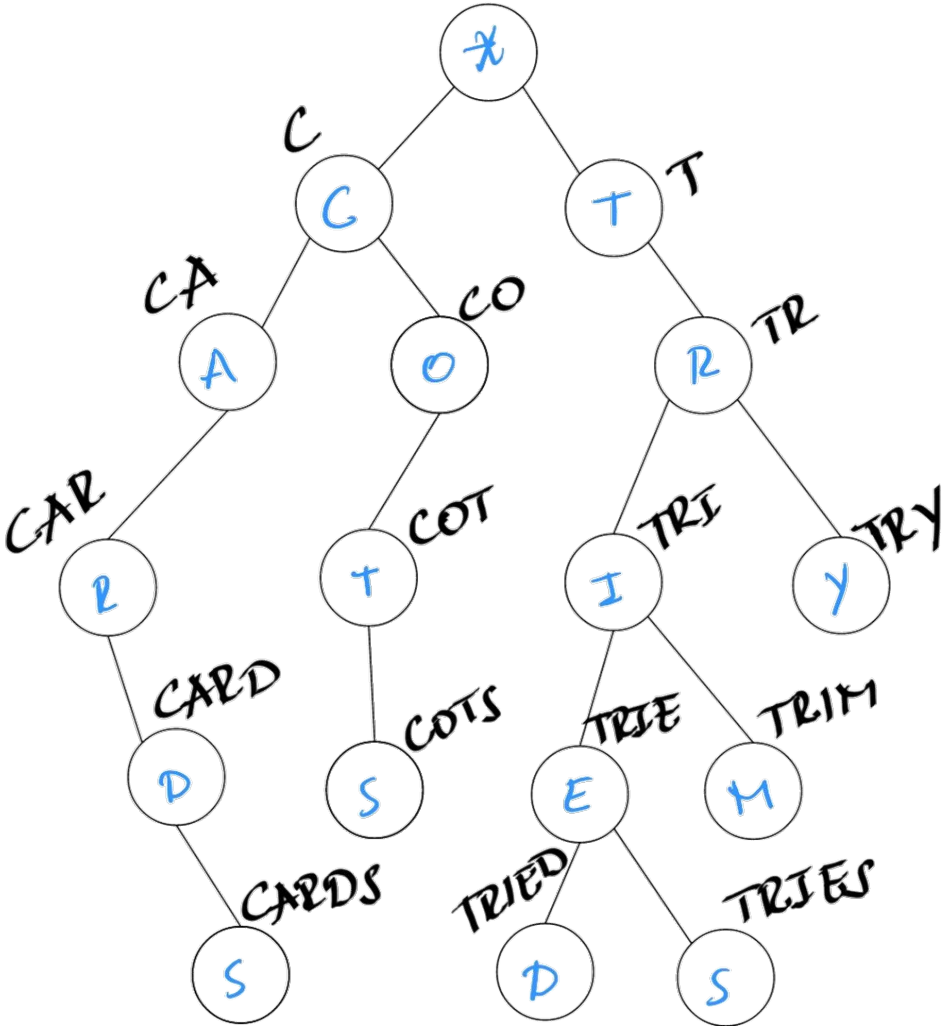

*
C
T
CA
CO
TR
A
O
R
CAR
COT
TRI
TRY
R
T
I
Y
CARD
COTS
TRIE
TRIM
D
S
E
M
CARDS
TRIED
TRIES
S
D
S

Trie

Objective

- Implement trie data structure
- Use trie with other algorithms
- Identify and solve problems using tries

Prerequisites

- Tree
- Recursion and Iteration
- Willingness to learn

Lecture Flow

1. Motivation Problem
2. Meet Tries
3. Structure of Trie
4. Operations on Trie
5. Time and Space Complexity
6. Practice
7. Quote of 2nd Education Phase!

