Introduction to

Information Retrieval

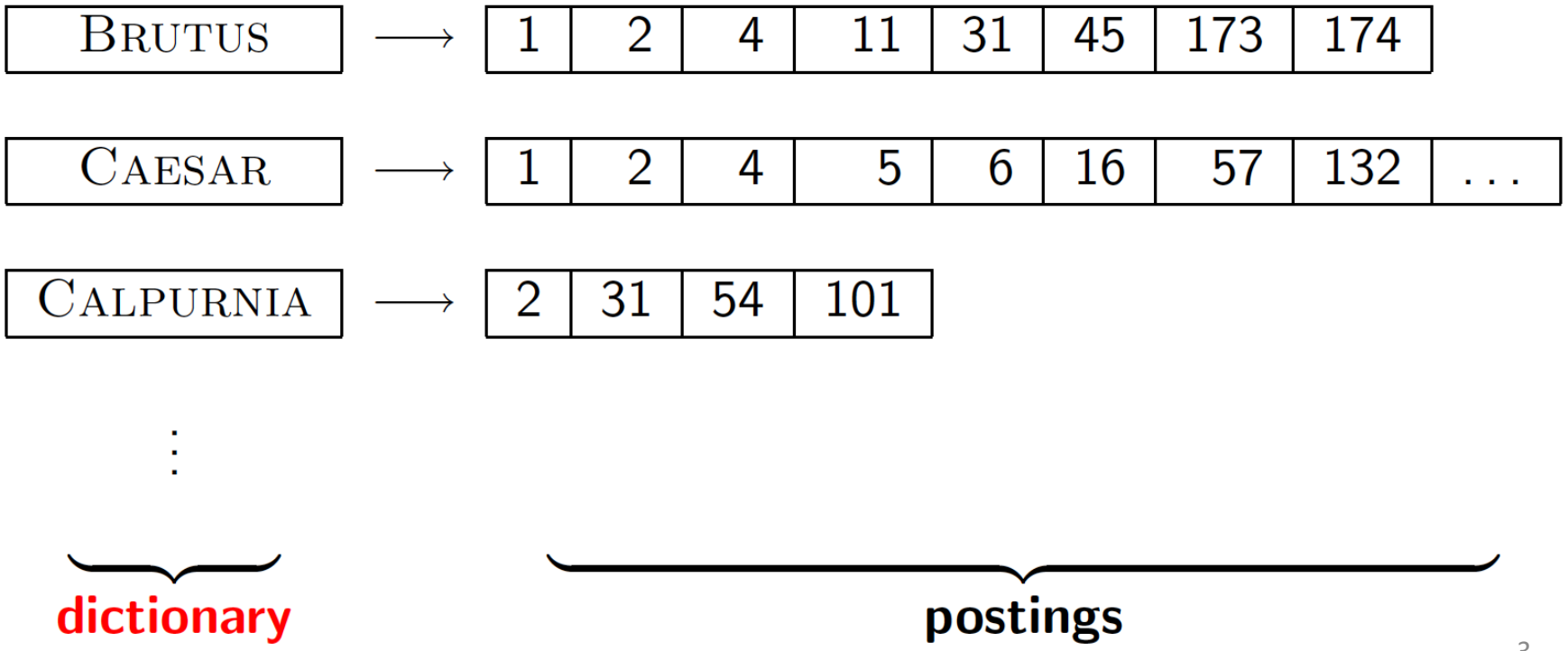
Dictionaries and tolerant retrieval

Agenda

- **Dictionaries and tolerant retrieval**
  - Search structures for dictionaries, Wildcard queries: General wildcard queries , k-gram indexes for wildcard queries
  - Spelling correction: Implementing spelling correction, Forms of spelling correction, Edit distance, k-gram indexes for spelling correction, Context sensitive spelling correction, Phonetic correction.

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	⟶
aachen	65	⟶
. . .	. . .	. . .
zulu	221	⟶

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:
  - Hash Tables
  - Search Trees
- The choice of solution (hashing, or search trees) is governed by a number of questions:
  - How many keys are we likely to have?
  - Is the number likely to remain static, or change a lot – and in the case of changes, are we likely to only have new keys inserted, or to also have some keys in the dictionary be deleted?
  - What are the relative frequencies with which various keys will be accessed?
- Some IR systems use hashtables, some trees

