtables
  - Trees
- Some IR systems use hashtables, some trees

Hashtables

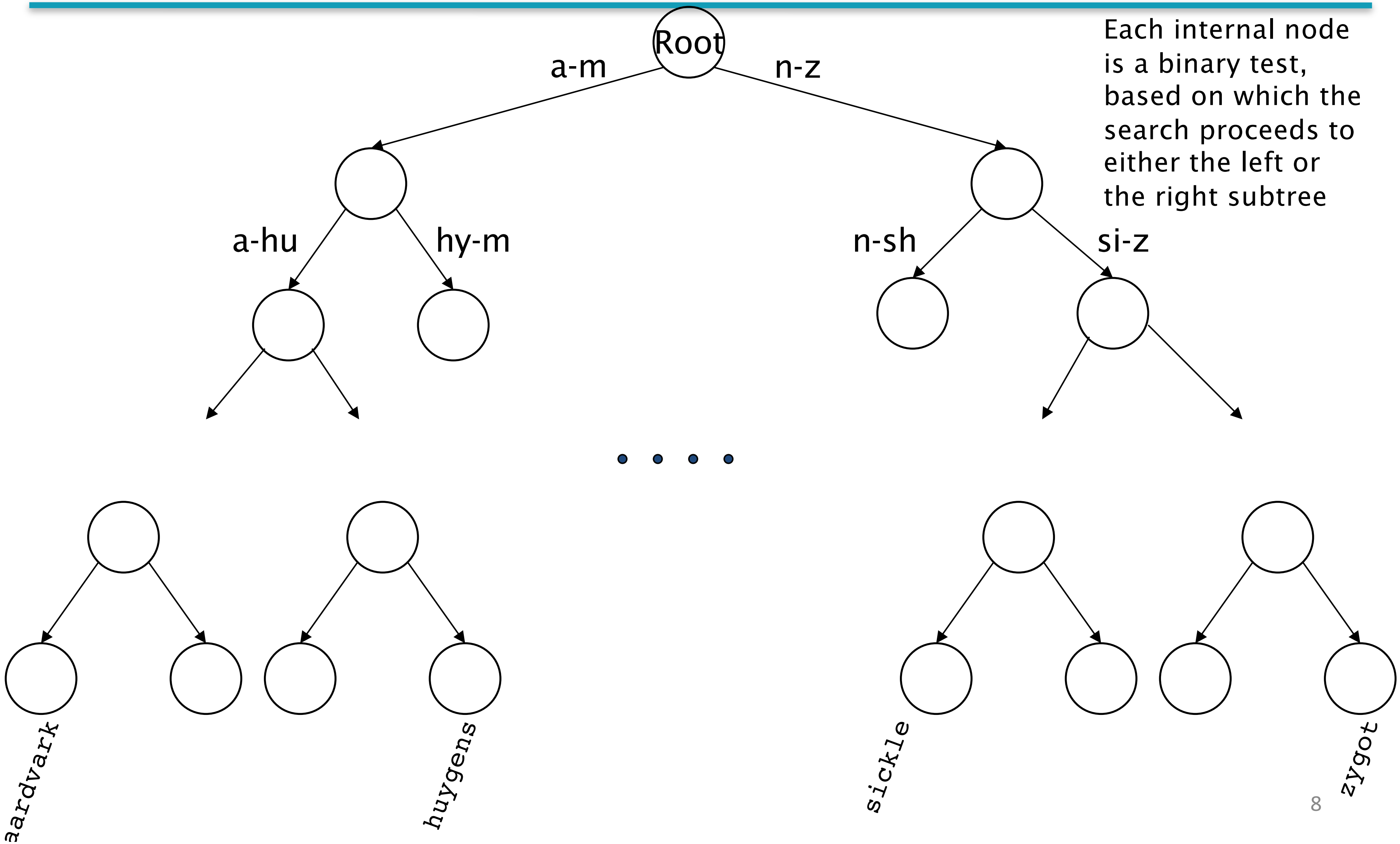
- Each vocabulary term is hashed to an integer over a large enough space that hash collisions are unlikely
  - (Assumed: you are familiar with hashtables)
- Pros:
  - Lookup is faster than for a tree: O(1)
- Cons:
  - No easy way to find minor variants (judgment/judgement) since they could be hashed to very different indexes:
  - No easy way to handle wild-card queries / prefix search
  - If vocabulary keeps growing, need to occasionally do the expensive operation of rehashing *everything*

