Tries (Prefix Trees)

Efficient Prefix-Based Data Structures

Minseok Jeon
DGIST

November 2, 2025

Outline

Introduction to Tries

What is a Trie?

Definition: A tree structure for efficient prefix-based queries.

Key Characteristics:

- Each node represents a **character**
- Path from root to node forms a **string/prefix**
- Children: Map from character to child node
- End marker: Flag indicating complete word

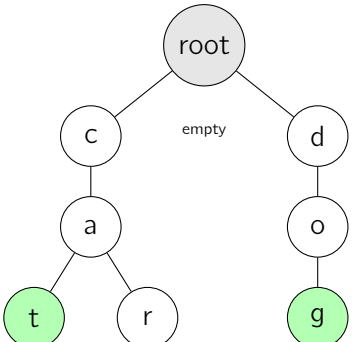
Also Known As:

- Prefix Tree
- Digital Tree
- Radix Tree (compressed variant)

Origin: Name comes from "re**trie**val"

Visual Example

Words: ["cat", "car", "card", "dog"]



Green nodes = end of word

Paths:

c-a-t = "cat"

c-a-r = "car"

c-a-r-d = "card"

d-o-g = "dog"

Why Use Tries?

Advantages:

- **Fast prefix queries:** O(m) where m = prefix length
- **Shared prefixes:** Memory efficient for common prefixes
- **Predictable performance:** No hash collisions
- **Alphabetical ordering:** Natural lexicographic order

Comparison with Hash Table:

Operation	Hash Table	Trie
Search	O(1) avg	O(m)
Insert	O(1) avg	O(m)
Prefix search	O(n)	O(m + k)
Ordered traversal	O(n log n)	O(n)
Space	O(n)	O(ALPHABET * n * m)

m = word length, k = results, n = number of words

Common Applications

Real-World Uses:

- **Autocomplete:** Search engines, text editors
- **Spell checking:** Word suggestions and corrections
- **IP routing:** Longest prefix matching in routers
- **Dictionary:** Fast word lookup and validation
- **Phone directory:** T9 predictive text
- **DNA sequencing:** Pattern matching in genomics

Node Structure and Implementation

Basic Trie Node Structure

```python
class TrieNode:
    def __init__(self):
        self.children = {}  # Map: char -> TrieNode
        self.is_end_of_word = False

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

Node Components:

- `children`: Dictionary mapping characters to child nodes
- `is_end_of_word`: Boolean flag marking complete words

Space per node: ∼100+ bytes (dict overhead + bool)

Array-Based Implementation

For fixed alphabets (e.g., lowercase a-z):

```python
class TrieNodeArray:
    def __init__(self):
        self.children = [None] * 26  # For lowercase a-z
        self.is_end_of_word = False

    def get_index(self, char):
        return ord(char) - ord('a')

    def get_child(self, char):
        index = self.get_index(char)
        return self.children[index]

    def set_child(self, char, node):
        index = self.get_index(char)
        self.children[index] = node
```

Node with Parent Pointers

Enhanced structure for traversal:

```
class TrieNodeWithParent:
    def __init__(self, char='', parent=None):
        self.char = char
        self.parent = parent
        self.children = {}
        self.is_end_of_word = False

    def get_word(self):
        """Reconstruct word from this node"""
        word = []
        node = self
        while node.parent is not None:
            word.append(node.char)
            node = node.parent
        return ''.join(reversed(word))
```

Core Operations

Insert Operation

```python
def insert(self, word):
    """Insert word into trie - O(m) where m = word length"""
    node = self.root

    for char in word:
        if char not in node.children:
            node.children[char] = TrieNode()
        node = node.children[char]

    node.is_end_of_word = True
```

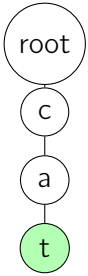
Algorithm:

1. Start at root
2. For each character in word:
   - If child exists, move to it
   - Otherwise, create new child node
3. Mark final node as end of word

Insert Visualization

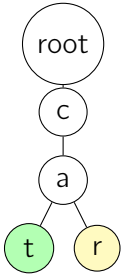
Inserting "car" into trie with "cat":

Before:



Only "cat" exists

After insert("car"):



Both "cat" and "car" exist

Key insight: Common prefix "ca" is shared between words

Search Operation

