Trie Trees und verwandte Techniken

Joe Koelbel
cgt103005

Gliederung

- Wörterbuchproblem
- Trie Trees
  - Allgemeines
  - Aufbau
  - Wörterbuchoperationen
  - Praktische Anwendungen
  - Optimierungen
  - Effizienz
- C# Dictionary
  - Hashing
  - Wörterbuchoperationen
  - Effizienz
- Balancierte Binäre Suchbäume
  - Rot-Schwarz-Baum
    - Aufbau
    - Wörterbuchoperationen
  - weitere balancierte binäre Suchbäume
  - Effizienz
  - Vergleich mit Trie

Wörterbuchproblem

n Datensätze sollen durch n Schlüssel eindeutig identifizierbar sein

Drei Operationen:

- insert: Fügt einen Datensatz + Schlüssel hinzu
- delete: Löscht einen Datensatz + Schlüssel
- find: Überprüft, ob ein Schlüssel vorhanden ist und liefert den dazugehörigen Datensatz

Trie Tree - Allgemeines

1959 von René de la Briandais beschrieben

1961 Namensgebung durch Edward Fredkin

re**trie**val

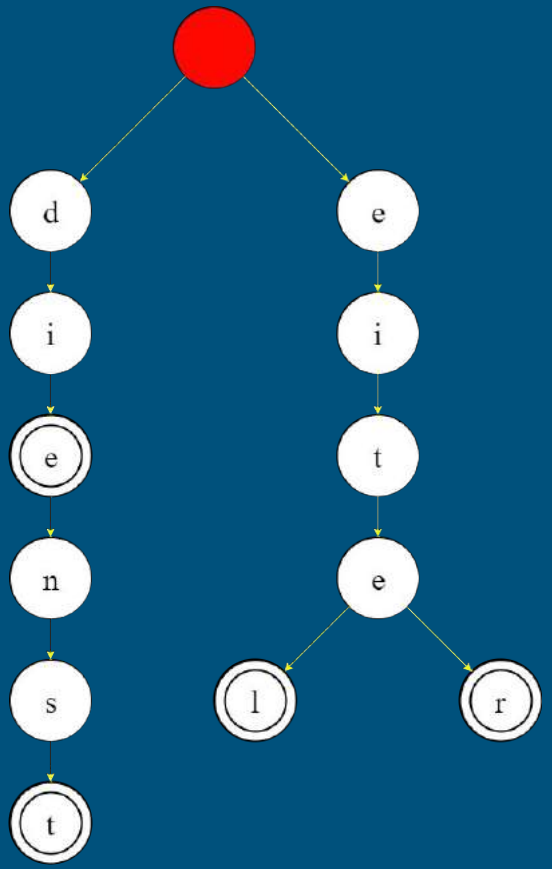
Lösung des Wörterbuchproblems für Strings

Trie Tree - Aufbau

- Leere Wurzel
- Jeder Knoten speichert einen Buchstaben
- Anzahl direkter Kindknoten <= Größe des verwendeten Alphabets
- Kindknoten speichert Referenz auf Elternknoten
- boolescher Wert: Knoten ist Ende eines Strings?

