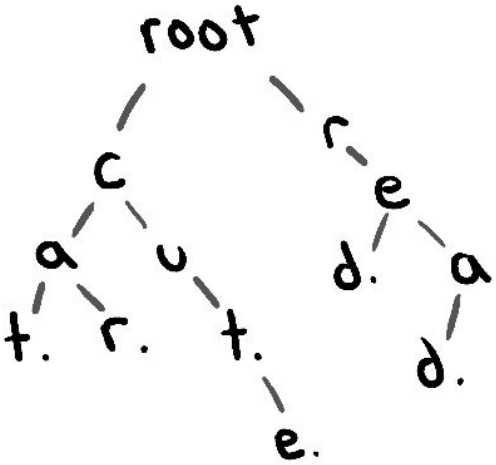
# Trie

By Max, Peter, Kenneth, and ChrisT

# What is a Trie?

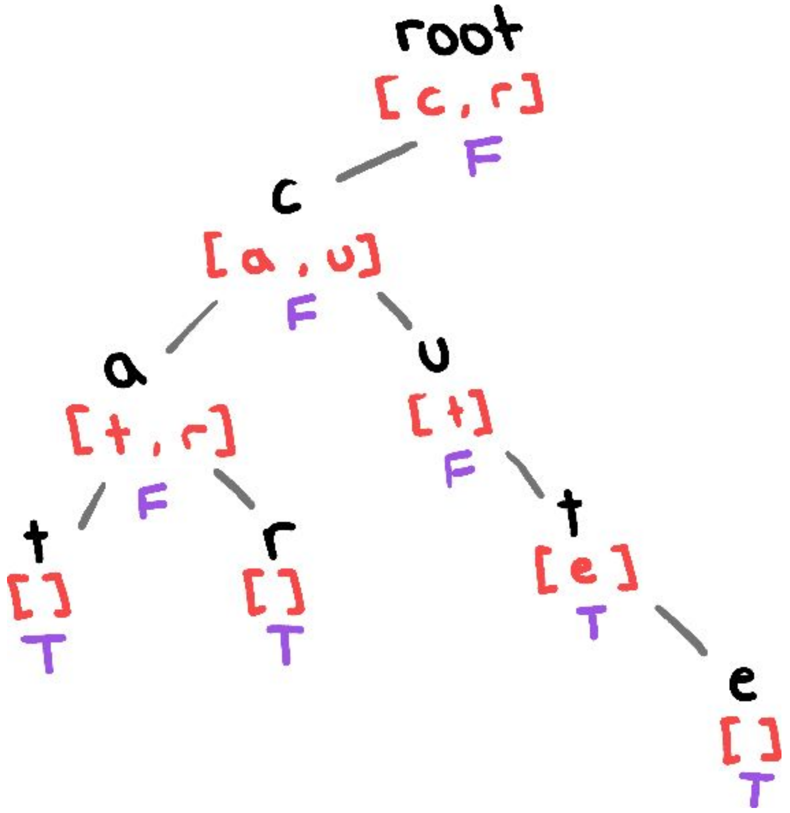
- Tree data structure used to store and perform lookup on strings
- Stores common prefixes as common ancestors in a tree



[“car”, “cat”,  “cut”, “cute”, “read”, “red”]

