
Algorithmen und Datenstrukturen

Tries und PATRICIA Tries

Prof. Dr. Ralf Möller

Universität zu Lübeck

Institut für Informationssysteme

Felix Kuhr (Übungen)

sowie viele Tutoren

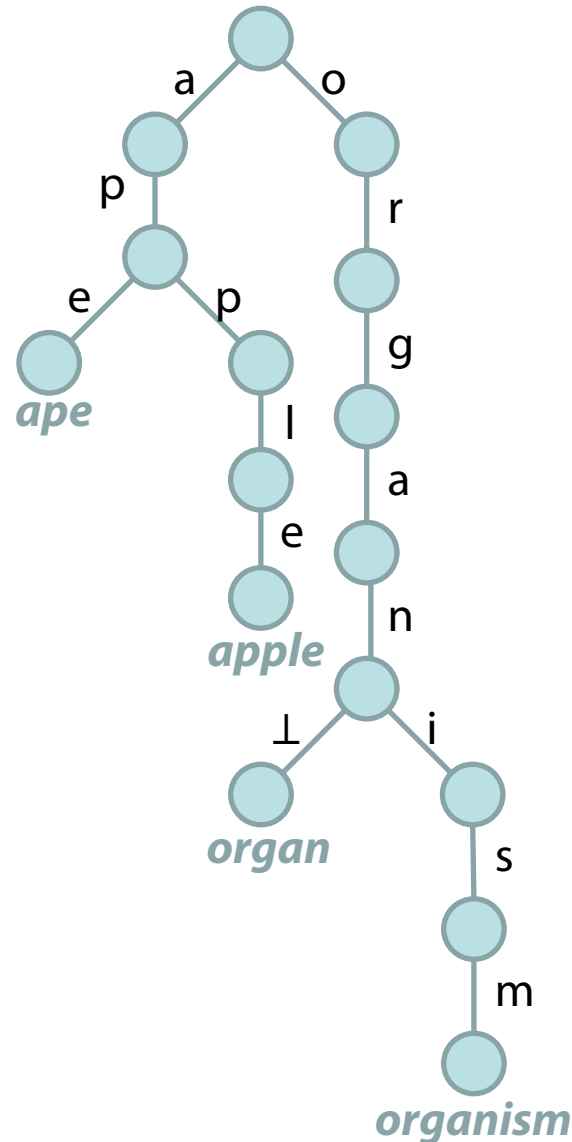
Danksagung

Die nachfolgenden Präsentationen wurden mit einigen Änderungen übernommen aus:

- „Algorithmen und Datenstrukturen“
gehalten von Sven Groppe an der UzL



Idee: Ausnutzung von gemeinsamen Präfixen



Trie

- Repräsentation von Mengen von Zeichenketten
 - Name stammt nach *Edward Fredkin* von re**TRIE**val
 - Anwendungen von Tries finden sich im Bereich des *Information Retrieval*
 - Informationsrückgewinnung aus bestehenden komplexen Daten (Beispiel Internet-Suchmaschine)
 - auch Radix-Baum oder Suffix-Baum genannt
 - Relativ **kompakte Speicherung von Daten** insbesondere mit **gemeinsamen Präfixen**
- 



Trie

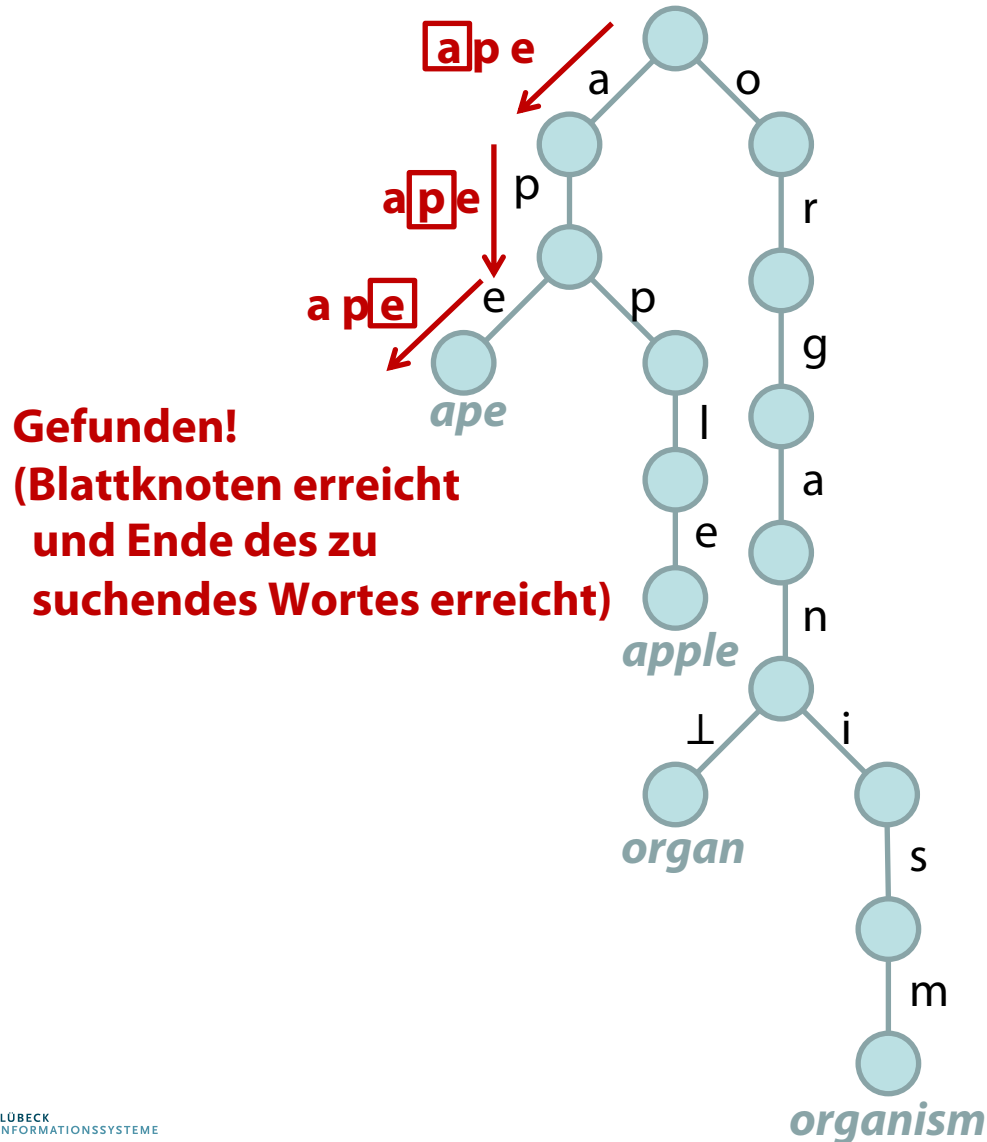
- Voraussetzung
 - Daten darstellbar als Folge von Elementen aus einem (endlichen) Alphabet
 - Beispiele
 - Zeichenkette „Otto“ besteht aus Zeichen ,O‘, ,t‘, ,t‘ und ,o‘ (Alphabet ist alle Zeichen)
 - Zahl 7 ist darstellbar als Folge von Bits 111 (Alphabet ist {0, 1})
 - Unter den Elementen aus dem Alphabet besteht eine Ordnung