Trees

- Simplest: binary tree

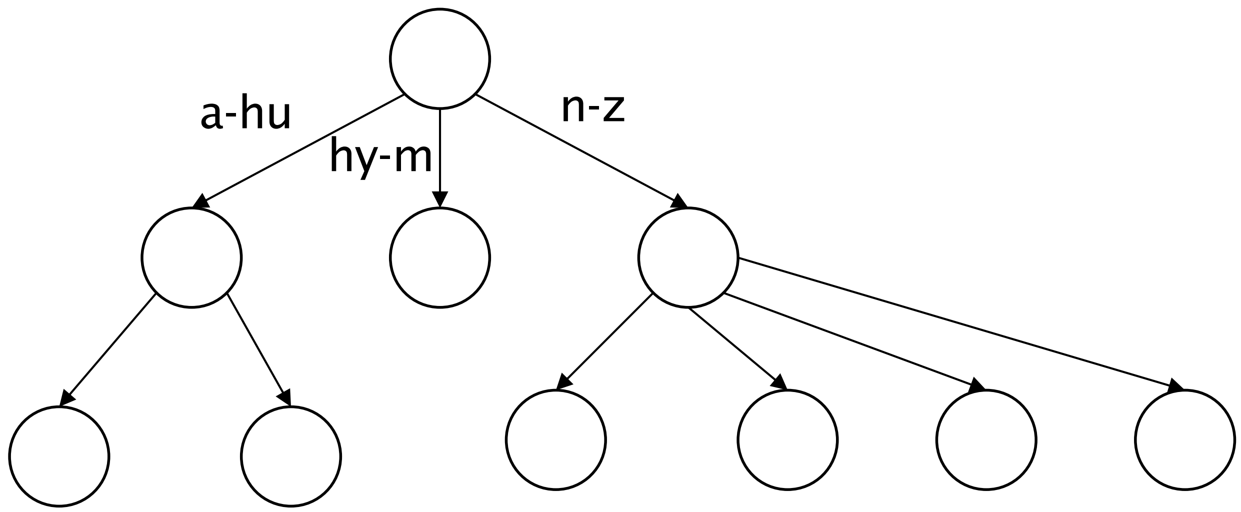
- More usual: B-trees

Tree: binary tree



Each internal node is a binary test, based on which the search proceeds to either the left or the right subtree

Tree: B-tree



- Definition: Every internal nodel has a number of children in the interval [*a*,*b*] where *a, b* are appropriate positive integers, e.g., [2,4].
- Each branch under an internal node represents a test for a range of character sequences (to guide the search)

Trees

- Trees require a standard ordering of characters and hence strings … but we typically have one for all languages

- Pros:
  - Solves the prefix problem (terms starting with *hyp*)
- Cons:
  - Slower than hashtables: O(log *M*) [and this requires *balanced* tree]
  - Rebalancing binary trees is expensive
    - But B-trees mitigate the rebalancing problem

WILD-CARD QUERIES

Users may give wild-card queries

- ***mon*:*** find docs having any word beginning with “mon”
- Why wild-card queries?
  - Users may not be sure of a spelling
  - Users may want to match multiple variants of a term, e.g., query judicia* for matching both judicial and judiciary
- Types of wild-card queries
  - Trailing wild-card queries, e.g., mon*
  - Leading wild-card queries, e.g., *mon
  - General wild-card queries, e.g., s*dney, fas*in*te
- Key challenge: which dictionary terms match a wild-card query?