# Constructing a Trie

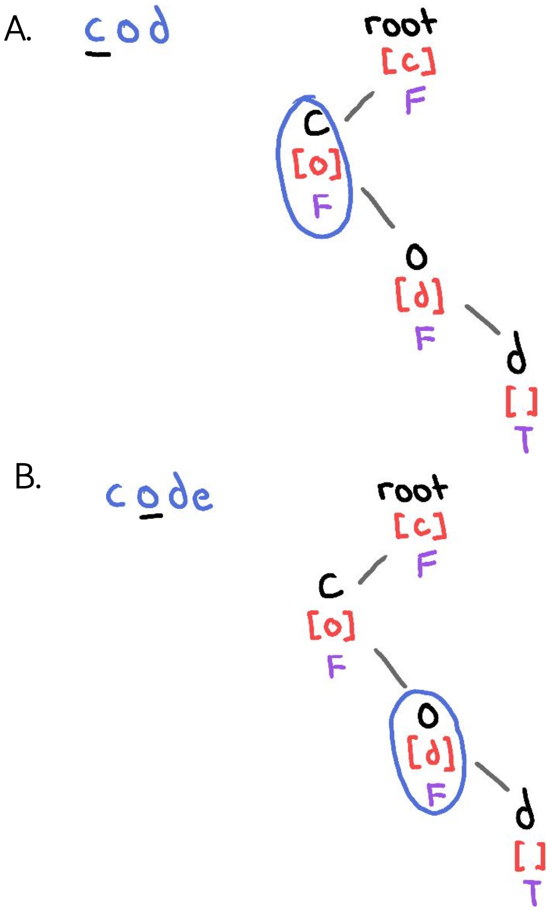
- How to represent letters and periods
- Node
  - Stores information
  - Links to other nodes
- Each node of the tree stores
  - An array of child nodes indicating the next letter
  - A boolean indicating if it is the end of a word



# Constructing a Trie

1. Start at the root node
2. Check if the current indexed letter exists inside the children array
3. If it does, go to step 5
4. If it does not, create the node and go to step 5
5. Go to the child node
6. If the current index is the last letter of the string, set boolean to true then go back to step 1
7. If the current index is not the last letter of the string, increase the index and go back to step 2

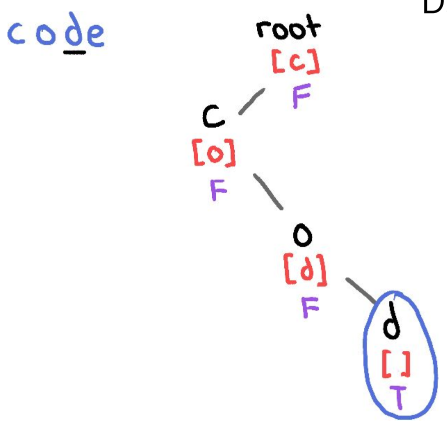
Adding “code” to a trie with “cod”



Adding “code” to a trie with “cod”

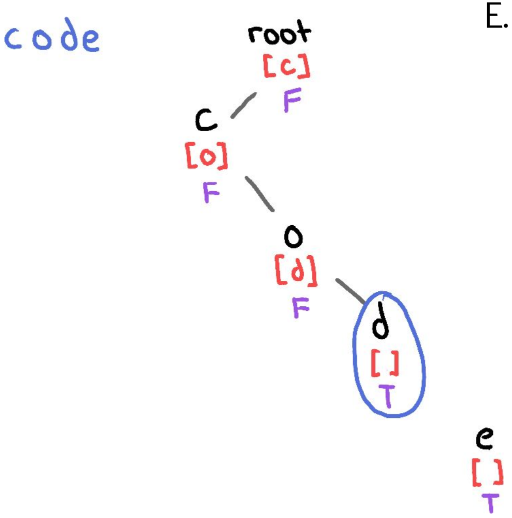
# Constructing a Trie

C.



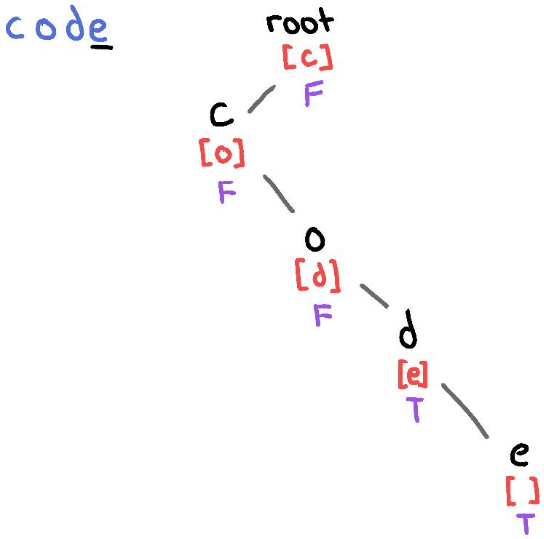
‘e’ is not a child node

D.