```python
def search(self, word):
    """Search for exact word - O(m)"""
    node = self.root

    for char in word:
        if char not in node.children:
            return False
        node = node.children[char]

    return node.is_end_of_word
```

Examples:

- `search("cat")` → True (complete word)
- `search("ca")` → False (prefix but not word)
- `search("cats")` → False (doesn't exist)

Important: Must check `is_end_of_word` flag!

Prefix Search (Starts With)

```python
def starts_with(self, prefix):
    """Check if any word starts with prefix - O(m)"""
    node = self.root

    for char in prefix:
        if char not in node.children:
            return False
        node = node.children[char]

    return True  # No need to check is_end_of_word
```

Difference from search:

- Don't check `is_end_of_word`
- Just verify path exists

Examples with trie containing ["cat", "car", "card"]:

- `starts_with("ca")` → True

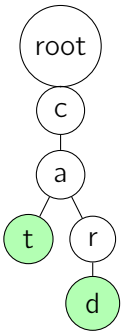
Delete Operation

```python
def delete(self, word):
    """Delete word from trie - O(m)"""
    def _delete(node, word, index):
        if index == len(word):
            # Reached end of word
            if not node.is_end_of_word:
                return False  # Word doesn't exist

            node.is_end_of_word = False

            # Return True if node has no children (can be deleted)
            return len(node.children) == 0

        char = word[index]
        if char not in node.children:
            return False  # Word doesn't exist

        child = node.children[char]
        should_delete_child = _delete(child, word, index + 1)
```

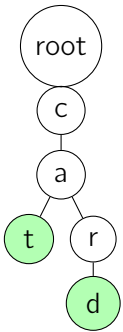
Delete Visualization

Deleting "car" from trie with ["cat", "car", "card"]:

Before delete("car"):



After delete("car"):



Key points:

- Node 'r' is NOT deleted (needed by "card")
- Only the `is_end_of_word` flag on 'r' is set to false

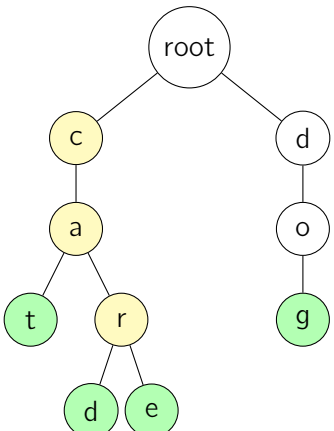
Prefix Queries and Autocomplete

Find All Words with Prefix

```python
def find_words_with_prefix(self, prefix):
    """Return all words starting with prefix"""
    node = self.root

    # Navigate to prefix node
    for char in prefix:
        if char not in node.children:
            return []
        node = node.children[char]

    # Collect all words from this node
    words = []
    self._collect_words(node, prefix, words)
    return words

def _collect_words(self, node, current_word, words):
    """DFS to collect all words from node"""
    if node.is_end_of_word:
        words.append(current_word)
```

Prefix Query Example

Trie with words: ["cat", "car", "card", "care", "dog"]



`find_words_with_prefix("car")` → ["car", "card", "care"]

Autocomplete with Limit

```python
def autocomplete(self, prefix, max_results=5):
    """Return top N autocomplete suggestions"""
    node = self.root

    # Navigate to prefix
    for char in prefix:
        if char not in node.children:
            return []
        node = node.children[char]

    # BFS to find closest words
    from collections import deque
    results = []
    queue = deque([(node, prefix)])

    while queue and len(results) < max_results:
        current, word = queue.popleft()

        if current.is_end_of_word:
```

Longest Common Prefix

```python
def longest_common_prefix(self, words):
    """Find longest common prefix of all words"""
    if not words:
        return ""

    # Insert all words
    for word in words:
        self.insert(word)

    # Traverse until branching or end
    node = self.root
    prefix = ""

    while len(node.children) == 1 and not node.is_end_of_word:
        char = next(iter(node.children))
        prefix += char
        node = node.children[char]
```

Wildcard Search