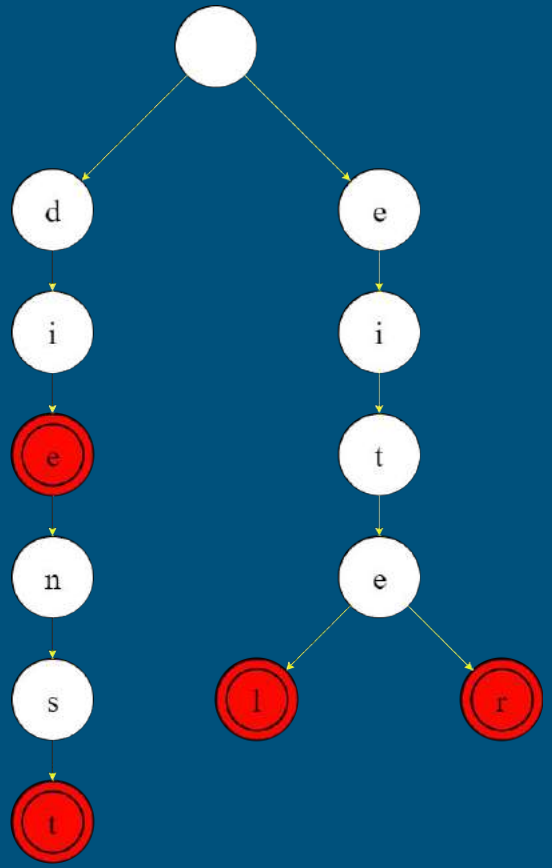
Trie Tree - Beispiel

Leere Wurzel



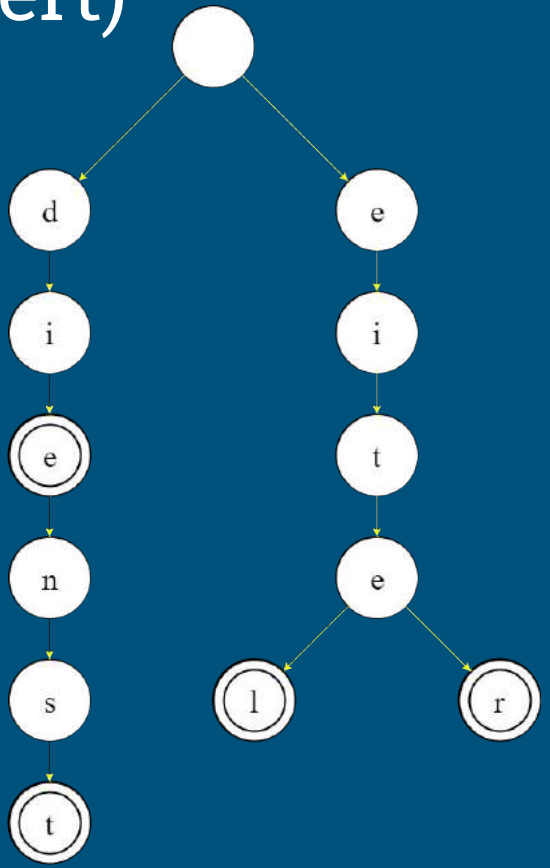
Trie Tree - Beispiel

boolescher Wert:
Knoten ist Ende eines Strings?



Trie Tree - Einfügen (insert)

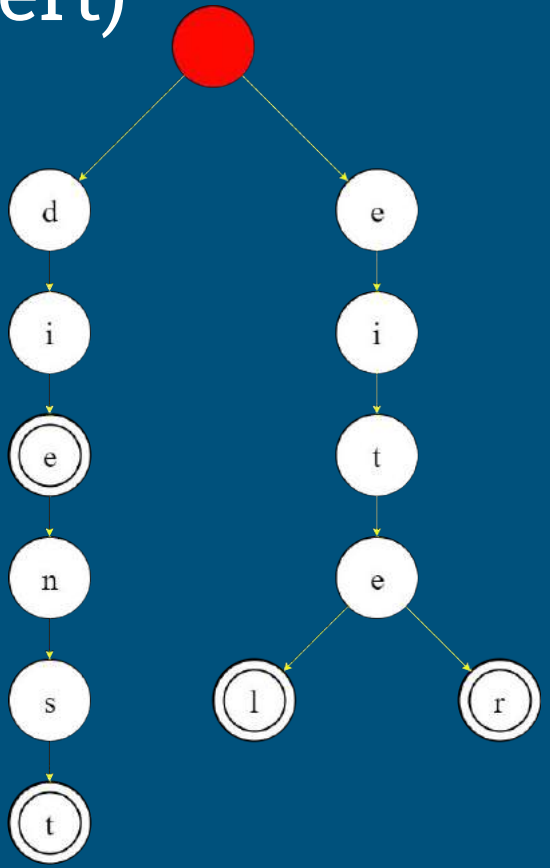
Einfügen von “wow”



Trie Tree - Einfügen (insert)

Einfügen von “wow”

Enthält Wurzel Kind ‘w’?