Trailing wild-card queries

- Trailing wild-card queries (mon*) easier
  - Use a binary tree (or B-tree) over the dictionary:
  - Retrieve all dictionary words in range: ***mon ≤ w < moo***
  - Then process postings lists of all such words

Leading wild-card queries

- ***mon:** find words ending in “mon”: slightly harder than trailing wild-card queries
- Maintain an additional B-tree for terms *backwards* (reverse B-tree)
  - Each root-to-leaf path corresponds to a term in the dictionary *written backwards*
  - Can retrieve all words in range: ***nom ≤ w < non.***

Query processing

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

Exercise: from what we have discussed, how can we enumerate all terms matching the query ***pro*cent*** ?

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

- How can we handle *'s in the middle of query term?
  - ***co*tion***
- One solution: look up ***co**** in a B-tree and ****tion*** in a reverse B-tree, and then intersect the two term sets
  - Expensive
- Better solution: transform wild-card queries so that the *'s occur at the end
  - This gives rise to the **Permuterm** Index

Permuterm index

- Introduce $ as a special symbol to mark the end of a term (a symbol that does not appear in the text)

- For term ***hello***, index under:
  - ***hello$, ello$h, llohe, lohel, o$hell, $hello***
  - Various rotations of each term (augmented with $) all link to the original vocabulary term

- Permuterm vocabulary: set of all rotated terms in the permuterm index

Handling wild-card queries with Permuterm index

- Given a wild-card query
  - Rotate the query so that the * symbol appears at the end of the string
  - Look up the rotated query in the permuterm index
- 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****

Queries having multiple wild-cards

- E.g., fi*mo*er
- First enumerate all dictionary terms that are in the permuterm index of er$fi*
- Not all such terms will have “mo” in the middle – need to filter out mismatched terms exhaustively

Permuterm query processing

- Rotate query wild-card to the right
- Now use B-tree lookup as before
- Once the permuterm index enables us to identify the original vocabulary terms matching a wild-card query, we can look up these terms in the usual way

- *Permuterm problem: ≈ quadruples lexicon size*

Empirical observation for English.

k-gram (e.g., Bigram) indexes

- Enumerate all *k*-grams (sequence of *k* chars) occurring in any term
- Use $ as a special character to denote the beginning and end of each 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

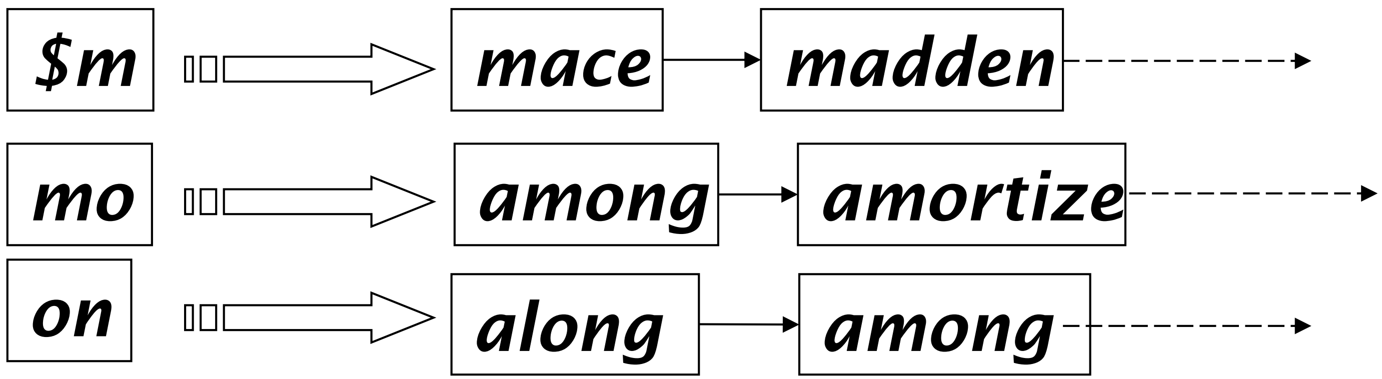
K-gram index

- Dictionary contains all k-grams that occur in any term in the vocabulary
- Maintain a ***second*** **inverted index *from k-grams to dictionary terms*** that match each k-gram
  - Each postings list points from a k-gram to all vocabulary terms containing that k-gram

- Example with k=2 shown on next slide

K-gram index (k=2) example

The *k*-gram index finds *terms* based on a query consisting of *k*-grams (here *k*=2).