```python
def search_with_wildcard(self, pattern):
    """Search with '.' as wildcard for any character"""
    def dfs(node, index):
        if index == len(pattern):
            return node.is_end_of_word

        char = pattern[index]

        if char == '.':
            # Try all children
            for child in node.children.values():
                if dfs(child, index + 1):
                    return True
            return False
        else:
            if char not in node.children:
                return False
            return dfs(node.children[char], index + 1)
```

Memory Optimization

Memory Usage Analysis

Standard Trie Space Complexity:

For words ["cat", "car", "card"]:

- Nodes: root + c + a + t + r + d = 6 nodes
- Each node: dict (hash table) + bool $\approx$ 100+ bytes
- Total: $\sim$600 bytes for 3 words

Worst Case: No shared prefixes

Words: ["abc", "def", "ghi"]

- Nodes: 1 + 3*3 = 10 nodes
- Space: O(ALPHABET_SIZE $\times$ N $\times$ M)
- N = number of words, M = average length

Best Case: Many shared prefixes

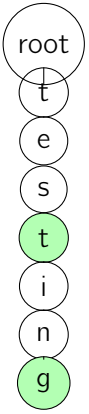
Words: ["test", "testing", "tester"]

- Much better space efficiency due to sharing

Radix Tree (Compressed Trie)

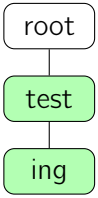
Idea: Merge chains of single-child nodes

Standard Trie:



9 nodes

Radix Tree:



3 nodes (67% reduction)

Radix Node Structure

```python
class RadixNode:
    def __init__(self, label=""):
        self.label = label  # String, not single char!
        self.children = {}
        self.is_end_of_word = False

class RadixTree:
    def __init__(self):
        self.root = RadixNode()
```

Key Difference:

- Standard Trie: Each node stores one character
- Radix Tree: Each node stores a string (edge label)

Applications:

- Git uses radix trees internally
- Linux kernel routing tables

Space Optimization Techniques

1. **Array-based for small alphabets**
   - Use fixed-size array instead of dictionary
   - Trade: wastes space if sparse, but faster access

2. **Bit packing for flags**
   - Pack multiple boolean flags into single integer
   - Saves memory when nodes have many flags

3. **Lazy initialization**
   - Don't create children dict until needed
   - `self.children = None` initially

4. **Use radix tree for long common prefixes**
   - Significantly reduces node count
   - Best for datasets with long shared prefixes

Enhanced Tries with Metadata

Frequency Trie

Store word frequencies for autocomplete ranking:

```python
class FrequencyTrieNode:
    def __init__(self):
        self.children = {}
        self.count = 0  # Number of times word inserted

class FrequencyTrie:
    def __init__(self):
        self.root = FrequencyTrieNode()

    def insert(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = FrequencyTrieNode()
            node = node.children[char]
        node.count += 1
```

Top-K Autocomplete with Frequency

```python
def top_k_with_prefix(self, prefix, k):
    """Get top k most frequent words with prefix"""
    node = self.root

    # Navigate to prefix
    for char in prefix:
        if char not in node.children:
            return []
        node = node.children[char]

    # Collect all words with frequencies
    words = []
    self._collect_with_freq(node, prefix, words)

    # Sort by frequency and return top k
    words.sort(key=lambda x: x[1], reverse=True)
    return [word for word, freq in words[:k]]

def _collect_with_freq(self, node, word, words):
```

Indexed Trie for Search Engines

```python
class IndexedTrieNode:
    def __init__(self):
        self.children = {}
        self.doc_ids = []  # List of document IDs

class SearchEngine:
    def __init__(self):
        self.trie = IndexedTrieNode()
        self.documents = {}

    def add_document(self, doc_id, text):
        """Index document for search"""
        self.documents[doc_id] = text
        words = text.lower().split()

        for word in words:
            node = self.trie
            for char in word:
                if char not in node.children:
```

Suffix Trie for Pattern Matching

```python
class SuffixTrie:
    """Store all suffixes of a string for fast pattern matching"""
    def __init__(self, text):
        self.root = TrieNode()
        self.text = text

