Topics: - trie intro
- search
- insert
- Frequency
- remove
- Shortest unique prefix

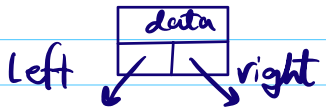
read like "TRY"

Trie is a tree data structure that

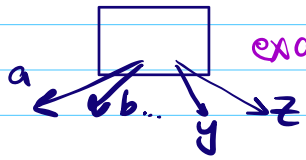
Stores data from top to bottom

types of trie
- tries of chars → today lower case 26 letters
- tries of bits → next sessions

vs. bin tree

TNode


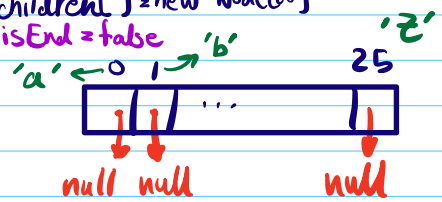
index

trie node


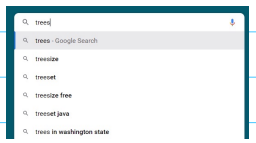
exactly 26 children

index
c - 'a' → 0 to 26

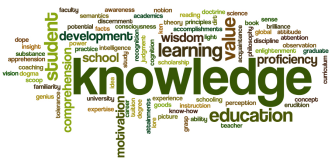
trie def.
```
class Node{
    bool isEnd;
    Node children[]
    Node(){
        children[] = new Node[26]
        isEnd = false
    }
}
```
index + 'a' → 'a' to 'z'
EOW: end of word



trie real applications.

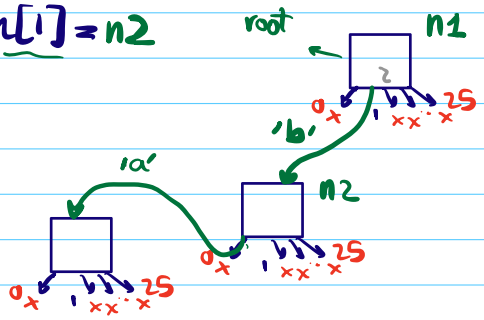


suggestion & autoComplete



word Count ✓

n1.children[1] = n2



index + 'a'
label of edge

This is a treex

spell checker ✓
't' - 'a'

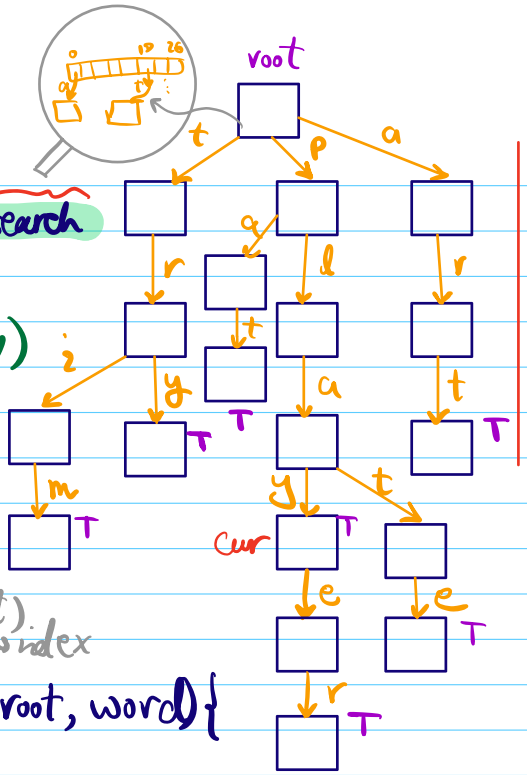
Impatient questions on tries
- How to build?
- How to use?
- Applications?
- How to update?
- How to represent a char?
- terminal node? end of word?
- Can we use hashmap instead?
- Compresion? Huffman tree? beyond the scope

example vocabelary

try
play
art
trim
plate
pat
player

(w)

playxer
0 1 2 3 4



trie lookup/search

TCs O(w)

SCs < O(wxV)

"play"

"try player"

player

(Node, int) index

true/false

search example

```
bool search(root, word){
    cur = root
    n = word.len
    for(i=0; i<n; i++){
        ch = word[i]
        index = ch - 'a' // [0,25]
        if(cur.children[index] == null){
            ret false
        }
        cur = cur.children[index]
    }
    ret cur.isEnd // if true
}
```

TCs O(n)

SCs ?