Processing wild-cards with k-gram index

- Query ***mon**** can now be run as
  - ***$m*** *AND* ***mo*** *AND* ***on***
- Gets terms that match 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.
- Fast, space efficient (compared to permuterm).

Processing wild-card queries

- As before, we must execute a Boolean query for each enumerated, filtered term.
- Wild-cards can result in expensive query execution (very large disjunctions…)
  - pyth* AND prog*
- If you encourage "laziness" people will respond!

Type your search terms, use '*' if you need to.
E.g., Alex* will match Alexander.

Recap till now

- Dictionary Data Structures:
  - Hashtables
  - B-trees

- Tolerant Retrieval
  - Wildcard queries
  - Permuterm index
  - k-gram index

SPELLING CORRECTION

Spell correction

- Two principal uses
  - Correcting document(s) being indexed
  - Correcting user queries to retrieve “right” answers
- Two main flavors:
  - Isolated word
    - Check each word on its own for misspelling: ***jacson***
    - Will not catch typos resulting in correctly spelled words
    - e.g., ***from*** → ***form***
  - Context-sensitive
    - Look at surrounding words,
    - e.g., ***I flew form Heathrow to Narita.***

Document correction

- Needed for OCR’ed documents or for correcting typing errors
  - Correction algorithms are tuned for this: rn/m
  - Can use domain-specific knowledge
    - OCR confuses O and D more often than it would confuse O and I
    - O and I are adjacent on the QWERTY keyboard, so more likely interchanged in typing
- Often we don’t prefer to change the documents; instead fix the query-document mapping

Query mis-spellings

- Our principal focus here
  - E.g., the query ***"IIT Khagarpur"***
- We can either
  - Retrieve documents indexed by the correct spelling (with a declaration of the changed query), OR
  - Return several suggested alternative queries with the correct spelling
    - *Did you mean … ?*

Isolated word correction

- Fundamental premise – there is a lexicon from which the correct spellings come
- Two basic choices for this
  - A standard lexicon such as
    - Webster's English Dictionary
    - An "industry-specific" lexicon – hand-maintained
  - The lexicon of the indexed corpus
    - E.g., all words on the web
    - All names, acronyms etc.
    - (Including the mis-spellings)

Isolated word correction

- Given a lexicon and a character sequence Q, return the words in the lexicon closest to Q
- What's "closest"?
- We'll study several alternatives
  1. Edit distance (Levenshtein distance)
  2. Weighted edit distance
  3. *n*-gram overlap

Edit distance

- Given two strings S_1 and S_2, the minimum number of operations to convert one to the other
- Operations are typically character-level
  - Insert, Delete, Replace, (Transposition)
- E.g., the edit distance from ***dof*** to ***dog*** is 1
  - From ***cat*** to ***act*** is 2 (Just 1 with transpose.)
  - from ***cat*** to ***dog*** is 3.
- Generally found by dynamic programming.

