Tries

Naïve brute force for searching a text of size n and a pattern of size m requires significant computing time. Preprocessing the pattern speeds up pattern matching queries.

If the text is large, immutable, and searched often (e.g., Shakespeare), we may want to preprocess the text itself.

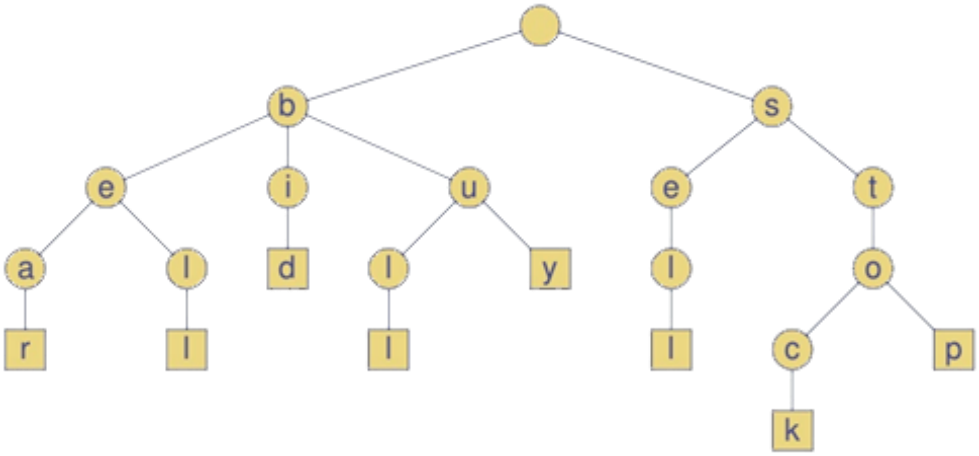
A trie is a compact data structure for representing a set of strings, such as all the words in a text. A trie supports pattern matching queries in time proportional to the pattern size.

Standard Tries

The standard trie for a set of strings S is an ordered tree such that:

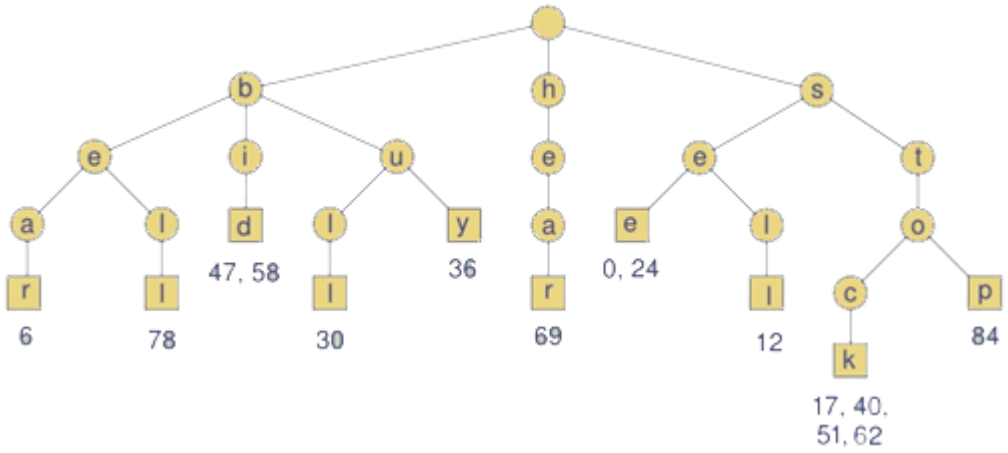
- Each node but the root is labeled with a character
- The children of a node are alphabetically ordered
- The paths from the root to the leaves yield the strings of S