O(1)

play

players

Q

Insert

i

arm trap

assignment

break?

spell check with hashset {all the vocabs}

lookup TCs O(w)+O(1) ~O(w)

SCs O(VxW)

hashSet<String>

try
play
art
trim
plate
pat
player

O(w)

avg w: len of word
V: # of words

not null

while {tmp = tmp.next}
{tmp = tmp.left}

→ changes to above code in (*)

→ remember to set isEnd

```
cur.children[ind] = new Node();
cur = cur.children[index]
```

SC

why not hash map?

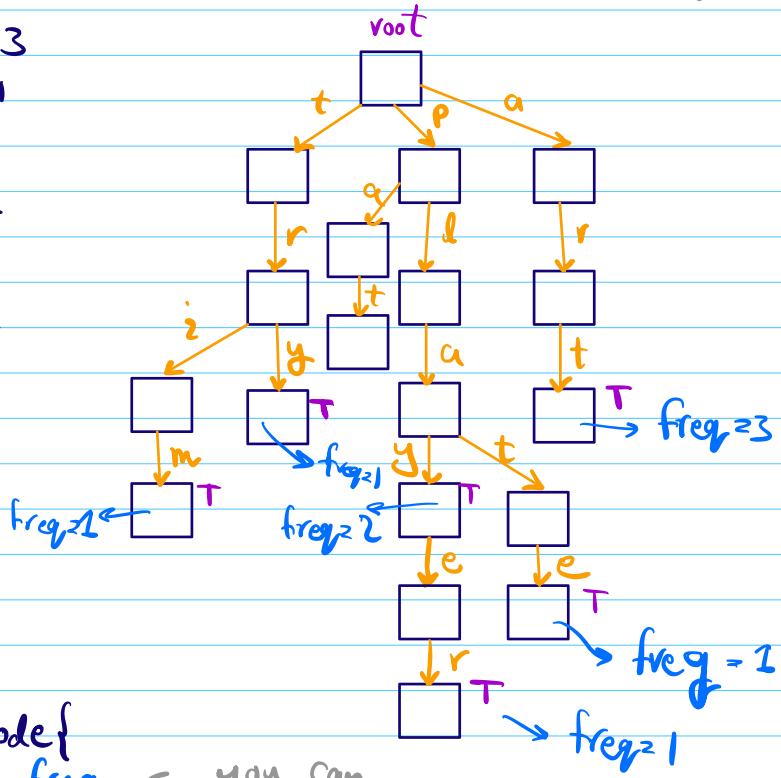
Frequency of words in a text

optional

ex
text:
art
try
play
art
trim
plate
pat
player

art
trim
play

art, 3
try, 1
⋮
play, 2
⋮
trim, 1




```
class Node{
    int freq   ← you can merge with isEnd.
    Node children[]
    Node(){
        children[]=new Node[26]
        freq=0
    }
}
```

