Trie Data Structure

Created by	j jubink
Created time	@February 26, 2025 10:36 PM
Tags	

🌱 Beginner Level - Trie Data Structure Practice Questions

Here are step-by-step **algorithms and Python solutions** for each **beginner-level** Trie question.

1️⃣ Insert a Word into a Trie

Algorithm

1. Start from the root node of the Trie.
2. Traverse each character in the given word:
    - If the character is **not** present, create a new TrieNode.
    - Move to the next TrieNode.
3. After processing all characters, mark the last node as an **end of the word**.

Python Code

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

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

    def insert(self, word):
        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

# Usage
trie = Trie()
trie.insert("apple")
trie.insert("app")
```

2️⃣ Search for a Word in a Trie

Algorithm

1. Start from the root node.
2. Traverse each character in the given word:
   - If the character **does not exist**, return `False`.
   - Move to the next TrieNode.
3. If the last node **is marked as an end of a word**, return `True`; otherwise, return `False`.

Python Code

```
python
CopyEdit
def search(self, word):
```

```
    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

# Usage
print(trie.search("apple"))  # True
print(trie.search("app"))    # True
print(trie.search("bat"))    # False
```

3️⃣ Check if a Given Prefix Exists in a Trie

Algorithm

1. Start from the root node.
2. Traverse each character in the given prefix:
    - If the character **does not exist**, return `False`.
    - Move to the next TrieNode.
3. If all characters are found, return `True`.

Python Code

```
python
CopyEdit
def starts_with(self, prefix):
    node = self.root
    for char in prefix:
        if char not in node.children:
            return False
        node = node.children[char]
    return True
```

```
# Usage
print(trie.starts_with("ap"))  # True
print(trie.starts_with("bat")) # False
```

4️⃣ Delete a Word from a Trie

Algorithm

1. Recursively traverse down the Trie following the characters of the word.
2. If the word exists:
    - Unmark the last node.
    - Delete nodes from bottom to top if they are no longer needed.

Python Code

```
python
CopyEdit
def delete(self, word):
    def _delete(node, word, depth):
        if depth == len(word):
            if not node.is_end_of_word:
                return False
            node.is_end_of_word = False
            return len(node.children) == 0  # Delete only if it's a leaf node

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

        should_delete = _delete(node.children[char], word, depth + 1)

        if should_delete:
```

```
            del node.children[char]
            return len(node.children) == 0  # Delete the parent node if it's empty

        return False

    _delete(self.root, word, 0)

# Usage
trie.delete("apple")
print(trie.search("apple"))  # False
```

5️⃣ Count the Number of Words in a Trie

Algorithm

1. Perform a depth-first search (DFS) on the Trie.
2. Count each node marked as an **end of a word**.

Python Code

```
python
CopyEdit
def count_words(self):
    def dfs(node):
        count = 1 if node.is_end_of_word else 0
        for child in node.children.values():
            count += dfs(child)
        return count
    return dfs(self.root)

# Usage
print(trie.count_words())  # Output: Number of words
```

6️⃣ Count Words Starting with a Given Prefix

Algorithm

1. Traverse the Trie following the given prefix.
2. Perform DFS from the last node of the prefix.
3. Count words marked as `is_end_of_word`.

Python Code

```
python
CopyEdit
def count_prefix(self, prefix):
    def dfs(node):
        count = 1 if node.is_end_of_word else 0
        for child in node.children.values():
            count += dfs(child)
        return count

    node = self.root
    for char in prefix:
        if char not in node.children:
            return 0
        node = node.children[char]

    return dfs(node)

# Usage
print(trie.count_prefix("ap"))  # Output: Count of words starting with "ap"
```

7️⃣ Check if a Trie is Empty

Algorithm

1. If the root node has no children, return `True` (empty Trie).
2. Otherwise, return `False`.

Python Code

```
python
CopyEdit
def is_empty(self):
    return len(self.root.children) == 0

# Usage
print(trie.is_empty())  # False
```

8️⃣ Find the Longest Common Prefix in a Trie

Algorithm

1. Start from the root node.
2. Traverse while the node has exactly **one child**.
3. Stop when a branching point or `is_end_of_word` is encountered.
4. Return the accumulated prefix.

Python Code

```
python
CopyEdit
def longest_common_prefix(self):
    node = self.root
    prefix = ""

    while len(node.children) == 1 and not node.is_end_of_word:
        char = list(node.children.keys())[0]
        prefix += char
```