Trie - Definition

- Sei Σ das **Alphabet** der zu speichernden Wörter inklusive $\perp$ für „leeres Zeichen“
- Dann ist ein **Trie ein Baum** mit folgenden **Bedingungen**
 - Jeder Knoten hat **0 bis maximal $|\Sigma|$ Kinder**
 - Jede Kante ist durch **genau einen Bezeichner aus Σ** gekennzeichnet
 - Die Kanten zu den Kindern eines Knotens haben **unterschiedliche Bezeichner**
 - d.h. es gibt *keine* Kanten eines Knotens mit demselben Bezeichner
 - in der Regel werden die Kanten sortiert dargestellt und gespeichert
 - **$\perp$ steht nur über einen Blattknoten** und auch nur wenn der Elternknoten des Blattknotens weitere Kindsknoten besitzt
- Der Wert eines Blattknotens ergibt sich aus der **Konkatenation der Kantenbezeichner** von der Wurzel zum Blattknoten

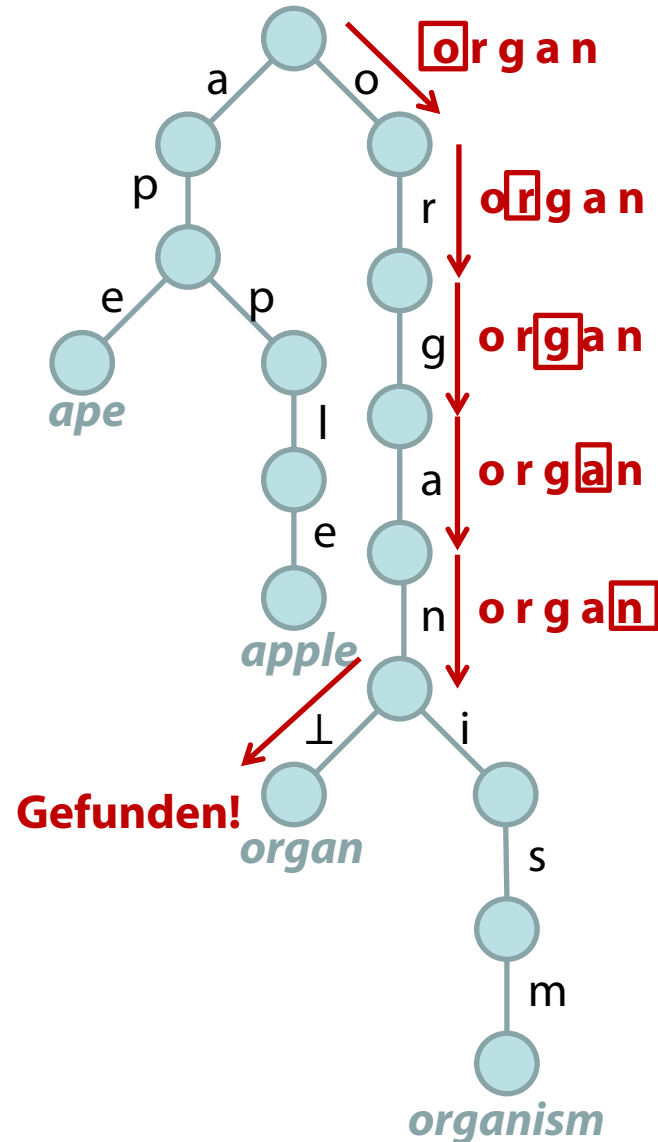
Beispiel: Suche nach **ape**



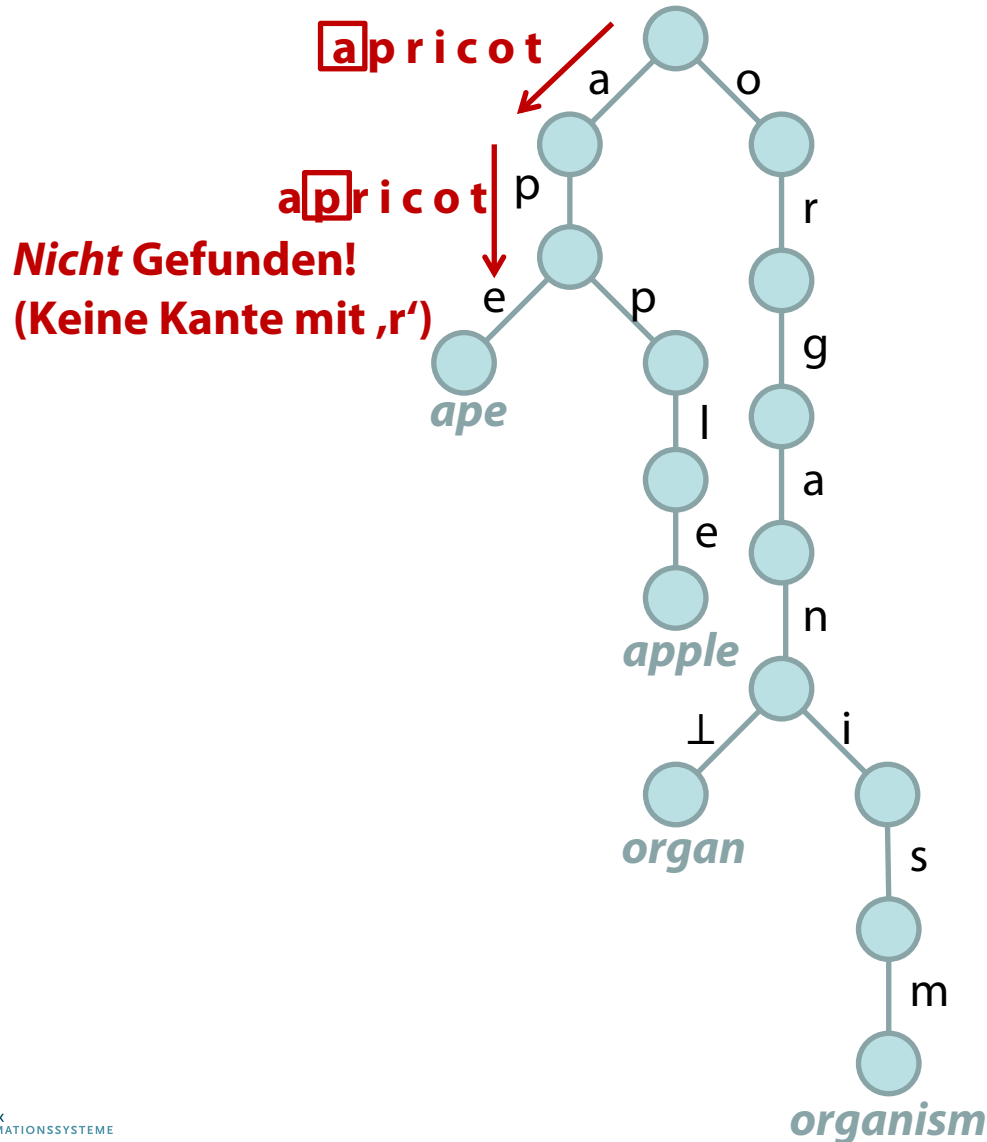
Suche in Tries

- Wir rufen die Suchmethode mit der Wurzel des Trie auf
- Die Suchmethode erhält als Parameter den momentanen Knoten des Trie sowie das restliche zu suchende Wort
 - Falls das restliche zu suchende Wort leer ist,
 - geben wir „**gefunden**“ zurück
 - falls der momentane Knoten ein Blattknoten ist
 - falls eine ausgehende Kante mit dem Bezeichner $\perp$ existiert
 - ansonsten geben wir „**nicht gefunden**“ zurück
 - Falls das restliche zu suchende Wort *nicht leer* ist
 - und falls eine ausgehende Kante existiert mit dem nächsten Zeichen des restlich zu suchenden Wortes, dann rufen wir die Suchmethode mit dem entsprechenden Kindsknoten und um das nächste Zeichen verkürzte restlich zu suchende Wort rekursiv auf
 - ansonsten geben wir nicht „**nicht gefunden**“ zurück