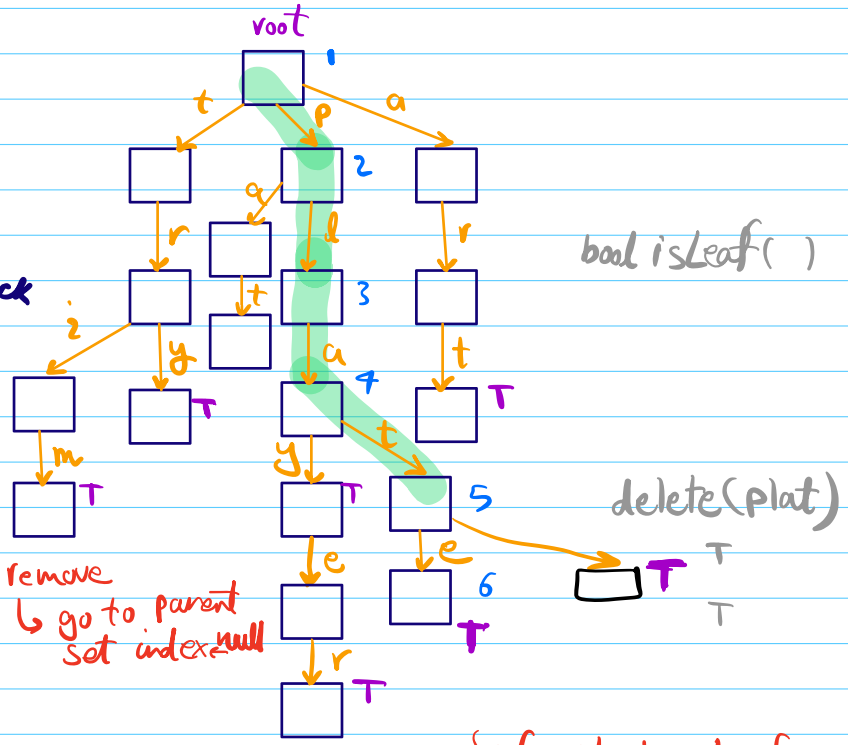
isEnd>0
terminal node

freq only assign or ++ if isEnd==true

remove word delition from a trie

delete (Plate)

delete (Play)



bool isLeaf( )

① search & build stack

Q always safe to do to remove

step1

② n_{last}.isEnd=false

if can clean up do that

stack<node>

<node, index>

remove
↳ go to parent
set index=null

S

safe to remove node ← { if it is leaf && isEnd==false

step 2 cleanup

```
while( cur!=null && cur is leaf ){
    cur = st.pop();
    if( cur is leaf & cur.isEnd==false )
    else break
    cur = (!st.empty()) ? st.peak() : null
}
```

assignment ← how?

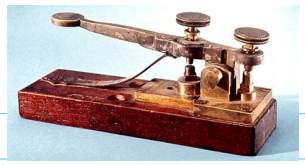
remove → get parent and set the ref=null

word size

TC: O(W)

SC: O(W)

Shortest Unique Prefix.



telegraph device

{lets, gone, lamp, cat, game}

le ga

le g la

lets game lake

P1 Given list of words, for each word find the shortest prefix that can uniquely identify the word.

ex {trie, trap, plate, part, cat, Place, tie, Party}

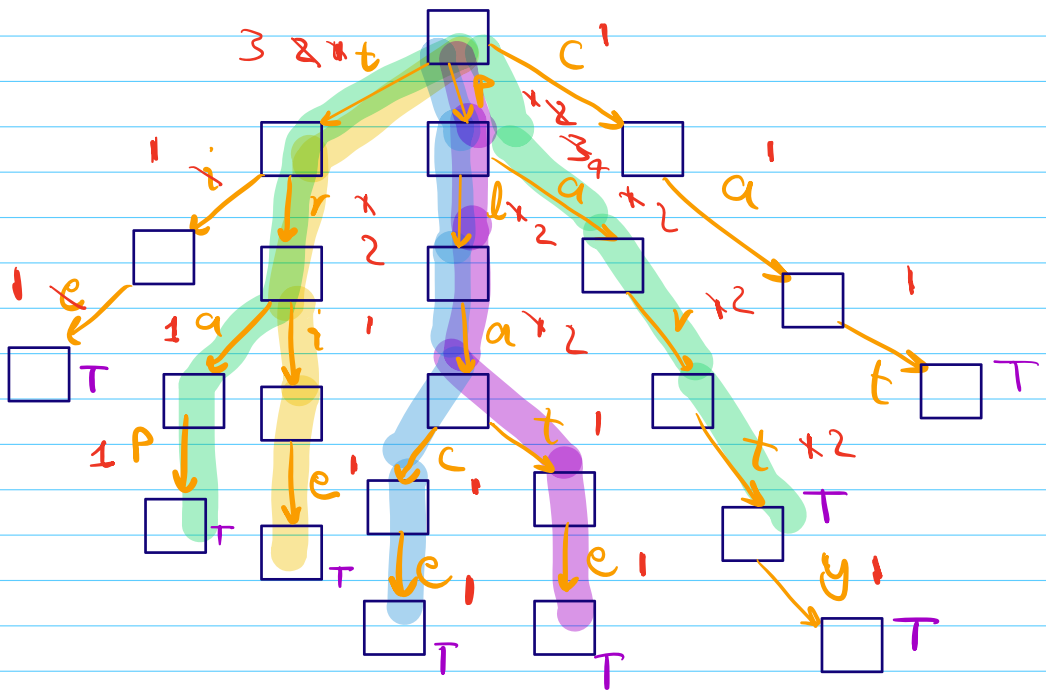
ans {tri, tra, plat, part, c, Plac, ti, Party}