Example: set of strings S = { bear, bell, bid, bull, buy, sell, stock, stop }



Word Matching With a Trie

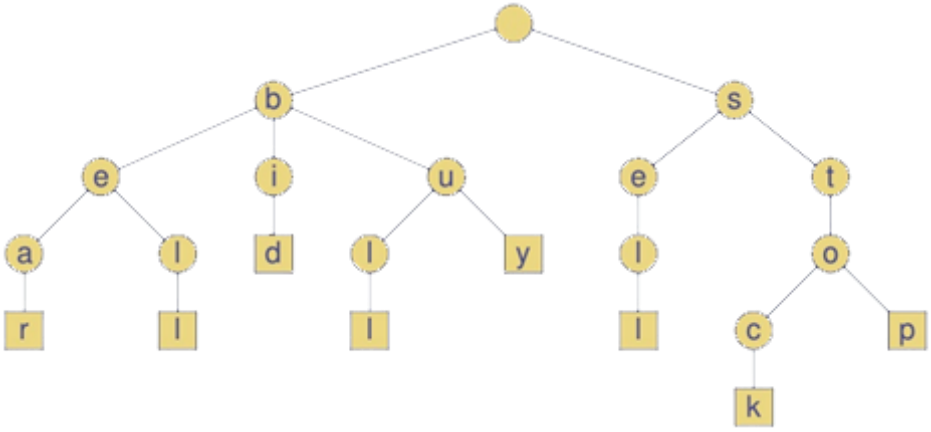
We insert the words of the text into a trie. Each leaf stores the occurrences of the associated word in the text.

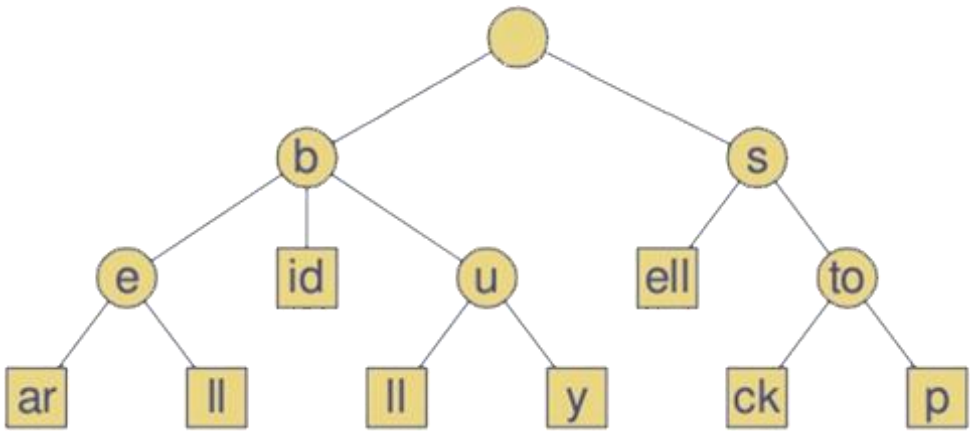


Compressed Tries

A compressed trie has internal nodes of degree at least two

It is obtained from standard trie by compressing chains of “redundant” nodes





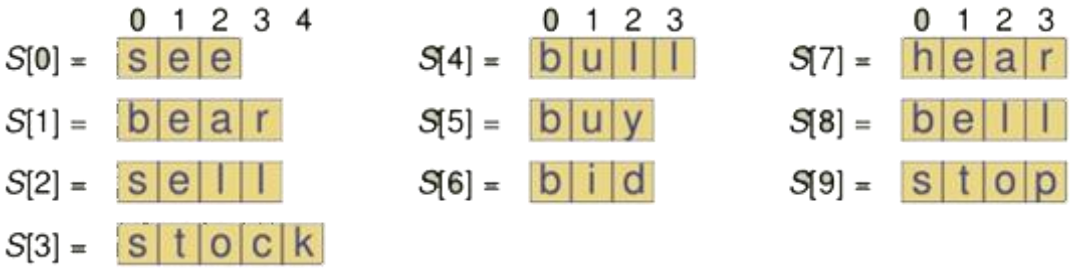
Compact Representation

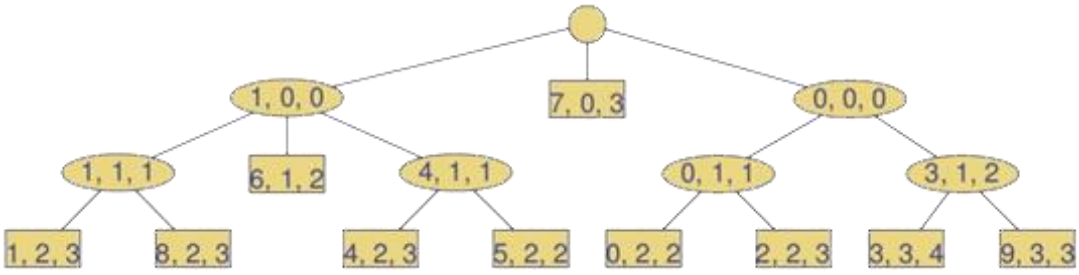
Compact representation of a compressed trie for an array of strings:

Stores at the nodes ranges of indices instead of substrings

Uses *s* space, where s is the number of strings in the array

Serves as an auxiliary index structure





String Searches

Begins with: where name like ‘x%’

Ends with: where name like ‘%x’

Substring: where name like ‘%x%’

Applications of Tries

Auto complete: User types “Rob” and you can type with all words that begin with Rob, or all contacts that begin with Rob, etc.

Sequence Assembly in Genetics Sequences

Sorting of Large Sets of Strings: BurstSort

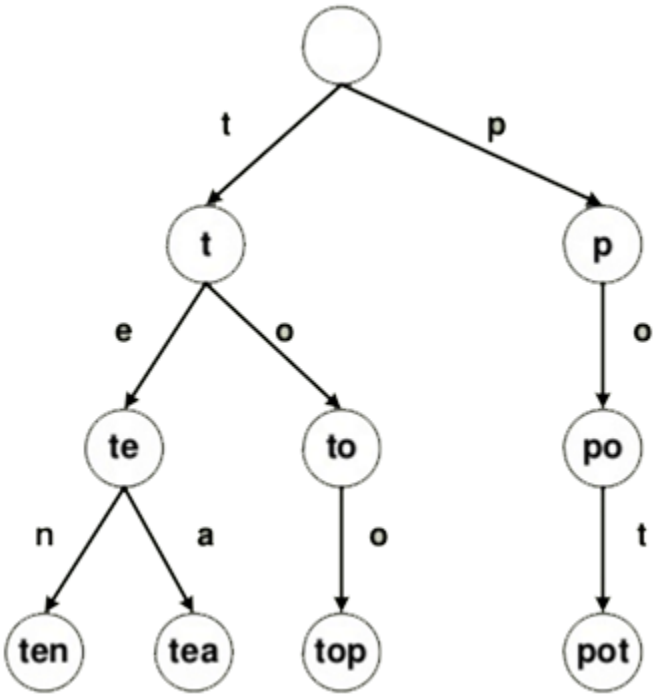
Slightly Different Version of Trie

A trie is a tree-based data structure for storing strings:

There is one **node** for every common **prefix**

The strings are stored in extra **leaf** nodes

Prefixes are not only stored at leaf nodes but also at internal nodes



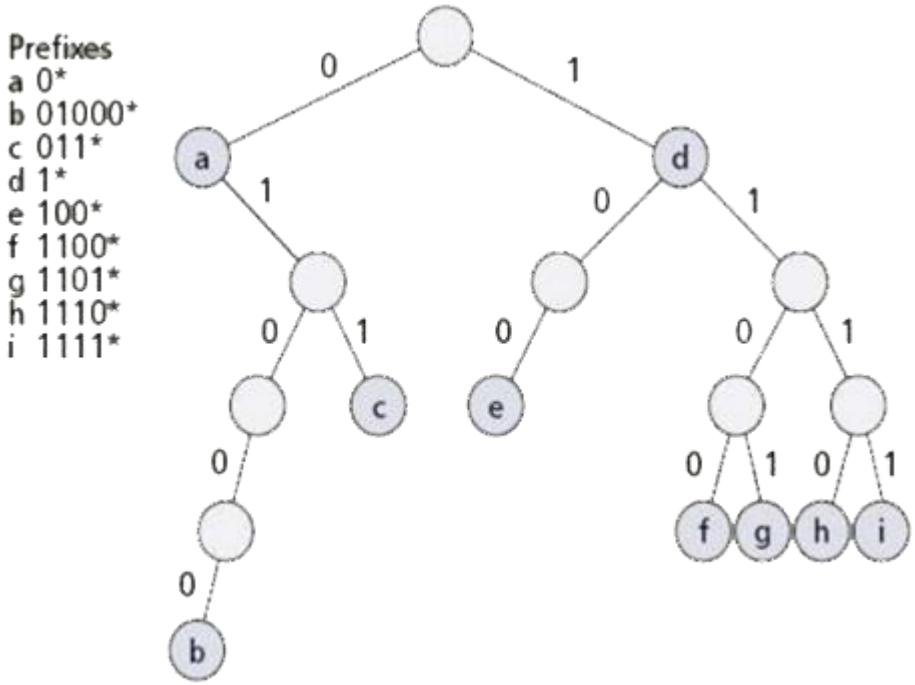
Binary Trie

Structure

- Each leaf contains a possible address
- Prefixes in the table are marked (dark)