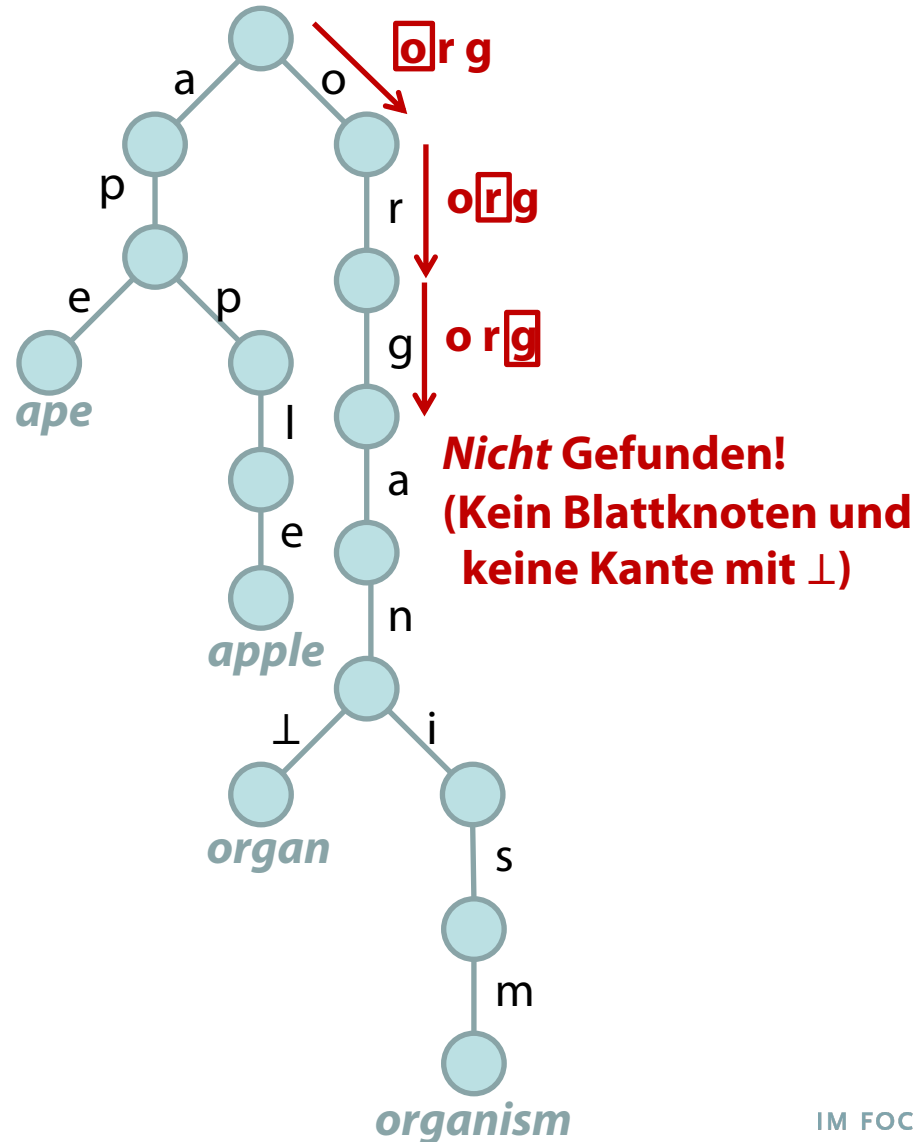
Beispiel: Suche nach **organ**



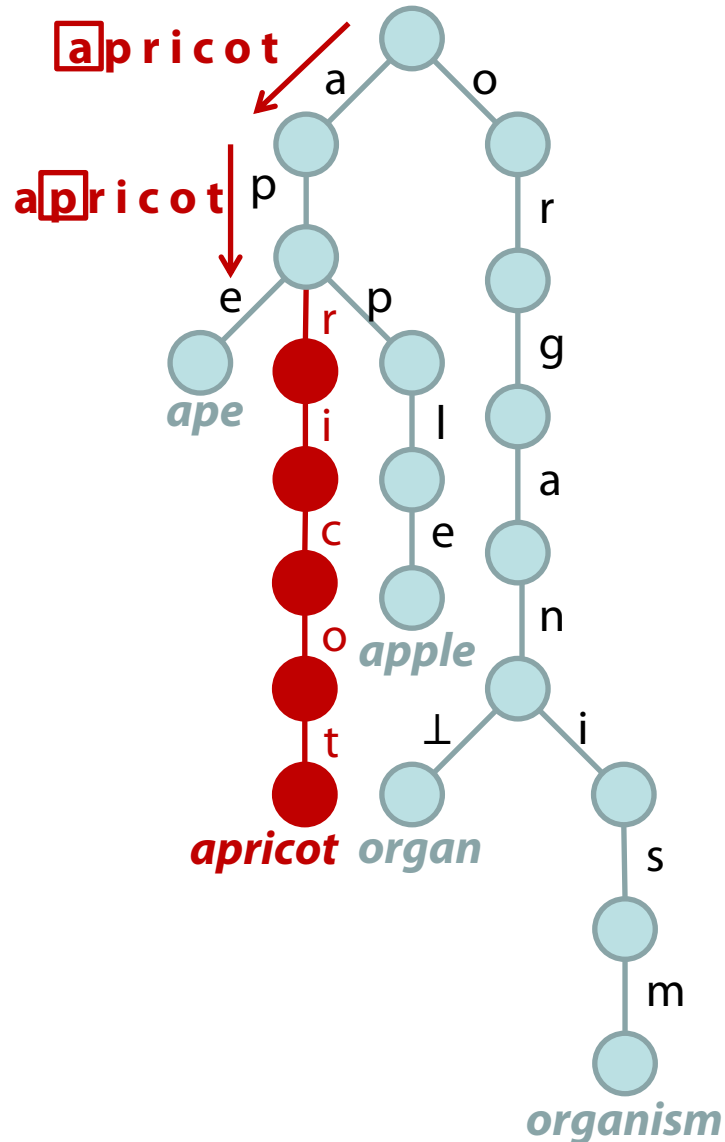
Beispiel: Suche nach **apricot**



Beispiel: Suche nach **org**



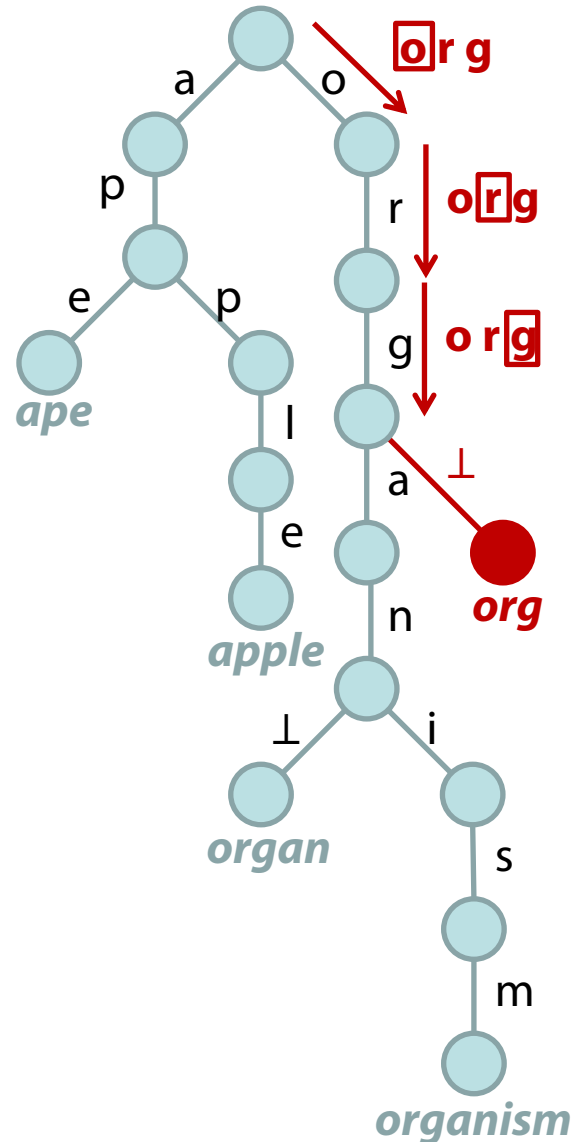
Einfügen: Beispiel **apricot**



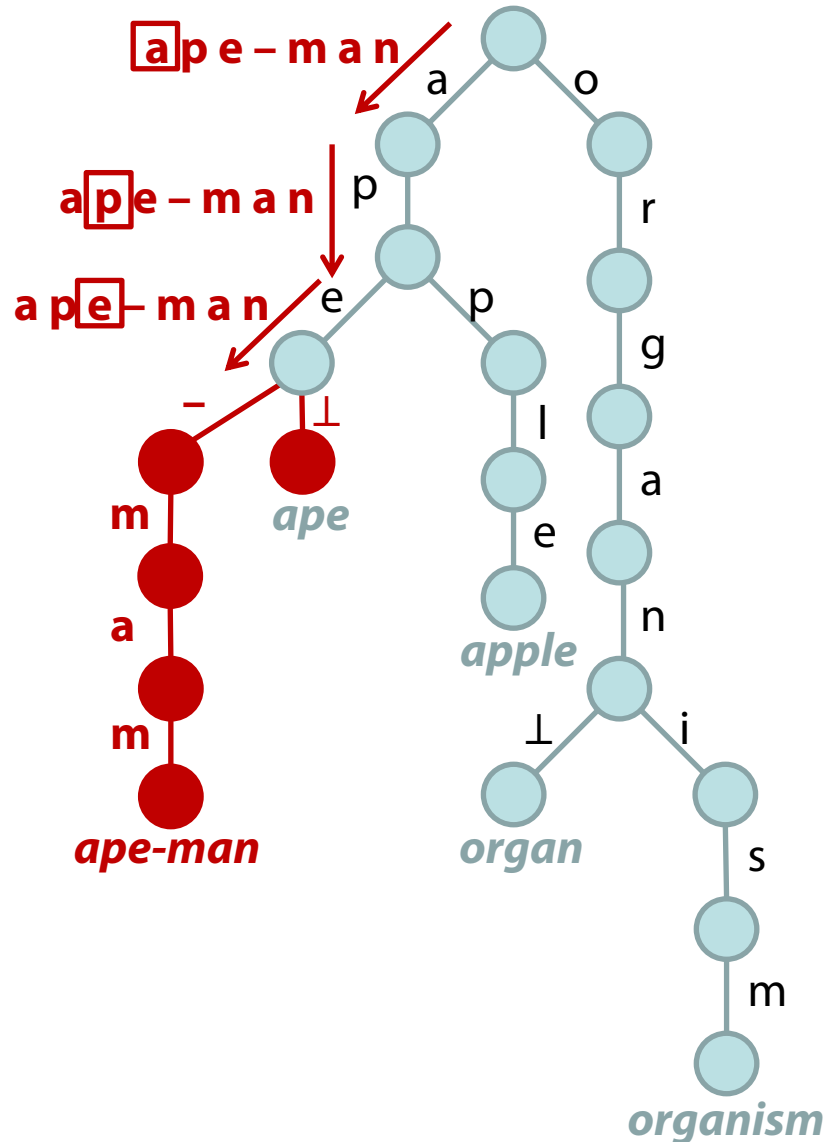
Einfügen in Tries

- Vereinfacht:
 - „Nach einzufügendem Wort suchen und das ergänzen, was fehlt“
- Sonderfälle
 - Füge zusätzlich ein Blattknoten über eine Kante mit Beschriftung $\perp$ zu einem Knoten v hinzu, falls
 - die Suche bei einem inneren Knoten v fertig ist
 - v ein Blattknoten ist und eine neue Kante ausgehend von v hinzugefügt werden muss