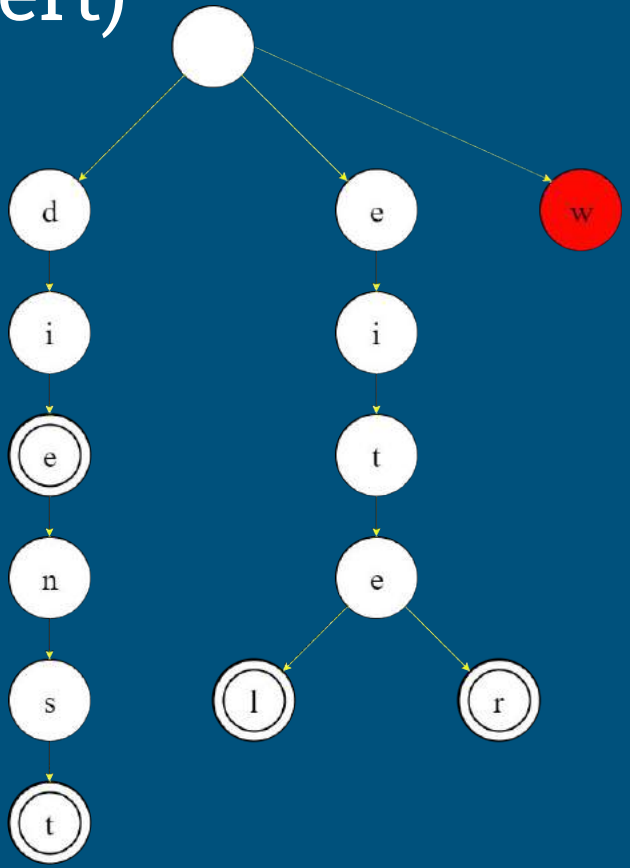


Trie Tree - Einfügen (insert)

Einfügen von “wow”

Enthält Wurzel Kind ‘w’?

Erzeuge neuen Knoten ‘w’



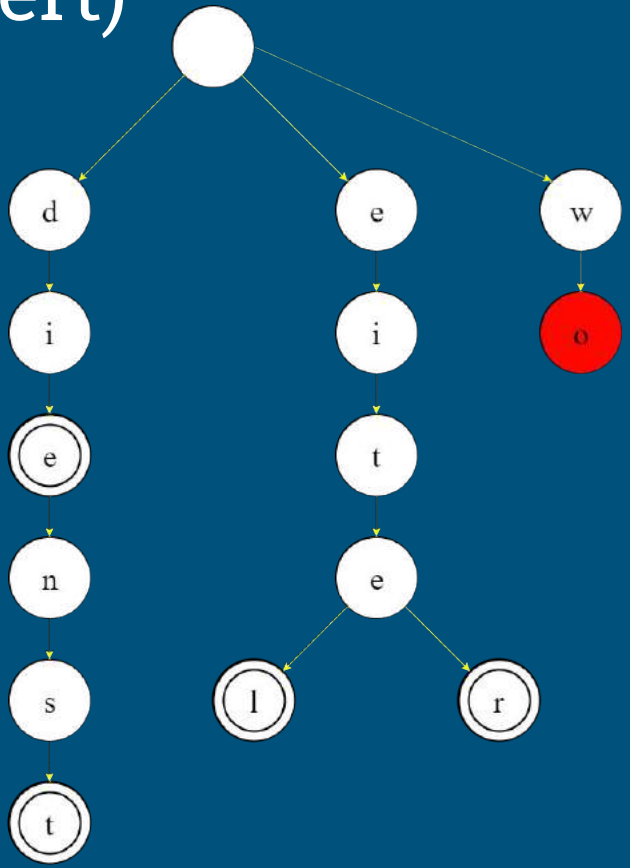
Trie Tree - Einfügen (insert)

Einfügen von “wow”

Enthält Wurzel Kind ‘w’?

