# 🌲 Tries (Prefix Tree) Interview Quick-Check Pattern

*A focused and visual guide to master prefix tree–based problems in interviews.*

---

## 1. 🧠 When to Use a Trie?

Ask yourself:

- ❓ Are you storing or searching **prefixes** or **words**?
- ❓ Do you need to **auto-complete**, **spell-check**, or **prefix match**?
- ❓ Is the question about **shared characters** or **counting overlaps**?
- ❓ Are you trying to **build a dictionary** or **validate substrings**?

> 💡 If you're solving **prefix-related** problems, **Tries** are your go-to structure.

---

## 2. 🔍 Common Problems Tries Are Perfect For

| Problem Type | Trie Use Case |
|---|---|
| Word insert/search/prefix match | Basic Trie structure |
| Auto-complete | Store and return words by prefix |
| Longest prefix match | Deepest match from root |
| Count words with prefix | Use node counters |
| Replace words in a sentence | Use trie for fast prefix replacement |
| Word Search II | Backtrack on trie |
| T9 / Keypad search | Convert digits to letters via trie |

---

## 3. 🏗️ Basic Trie Node & Structure

```
class TrieNode {
  constructor() {
    this.children = {}; // maps character to TrieNode
    this.isWord = false; // marks end of word
  }
}

class Trie {
  constructor() {
    this.root = new TrieNode();
  }

  insert(word) {
    let node = this.root;
    for (let char of word) {
      if (!node.children[char]) node.children[char] = new
TrieNode();
      node = node.children[char];
    }
    node.isWord = true;
  }

  search(word) {
    let node = this._traverse(word);
    return node !== null && node.isWord;
  }

  startsWith(prefix) {
    return this._traverse(prefix) !== null;
  }

  _traverse(prefix) {
    let node = this.root;
    for (let char of prefix) {
      if (!node.children[char]) return null;
      node = node.children[char];
    }
    return node;
  }
```

```
}
```

---

## 4. ✅ Advanced Trie Use Case: Replace Words in a Sentence

```
function replaceWords(dictionary, sentence) {
  const trie = new Trie();
  for (let word of dictionary) trie.insert(word);

  return sentence
    .split(' ')
    .map(word => {
      let node = trie.root, prefix = '';
      for (let char of word) {
        if (!node.children[char] || node.isWord) break;
        node = node.children[char];
        prefix += char;
      }
      return node.isWord ? prefix : word;
    })
    .join(' ');
}
```

---

## 5. 🧱 Edge Cases to Watch For

- Word that's a **prefix** of another (e.g. `app` and `apple`)
- Empty string insertion or search
- Uppercase vs lowercase characters
- Unicode/multibyte characters (beyond a–z)
- Multiple paths with overlapping characters
- Trie depth and memory (space heavy with large dictionaries)

🧠 Always visualize the trie layer by layer to understand behavior.

---

## 6. 🧠 Mental Model for Tries

### Think of a trie like a shared-path tree for strings.

| Question Says… | You Think… |
|---|---|
| “Starts with”, “prefix match” | Trie traversal |
| “Autocomplete” | DFS from prefix node |
| “Replace with root word” | Traverse until `node.isWord == true` |
| “Count words starting with…” | Count nodes from prefix node |
| “Check if word exists” | Search full path, check `isWord` |
| “Build compressed dictionary” | Trie + path compression |

---

## 🔁 Problem Solving Loop

1. 📌 Do I need to **store words efficiently**?
2. 🚀 Is this a **prefix search**, **insert**, or **path-based matching**?
3. 💡 Do I need to **early-stop** traversal once a word is matched?
4. 🛠️ Can I use a **trie traversal helper function**?
5. 🔎 Is the output based on **prefix**, **exact match**, or **returning suggestions**?

---

## ✅ Final Interview Checklist

- Did I structure each node with a `children` map and `isWord` flag?
- Is the input large enough to require a Trie (vs. plain array)?
- Am I correctly distinguishing full words vs prefixes?
- Have I handled corner cases (empty strings, case, overlapping paths)?

- Do I need to **build, search, or traverse** the trie?