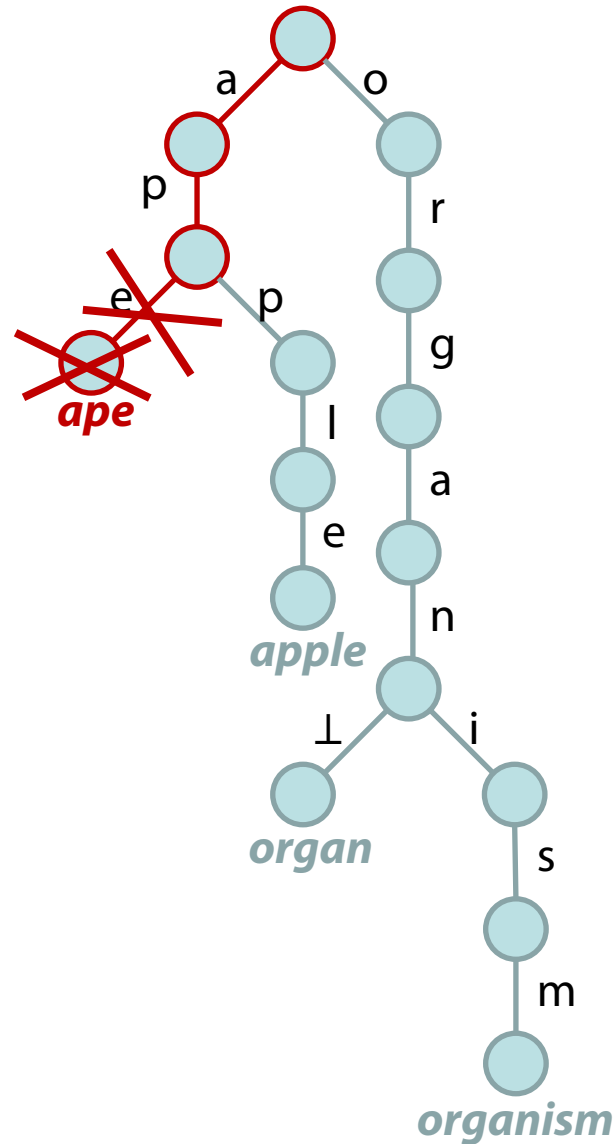
Beispiel: Einfügen von **org**



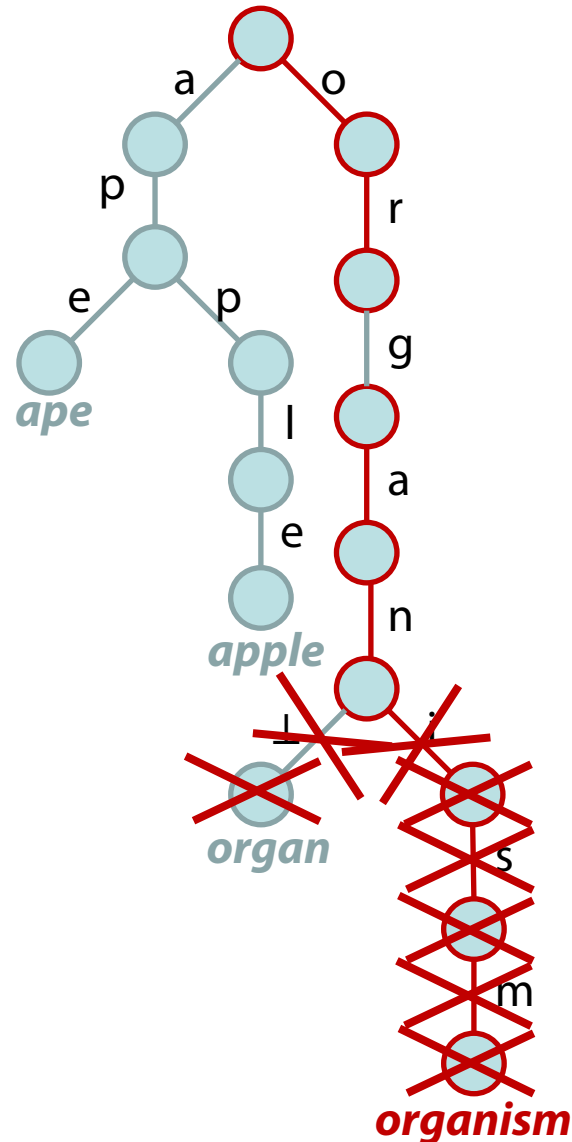
Beispiel: Einfügen von **ape-man**



Löschen von *ape*



Löschen von *organism*

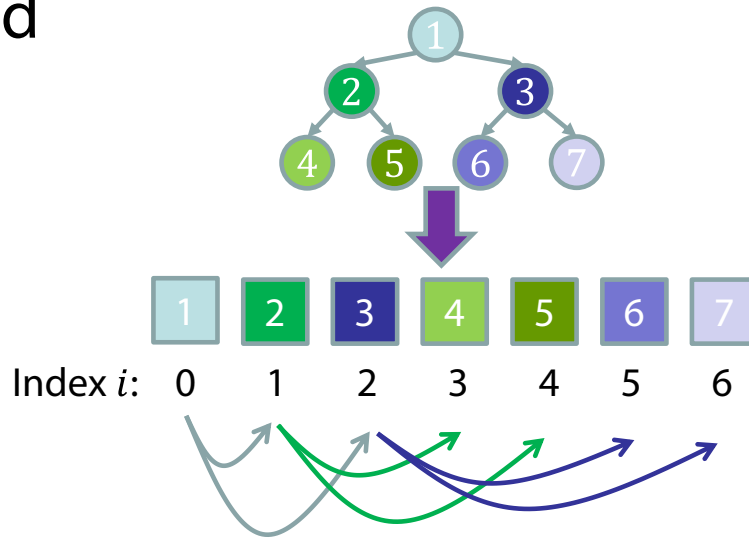


Löschen in Tries

- Zuerst erfolgt eine Suche nach dem zu löschenden Wort
- Beginnend mit dem Blattknoten, der das zu löschende Wort repräsentiert, wird solange der aktuelle Knoten und die Kante zu seinem Elternknoten gelöscht, bis mehr als ein Kindsknoten im Elternknoten vorhanden ist
 - Anschließender Sonderfall:
 - Bei nur einem Kindsknoten, welches über eine $\perp$ -Kante verbunden ist, wird dieser ebenfalls gelöscht

Implementierungsmöglichkeiten

1. Feld



2. Zeigerstruktur

- Kinder in Liste
- Kinder in Feld

Analyse der Implementationsmöglichkeiten

- Kinder in **Liste**
 - **Vorteile:** Geringer Platzbedarf $O(n)$
 - **Nachteil:** Suchen, Einfügen und Entfernen in $O(|\Sigma|)$ pro betrachteten Knoten