children[26]

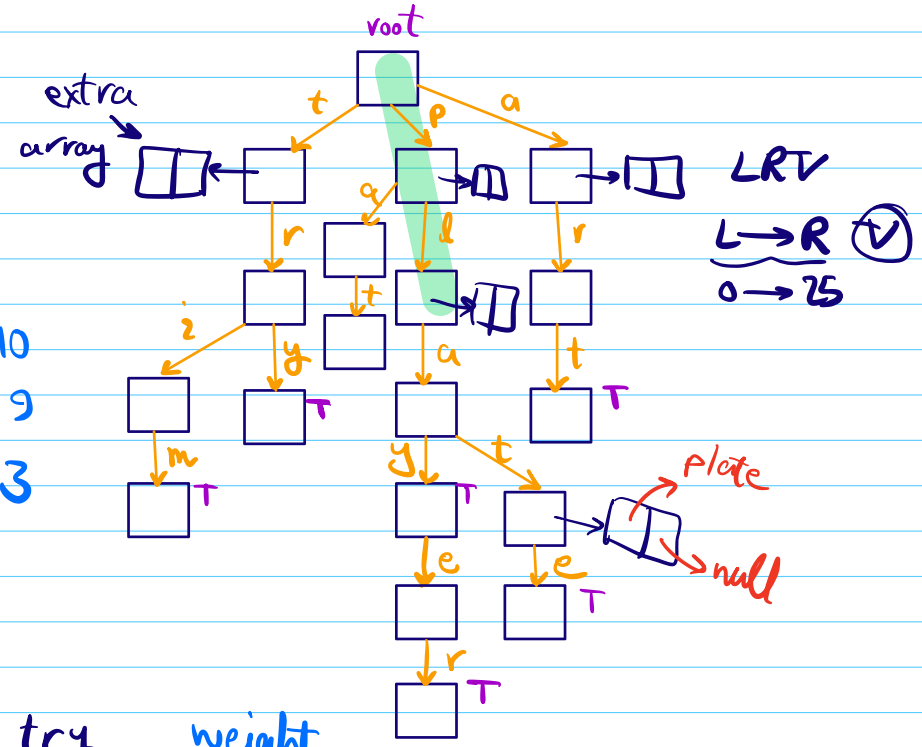
children_count[26]

3 min

Note Freq is diffivent from word count



hints on auto complete



config → max number of suggestion
- play, 10
- player, 9
- plate, 3

vocab
- try
- play
- art
- trim
- plate
- pat
- player

weight

Advanced DSA : Tries 1: Trie of Character_base_25-Oct-2023-13:38

Edit Preview

Advanced DSA : Tries 1: Trie of Character

---

title: Introduction to Trie
description:
duration: 300
card_type: cue_card

Introduction

Definition

A trie, also known as a **prefix tree**, is a tree-like data structure used for efficiently storing a dynamic set of strings or keys, usually associated with text. The term "trie" comes from the word "retrieval" because one of its main applications is in searching and retrieving words or strings based on a prefix or a sequence of characters. This data structure organizes strings by their common prefixes and is characterized by nodes that can have multiple children, typically one child for each possible character in the alphabet.