Hash Tables

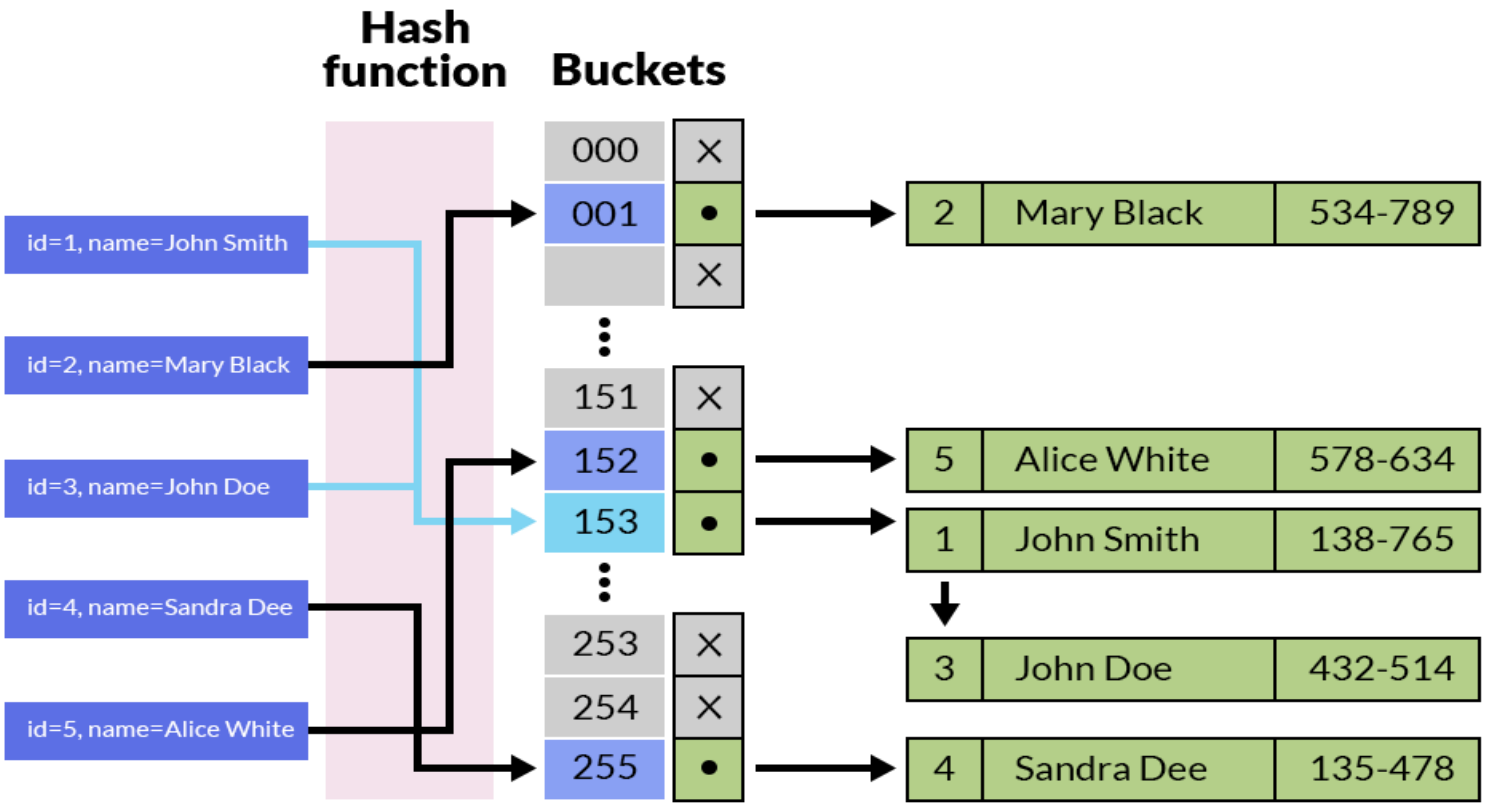
- Hashing has been used for dictionary lookup in some search engines.
- Each vocabulary term (key) is hashed into an integer over a large enough space that hash collisions are unlikely; collisions if any are resolved by auxiliary structures that can demand care to maintain.
- At query time, we hash each query term separately and following a pointer to the corresponding postings, taking into account any logic for resolving hash collisions.

- In particular, we cannot seek (for instance) all terms beginning with the prefix automat

- Finally, in a setting (such as the Web) where the size of the vocabulary keeps growing, a hash function designed for current needs may not suffice in a few years' time.

Hash tables

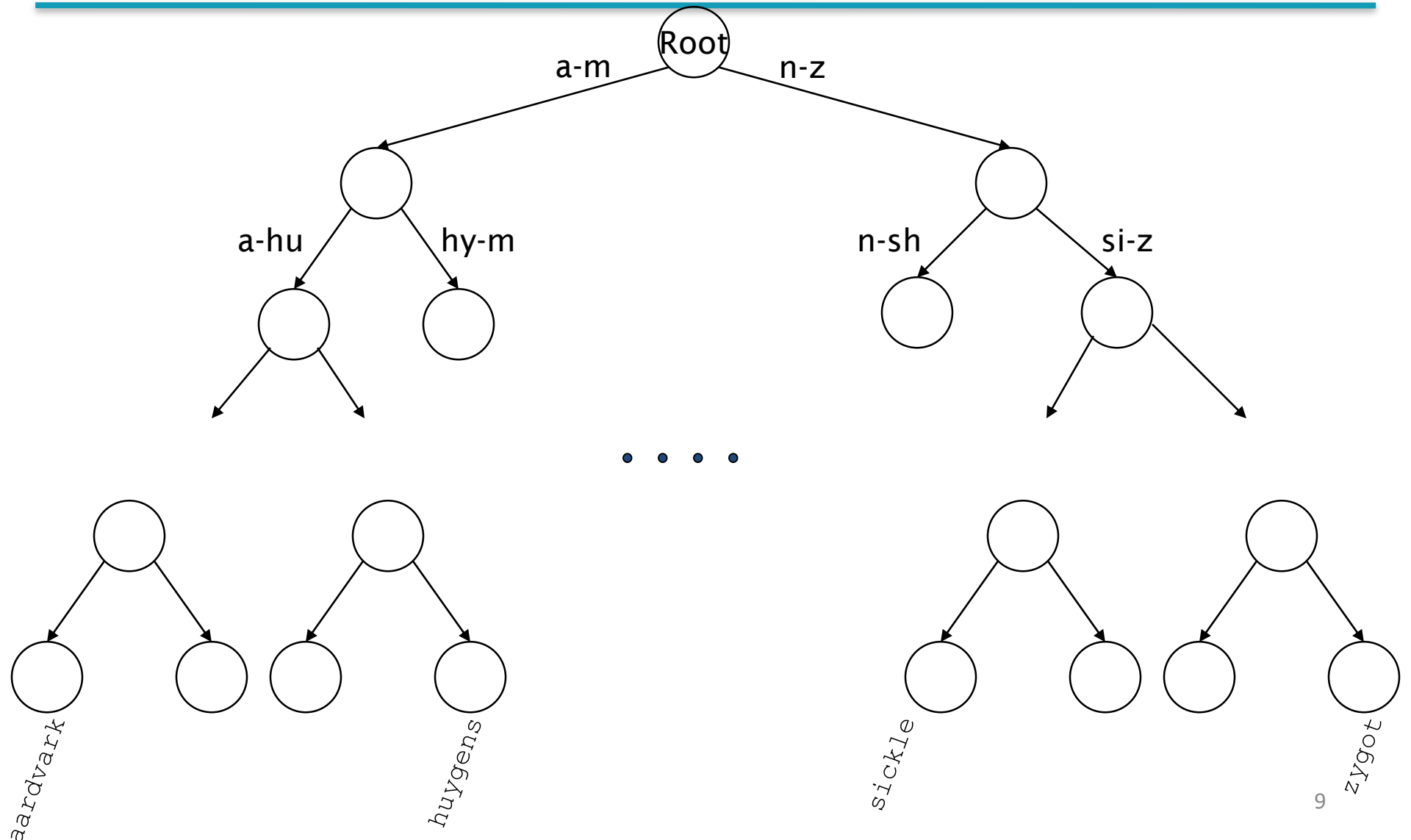
Hash Function : $H(s)=\sum_{k=0}^{n-1} chk$