```
        node = node.children[char]

    return prefix

# Usage
print(trie.longest_common_prefix())  # Output: Longest common prefix
```

9️⃣ Print All Words in a Trie

Algorithm

1. Perform a DFS traversal.
2. Accumulate characters while traversing.
3. When `is_end_of_word` is encountered, print the accumulated word.

Python Code

```
python
CopyEdit
def get_all_words(self):
    def dfs(node, path, words):
        if node.is_end_of_word:
            words.append("".join(path))
        for char, child in node.children.items():
            dfs(child, path + [char], words)

    words = []
    dfs(self.root, [], words)
    return words

# Usage
print(trie.get_all_words())  # Output: ['app', 'apple']
```

🔟 Find the Shortest Unique Prefix for Each Word

Algorithm

1. Store the frequency of each character in the Trie.
2. The shortest unique prefix of a word is the first node where frequency = 1.

Python Code

```
python
CopyEdit
def shortest_unique_prefix(self, word):
    node = self.root
    prefix = ""
    for char in word:
        prefix += char
        node = node.children[char]
        if len(node.children) == 1 and node.is_end_of_word:
            return prefix
    return prefix

# Usage
print(trie.shortest_unique_prefix("apple"))  # Output: "ap"
```

🌟 Intermediate Level - Trie Data Structure Practice Questions

Now, let's move to **intermediate-level** Trie problems with **step-by-step algorithms and Python solutions**. 🚀

1️⃣ Implement Auto-Complete Feature Using Trie

Algorithm

1. Insert words into the Trie.
2. Traverse the Trie following the given prefix.
3. Use **Depth-First Search (DFS)** to collect all words from that node.

Python Code

```
python
CopyEdit
def autocomplete(self, prefix):
    def dfs(node, path, results):
        if node.is_end_of_word:
            results.append("".join(path))
        for char, child in node.children.items():
            dfs(child, path + [char], results)

    node = self.root
    for char in prefix:
        if char not in node.children:
            return []  # No words found
        node = node.children[char]

    results = []
    dfs(node, list(prefix), results)
    return results

# Usage
trie.insert("apple")
trie.insert("appetite")
trie.insert("apply")
print(trie.autocomplete("app"))  # Output: ['apple', 'appetite', 'apply']
```

2️⃣ Find All Words Matching a Given Pattern

Algorithm

1. Use a Trie to store words.
2. Traverse the Trie based on the given pattern.
3. Use DFS to find all matching words.

Python Code

```
python
CopyEdit
def words_with_pattern(self, pattern):
    def dfs(node, path, results):
        if node.is_end_of_word:
            results.append("".join(path))
        for char, child in node.children.items():
            dfs(child, path + [char], results)

    node = self.root
    for char in pattern:
        if char not in node.children:
            return []
        node = node.children[char]

    results = []
    dfs(node, list(pattern), results)
    return results

# Usage
print(trie.words_with_pattern("ap"))  # Output: ['apple', 'appetite', 'apply']
```

3️⃣ Count the Number of Distinct Substrings Using Trie

Algorithm

1. Insert all **suffixes** of the string into the Trie.
2. Count the number of nodes in the Trie (each node represents a unique substring).

Python Code

```
python
CopyEdit
def count_distinct_substrings(self, word):
    count = 0
    for i in range(len(word)):
        node = self.root
        for j in range(i, len(word)):
            if word[j] not in node.children:
                node.children[word[j]] = TrieNode()
                count += 1  # New substring found
            node = node.children[word[j]]
    return count + 1  # Include empty substring

# Usage
print(trie.count_distinct_substrings("apple"))  # Output: Number of unique sub
strings
```

4️⃣ Find the Longest Word in a Trie

Algorithm

1. Perform DFS.
2. Track the longest word encountered where **all prefixes exist in the Trie**.

Python Code

```
python
CopyEdit
```

```
def longest_word(self):
    def dfs(node, path):
        nonlocal longest
        if node.is_end_of_word:
            word = "".join(path)
            if len(word) > len(longest):
                longest = word
        for char, child in node.children.items():
            dfs(child, path + [char])

    longest = ""
    dfs(self.root, [])
    return longest

# Usage
trie.insert("banana")
trie.insert("ban")
trie.insert("band")
print(trie.longest_word())  # Output: 'banana'
```

5️⃣ Find the Shortest Unique Prefix for Each Word in a Trie

Algorithm

1. Store frequency of each character in Trie.
2. The first node where **frequency = 1** is the shortest unique prefix.

Python Code