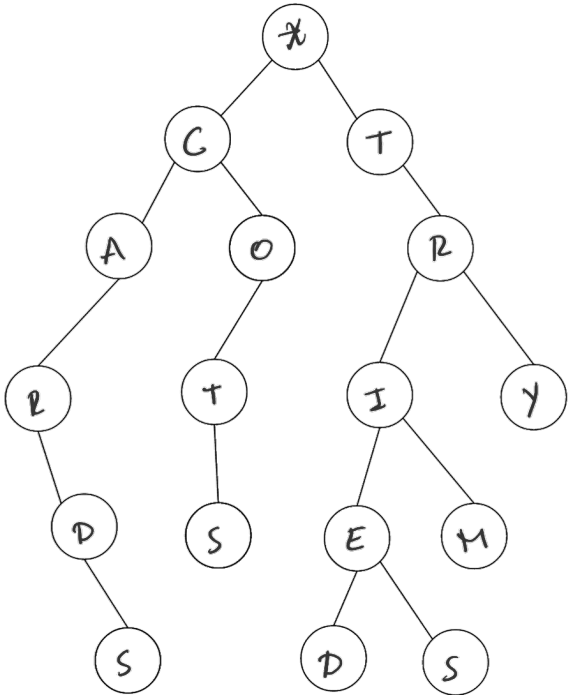
Can you implement spelling checker ?

What are tries?

Tries are just **trees** used for storing a dynamic set of strings, usually associated with their **frequencies or additional information**.

The word "**trie**" comes from the word "**re**trie**val**."

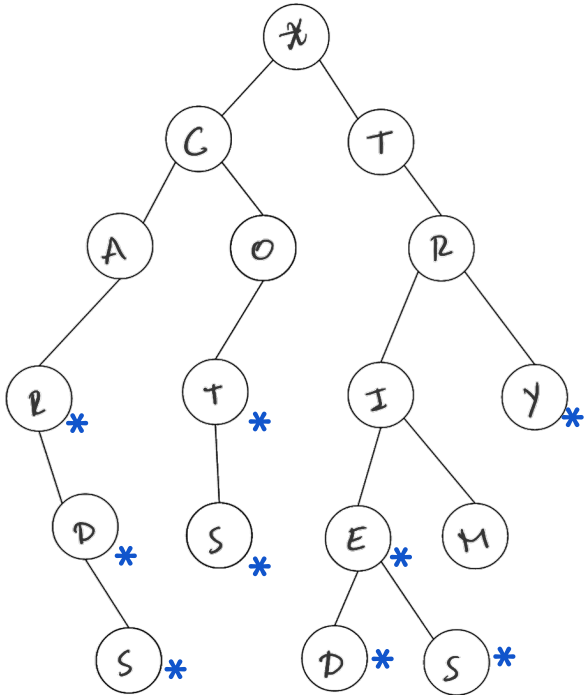
Aka. **prefix tree**, **digital tree**



A2SV

Why trie?

Suppose we use this trie to store this very small version of the English Language. We can use it for **lookups**.



My Dictionary

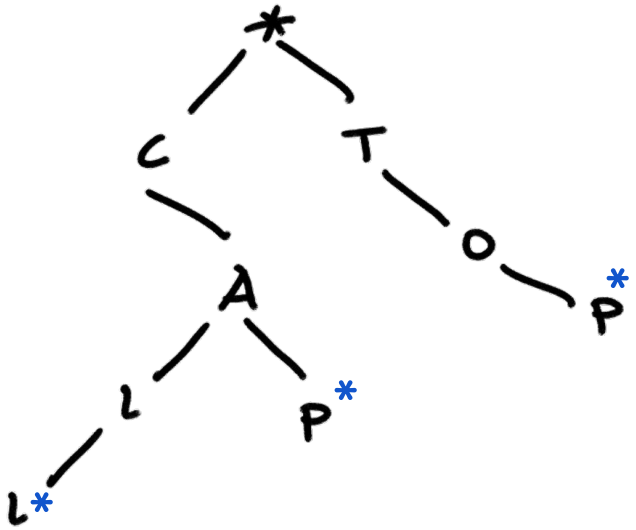
- CAR
- CARD
- CARDS
- COT
- COTS
- TRIE
- TRIED
- TRIES
- TRY