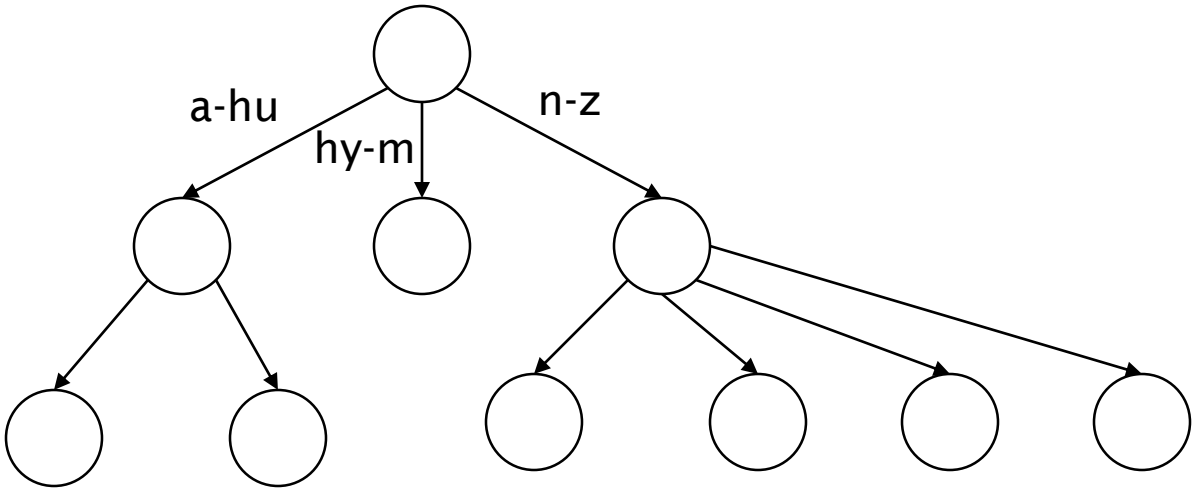
Hashtables

- Each vocabulary term is hashed to an integer
- Pros:
  - Lookup is faster than for a tree: O(1)
- Cons:
  - No easy way to find minor variants:
    - judgment/judgement
  - No prefix search - Eg. words beginning with help
  - If vocabulary keeps growing, need to occasionally do the expensive operation of rehashing *everything*

