

Trie (Prefixbaum)

1 Grundidee

Ein **Trie** (Prefixbaum) ist eine Baumstruktur zur effizienten Speicherung und Suche von **Strings** (Wörtern). Jeder Pfad von der Wurzel zu einem Knoten entspricht einem **Prefix**.

- Kanten sind mit **Zeichen** beschriftet
- Knoten repräsentieren ein Prefix
- Ein spezielles Flag markiert das **Ende eines Wortes**

→ Sehr gut für:

- Prefix-Suche (Autocomplete)
- Wörterbücher
- schnelle Mitgliedschaftstests für Strings

2 Voraussetzungen

- Schlüssel sind **Strings** (oder Sequenzen von Symbolen)
- Alphabet / Zeichenvorrat ist definiert
- Speicher für Knoten + Kind-Referenzen (z. B. Dict/Map)

3 Laufzeiten & Eigenschaften

Eigenschaft	Wert
Suche	$O(L)$
Einfügen	$O(L)$
Löschen	$O(L)$ (mit Aufräumen)
Speicherbedarf	$O(\text{summe aller Zeichen})$
In-place	nein
Stabil	–

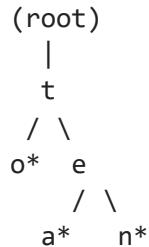
Hinweis: L = Länge des Suchworts, nicht Anzahl gespeicherter Wörter. Das ist der große Vorteil gegenüber vielen Vergleichsstrukturen.

4 Schritt-für-Schritt-Beispiel

Wir fügen ein:

to, tea, ten

Vereinfachte Darstellung:



* bedeutet: Wort endet hier (End-of-Word Flag).

Prefix-Suche "te"

- folgt Pfad: t → e
 - liefert alle Wörter darunter: tea, ten
-

5 Besonderheiten / Prüfungsrelevante Hinweise

- Tries sind ideal für **Prefix-Operationen**
 - Laufzeit hängt von der **Wortlänge** ab, nicht von n
 - Speicher kann hoch sein, weil viele Knoten entstehen
 - Häufige Optimierung: **Patricia Trie** (komprimierter Trie)
-

6 Vor- und Nachteile

Vorteile

- schnelle Suche O(L)
- perfekte Prefix-Suche (Autocomplete)
- keine Hash-Kollisionen

Nachteile

- hoher Speicherverbrauch
- Implementierung komplexer als Hash-Set
- abhängig vom Alphabet (viele Kinder möglich)



Merksatz für die Prüfung

Ein Trie speichert Wörter zeichenweise und erlaubt Suche und Einfügen in $O(L)$, besonders stark bei Prefix-Abfragen.

7 Python-Implementierung

```
In [1]: class TrieNode:
    def __init__(self):
        self.children = {}          # char -> TrieNode
        self.is_end = False         # Wort endet hier?

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

        def insert(self, word):
            node = self.root
            for ch in word:
                if ch not in node.children:
                    node.children[ch] = TrieNode()
                node = node.children[ch]
            node.is_end = True

        def search(self, word):
            node = self._find_node(word)
            return node is not None and node.is_end

        def starts_with(self, prefix):
            return self._find_node(prefix) is not None

        def _find_node(self, s):
            node = self.root
            for ch in s:
                if ch not in node.children:
                    return None
                node = node.children[ch]
            return node

    # Beispiel
trie = Trie()
for w in ["to", "tea", "ten"]:
    trie.insert(w)

print(trie.search("tea"))      # True
print(trie.search("te"))       # False
print(trie.starts_with("te"))  # True
```

True
False
True

Patricia Trie (komprimierter Trie / Radix Tree)

1 Grundidee

Ein **Patricia Trie** (oft auch **komprimierter Trie** oder **Radix Tree**) ist eine optimierte Variante des Tries. Er **komprimiert** Pfade, bei denen Knoten nur **ein einziges Kind** haben, zu **einer Kante mit einem String-Label**.

→ Ergebnis:

- weniger Knoten
- weniger Speicher
- schnellere Traversierung in der Praxis

2 Voraussetzungen

- Schlüssel sind **Strings** (oder Sequenzen)
- Fähigkeit, **gemeinsame Prefixe** zu bestimmen (Prefix-Splitting)
- Kinder werden typischerweise in einer Map/Dictionary gespeichert

3 Laufzeiten & Eigenschaften

Eigenschaft	Wert
Suche	$O(L)$
Einfügen	$O(L)$
Löschen	$O(L)$ (mit Aufräumen)
Speicherbedarf	deutlich kleiner als Trie (typisch)
In-place	nein
Stabil	–

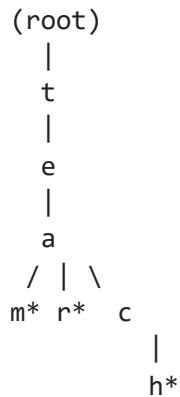
Hinweis: Wie beim Trie hängt die Laufzeit von der **Wortlänge L** ab, nicht von der Anzahl Wörter. In der Praxis oft schneller, weil weniger Knoten besucht werden.