create ‘e’ node

E.



connect ‘e’ node

# Applications

Similar to a hash table for strings:

- Supports insertions and deletions
- 0 hash collisions

Traversing the trie can sort all the strings

A type of trie called the suffix tree can support many string operations, like string searching

# Chris said some things

- BBST-esque operations
  - The tree structure allows for queries similar to order statistic tree
  - This is achieved with a bitwise trie (instead of letters use bits)
  - insert, delete, succ/predecessor
  - k$^{th}$ smallest element and # elements ≤ k (order of k)

Insertion and deletion is the same

Successor/predecessor: find the next/previous leaf node, first by going up until you are able to go back down

# Chris said some things

- BBST-esque operations
  - k$^{th}$ smallest element and # elements ≤ k (order of k)

Similar to regular order statistic tree: in every node store the number of leaves in the left subtree and right subtree

Using this you can “binary search” for the k$^{th}$ smallest element

To find the order of k, repeatedly add the subtree sizes of the subtrees on the left of k

# Chris said some things

- Sparse segtree
    - Same as normal segment tree
    - Instead of implementing the tree structure with an array, use a normal tree
    - Only allocate nodes when necessary

# Sample Problem

https://dmoj.ca/problem/fhc15c1p2

Since you crave state-of-the-art technology, you've just purchased a phone with a great new feature: autocomplete!

Your phone's version of autocomplete has some pros and cons. On the one hand, it's very cautious. It only autocompletes a word when it knows exactly what you're trying to write. On the other hand, you have to teach it every word you want to use.

You have N distinct words that you'd like to send in a text message in order. Before sending each word, you add it to your phone's dictionary. Then, you write the smallest non-empty prefix of the word necessary for your phone to autocomplete the word. This prefix must either be the whole word, or a prefix which is not a prefix of any other word yet in the dictionary.

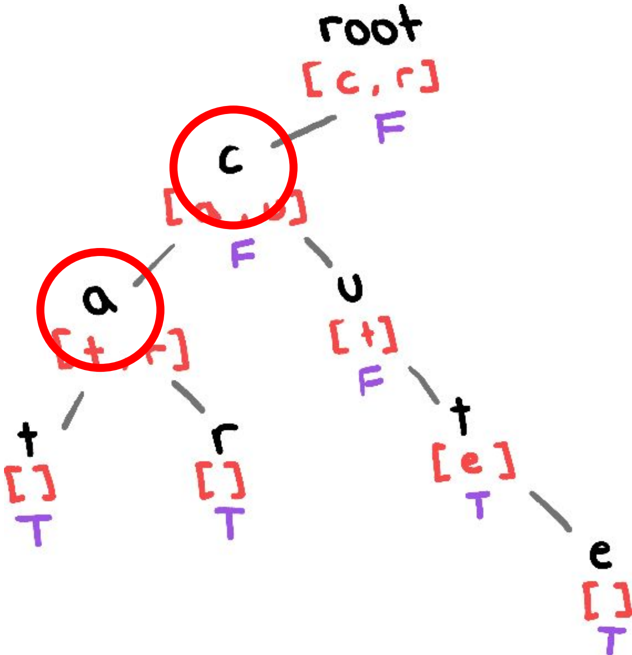
What's the minimum number of letters you must type to send all N words?

# Sample Problem

Use the trie to see how many words have a given prefix.

When exactly 1 had has a prefix, then the prefix in the trie will have 1 leaf in its subtree.

Knowing this, you can traverse the tree to get the answer.



# Practice

- https://dmoj.ca/problem/fhc15c1p2
- https://dmoj.ca/problem/tle16c7p6
- https://mcpt.ca/problem/xorm
- https://codeforces.com/contest/282/problem/E
- https://codeforces.com/contest/665/problem/E
- https://codeforces.com/contest/1416/problem/C
- https://codeforces.com/contest/455/problem/B
- https://dmoj.ca/problem/ds4
- https://mcpt.ca/problem/xorpath (needs centroid decomp)