Tree: binary tree

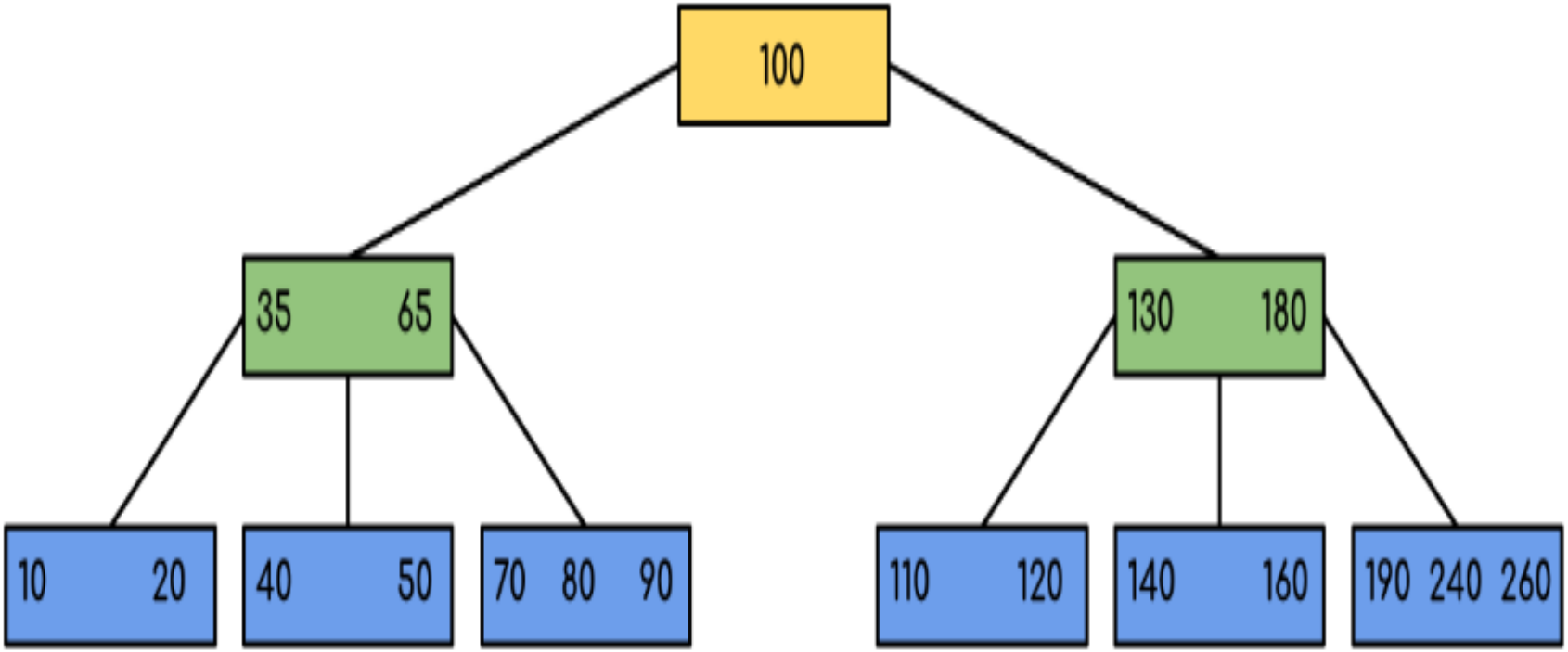


Tree: B-tree



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

Tree: B-tree



Trees

- Simplest: binary tree
  - time complexity for searching is **O(logN)** in average case.
  - The space complexity of a binary search tree is **O (n)** in both the average and the worst cases. with 'n' being the depth of the tree(number of nodes present in a tree)
- More usual: B-trees
  - B-Tree is a self-balancing search tree.
  - Time complexity for search, insert and delete is O(log n)
  - B-tree implementation has O(n) space complexity
- Trees require a standard ordering of characters and hence strings … but we typically have one
- Pros:
  - Solves the prefix problem (terms starting with *hyp*)
- Cons:
  - Slower: O(log *M*) [and this requires *balanced* tree]
  - Rebalancing binary trees is expensive
    - But B-trees mitigate the rebalancing problem

WILD-CARD QUERIES

Wildcard queries are used in any of the following situations:

- (1) the user is uncertain of the spelling of a query term (e.g., Sydney vs. Sidney, which leads to the wildcard query S*dney)
- (2) the user is aware of multiple variants of spelling a term and (consciously) seeks documents containing any of the variants (e.g., color vs. colour)
- (3) the user seeks documents containing variants of a term that would be caught by stemming, but is unsure whether the search engine performs stemming (e.g., judicial vs. judiciary, leading to the wildcard query judicia*);
- (4) the user is uncertain of the correct use of a foreign word or phrase (e.g., the query Universit* Stuttgart).

Wild-card queries: *

- ***Trailing wildcard query***
- ***mon*:*** find all docs containing any word beginning with "mon".
- Easy with binary tree (or B-tree) : retrieve all words in range: ***mon ≤ w < moo***
- ***Leading wildcard queries***
- ****mon:*** find words ending in "mon": harder
  - Maintain an additional B-tree for terms *backwards.*
  - *reverse B-tree* on the dictionary.
  - term lemon would, in the B-tree, be represented by the path root-n-o-m-e-l.

  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*** ?

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.
- E.g., consider the query:

  se*mon

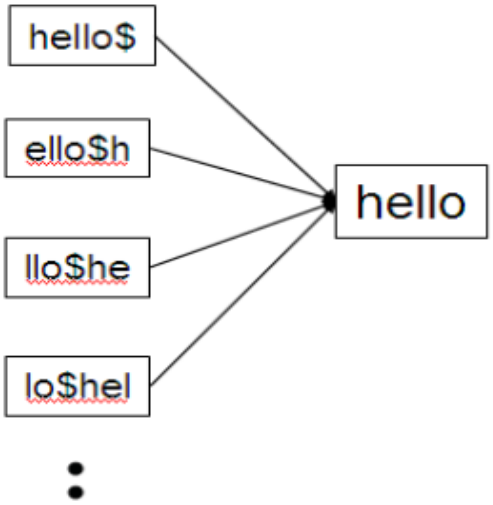
  To do this, we use the regular B-tree to enumerate the set *W* of dictionary terms beginning with the prefix se, then the reverse B-tree to enumerate the set *R* of terms ending with the suffix mon.
- Next, we take the intersection $W \cap R$ of these two sets, to arrive at the set of terms that begin with the prefix se and end with the suffix mon.
- Finally, we use the standard inverted index to retrieve all documents containing any terms in this intersection.