Erzeuge neuen Knoten ‘w’

Erzeuge neuen Knoten ‘o’



Trie Tree - Einfügen (insert)

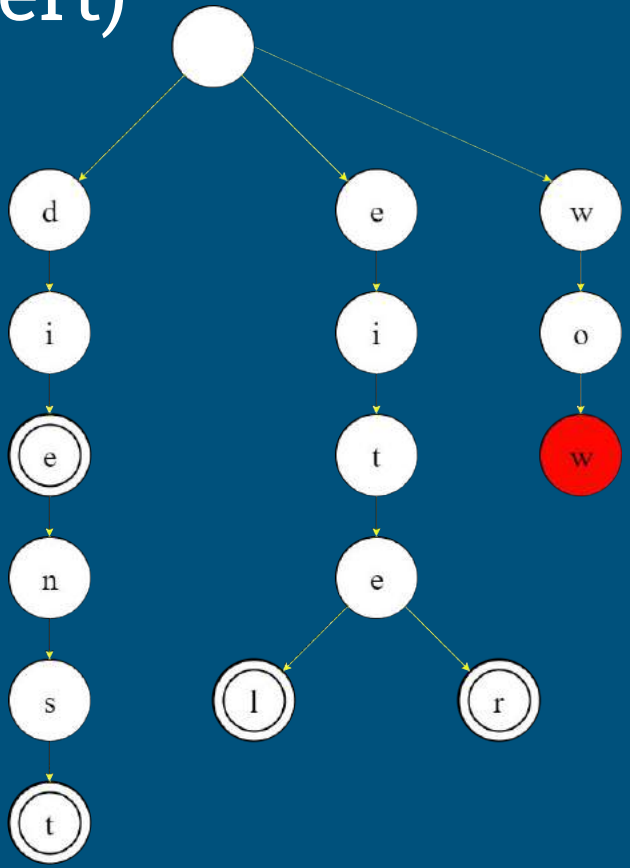
Einfügen von “wow”

Enthält Wurzel Kind ‘w’?

Erzeuge neuen Knoten ‘w’

Erzeuge neuen Knoten ‘o’

Erzeuge neuen Knoten ‘w’



Trie Tree - Einfügen (insert)

Einfügen von “wow”

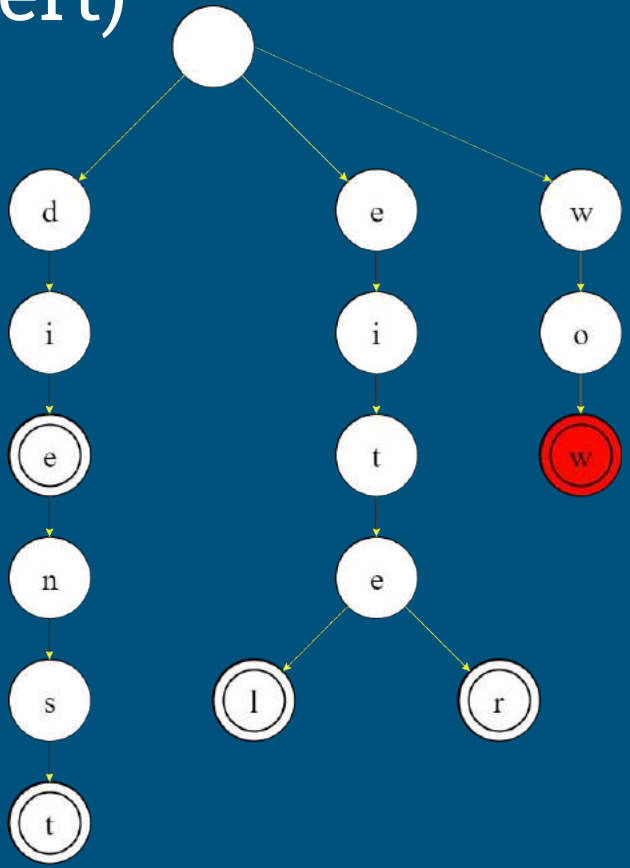
Enthält Wurzel Kind ‘w’?