Weighted edit distance

- Similar to above, but the weight of an operation depends on the character(s) involved
  - Meant to capture OCR or keyboard errors
    Example: ***m*** more likely to be mis-typed as ***n*** than as ***q***
  - Therefore, replacing ***m*** by ***n*** is a smaller edit distance than replacing ***m*** by ***q***
  - This may be formulated as a probability model
- Requires weight matrix as input
- The dynamic programming approach can be modified to handle weights

Using edit distances for correction

- Given query, first enumerate all character sequences within a preset (may be weighted) edit distance (e.g., 2)
- Intersect this set with list of "correct" words
- Show terms you found to user as suggestions
- Alternatively,
  - Look up all possible corrections in our inverted index and return all docs … slow
  - Retrieve with a single most likely correction
- The alternatives disempower the user, but save a round of interaction with the user

n-gram overlap

- Enumerate all the *n*-grams in the query string as well as in the lexicon
- Use the *n*-gram index (recall what we discussed in wild-card search) to retrieve all lexicon terms matching any of the query *n*-grams

- Variations
  - Can threshold by number of matching *n*-grams
  - Weight by keyboard layout, common typos, etc.

Example with trigrams

- Suppose the text is ***november***
  - Trigrams are *nov, ove, vem, emb, mbe, ber*
- The query is ***december***
  - Trigrams are *dec, ece, cem, emb, mbe, ber*
- So 3 trigrams overlap (of 6 in each term)
- How can we turn this into a normalized measure of overlap?

One option – Jaccard coefficient

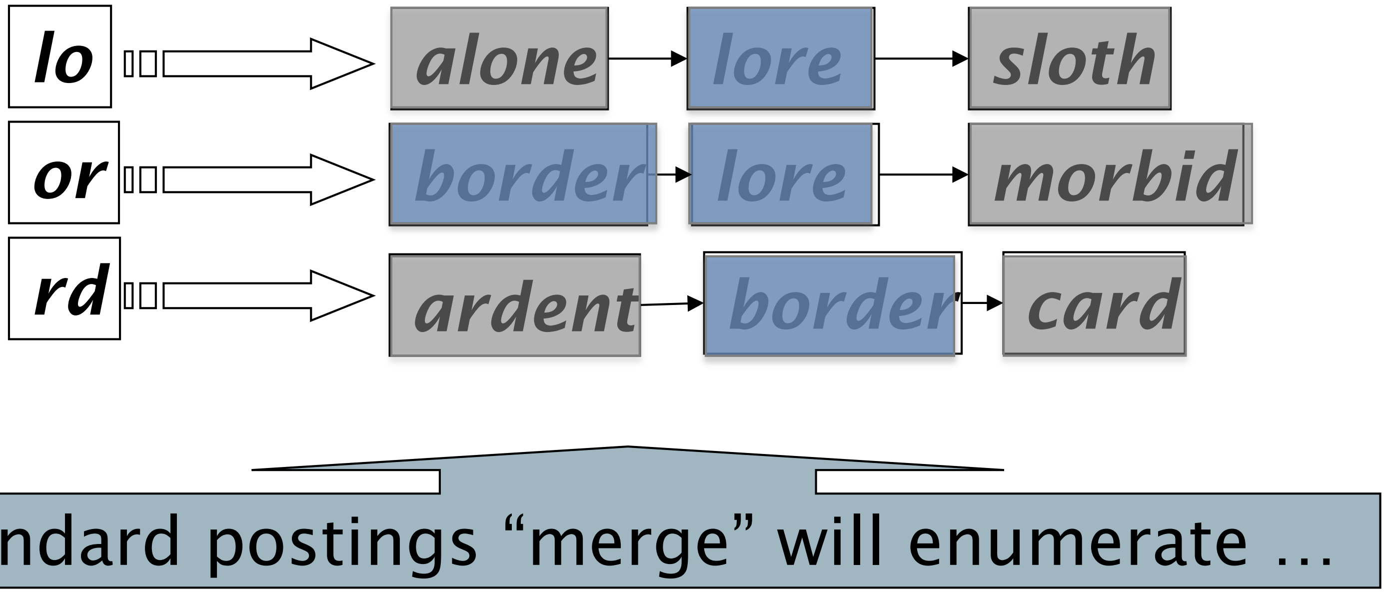
- A commonly-used measure of overlap
- Let *X* and *Y* be two sets; then the J.C. is

$$\left|X \cap Y\right| / \left|X \cup Y\right|$$

- Equals 1 when *X* and *Y* have the same elements and zero when they are disjoint
- *X* and *Y* don't have to be of the same size
- Always assigns a number between 0 and 1
  - Now threshold to decide if you have a match
  - E.g., if J.C. > 0.8, declare a match

Matching trigrams

- Consider the query ***lord*** – we wish to identify words matching 2 of its 3 bigrams (***lo, or, rd***)



Adapt this to using Jaccard (or another) measure.