Example

Here's an example of a trie to store a few words: "cat", "car", "bat", and "ball".

Explanation:

- The root node has three children: 'c', 'b', and 'b'.
- The first word, "cat", starts with 'c', so we follow the 'c' child.
- The second level has 'a' as a child node, so we follow it.
- The third level has 't' as a child node, and since this is the end of the word, we mark it as an end node.
- Similarly, we insert "car", "bat", and "ball", following the respective paths.

Practical Applications :

Spell Checker:

Spell checker can be implemented to check if a given word exists in a dictionary or a list of valid words.
1.Trie Creation:

- Build a Trie data structure where each node represents a character in a word.
- Insert each word from the dictionary into the Trie by creating nodes for each character and marking the last node of each word as the end of a valid word.

2.Spell Checking:

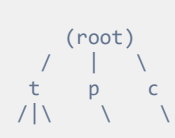
- To check the spelling of a word:
- Start at the root of the Trie.
- Traverse the Trie character by character.
- Move to the corresponding child node for each character in the word.
- If you can't find a child node for a character, the word is not in the dictionary, and it's misspelled.
- If you successfully traverse the Trie to the end of the word, and the last node is marked as the end of a valid word, then the word is spelled correctly.

Time complexity:
Spell checking with a Trie has a constant and predictable time complexity of **O(m)**, where "m" is the length of the word you're checking. This efficiency is independent of the dictionary's size, making Tries an excellent choice for spell checking, even with large dictionaries.

Auto-Complete :

When a user starts typing a word or phrase, you want to suggest possible completions based on what they've typed so far. A trie can be an excellent data structure for implementing this feature efficiently.
Consider the dictionary : "try," "trim," "trie," "play," "trying," "plate," "car," "par," and "trimmer."

Here's the representation of the trie :



```
      r i r    l    a
      | | |    |    |
      y e i    a    r
             /|\
            n g e
            | | |
            g i r
            | | |
            r m m
            | |
            e r
```

This Trie structure allows for efficient autocomplete suggestions for words that share the same prefix. For example:

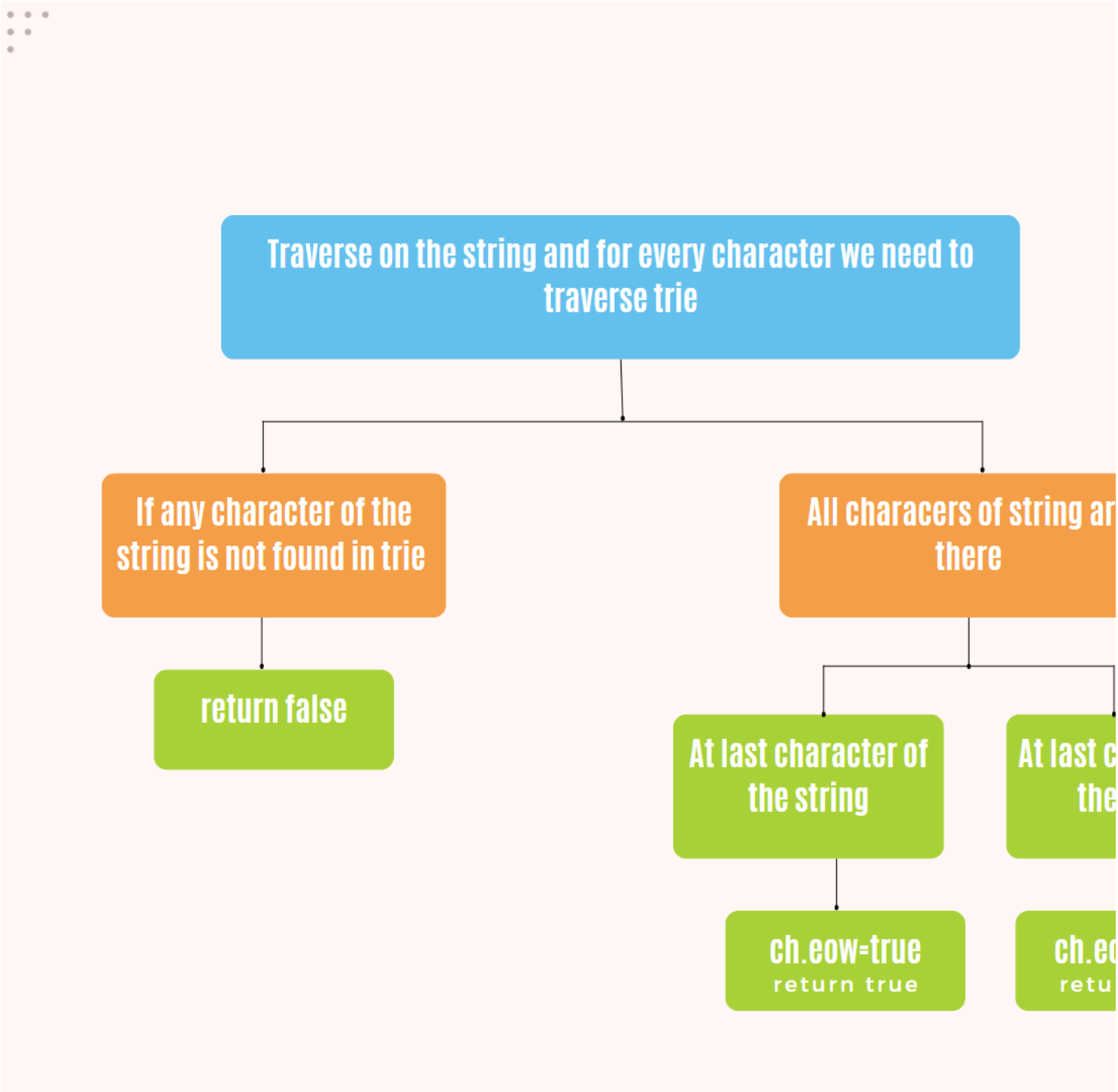
- If a user types "tri," it quickly find "try," "trim," and "trie" as autocomplete suggestions.
- If a user types "pl," it provides "play" and "plate" as suggestions.
- If a user types "ca," it suggests "car."
- If a user types "par," it suggests "par."

Pseudocode

```
class Node {
public:
    char data;             // A character stored in the node
    Node* children[26];    // An array of child nodes for each letter of the alphabet
    bool is_end_of_word;   // A marker to denote if this node marks the end of a word

    Node(char ch) {
        data = ch;
        for (int i = 0; i < 26; i++) {
            children[i] = nullptr;  // Initialize all child nodes to nullptr
        }
        is_end_of_word = false;  // Initialize the end-of-word marker to false
    }
};
```

Adding the `is_end_of_word` marker to a Trie node is like placing a flag to indicate if the current node represents the end of a complete word. This flag helps efficiently identify full words when searching or suggesting words in a Trie, making it handy for tasks like spell checking and autocomplete.



---

title: Insertion in Trie
description:
duration: 600
card_type: cue_card

Insert in Trie

Inserting a value into a Trie is a fundamental operation in data structures and algorithms. Each node in a Trie represents a single character, and the path from the root of the Trie to a node spells out a prefix of one or more keys. Here's how to insert a value into a Trie:

- Start at the root node of the Trie.
- For each character in the value you want to insert:
  a. Check if the current character is already a child of the current node.
  b. If it is, move to that child node.
  c. If it isn't, create a new node for that character and add it as a child of the current node, then move to the new node.
- Repeat step 2 for each character in the value.
- After inserting all characters, mark the node corresponding to the last character of the value as the end of a word (or key). This step is important for searching and determining if the Trie contains a particular key.

Pseudocode

```
 void insert(string word) {
        TrieNode * current = root;

        for (char c : word) {
            if (current -> children.find(c) == current -> children.end()) {
                current -> children[c] = new TrieNode();
            }
            current = current -> children[c];
        }

        current -> isEndOfWord = true;
    }
```

---

title: Quiz 1
description:
duration: 45
card_type: quiz_card

Question

In a Trie, during the insertion of a word, what does each node typically represent?