Erzeuge neuen Knoten ‘w’

Erzeuge neuen Knoten ‘o’

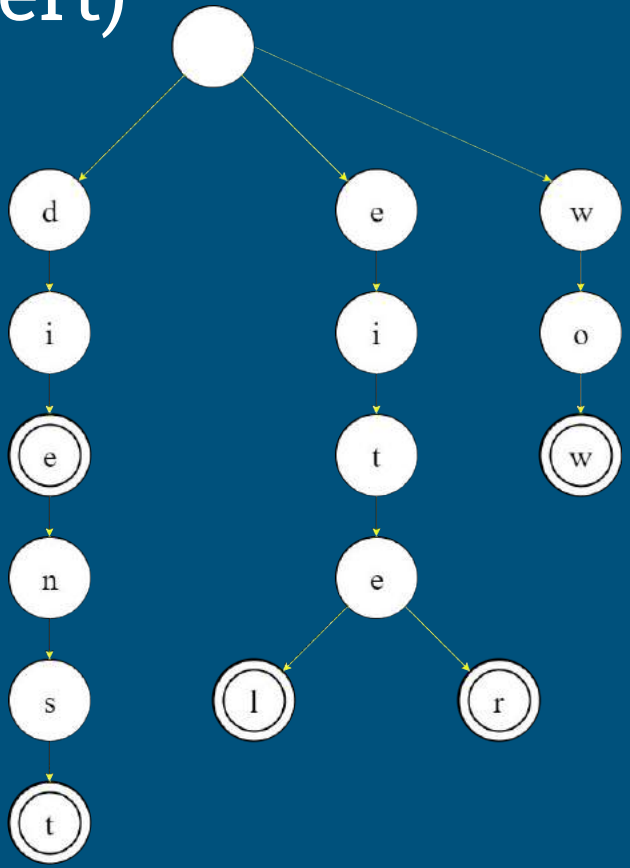
Erzeuge neuen Knoten ‘w’

Setze boolean



Trie Tree - Einfügen (insert)

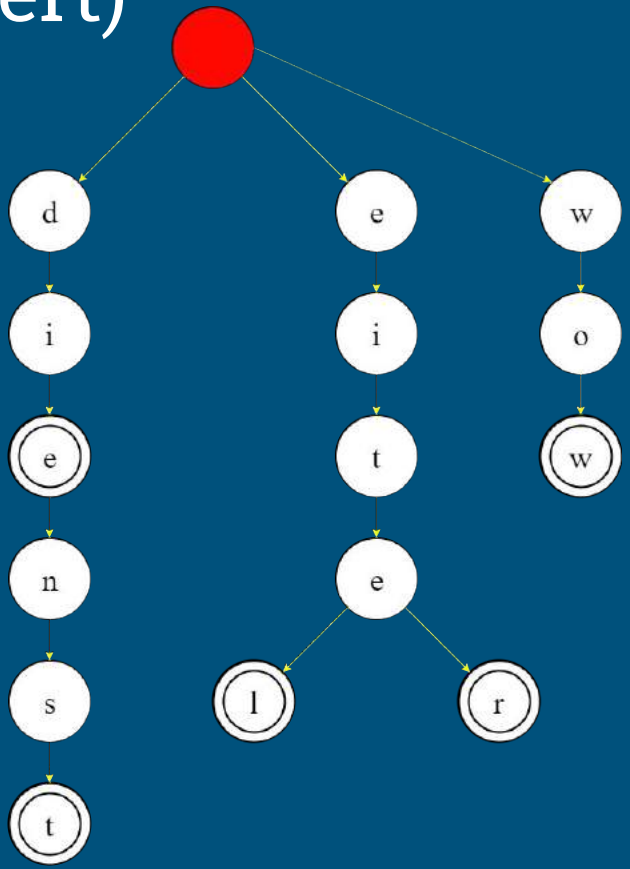
Einfügen von “dieter”



Trie Tree - Einfügen (insert)

Einfügen von “dieter”

Enthält Wurzel Kind ‘d’?