Let's build our spelling checker

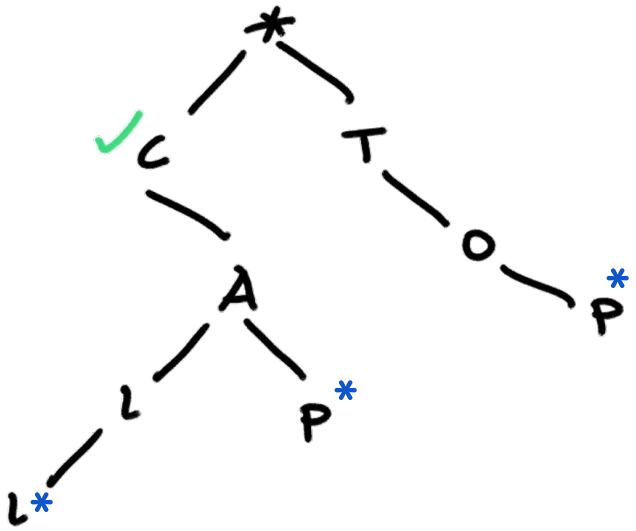
Given the following word bank for simplicity **[CALL, CAP, TOP]**

You are expected to build a feature that, as soon as a user writes a character that causes a word to become **invalid**, you should underline the word in **red**. A word becomes invalid if it does not exist in our word bank.