Context-sensitive spell correction

- Consider the phrase query ***"flew form Heathrow"***
- We'd like the IR system to respond:

  Did you mean "***flew from Heathrow***"?

because no docs matched the query phrase.

- Need surrounding context to catch this error

Context-sensitive correction

- Query: ***flew form Heathrow***
  - Note: we do not know which word(s) is/are in error
- First idea: retrieve dictionary terms close (e.g., in terms of weighted edit distance) to each query term
- Now try all possible resulting phrases with one word “fixed” at a time
  - ***flew from heathrow***
  - ***fled form heathrow***
  - ***flea form heathrow***
  - .. and so on
- Hit-based spelling correction: Suggest alternative(s) that have lots of hits (from query logs / corpus)

Exercise

- Suppose that for ***"flew form Heathrow"*** we have 7 alternatives for flew, 19 for form and 3 for heathrow.

How many "corrected" phrases will we enumerate in this scheme?

Another approach

- Break phrase query into a conjunction of biwords (we discussed in Lecture 2).
- Look for biwords that need only one term corrected.
- Enumerate only phrases containing “common” biwords.

General issues in spell correction

- We enumerate multiple alternatives for "Did you mean?"
- Need to figure out which to present to the user
  - The alternative hitting most docs
  - Query log analysis
- More generally, rank alternatives probabilistically

$$\text{argmax}_{corr}\ P(corr \mid query)$$

  - From Bayes rule, this is equivalent to

$$\text{argmax}_{corr}\ P(query \mid corr) * P(corr)$$

Noisy channel   Language model

SOUNDEX

Soundex

- Class of heuristics to expand a query into phonetic equivalents
  - Language specific – mainly for names :
  - E.g., ***chebyshev*** $\rightarrow$ ***tchebycheff***
- Invented for the U.S. census … in 1918

Soundex – typical algorithm

- Turn every token to be indexed into a 4-character reduced form
- Do the same with query terms
- Build and search an index on the reduced forms
  - (when the query calls for a soundex match)

- http://www.creativyst.com/Doc/Articles/SoundEx1/SoundEx1.htm#Top

Soundex – typical algorithm

1. Retain the first letter of the word.
2. Change all occurrences of the following letters to '0' (zero):
   'A', E', 'I', 'O', 'U', 'H', 'W', 'Y'.
3. Change letters to digits as follows:

- B, F, P, V → 1
- C, G, J, K, Q, S, X, Z → 2
- D,T → 3
- L → 4
- M, N → 5
- R → 6

Soundex continued

4. Remove all pairs of consecutive digits.
5. Remove all zeros from the resulting string.
6. Pad the resulting string with trailing zeros and return the first four positions, which will be of the form <uppercase letter> <digit> <digit> <digit>.

E.g., ***Herman*** becomes H655.

Will ***hermann*** generate the same code?

Soundex

- Soundex is the classic algorithm, provided by most databases (Oracle, Microsoft, …)
- How useful is soundex? Not very – for information retrieval
- Okay for "high recall" tasks (e.g., Interpol), though biased to names of certain nationalities
- Zobel and Dart (1996) show that other algorithms for phonetic matching perform much better in the context of IR

What queries can we process?

- We have
  - Positional inverted index with skip pointers
  - Wild-card index
  - Spell-correction
  - Soundex
- Queries such as

(SPELL(moriset) /3 toron*to) OR *SOUNDEX* ***(chaikofski)***