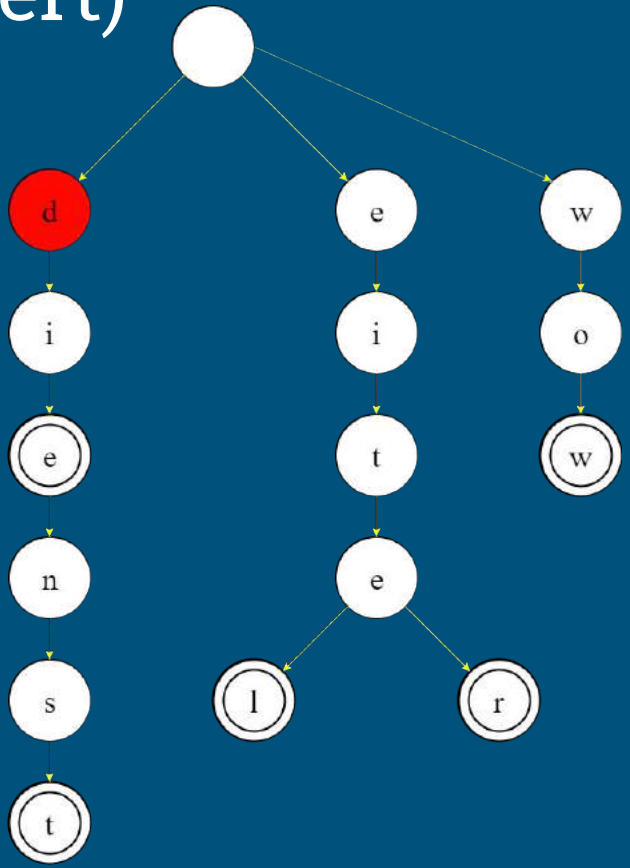


Trie Tree - Einfügen (insert)

Einfügen von “dieter”

Enthält Wurzel Kind ‘d’?

Enthält Knoten ‘d’ Kind ‘i’?



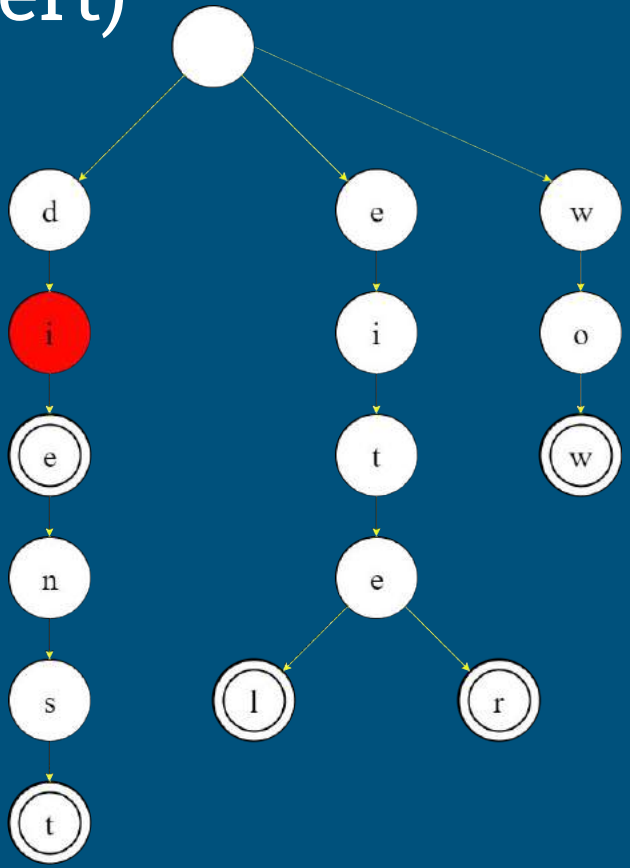
Trie Tree - Einfügen (insert)

Einfügen von “dieter”

Enthält Wurzel Kind ‘d’?

Enthält Knoten ‘d’ Kind ‘i’?

Enthält Knoten ‘i’ Kind ‘e’?