Search

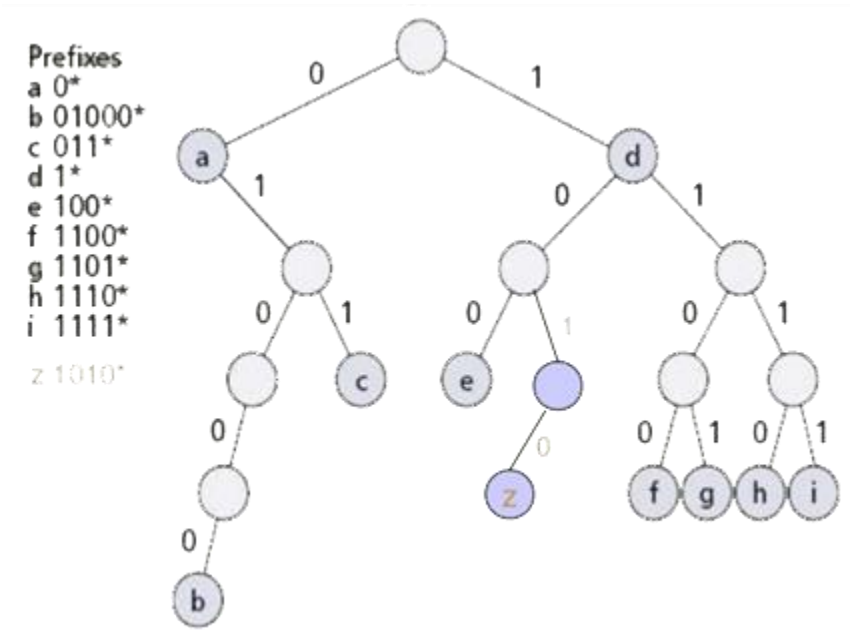
- Traverse the tree according to destination address
- Most recent marked node is the current longest prefix
- Search ends when a leaf node is reached



Binary Trie

Update

- Search for the new entry
- Search ends when a leaf node is reached
- If there is no branch to take, insert new node(s)



Compressed Binary Trie

Goal: Eliminate long sequences of 1-child nodes

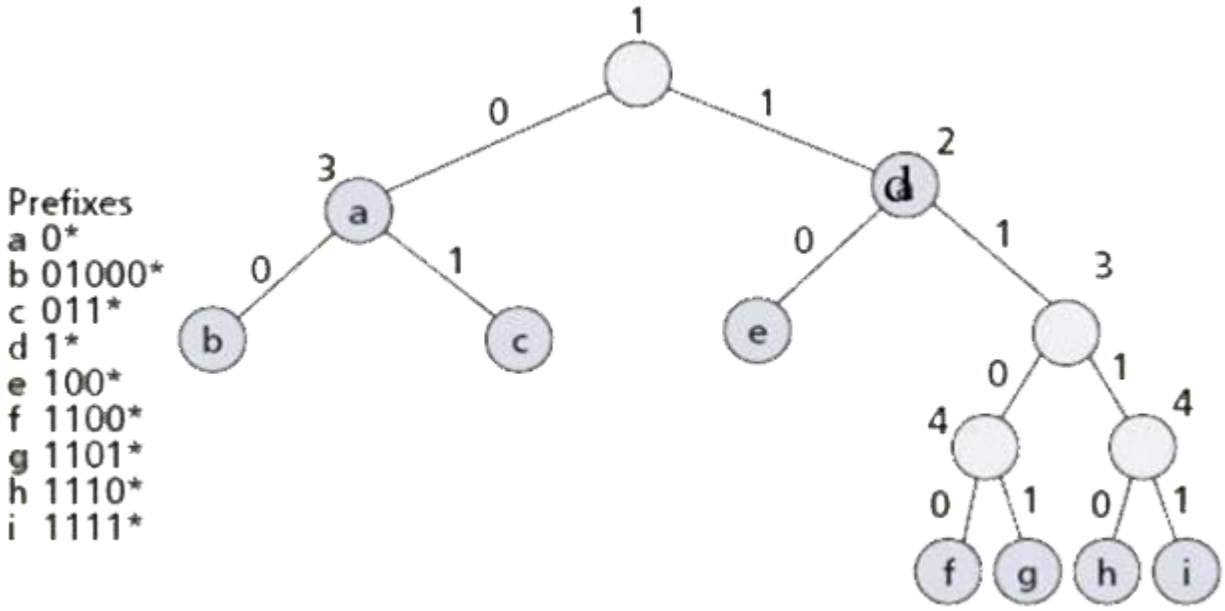
Path compression -> collapses 1-child branches

