Bitflip's Ultimate Guide to Prefix Trees (Tries)

Python and Java implementations found at the bottom of this PDF

Prefix Tree Functions

Insert

- To insert a word, start from the root and move through each character.
- If a letter's node doesn't exist yet, create it.
- When you reach the last character, mark that node as the end of a word (endOfWord = True).

Search

- To search for a word, start at the root and move down letter by letter.
- If a letter doesn't exist, return False.
- If you reach the last node, return whether endOfWord is True.

StartsWith

- Similar to search, but we don't care about endOfWord.
- If all prefix characters exist, return True.

Get All Words (DFS)

- Once you've navigated to a given prefix node, recursively explore every branch below it.
- Each time you hit a node marked endOfWord, you've found a full word.

Keywords & Key Indicators

- "prefix"
- "autocomplete"
- "dictionary"
- "starts with"
- "suggest words"
- "spell checker"
- "search engine"
- "word search"

Practice Problem Roadmap

- 208. Implement Trie (Prefix Tree) - Start here.
- 211. Design Add and Search Words Data Structure – wildcard support (.).
- 212. Word Search II – combine DFS with Trie for board searches.
- 648. Replace Words – find root prefixes to replace words in a sentence.
- 676. Implement Magic Dictionary – Trie + character substitution logic.
- 745. Prefix and Suffix Search – advanced Trie usage (two-trie or suffix-trie).
- 1268. Search Suggestions System – autocomplete results sorted lexicographically.
- 472. Concatenated Words – use Trie + recursion to detect compound words.
- 336. Palindrome Pairs – Trie + palindrome checking pattern.

Time Complexity

METHOD	WORST CASE
insert()	O(L)
startsWith()	O(L)
search()	O(L)
findAll()	DFS - O(# of letters in tree)

Python Implementation

```python
class TrieNode:
    def __init__(self):
        self.children = {}
        self.end_of_word = False

    def insert(self, word: str):
        node = self
        for c in word:
            if c not in node.children:
                node.children[c] = TrieNode()
            node = node.children[c]
        node.end_of_word = True

    def search(self, word: str) -> bool:
        node = self
        for c in word:
            if c not in node.children:
                return False
            node = node.children[c]
        return node.end_of_word

    def starts_with(self, prefix: str) -> bool:
        node = self
        for c in prefix:
            if c not in node.children:
                return False
            node = node.children[c]
        return True

    def get_words_with_prefix(self, prefix: str) -> list[str]:
        node = self
        for c in prefix:
            if c not in node.children:
                return []  # prefix not found
            node = node.children[c]

        results = []
        self._dfs(node, prefix, results)
        return results

    def _dfs(self, node, path, results):
        if node.end_of_word:
            results.append(path)
        for c, child in node.children.items():
            self._dfs(child, path + c, results)
```

Java Implementation

```java
import java.util.*;

class TrieNode {
    public TrieNode[] children;
    public boolean endOfWord = false;

    public TrieNode() {
        children = new TrieNode[26];
    }

    public void insert(String word) {
        TrieNode node = this;
        for (int i = 0; i < word.length(); ++i) {
            char c = word.charAt(i);
            if (node.children[c - 'a'] == null) {
                node.children[c - 'a'] = new TrieNode();
            }
            node = node.children[c - 'a'];
        }
        node.endOfWord = true;
    }

    public boolean search(String word) {
        TrieNode node = this;
        for (int i = 0; i < word.length(); ++i) {
            char c = word.charAt(i);
            if (node.children[c - 'a'] == null) return false;
            node = node.children[c - 'a'];
        }
        return node.endOfWord;
    }

    public boolean startsWith(String prefix) {
        TrieNode node = this;
        for (int i = 0; i < prefix.length(); ++i) {
            char c = prefix.charAt(i);
            if (node.children[c - 'a'] == null) return false;
            node = node.children[c - 'a'];
        }
        return true;
    }

    public List<String> getWordsWithPrefix(String prefix) {
        List<String> results = new ArrayList<>();
        TrieNode node = this;

        for (int i = 0; i < prefix.length(); ++i) {
            char c = prefix.charAt(i);
            if (node.children[c - 'a'] == null) {
                return results; // prefix not found
            }
            node = node.children[c - 'a'];
        }

        // DFS from that node
        dfs(node, prefix, results);
        return results;
```

```
    }

    private void dfs(TrieNode node, String path, List<String> results) {
        if (node.endOfWord) results.add(path);
        for (int i = 0; i < 26; ++i) {
            if (node.children[i] != null) {
                dfs(node.children[i], path + (char) ('a' + i), results);
            }
        }
    }
}
```