DICTIONARIES AND TOLERANT RETRIEVAL

Presented By : Sanket Nagrale (185053)
Shubhang Bhagat(185060)
Sachin Wattamwar(185070)
Sanket Wable(185088)

Dictionaries

Dictionaries Data Structures :

1) Hash tables
2) Trees

Criteria for when to use hashes vs. trees:

- Is there a fixed number of terms or will it keep growing?
- What are the relative frequencies with which various keys will be accessed?
- How many terms are we likely to have?

Hashes

Each vocabulary term is hashed into an integer.

Pros:

- Lookup is fast (faster than in a search tree)
- Lookup time is constant .

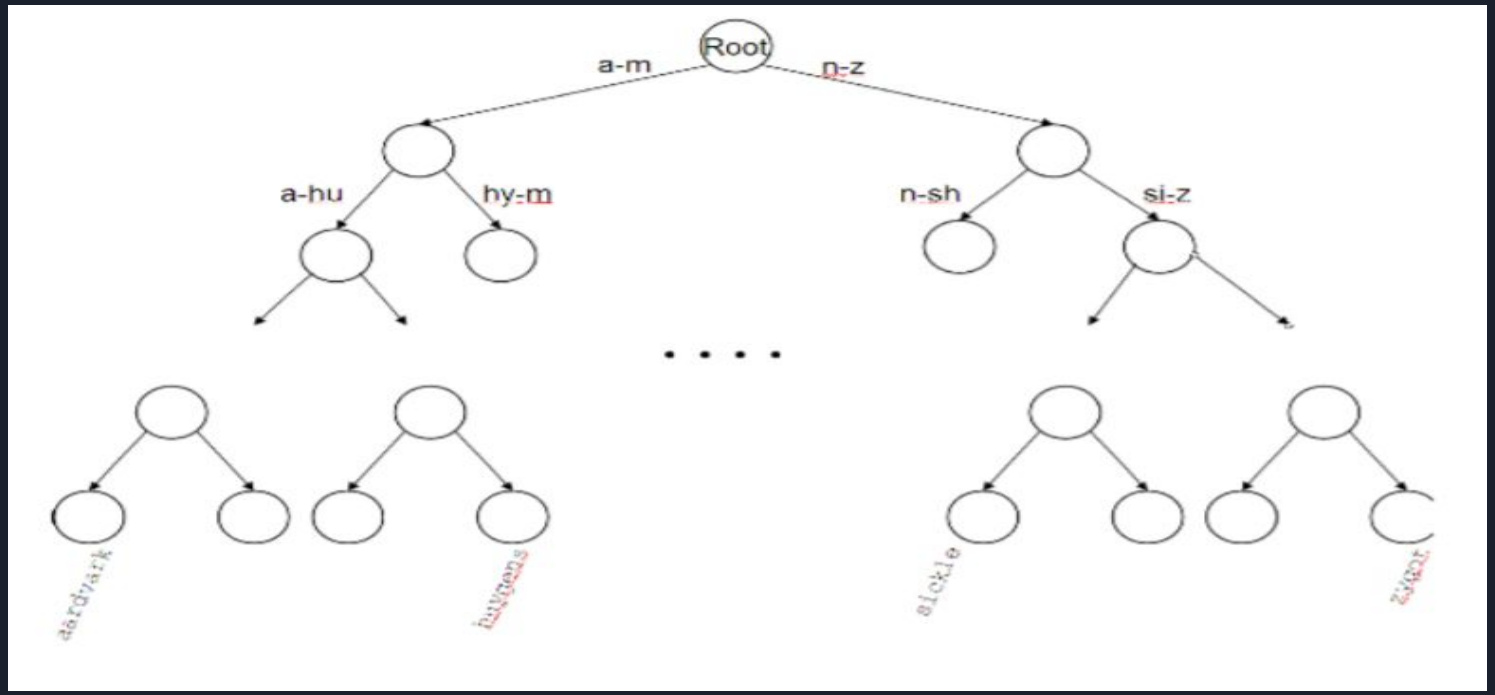
Cons:

- no way to find minor variants (result vs. résult)
- no prefix search (all terms starting with auto)
- need to rehash everything periodically if vocabulary keeps growing

Trees

- Trees solve the prefix problem (find all terms starting with auto).
- Simplest tree: binary tree
- Search is slightly slower than in hashes: O(logM), where M is the size of the vocabulary.
- O(logM) only holds for balanced trees.
- Re-balancing binary trees is expensive.
- B-trees mitigate the rebalancing problem.
- B-tree definition: every internal node has a number of children in the interval [a, b] where a, b are appropriate positive integers, e.g., [2, 4].

Binary Tree



Wildcard queries

- mon*: find all docs containing any term beginning with mon
- Easy with B-tree dictionary: retrieve all terms t in the range: mon ≤ t < moo
- *mon: find all docs containing any term ending with mon
- Maintain an additional tree for terms backwards
- Then retrieve all terms t in the range: nom ≤ t < non
- Result: A set of terms that are matches for wildcard query
- Then retrieve documents that contain any of these terms

How to handle * in the middle of a term

- Example: m*nchen
- We could look up m* and *nchen in the B-tree and intersect the two term sets.
- Expensive
- Alternative: permuterm index
- Basic idea: Rotate every wildcard query, so that the * occurs at the end.
- Store each of these rotations in the dictionary, say, in a B-tree