```
python
CopyEdit
def shortest_unique_prefix(self, word):
    node = self.root
```

```
    prefix = ""
    for char in word:
        prefix += char
        if len(node.children) == 1 and node.is_end_of_word:
            return prefix
        node = node.children[char]
    return prefix

# Usage
print(trie.shortest_unique_prefix("apple"))  # Output: "ap"
```

6️⃣ Implement a Spell Checker Using Trie

Algorithm

1. Store a dictionary of valid words in the Trie.
2. Check if the word exists.
3. If not found, suggest corrections based on **Levenshtein Distance**.

Python Code

```
python
CopyEdit
def spell_check(self, word):
    if self.search(word):
        return f"'{word}' is correctly spelled."

    suggestions = self.autocomplete(word[:len(word) // 2])
    return f"Did you mean: {suggestions}?"

# Usage
print(trie.spell_check("appl"))  # Output: Did you mean: ['apple', 'apply']
```

7️⃣ Implement a Word Break Problem Using Trie

Algorithm

1. Use a Trie to store a dictionary of words.
2. Recursively check if the string can be split into valid words.

Python Code

```
python
CopyEdit
def word_break(self, sentence):
    def dfs(index):
        if index == len(sentence):
            return True
        node = self.root
        for i in range(index, len(sentence)):
            if sentence[i] not in node.children:
                return False
            node = node.children[sentence[i]]
            if node.is_end_of_word and dfs(i + 1):
                return True
        return False

    return dfs(0)

# Usage
trie.insert("apple")
trie.insert("pen")
print(trie.word_break("applepen"))  # Output: True
```

8️⃣ Find the Most Frequent Word in a Trie

Algorithm

1. Store a frequency counter in each node.
2. Perform DFS to find the most frequently inserted word.

Python Code

```
python
CopyEdit
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end_of_word = False
        self.frequency = 0  # Store frequency count

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

    def insert(self, word):
        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
        node.frequency += 1  # Increase frequency

    def most_frequent_word(self):
        def dfs(node, path):
            nonlocal max_freq, most_frequent
            if node.is_end_of_word and node.frequency > max_freq:
                max_freq = node.frequency
                most_frequent = "".join(path)
            for char, child in node.children.items():
                dfs(child, path + [char])
```

```
        max_freq = 0
        most_frequent = ""
        dfs(self.root, [])
        return most_frequent

# Usage
trie = Trie()
trie.insert("apple")
trie.insert("apple")
trie.insert("banana")
trie.insert("apple")
trie.insert("banana")
print(trie.most_frequent_word())  # Output: 'apple'
```

🌟 Advanced Level - Trie Data Structure Practice Questions

Now, let's move on to **advanced-level** Trie problems with **detailed algorithms and Python solutions**. 🚀

1️⃣ Implement a Ternary Search Trie (TST)

Algorithm

1. Similar to a regular Trie, but each node has **three pointers**:
    - Left: Characters **less than** the current node.
    - Middle: Characters **equal** to the current node.
    - Right: Characters **greater than** the current node.
2. Insert, search, and traverse words using these three pointers.

Python Code

```python
python
CopyEdit
class TSTNode:
    def __init__(self, char):
        self.char = char
        self.is_end_of_word = False
        self.left = self.middle = self.right = None

class TernarySearchTrie:
    def __init__(self):
        self.root = None

    def insert(self, word):
        def insert_recursive(node, word, index):
            if index == len(word):
                return node

            char = word[index]
            if not node:
                node = TSTNode(char)

            if char < node.char:
                node.left = insert_recursive(node.left, word, index)
            elif char > node.char:
                node.right = insert_recursive(node.right, word, index)
            else:
                if index + 1 == len(word):
                    node.is_end_of_word = True
                else:
                    node.middle = insert_recursive(node.middle, word, index + 1)
            return node

        self.root = insert_recursive(self.root, word, 0)

    def search(self, word):
```

```
        def search_recursive(node, word, index):
            if not node:
                return False
            char = word[index]
            if char < node.char:
                return search_recursive(node.left, word, index)
            elif char > node.char:
                return search_recursive(node.right, word, index)
            else:
                if index == len(word) - 1:
                    return node.is_end_of_word
                return search_recursive(node.middle, word, index + 1)

        return search_recursive(self.root, word, 0)