        # Insert all suffixes
        for i in range(len(text)):
            self._insert_suffix(text[i:], i)

    def _insert_suffix(self, suffix, start_index):
        node = self.root
        for char in suffix:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.start_index = start_index

    def find_pattern(self, pattern):
```

Real-World Applications

Spell Checker

```python
class SpellChecker:
    def __init__(self, dictionary):
        self.trie = Trie()
        for word in dictionary:
            self.trie.insert(word.lower())

    def is_correct(self, word):
        """Check if word is spelled correctly"""
        return self.trie.search(word.lower())

    def suggest_corrections(self, word, max_distance=2):
        """Suggest corrections using edit distance"""
        suggestions = []

        def dfs(node, current, remaining_edits):
            if remaining_edits < 0:
                return

            if node.is_end_of_word and current:
```

Spell Checker Example

Dictionary: ["cat", "car", "card", "care", "careful"]

Operations:

- `is_correct("car")` → True
- `is_correct("cra")` → False
- `suggest_corrections("cra")` → ["car", "care", "card"]

Edit Distance Techniques:

- **Substitution:** "cra" → "car" (distance = 1)
- **Insertion:** "cra" → "care" (distance = 1)
- **Deletion:** "crat" → "cat" (distance = 1)

Complexity: O(m × ALPHABET_SIZE^d) where d = max distance

IP Routing Table

```python
class IPRoutingTable:
    """Longest prefix matching for IP routing"""
    def __init__(self):
        self.root = TrieNode()

    def add_route(self, ip_prefix, next_hop):
        """Add routing entry (e.g., "192.168.0.0/16")"""
        binary = self._ip_to_binary(ip_prefix)

        node = self.root
        for bit in binary:
            if bit not in node.children:
                node.children[bit] = TrieNode()
            node = node.children[bit]

        node.next_hop = next_hop
        node.is_end_of_word = True

    def lookup(self, ip_address):
```

IP Routing Example

Routing table entries:

- 192.168.0.0/16 → Router A
- 192.168.1.0/24 → Router B
- 192.168.1.128/25 → Router C

Lookups (longest prefix match):

- 192.168.1.200 → Router B (/24 match)
- 192.168.1.150 → Router C (/25 match, longest!)
- 192.168.2.1 → Router A (/16 match)

Why Trie?

- Fast O(32) lookup for IPv4 (32 bits)
- Naturally finds longest prefix
- Efficient for large routing tables

Autocomplete System

```python
class AutocompleteSystem:
    """Google-style autocomplete with frequency ranking"""
    def __init__(self):
        self.trie = FrequencyTrie()
        self.current_prefix = ""

    def input(self, char):
        """Process one character input"""
        if char == '#':
            # End of sentence, save it
            self.trie.insert(self.current_prefix)
            self.current_prefix = ""
            return []

        self.current_prefix += char

        # Return top 3 suggestions
        return self.trie.top_k_with_prefix(self.current_prefix, 3)
```

Word Break Problem

```python
def word_break(s, word_dict):
    """Check if string can be segmented into dictionary words
    Example: "leetcode" with ["leet", "code"] -> True
    """
    trie = Trie()
    for word in word_dict:
        trie.insert(word)

    n = len(s)
    dp = [False] * (n + 1)
    dp[0] = True  # Empty string

    for i in range(1, n + 1):
        node = trie.root
        for j in range(i - 1, -1, -1):
            if not dp[j]:
                continue

            char = s[j]
```

File System Path Matching

```python
class FileSystemTrie:
    """Efficient file path matching with wildcards"""
    def __init__(self):
        self.root = TrieNode()

    def add_path(self, path):
        """Add file path like /home/user/file.txt"""
        parts = path.strip('/').split('/')
        node = self.root

        for part in parts:
            if part not in node.children:
                node.children[part] = TrieNode()
            node = node.children[part]