Choices

- [x] A character in the word.
- [ ] A whole word.
- [ ] A sentence.
- [ ] A phrase.

---

title: Search in Trie
description:
duration: 900
card_type: cue_card

Search in Trie

Searching for a word in a Trie involves traversing the Trie starting from the root and following the path that corresponds to the characters in the word.

Pseudocode

```
 bool search(string word) {
        TrieNode * current = root;

        for (char c : word) {
            int idx = c - 'a';
            if (current -> children[idx] == nullptr) {
                return false; // Character not found
            }
            current = current -> children[idx];
        }

        return current -> isEndOfWord; // Check if it's the end of a word
    }
};
```

Explanation:

1. Start at the root of the Trie.
2. For each character in the input word:
    - Calculate the index (idx) corresponding to the character.
    - If the child node at idx is nullptr, return false.
    - Otherwise, update current to the child node at idx.
3. After processing all characters:
    - Return true if current marks the end of a word.
    - Return false otherwise.

---

title: Deletion in Trie
description:
duration: 600
card_type: cue_card

Deletion in Trie

Deletion in a Trie data structure involves removing a specific key (usually a word or a sequence of characters) from the Trie. The goal of deletion is to ensure that the key is no longer present in the Trie. Here's a general approach to delete a value from a Trie:

- Start at the root node of the Trie.

- Traverse the Trie to find the node that corresponds to the value you want to delete. Keep track of the nodes you visit and the characters in the value.
- If you can't find the value, it's not present in the Trie, so there's nothing to delete.
- If you find the node corresponding to the value:
  a. Mark the `isEndOfWord` flag of the corresponding node as false to indicate that it's no longer the end of a word.
  b. If the node has no other children nodes (i.e., it's not a part of other words), you can safely remove the node and its parent node's reference to it.
  c. Continue this process moving upwards in the Trie, checking if each node can be removed.

Pseudocode

```
 void deleteNode(string word) {
        TrieNode * current = root;
        TrieNode * temp = nullptr;

        for (int i = 0; i < word.length(); i++) {
            int count = 0;

            // Count non-null children
            for (char j = 'a'; j <= 'z'; j++) {
                if (current -> children.find(j) != current -> children.end()) {
                    count++;
                }
            }

            // Check if more than one child or current node is end of another word
            if (count > 1 || current -> isEndOfWord) {
                temp = current;
            }

            char nc = word[i];
            int idx = nc - 'a';

            current = current -> children[idx];
            current -> isEndOfWord = false;
        }

        int count = 0;

        // Count non-null children
        for (char j = 'a'; j <= 'z'; j++) {
            if (current -> children.find(j) != current -> children.end()) {
                count++;
            }
        }

        if (count > 0) {
            return;
        } else {
            temp -> children[word.back() - 'a'] = nullptr;
        }
    }
};
```

Explanation:

1. The code initializes `current` as the root and `temp` as `nullptr` to keep track of the parent node.
2. It iterates through each character in the `word` to be deleted, just like in the previous version.
3. Inside the loop, it calculates the character index `idx` and character `nc` as before.
4. It checks the same conditions for potential node deletion: more than one non-null child or `current` being the end of another word.
5. It then updates `current` to the next node and marks it as not the end of a word.
6. After the loop, it directly checks if `current` has no non-null children (`current->children.size() == 0`). If this condition is met, it erases the corresponding child node from the parent temp.

---

title: Quiz 2
description:
duration: 45
card_type: quiz_card

Question

What is the purpose of the `isEndOfWord` flag in Trie nodes during deletion?

Choices

- [ ] It marks the root node of the Trie.
- [ ] It indicates the presence of an incomplete word.
- [ ] It speeds up the deletion process.
- [x] It signifies the end of a valid word.