Step 1: Represent the word bank using a trie



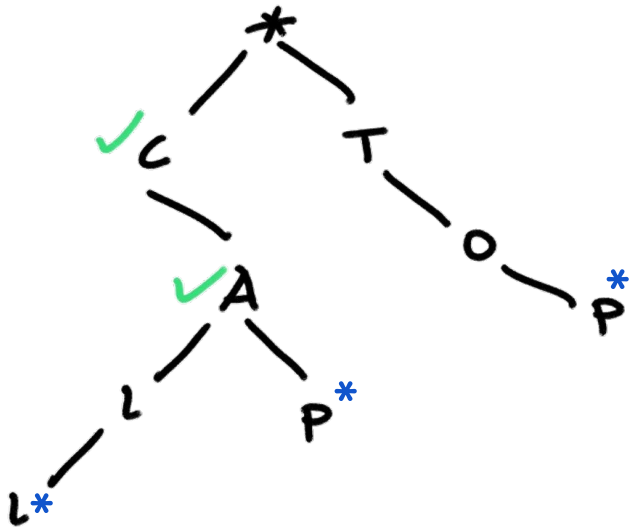
Step 2: Validate as user types



User types

C

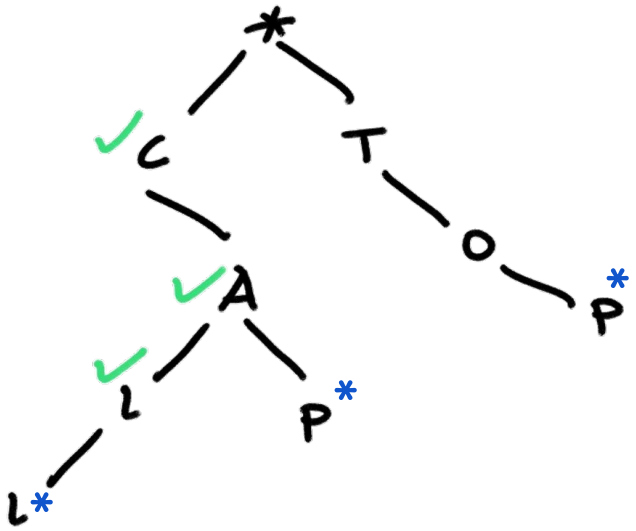
Step 2: Validate as user types



User types

C A

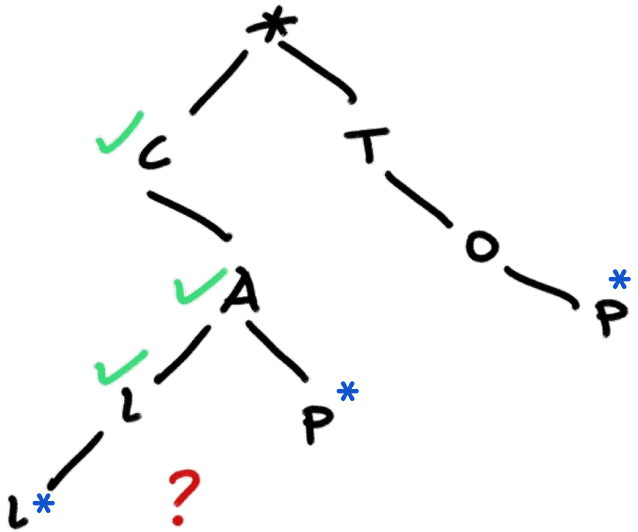
Step 2: Validate as user types



User types

CAL

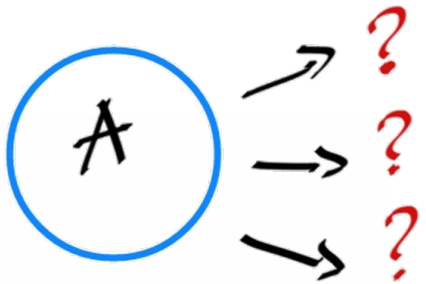
Step 2: Validate as user types



User types

CALP
✓✓✓✗

Why tries when we have hashmaps and sets?



Structure of a trie node

Trie Node

```
class TrieNode:

    # Trie node class

    def __init__(self):

        self.children = [None for _ in range(26)]


        # is_end is True if node represent the end of the word

        self.is_end = False
```

Trie Node

What else can we use to represent the **children** property in a **TrieNode** class?

```
self.children = {}
```

Pros/Cons?

Depending on the problem, **your TrieNode can have different properties!**

Operations on tries

Insertion

When we insert a target value into a **BST**, in each node, we need to decide which child node to go according to the relationship between the value of the node and the target value.

Similarly, when we insert a target value into a Trie, we will also decide which path to go depending on the target value we insert.

Pseudocode

```
1. Initialize: cur = root
2. for each char c in target string S:
3.      if cur does not have a child c:
4.          cur.children[c] = new Trie node
5.      cur = cur.children[c]
6. cur.is_end = True
7. cur is the node which represents the end of string S
```

Visualization

Implement insertion to a trie

Playground

Solution

```
class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word: str) -> None:
        current = self.root
        for ch in word:
            idx = ord(ch) - ord('a')
            if not current.children[idx]:
                current.children[idx] = TrieNode()
            current = current.children[idx]
        current.is_end = True
```

Search prefix

We can go down the tree until we exhaust the given prefix.

- If we can not find the child node we want, **search fails**.
- Otherwise, **search succeeds.**

Pseudocode

```
1. Initialize: cur = root
2. for each char c in target string S:
3.   if cur does not have a child c:
4.     search fails
5.   cur = cur.children[c]
6. search is successful
```

Visualization

Search word

We can treat this word as a prefix and search in the same way we mentioned above. But we need additional check.

- If the prefix search fails, then the **word search fails.**
- If the prefix search succeeds, we need to check if the target word is only a prefix of words in the trie or it is exactly a word. Check the **is_end** flag on the final node.

Visualization

Implement search in a trie

Question

Solution

```
def startsWith(self, prefix: str) -> bool:
    current = self.root
    for ch in prefix:
        idx = ord(ch) - ord('a')
        if not current.children[idx]:
            return False
        current = current.children[idx]
    return True
```

Solution

```
def search(self, word: str) -> bool:
    current = self.root
    for ch in word:
        idx = ord(ch) - ord('a')
        if not current.children[idx]:
            return False
        current = current.children[idx]
    return current.is_end
```

Deletion

The following are possible conditions when deleting word from a trie.