Query processing

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

Permuterm index

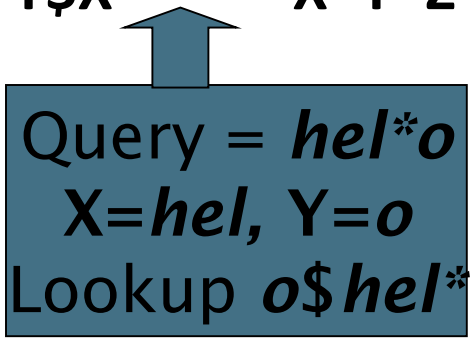
- For term ***hello***, index under:
  - ***hello$, ello$h, llohe, lohel, o$hell, $hello*** where $ is a special symbol - We refer to the set of rotated terms in the permuterm index as the *permuterm vocabulary*.



A portion of a permuterm index.

- Consider the wildcard query m*n.
- The key is to *rotate* such a wildcard query so that the * symbol appears at the end of the string – thus the rotated wildcard query becomes n$m*.
- Next, we look up this string in the permuterm index, where seeking n$m* (via a search tree) leads to rotations of (among others) the terms man and moron.

- 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!



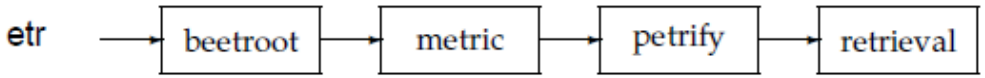
Permuterm query processing

- The permuterm index enables us to identify the original vocabulary terms matching a wildcard query.
- We look up these terms in the standard inverted index to retrieve matching documents. We can thus handle any wildcard query with a single * symbol.

- What about a query such as fi*mo*er? In this case we first enumerate the terms in the dictionary that are in the permuterm index of er$fi*.
- Not all such dictionary terms will have the string mo in the middle - we filter these out by exhaustive enumeration, checking each candidate to see if it contains mo.
- In this example, the term fishmonger would survive this filtering but filibuster would not.
- We then run the surviving terms through the standard inverted index for document retrieval.
- ***Permuterm problem:*** **dictionary becomes quite large, including as it does all rotations of each term - ten-fold space increase**

k-gram indexes for wildcard queries

- A *k-gram* is a sequence of *k* characters.
- Thus cas, ast and stl are all 3-grams occurring in the term castle.
- We use a special character $ to denote the beginning or end of a term, so the full set of 3-grams generated for castle is: $ca, cas, ast, stl, tle, le$.
- In a *k-gram index*, the dictionary contains all *k*-grams that occur in any term in the vocabulary.
- Each postings list points from a *k*-gram to all vocabulary terms containing that *k*-gram. For instance, the 3-gram etr would point to vocabulary terms such as metric and retrieval.



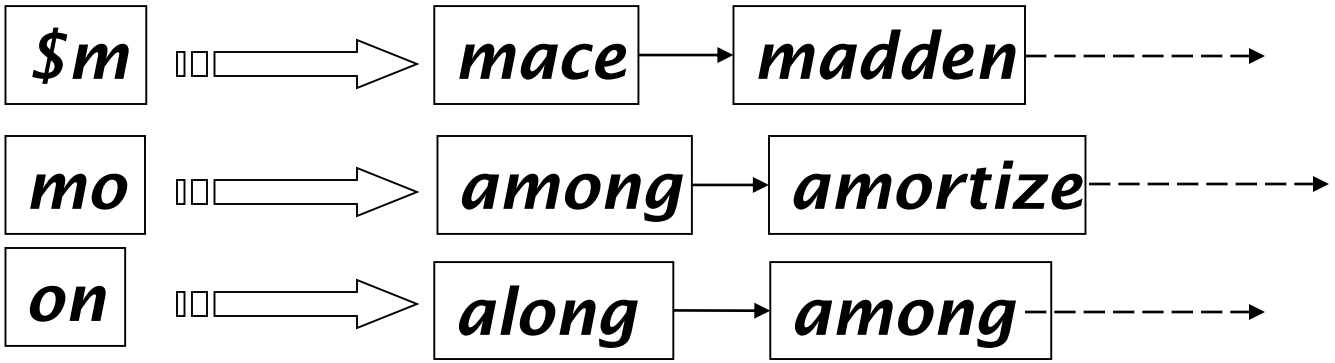
- How does such an index help us with wildcard queries?
- Consider the wildcard query re*ve.
- We are seeking documents containing any term that begins with re and ends with ve.
- Accordingly, we run the Boolean query $re AND ve$.
- This is looked up in the 3-gram index and yields a list of matching terms such as relive, remove and retrieve.
- Each of these matching terms is then looked up in the standard inverted index to yield documents matching the query.

Processing wild-cards

- 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).