# Usage
tst = TernarySearchTrie()
tst.insert("apple")
tst.insert("app")
print(tst.search("apple"))  # Output: True
print(tst.search("appl"))   # Output: False
```

2️⃣ Implement a Compressed Trie (Radix Tree)

Algorithm

1. Merge consecutive nodes with **single children** into one node.
2. Reduce memory usage by **storing common prefixes** instead of separate characters.
3. Implement insert, search, and delete operations efficiently.

Python Code

```
python
CopyEdit
class RadixTrieNode:
    def __init__(self):
        self.children = {}
        self.is_end_of_word = False

class RadixTrie:
    def __init__(self):
        self.root = RadixTrieNode()

    def insert(self, word):
        node = self.root
        while word:
            for key in node.children.keys():
                common_prefix = self.common_prefix(word, key)
                if common_prefix:
                    if common_prefix == key:
                        word = word[len(common_prefix):]
                        node = node.children[key]
                    else:
                        # Split into a new intermediate node
                        old_child = node.children[key]
                        new_node = RadixTrieNode()
                        node.children[common_prefix] = new_node
                        new_node.children[key[len(common_prefix):]] = old_child
                        del node.children[key]
                        node = new_node
                        word = word[len(common_prefix):]
                    break
            else:
                node.children[word] = RadixTrieNode()
                node.children[word].is_end_of_word = True
                break
```

```
    def common_prefix(self, str1, str2):
        i = 0
        while i < min(len(str1), len(str2)) and str1[i] == str2[i]:
            i += 1
        return str1[:i]

    def search(self, word):
        node = self.root
        while word:
            for key in node.children.keys():
                if word.startswith(key):
                    word = word[len(key):]
                    node = node.children[key]
                    if not word:
                        return node.is_end_of_word
                    break
            else:
                return False
        return False

# Usage
trie = RadixTrie()
trie.insert("apple")
trie.insert("app")
print(trie.search("apple"))  # Output: True
print(trie.search("app"))    # Output: True
print(trie.search("appl"))   # Output: False
```

3️⃣ Implement a Trie-Based DNS Resolver

Algorithm

1. Store domain names **from right to left** in a Trie.
2. Match queries with the longest available prefix in the Trie.

3. Use backtracking to resolve subdomains.

Python Code

```
python
CopyEdit
class DNSResolver:
    def __init__(self):
        self.trie = Trie()

    def add_domain(self, domain):
        reversed_domain = ".".join(domain.split(".")[::-1])
        self.trie.insert(reversed_domain)

    def resolve(self, domain):
        reversed_domain = ".".join(domain.split(".")[::-1])
        return self.trie.search(reversed_domain)

# Usage
resolver = DNSResolver()
resolver.add_domain("google.com")
resolver.add_domain("mail.google.com")
print(resolver.resolve("mail.google.com"))  # Output: True
print(resolver.resolve("drive.google.com")) # Output: False
```

4️⃣ Implement a Dictionary with Wildcard Search

Algorithm

1. Use a Trie to store words.
2. When encountering a wildcard `'.'`, explore **all possible paths**.

Python Code

```
python
CopyEdit
def search_with_wildcard(self, word):
    def dfs(node, index):
        if index == len(word):
            return node.is_end_of_word
        char = word[index]
        if char == ".":
            return any(dfs(child, index + 1) for child in node.children.values())
        if char in node.children:
            return dfs(node.children[char], index + 1)
        return False

    return dfs(self.root, 0)

# Usage
trie.insert("hello")
trie.insert("help")
print(trie.search_with_wildcard("he.lo"))  # Output: True
print(trie.search_with_wildcard("hel."))   # Output: True
print(trie.search_with_wildcard("h.ll"))   # Output: False
```

5️⃣ Longest Repeating Substring Using Trie

Algorithm

1. Insert **all suffixes** into the Trie.
2. Identify the deepest node with more than one occurrence.

Python Code

```
python
CopyEdit
```

```
def longest_repeating_substring(self, text):
    longest = ""
    for i in range(len(text)):
        node = self.root
        substring = ""
        for j in range(i, len(text)):
            char = text[j]
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
            substring += char
            if node.is_end_of_word:
                longest = max(longest, substring, key=len)
        node.is_end_of_word = True
    return longest

# Usage
print(trie.longest_repeating_substring("banana"))  # Output: "ana"
```