Path Compression:

- Required to store additional information with nodes -> Bit number field is added to node
- Bit string of prefixes must be explicitly stored at nodes
    - Need to make comparison when searching the tree

Search: “010110”

- Root node: Inspect 1st bit and move left
- “a” node:
    - Check with prefix of a (“0*”) and find a match
    - Inspect 3rd bit and move left
- “b” node:
    - Check with prefix of b (“01000*”) and determine that there is no match
    - Search stops. Longest prefix match is with a



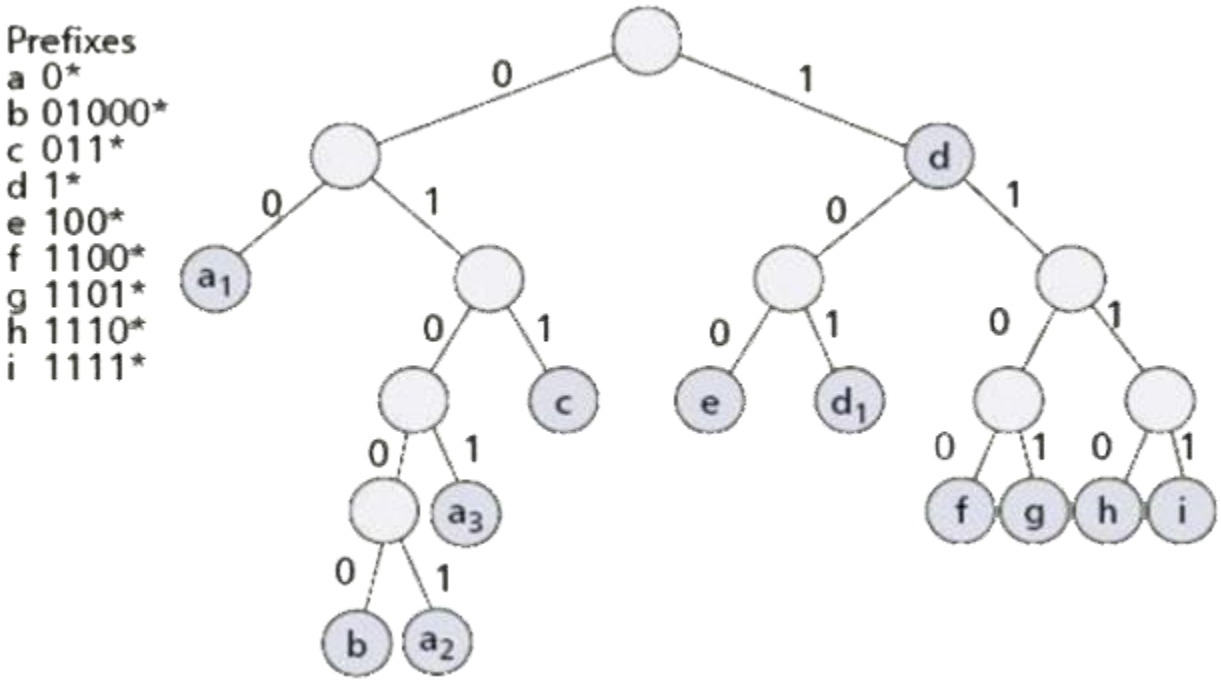
Disjoint-Prefix Binary Trie

Multiple matches in longest prefix rule require backtracking of search

Goal: Transform tree as to avoid multiple matches

Disjoint prefix:

- Nodes are split so that there is only one match for each prefix ("Leaf pushing")
- Consequence: Internal nodes do not match with prefixes
- Results:
    - a (0*) is split into: a1 (00*), a3 (010*), a2 (01001*)
    - d (1*) is represented as d1 (101*)



Variable-Stride Multibit Trie

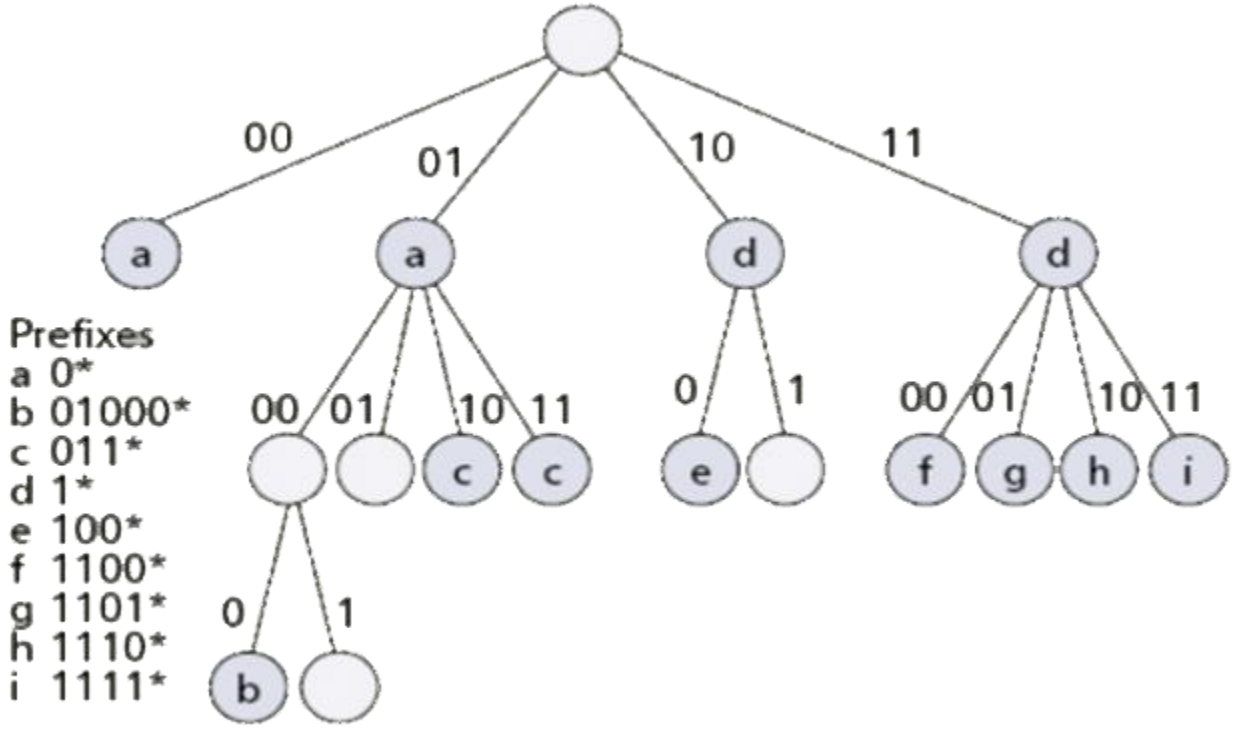
Goal: Accelerate lookup by inspecting more than one bit at a time

"Stride": number of bits inspected at one time

With k-bit stride, node has up to 2k child nodes

2-bit stride:

- 1-bit prefix for a (0*) is split into 00* and 01*
- 1-bit prefix for d (1*) is split into 10* and 11*
- 3-bit prefix for c has been expanded to two nodes
- Why are the prefixes for b and e not expanded?



Conclusions: Tries

Excellent data structure for managing Strings

Supports prefix and suffix kind of lookups

Extremely fast – After the Trie has been built, the search time is m where m is the size of the pattern.

Can be used to build indexes

Various applications in areas that use Strings (Literature/Dictionary/Content, as well as Networks and Bioinformatics)