- Kinder in **Feld mit Größe der Kinderanzahl**
 - **Vorteil:** Geringer Platzbedarf $O(n)$,
Suchen in $O(\log |\Sigma|)$ pro Knoten (binäre Suche unter den Kindern)
 - **Nachteile:** Einfügen und Entfernen in $O(|\Sigma|)$ pro betr. Knoten
- Kinder in **Feld mit Größe $|\Sigma|$**
 - **Vorteile:** Direkte Adressierung der Kante mit gegebener Beschriftung
(Falls Zeichen k -tes Zeichen im Alphabet ist, so betrachte k -tes Kind)
Suchen, Einfügen und Entfernen in $O(1)$ pro betr. Knoten
 - **Nachteil:** Platzbedarf von $O(n \cdot |\Sigma|)$

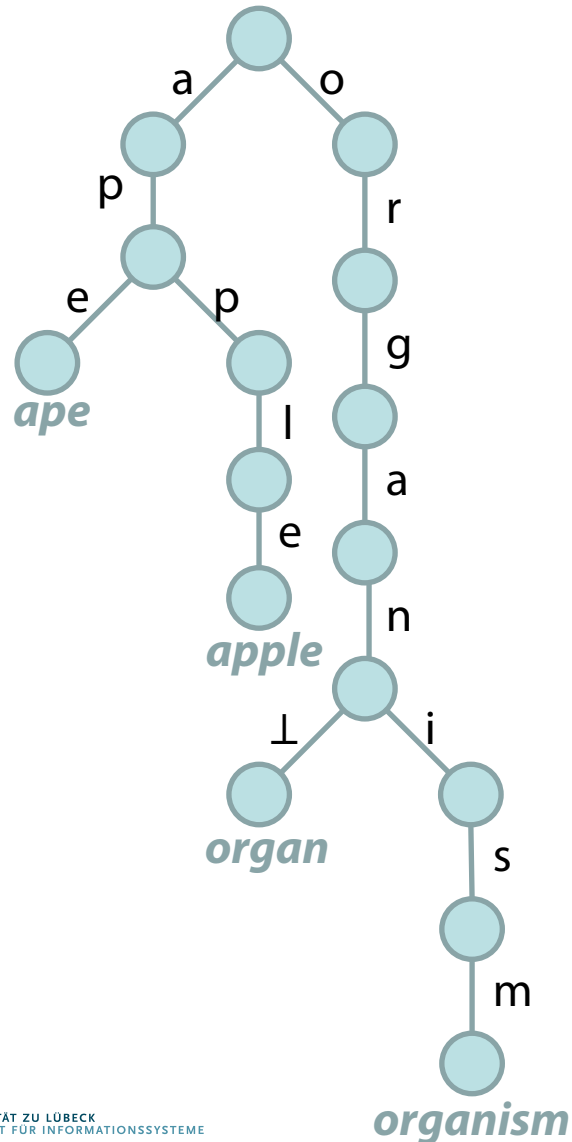
Komplexität der Basisoperationen (Suchen, Einfügen, Löschen)

- Alle Basisoperationen hängen ab
 - von der Tiefe t des Tries
 - t ist gleich der (maximalen) Länge der eingefügten Wörter
 - sowie der Kosten $f(|\Sigma|)$ pro Knoten für diese Operation
 - Abhängig von Speicherstruktur des Tries (siehe vorherige Folie)
 - $O(t \cdot f(|\Sigma|))$
- Bei geeigneter Implementation oder kleinem $|\Sigma|$:
 - $O(t)$