        node.is_end_of_word = True

    def find_files(self, pattern):
        """Find files matching pattern (e.g., /home/*/file.txt)"""
```

Complexity Analysis

Time Complexity Summary

Operation	Time Complexity	Notes
Insert	O(m)	m = word length
Search	O(m)	Exact word search
Delete	O(m)	Recursive cleanup
Prefix search	O(p)	p = prefix length
Find all with prefix	O(p + n)	n = matching words
Autocomplete (top k)	O(p + k)	k = results returned
Wildcard search	$O(m \times b^w)$	w = wildcards, b = branching

Key Insight: Performance depends on word/prefix length, NOT total words!
This makes tries excellent for large dictionaries.

Space Complexity Summary

Standard Trie:

- **Best case:** O(m) where m = longest word (all words share prefix)
- **Average case:** O(ALPHABET_SIZE $\times$ n $\times$ m)
- **Worst case:** O(ALPHABET_SIZE $\times$ n $\times$ m) (no shared prefixes)

Radix Tree:

- O(n) nodes where n = number of words
- Much better space efficiency

Space Optimization Trade-offs:

Implementation	Space	Speed
Dictionary-based	High	Fast
Array-based (sparse)	Very High	Fastest
Array-based (dense)	Medium	Fastest
Radix tree	Low	Medium

Comparison with Other Data Structures

Feature	Hash Table	BST	Trie
Insert	O(1)	O(log n)	O(m)
Search	O(1)	O(log n)	O(m)
Delete	O(1)	O(log n)	O(m)
Prefix search	O(n)	O(n)	O(p + k)
Sorted order	No	Yes	Yes
Space	O(n)	O(n)	O(n × m)
Collisions	Yes	No	No

When to use Trie:

- Need prefix-based queries (autocomplete, spell check)
- Many words with common prefixes
- Need predictable performance (no hash collisions)
- Lexicographic ordering matters

Summary

Key Concepts Recap

Trie Fundamentals:

- Tree where each node = character
- Path from root = string/prefix
- Efficient prefix queries: O(m) not O(n)

Core Operations:

- Insert, Search, Delete: All O(m)
- Prefix search: O(p)
- Find all with prefix: O(p + n)

Variants:

- Dictionary-based (flexible alphabet)
- Array-based (fixed alphabet, faster)
- Radix tree (compressed, space-efficient)
- Frequency trie (ranking autocomplete)

Applications Recap

Major Use Cases:

- **Autocomplete:** Search engines, text editors
- **Spell checking:** Word processors
- **IP routing:** Network routers (longest prefix match)
- **Text search:** Pattern matching, substring search
- **Dictionary:** Fast word validation
- **DNA sequencing:** Genomic pattern matching

When NOT to use Trie:

- Simple exact-match lookups (use hash table)
- No prefix queries needed
- Memory is very limited
- Small, simple datasets

Practice Problems

Basic:

- Implement insert, search, starts_with operations
- Find longest common prefix of array of strings
- Count words with given prefix

Intermediate:

- Implement autocomplete with top-k results
- Word search with wildcards
- Design search autocomplete system (LeetCode 642)
- Word break problem using trie

Advanced:

- Implement radix tree with node splitting
- Design spell checker with edit distance
- Suffix tree construction and pattern matching
- Trie with frequency-based ranking

Implementation Tips

Best Practices:

- Use dictionary for variable alphabets (UTF-8, symbols)
- Use array for fixed alphabets (a-z, 0-9)
- Consider radix tree if memory is concern
- Add metadata (frequency, indices) for advanced features

Common Pitfalls:

- Forgetting to check `is_end_of_word` in search
- Not handling empty string correctly
- Memory leaks in delete operation
- Not considering case sensitivity

Optimization Strategies:

- Cache frequently accessed nodes
- Use lazy initialization for children
- Implement iterative versions to avoid recursion overhead
- Profile and optimize hot paths

Further Learning

Related Topics:

- **Suffix Trees:** All suffixes of a string
- **Ternary Search Trees:** Space-efficient alternative
- **Patricia Trees:** Practical Algorithm to Retrieve Information
- **Aho-Corasick:** Multiple pattern matching

Resources:

- Implement all operations from scratch
- LeetCode Trie problems (208, 211, 212, 642, 648)
- Real-world project: Build autocomplete system
- Study open-source implementations (Redis, Elasticsearch)

Projects:

- Search autocomplete with frequency ranking
- Spell checker with suggestions
- T9 predictive text system
- Simple search engine with trie indexing

Thank You!

Questions?

Tries: Mastering Prefix-Based Queries