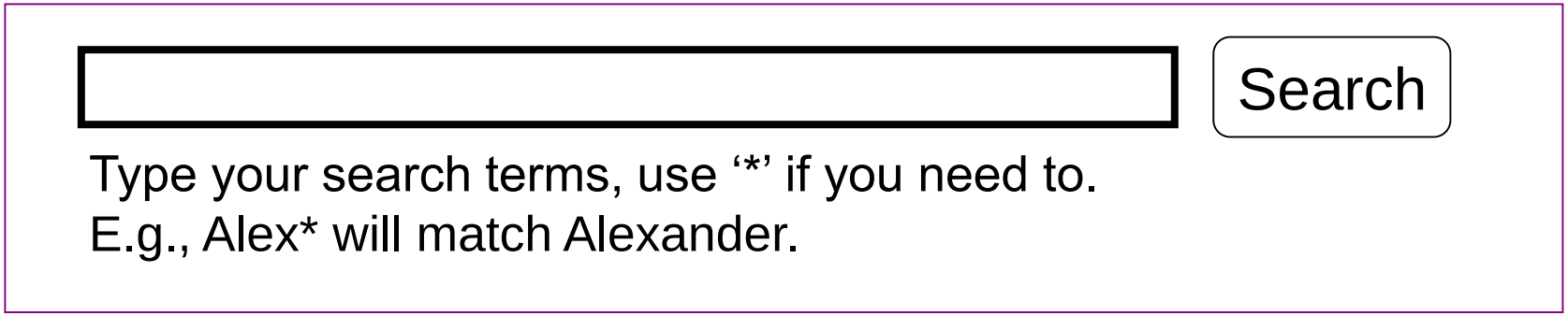
Bigram index example

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



Processing wild-card queries

- As before, we must execute a Boolean query for each enumerated, filtered term.
- Even without Boolean combinations of wildcard queries, the processing of a wildcard query can be quite expensive, because of the added lookup in the special index, filtering and finally the standard inverted index.
- Wild-cards can result in expensive query execution (very large disjunctions…)
  - pyth* AND prog*
- If you encourage "laziness" people will respond!



Recap

- Dictionary Data Structures: Hashtables, B-trees
- Tolerant Retrieval
  - Wildcard queries (permuterm index, k-gram index)
  - Spell correction contd …

SPELLING CORRECTION

Implementing spelling correction

- Of various alternative correct spellings for a mis-spelled query, choose the "nearest" one.
- When two correctly spelled queries are tied (or nearly tied), select the one that is more common. For instance, grunt and grant both seem equally used words as corrections for grnt.
  - Consider the number of occurrences of the term
  - Consider the most common among queries typed in by other users

- Spelling correction algorithms build on these computations of proximity; their functionality is then exposed to users in one of several ways:

1. On the query carot always retrieve documents containing carot as well as any "spell-corrected" version of carot
2. As in 1 but only when the query term carot is not in the dictionary
3. As in 1 but only when the original query returned fewer than a preset number of documents (say fewer than five documents).
4. When the original query returns fewer than a preset number of documents, the search interface presents a *spelling suggestion* to the end user: this suggestion consists of the spell-corrected query term(s). Thus, the search engine might respond to the user: "Did you mean carrot?"

Spell correction

- 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***
    - **flew form Heathrow** contains a mis-spelling of the term from – because each term in the query is correctly spelled in isolation.
  - Context-sensitive
    - Look at surrounding words,
    - e.g., ***I flew <u>form</u> Heathrow to Narita.***

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)

Techniques for addressing isolated-term 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
  - Edit distance (Levenshtein distance)
  - Weighted edit distance
  - *k*-gram index (overlap)

Techniques for addressing isolated-term correction

- **Edit distance:**
- Given two character strings *s*1 and *s*2, the *edit distance* EDIT DISTANCE between them is the minimum number of *edit operations* required to transform *s*1 into *s*2.
- Most commonly, the edit operations allowed for this purpose are:

1. insert a character into a string
2. delete a character from a string and
3. replace a character of a string by another character;

for these operations, edit distance is sometimes known as *Levenshtein distance*.

- For example, the edit distance between cat and dog is 3, ***cat*** to ***act*** is 2, ***dof*** to ***dog*** is 1
- Generally found by dynamic programming.

Weighted edit distance

- Edit distance can be generalized to allowing different weights for different kinds of edit operations
- 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 by ***q***
  - This may be formulated as a probability model
- Requires weight matrix as input
- Modify dynamic programming to handle weights

Levenshtein distance

```
EDITDISTANCE(s1, s2)
 1  int m[i, j] = 0
 2  for i ← 1 to |s1|
 3  do m[i, 0] = i
 4  for j ← 1 to |s2|
 5  do m[0, j] = j
 6  for i ← 1 to |s1|
 7  do for j ← 1 to |s2|
 8       do m[i, j] = min{m[i − 1, j − 1] + if (s1[i] = s2[j]) then 0 else 1fi,
 9                        m[i − 1, j] + 1,
10                        m[i, j − 1] + 1}
11  return m[|s1|, |s2|]
```

Dynamic programming algorithm for computing the edit distance between strings *s*1 and *s*2.

		f	a	s	t
	0	1 1	2 2	3 3	4 4
c	1 1	1 2 2 1	2 3 2 2	3 4 3 3	4 5 4 4
a	2 2	2 2 3 2	1 3 3 1	3 4 2 2	4 5 3 3
t	3 3	3 3 4 3	3 2 4 2	2 3 3 2	2 4 3 2
s	4 4	4 4 5 4	4 3 5 3	2 3 4 2	3 3 3 3

Example Levenshtein distance computation. The 2 × 2 cell in the [*i*, *j*] entry of the table shows the three numbers whose minimum yields the fourth.

		a	l	i	c	e
	0	1 1	2 2	3 3	4 4	5 5
p	1 / 1	1 2 / 2 1	2 3 / 2 2	3 4 / 3 3	4 5 / 4 4	5 6 / 5 5
a	2 / 2	1 2 / 3 1	2 3 / 2 2	3 4 / 3 3	4 5 / 4 4	5 6 / 5 5
r	3 / 3	3 2 / 4 2	2 3 / 3 2	3 4 / 3 3	4 5 / 4 4	5 6 / 5 5
i	4 / 4	4 3 / 5 3	3 3 / 4 3	2 4 / 4 2	4 5 / 3 3	5 6 / 4 4
s	5 / 5	5 4 / 6 4	4 4 / 5 4	4 3 / 5 3	3 4 / 4 3	4 5 / 4 4

Using edit distances for correction

- The spelling correction problem however demands more than computing edit distance
- Given a set S of strings (corresponding to terms in the vocabulary) and a query string *q*, we seek the string(s) in *V* of least edit distance from *q*.
  - Compute the edit distance from *q* to each string in *V*
  - Select string(s) of minimum edit distance
  - Very expensive
- Accordingly, a number of heuristics are used in practice to efficiently retrieve vocabulary terms likely to have low edit distance to the query term(s).
  - to restrict the search to dictionary terms beginning with the same letter as the query string; the hope would be that spelling errors do not occur in the first character of the query
  - to use a version of the permuterm index - Consider the set of all rotations of the query string *q*. For each rotation *r* from this set, we traverse the B-tree into the permuterm index, thereby retrieving all dictionary terms that have a rotation beginning with *r*.