---

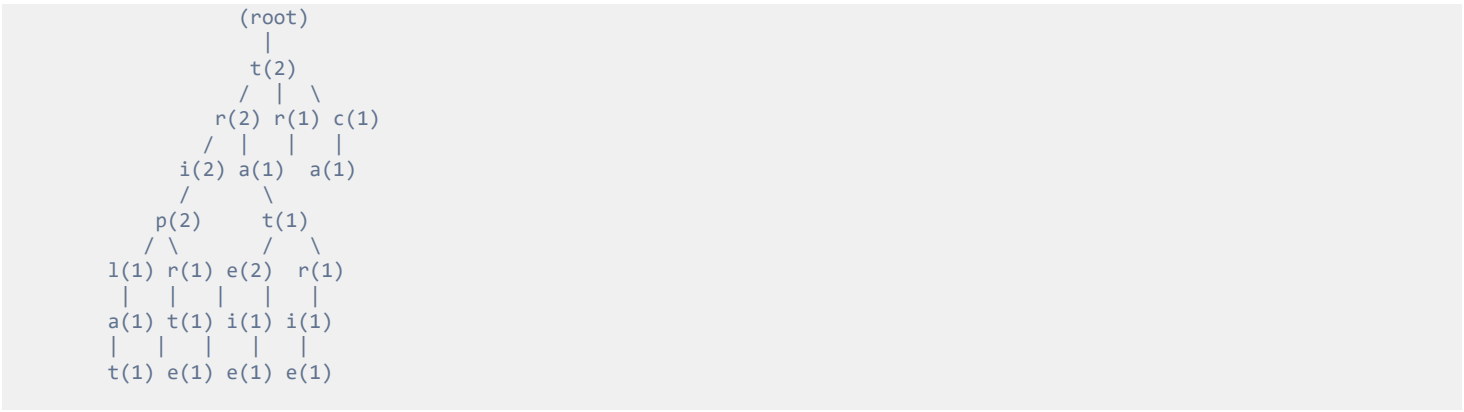
title: Shortest Unique Prefix
description:
duration: 600
card_type: cue_card

Shortest Unique Prefix

Finding the shortest unique prefix in a Trie for a given word or set of words involves traversing the Trie to identify the point where the given word or words diverge from other words in the Trie.

Observation :
Search for each word in the Trie character by character, and stop when you encounter a character node with a frequency of 1. The prefix built up to that point is the shortest unique prefix for the word.

Example:
Let's consider a simple example with a Trie containing the words "tri," "trap," "plate," "cat," "part," "place," and "trie."



The frequencies reflect the number of times each character appears in the Trie for the given words.

In this Trie:

"tri" shares the prefix "tri" with "trie."
"trap" shares the prefix "tra" with "trie" and "trap."
"plate" shares the prefix "pla" with "place" and "plate."
"cat" shares the prefix "ca" with "cat."
"part" shares the prefix "par" with "part."
"place" shares the prefix "pla" with "place" and "plate."
"trie" shares the prefix "tri" with "trie."

Now, you can find the shortest unique prefixes for the given words as follows:

1.**Word: "cat"**

- Start at the root.
- The first character 'c' appears only once in the Trie (frequency 1).
- Move to the 'a' node.
- The second character 'a' appears only once in the Trie (frequency 1).
- Stop here. The shortest unique prefix for "cat" is "cat" because both 'c' and 'a' occur only once in the Trie.

2.**Word: "place"**

- Start at the root.
- The first character 'p' appears multiple times in the Trie (frequency 2).
- Move to the 'l' node.
- The second character 'l' appears only once in the Trie (frequency 1).
- Move to the 'a' node.
- The third character 'a' appears only once in the Trie (frequency 1).
- Stop here. The shortest unique prefix for "place" is "plac" because 'p', 'l', and 'a' are unique at this point.

3.Word: "trie"

- Start at the root.
- The first character 't' appears multiple times in the Trie (frequency 2).
- Move to the 'r' node.
- The second character 'r' appears multiple times in the Trie (frequency 2).
- Move to the 'i' node.
- The third character 'i' appears multiple times in the Trie (frequency 2).
- Stop here. The shortest unique prefix for "trie" is "trie" because 't', 'r', and 'i' are all unique at this point.

Similarly, you can find the shortest unique prefixes for other words as well.