4 Schritt-für-Schritt-Beispiel (Kompression)

Wörter:

team, tear, teach

Normaler Trie



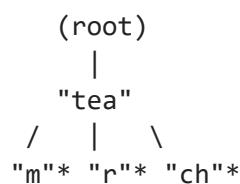
Patricia Trie (komprimiert)

Hier gibt es in diesem Mini-Beispiel nur wenig zu komprimieren, weil der Trie bereits kurz ist.
Besseres Beispiel für Kompression:

Normaler Trie hätte Pfade:

- $t \rightarrow e \rightarrow a \rightarrow m^*$
- $t \rightarrow e \rightarrow a \rightarrow r^*$
- $t \rightarrow e \rightarrow a \rightarrow c \rightarrow h^*$

Patricia Trie komprimiert den gemeinsamen Prefix:



(Die Kantenlabels sind Strings statt einzelne Zeichen.)

5 Besonderheiten / Prüfungsrelevante Hinweise

- Auch bekannt als:
 - **Radix Tree**

■ Compressed Trie

- Speicherung erfolgt mit **Kantenlabels** (Strings)
 - Typische Prüfungsfrage: *Was wird komprimiert?*
 - Alle Pfade mit „Single-Child“-Ketten werden zusammengefasst.
 - Sehr nützlich für große Wörterbücher und Prefix-Suchen
-

6 Vor- und Nachteile

Vorteile

- deutlich weniger Speicher als Trie
- oft schneller in der Praxis (weniger Knoten)
- weiterhin Prefix-Suche möglich

Nachteile

- Einfügen/Löschen ist komplexer (Split von Labels)
 - Implementierung fehleranfälliger
-

7 Merksatz für die Prüfung

Ein Patricia Trie komprimiert Trie-Pfade mit nur einem Kind zu Kantenlabels und spart so Speicher, bei gleicher O(L)-Logik.

7 Python-Implementierung (vereinfachtes Lehrbeispiel)

```
In [2]: class PatriciaNode:  
    def __init__(self):  
        # edges: label -> child_node  
        self.edges = {}  
        self.is_end = False  
  
class PatriciaTrie:  
    def __init__(self):  
        self.root = PatriciaNode()  
  
    def insert(self, word):  
        node = self.root  
        while True:  
            # finde eine Kante, deren Label ein gemeinsames Prefix mit word hat  
            for label, child in list(node.edges.items()):  
                common = self._common_prefix(label, word)
```

```

    if common:
        # Fall 1: Label passt komplett -> gehe tiefer
        if common == label:
            node = child
            word = word[len(common):]
            if word == "":
                node.is_end = True
            break

        # Fall 2: Partial Match -> Split notwendig
        # label = common + label_rest
        label_rest = label[len(common):]
        word_rest = word[len(common):]

        # Neuer Zwischenknoten für common
        mid = PatriciaNode()

        # Ersetze alte Kante label durch common
        del node.edges[label]
        node.edges[common] = mid

        # Alte Kante wird von mid weitergeführt
        mid.edges[label_rest] = child

        # Neue Kante für das neue Wort
        if word_rest == "":
            mid.is_end = True
        else:
            new_child = PatriciaNode()
            new_child.is_end = True
            mid.edges[word_rest] = new_child
        return

    else:
        # keine passende Kante gefunden -> neue Kante hinzufügen
        new_child = PatriciaNode()
        new_child.is_end = True
        node.edges[word] = new_child
        return

def search(self, word):
    node = self.root
    while word:
        found = False
        for label, child in node.edges.items():
            if word.startswith(label):
                word = word[len(label):]
                node = child
                found = True
                break
        if not found:
            return False
    return node.is_end

def _common_prefix(self, a, b):
    i = 0
    while i < len(a) and i < len(b) and a[i] == b[i]:

```

```
i += 1
return a[:i]

# Beispiel
pt = PatriciaTrie()
for w in ["team", "tear", "teach"]:
    pt.insert(w)

print(pt.search("team"))    # True
print(pt.search("tea"))     # False
print(pt.search("teach"))   # True
print(pt.search("ten"))     # False
```

```
True
False
True
False
```