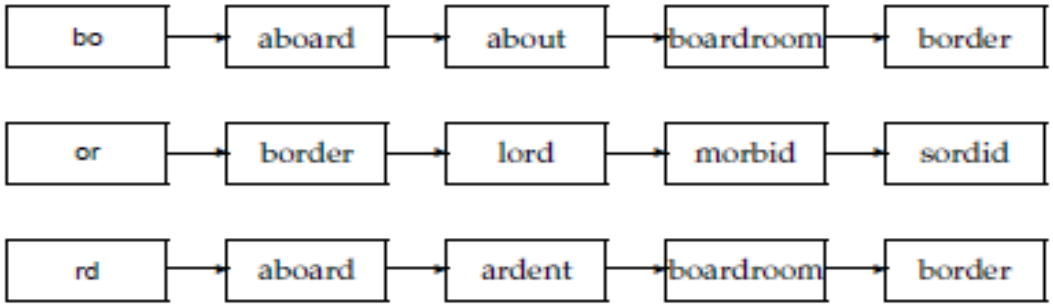
Using edit distances for correction

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

- **What are its benefits?**
  - It is a simple and intuitive measure of string similarity.
  - It can be computed efficiently using dynamic programming.
  - It is robust to small changes in the input strings.
- **What are its limitations?**
  - It does not take into account the semantic meaning of the words.
  - It can be sensitive to the length of the strings.
  - It does not handle transpositions (swapping two adjacent characters) well.
  - Sometimes other string metrics may be more appropriate depending on the specific requirements.
  - The cost of each operation is usually set to 1, but choosing the best cost can be application-dependent and is not always straightforward.

k-gram index (overlap)

- To further limit the set of vocabulary terms, we use k-gram index.
- Once we retrieve such terms, we can then find the ones of least edit distance from query *q*.
- Enumerate all the *k*-grams in the query string as well as in the lexicon
- Retrieve vocabulary terms that have many *k*-grams in common with the query.



The 2-gram index in the above Figure shows the postings for the three bigrams in the query bord. Suppose we wanted to retrieve vocabulary terms that contained at least two of these three bigrams, we would enumerate aboard, boardroom and border

- This straightforward application of the linear scan intersection of postings immediately reveals that the : terms like boardroom for bord, get enumerated.
- We require more measures to determine the overlap in *k*-grams between a vocabulary term and *q*.
- Measure of overlap is the *Jaccard coefficient* for measuring the overlap between two sets *A* and *B*, defined to be $|A \cap B|/|A \cup B|$.
- The Jaccard coefficient can be a value between 0 and 1, with 0 indicating no overlap and 1 complete overlap between the sets.
- The two sets we consider are the set of *k*-grams in the query *q*, and the set of *k*-grams in a vocabulary term

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?