- Word may not be there in trie.
  - Delete operation should not modify the trie.

- Word present as unique key (no part of key contains another word(prefix), nor the word itself is prefix of another word in the trie).
  - Delete all the nodes.

- Key is prefix key of another long key in the trie.
  - Unmark the final node **is_end** flag.

- Key present in trie, having at least one other word as prefix key.
  - Delete nodes from end of key until first leaf node of longest prefix key.

Implement delete on a trie

Playground

Time complexity

- Insert - **O(n)**
- Search - **O(n)**
- Delete - **O(n)**
    - where **n** is length of longest word

Hashmap implementation

Instead of using classes, we can use hashmaps to implement tries.

Hashmap implementation

```
Trie = lambda: defaultdict(Trie)
trie = Trie()

def insert(trie, word) -> None:
    current = trie
    for ch in word:
        if ch not in current:
            current[ch] = Trie()
        current = current[ch]
    current[“is_end”] = True
```

Comparison with hashtable

The space complexity of hash table is **O(M * N)**. If you want hash table to have the same function with Trie, you might need to store several copies of the key.

For instance, you might want to store **"a", "ap", "app", "appl"** and also **"apple"** for a keyword **"apple"** in order to search by prefix. The space complexity can be even much larger in that case.

The space complexity of Trie is **O(M * N)** as we estimated above. But actually far smaller than the estimation since there will be a lot of words have the similar prefix in real cases.

Trie wins in most cases!

Space complexity

If we have **M words** to insert in total and the **length of words is at most N**, there will be at most **M*N nodes** in the **worst case** (any two words don't have a common prefix).

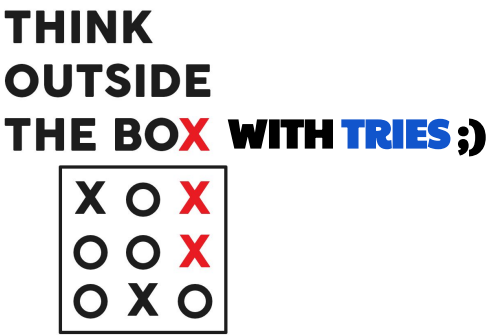
Let's assume that there are **maximum K unique characters** (K is equal to 26 when considering ascii lowercase strings, but might differ in different cases). Therefore, the space complexity will be **O(M*N*K).**

It seems that Trie is really **space consuming**, however, the **real space complexity of Trie is much smaller than our estimation**, especially when the distribution of words is dense.

Remark!

Tries are extremely flexible data structures, yet you **rarely use them by themselves** to solve problems. You'll use them more frequently in conjunction with other data structures and algorithms.

You will most likely add more attributes to the TrieNode or Trie class!



Bits and Tries

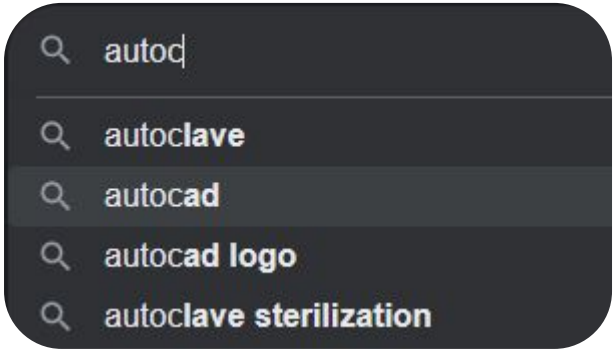
- Problem

Practice

- Longest Common Prefix

Can you implement search suggestion system?

Try Here!



Pair Programming

- Longest Word in Dictionary
- Map Sum Pairs

More Practice Questions!

- Replace Words
- Design Add and Search Words Data Structure
- Number of Matching Subsequences
- Prefix and Suffix Search
- Sum of Prefix Scores of Strings
- Word Break
- Word Break II


If you can't do great things, do small things in a great way.
Napoleon Hill
quotefancy
A2SV