PREFIX SEARCH

GIVEN A DICTIONARY D OF K STRINGS, WHERE THE SUM OF LENGTHS OF ALL THE STRINGS IS N, WE WANT TO STORE THEM IN A WAY THAT EFFICIENTLY SUPPORT PREFIX SEARCHES FOR A PATTERN P.

EX

prefix search of pa of $S = \{$abaco, box, paolo, patrizia$\}$

This type of search is usually done with tries (actually, with Patricia trees or also called Radix Trees, that are a compacted version of a trie)

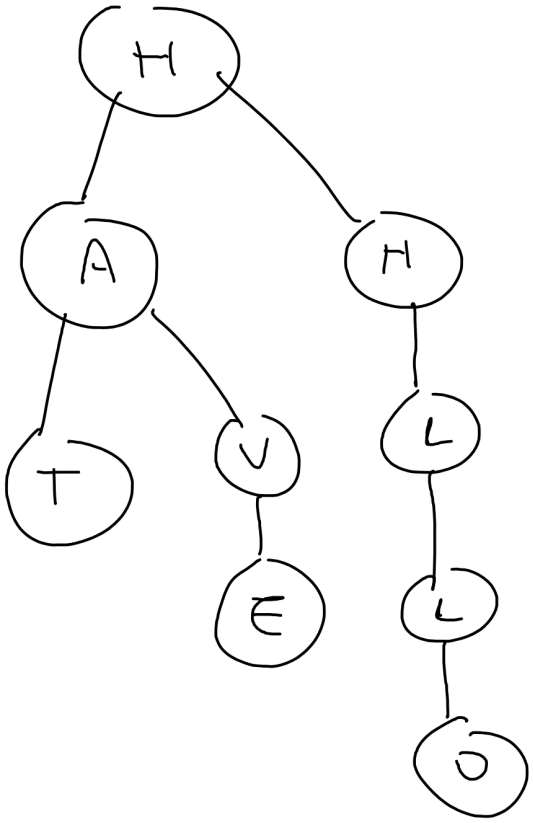
Difference between a trie and a compacted tree?

The compacted tree labels the edges, not the nodes. This implies less space usage (you can compact multiple labels on a single edge), but require a more complex implementation

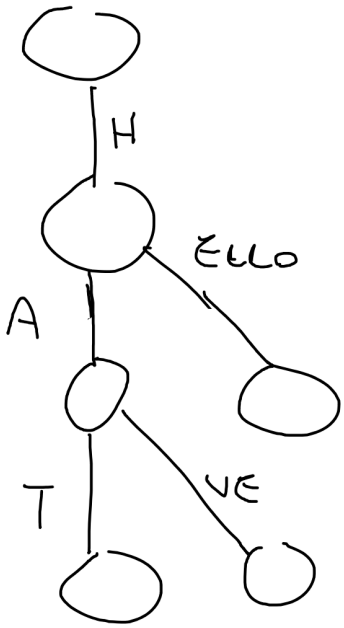
EXAMPLE

$S = \{Hello, Hat, Have\}$

TRIE



Compact tree



Search cost $O(p)$ (scan path that matches the prefix)

However, cache miss at any node

2-level Indexing

Partition the strings we want to search on in blocks of size B, keep in memory the first element of each block, keep in disk the rest and store in the leaf of the trie a pointer to those first elements.

Therefore, the search is divided in 2 phases:

1) Search in trie for the lexicographic position of the queried prefix

2) Retrieve from disk the corresponding block and scan it for prefix search

(ONLY ONE I/O)

Advantages

- Fewer comparison in memory (comparing only the first element of each block) + 1 I/O
- Less space used (same motivo)

EXAMPLE

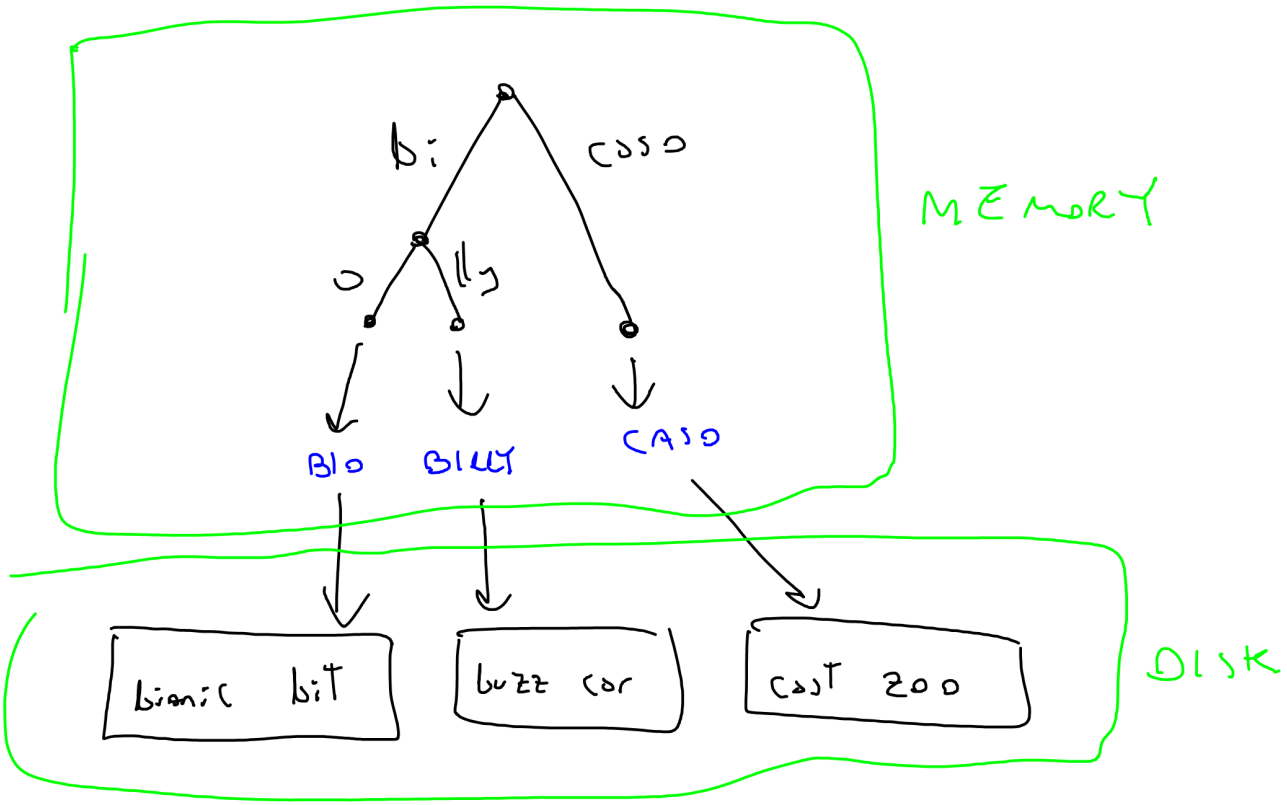
$S = \{ bio, bionic, bit, billy, buzz, car, caso, cast, zoo \}$

$B = 3$

$B_1 = \{ bio, bionic, bit \}$

$B_2 = \{ billy, buzz, car \}$

$B_3 = \{ caso, cast, zoo \}$



Note:

To further reduce space usage, we could compress the elements in disk by using front encoding, starting from the first element of each block:

BIO 3 BIONIC 2 BIT ; BILLY 1 BUZZ CAR ; CASO 3 CAST ZOO