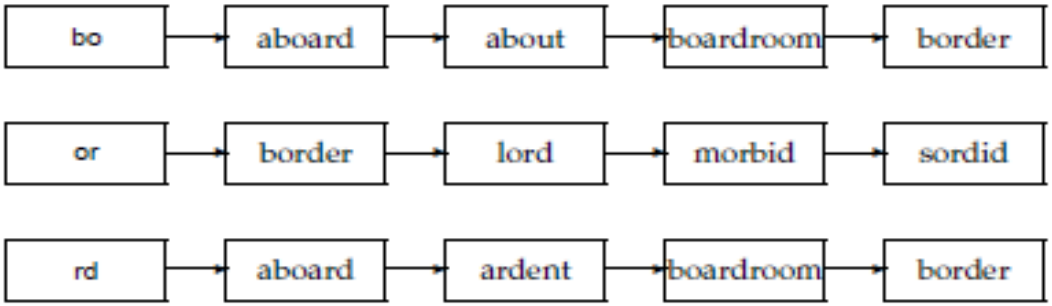
One option – Jaccard coefficient

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

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

- 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

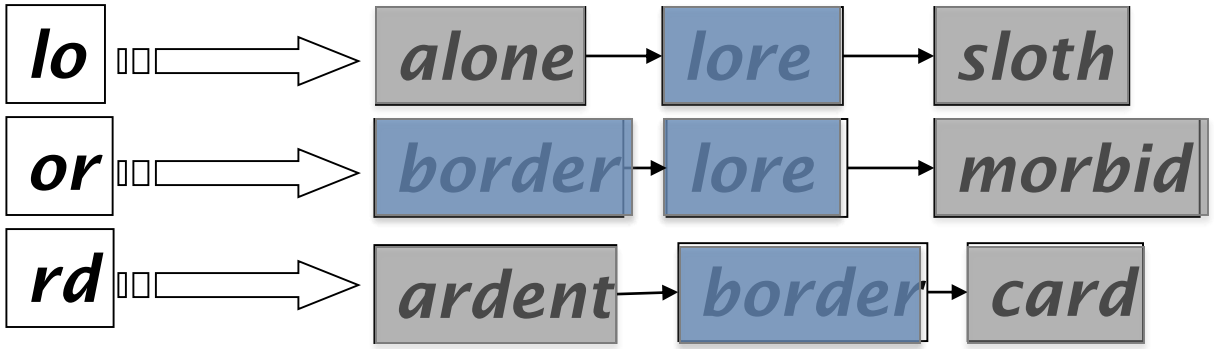
Jaccard coefficient



- Consider the point when the postings scan for query *q* = bord reaches term *t* = boardroom.
- We know that two bigrams match.
- If the postings stored the (pre-computed) number of bigrams in boardroom (namely, 8),
- Jaccard coefficient = 2/(8+3−2);
- the numerator is obtained from the number of postings hits (2, from bo and rd)
- While the denominator is the sum of the number of bigrams in bord and boardroom, less the number of postings hits.

Matching bigrams

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



Standard postings "merge" will enumerate …

Adapt this to using Jaccard (or another) measure.

How to use *k*-gram index

- First use the *k*-gram index to enumerate a set of candidate vocabulary terms that are potential corrections of *q*.
- We then compute the edit distance from *q* to each term in this set, selecting terms from the set with small edit distance to *q*.

Context-sensitive spell correction

- Text: ***I flew from Heathrow to Narita.***
- Consider the phrase query ***"flew form Heathrow"***
- We'd like to respond

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

because no docs matched the query phrase.

Context-sensitive correction

- Need surrounding context to catch this.
- First idea: retrieve dictionary terms close (in 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***
- This enumeration can be expensive if we find many corrections of the individual terms.
- We can retain only the most frequent combinations in the collection or in the query logs
- We can use the biword statistics in the collection or in the query

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 (Lecture 2).
- Look for biwords that need only one term corrected.
- Enumerate only phrases containing “common” biwords.

SOUNDEX

Soundex

- *Phonetic* correction: misspellings that arise because the user types a query that sounds like the target term.
- Class of heuristics to expand a query into phonetic equivalents
  - Language specific – mainly for names :
  - E.g., ***chebyshev*** → ***chebicheff***
- The idea owes its origins to work in international police departments from the early 20th century, seeking to match names for wanted criminals despite the names being spelled differently in different countries.

Soundex – typical algorithm

- The main idea here is to generate, for each term, a "phonetic hash" so that similar-sounding terms hash to the same value.
- Algorithms for such phonetic hashing are commonly collectively known as *soundex* algorithms.
- Following is an original soundex SOUNDEX algorithm, with various variants, built on the following scheme:

1. Turn every term to be indexed into a 4-character reduced form. Build an inverted index from these reduced forms to the original terms; call this the soundex index.
2. Do the same with query terms.
3. When the query calls for a soundex match, search this soundex index.

Such algorithms are especially applicable to searches on the names of people.

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 $\rightarrow$ 1
- C, G, J, K, Q, S, X, Z $\rightarrow$ 2
- D,T $\rightarrow$ 3
- L $\rightarrow$ 4
- M, N $\rightarrow$ 5
- R $\rightarrow$ 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 continued

- For an example of a soundex map, Hermann maps to H655.
- Given a query (say herman), we compute its soundex code and then retrieve all vocabulary terms matching this soundex code from the soundex index, before running the resulting query on the standard inverted index.
- This algorithm rests on a few observations:

(1) vowels are viewed as interchangeable, in transcribing names;

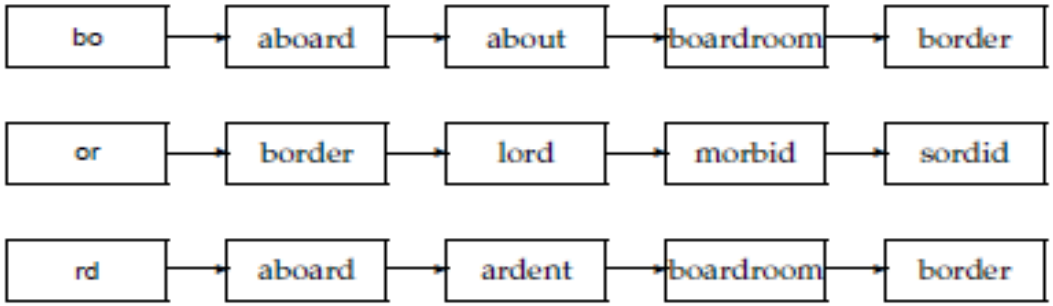
(2) consonants with similar sounds (e.g., D and T) are put in equivalence classes.

This leads to related names often having the same soundex codes.

https://www.ics.uci.edu/~dan/genealogy/Miller/javascrp/soundex.htm

Assignment 2

- Find two differently spelled proper nouns whose soundex codes are the same.
- Find two phonetically similar proper nouns whose soundex codes are different
- Compute the edit distance between paris and alice. Write down the 5 × 5 array of distances between all prefixes as computed by the algorithm
- Write pseudocode showing the details of computing on the fly the Jaccard coefficient while scanning the postings of the *k*-gram index
- Compute the Jaccard coefficients between the query bord and each of the terms in Figure below that contain the bigram or.



- Write down the entries in the permuterm index dictionary that are generated by the term original.
- Give an example of a sentence that falsely matches the wildcard query mon*h if the search were to simply use a conjunction of bigrams.