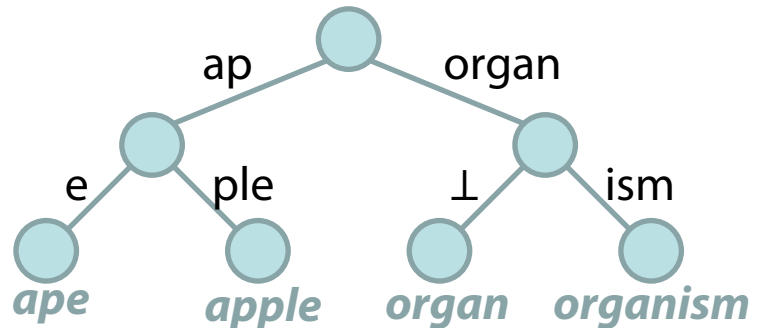
Patricia Tries

- In Tries haben viele Knoten nur 1 Kind und es bilden sich oft lange “Ketten” mit solchen Knoten
- **Idee:** Diese lange “Ketten” zusammenfassen
 - **Konsequenz:** Kanten sind nicht nur mit einem Zeichen, sondern mit Teilwörtern beschriftet

Trie:



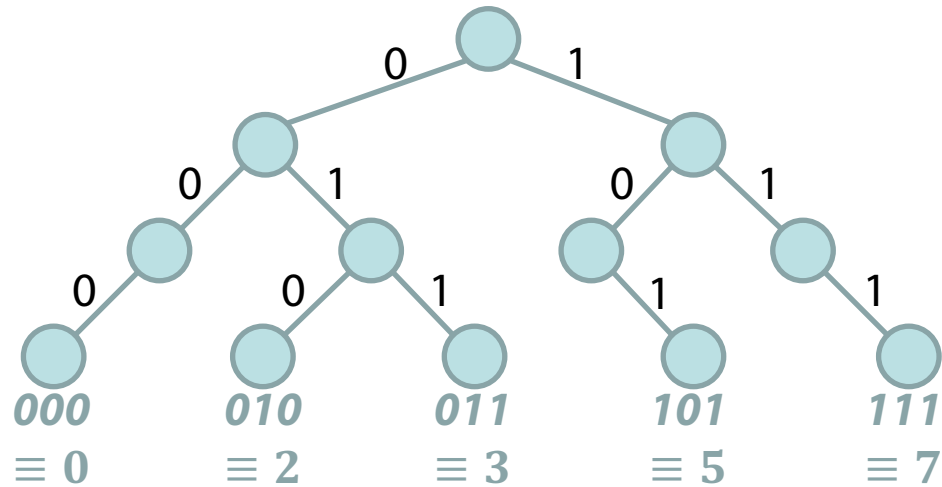
Patricia Trie:



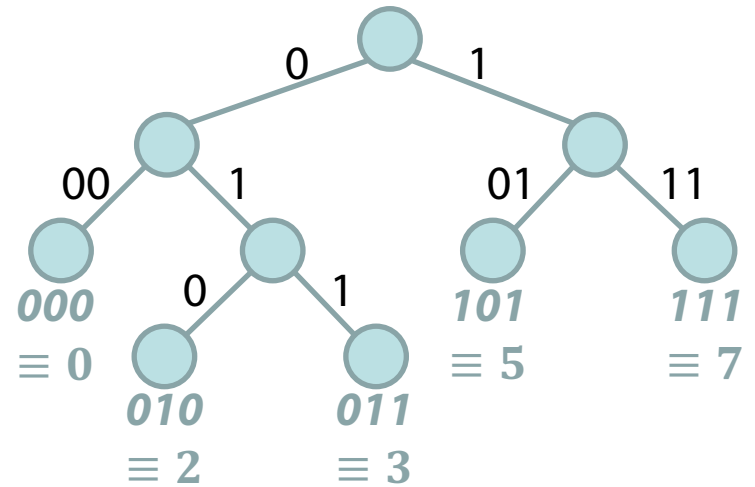
Bedingungen:

- Ausgehende Kanten eines Knotens starten *niemals* mit demselben Zeichen!
- Knoten haben 0 oder 2 bis $|\Sigma|$ viele Kinder, *niemals* hat ein Knoten 1 Kind!

Trie (binär):



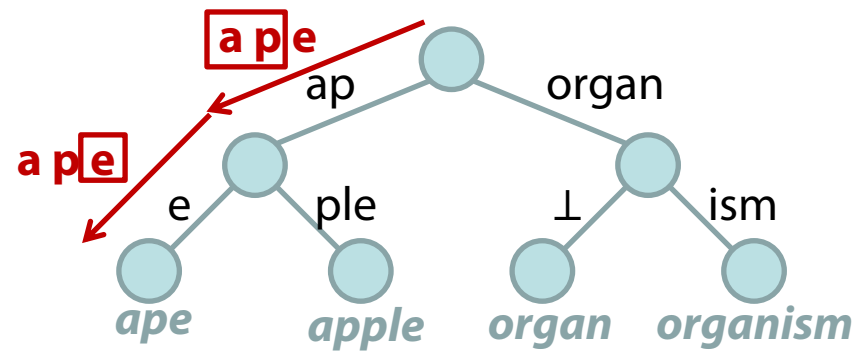
Patricia Trie:



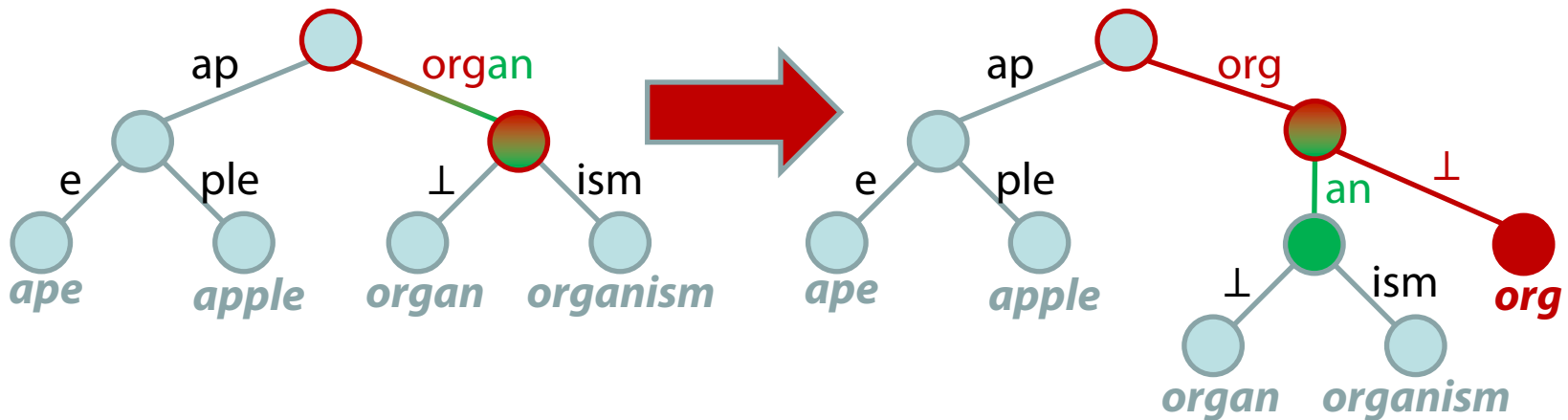
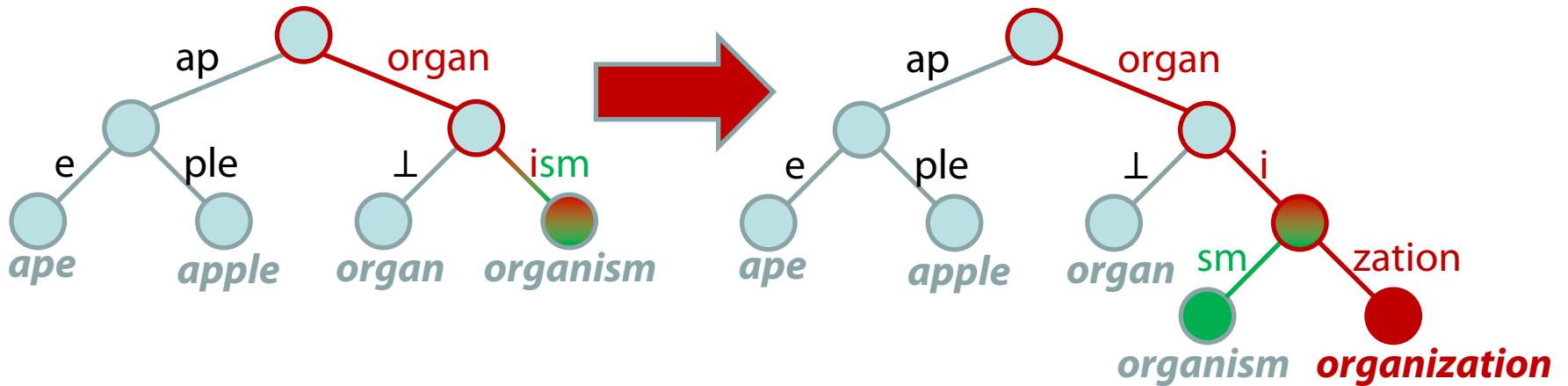
Konsequenzen für die Basisoperationen

- Suchen
 - Anstatt Vergleich von Zeichen nun Vergleich von Teilzeichenketten, sonst keine wesentliche Änderung gegenüber Suchen in Tries
- Einfügen
 - **Neuer Sonderfall:** Aufteilen eines Knotens, „falls Bezeichner der eingehenden Kante (echte) Teilzeichenkette des einzufügenden Wortes ist“
- Löschen
 - **Neuer Sonderfall:** Nach Löschen überprüfen, ob Knoten vereinigt werden können

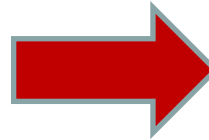
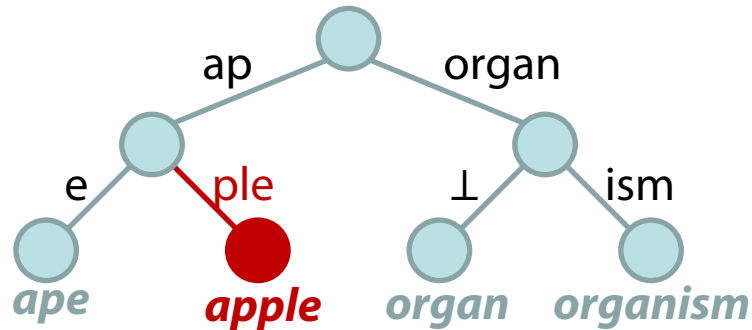
Beispiel Suchen nach *ape*



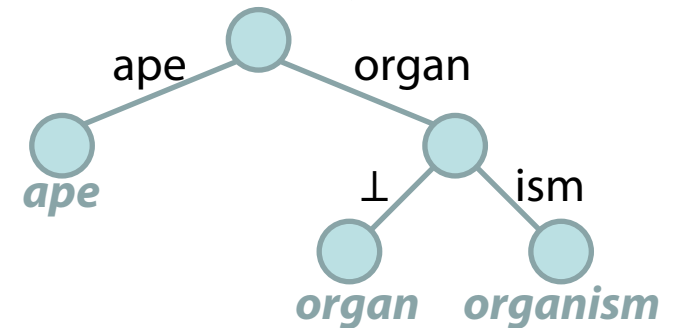
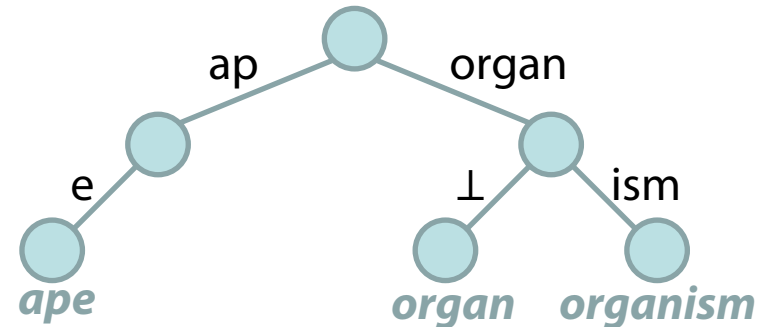
Sonderfall beim Einfügen von **organization** bzw. **org**



Beispiel für neuen Sonderfall beim Löschen von *apple*



Zwischenschritt:



Komplexität

- Ergebnisse für Basisoperationen korrelieren asymptotisch mit denen der Tries
 - D.h. bei geeigneter Implementierung $O(t)$ mit t Tiefe des Patricia Tries, welches mit der maximalen Länge der eingefügten Wörter korreliert
- Da Patricia Tries i.d.R. weniger Knoten haben, sind in der Praxis
 - Patricia Tries *schneller* und
 - haben *geringeren Speicherplatzverbrauch*

Zusammenfassung

- Trie
 - n-äres Alphabet
 - Speichern von Zeichenketten
 - binäres Alphabet
 - Speichern von z.B. Zahlen
- Patricia Trie
 - kompaktere Darstellung
 - n-äres Alphabet
 - binäres Alphabet