Trie Tree - Einfügen (insert)

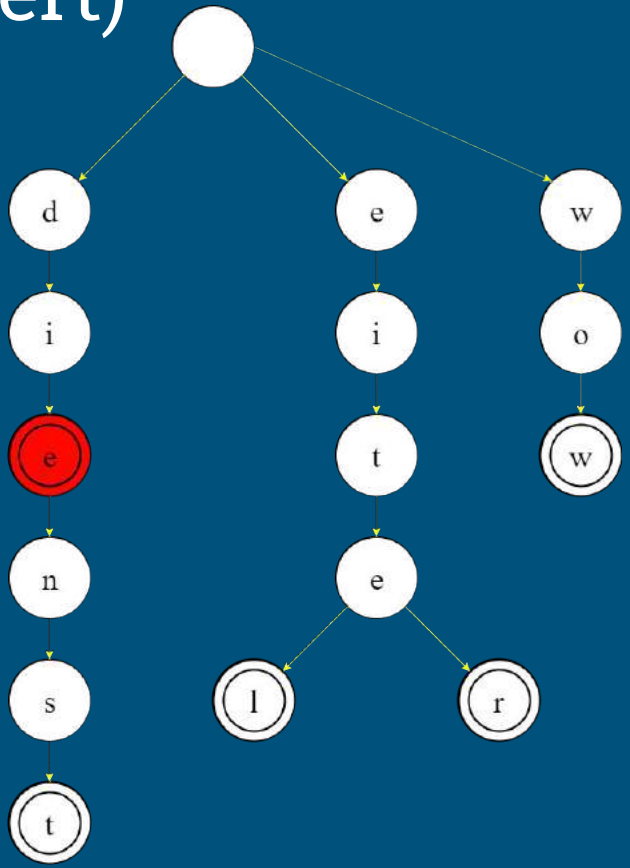
Einfügen von “dieter”

Enthält Wurzel Kind ‘d’?

Enthält Knoten ‘d’ Kind ‘i’?

Enthält Knoten ‘i’ Kind ‘e’?

Enthält Knoten ‘e’ Kind ‘t’?



Trie Tree - Einfügen (insert)

Einfügen von “dieter”

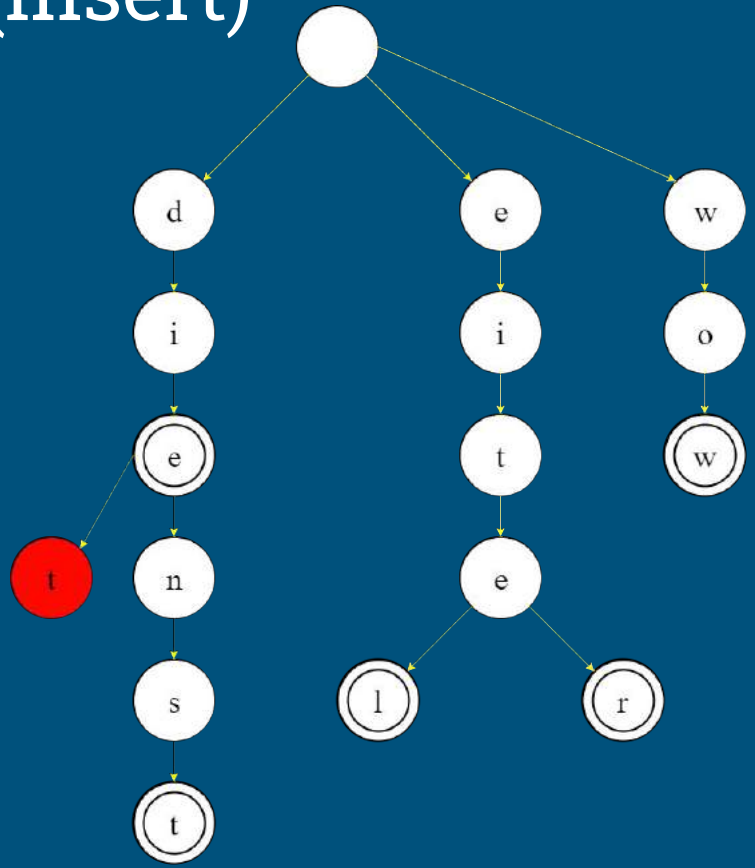
Enthält Wurzel Kind ‘d’?

Enthält Knoten ‘d’ Kind ‘i’?

Enthält Knoten ‘i’ Kind ‘e’?

Enthält Knoten ‘e’ Kind ‘t’?

Erzeuge neuen Knoten ‘t’



Trie Tree - Einfügen (insert)

Einfügen von “dieter”

Enthält Wurzel Kind ‘d’?

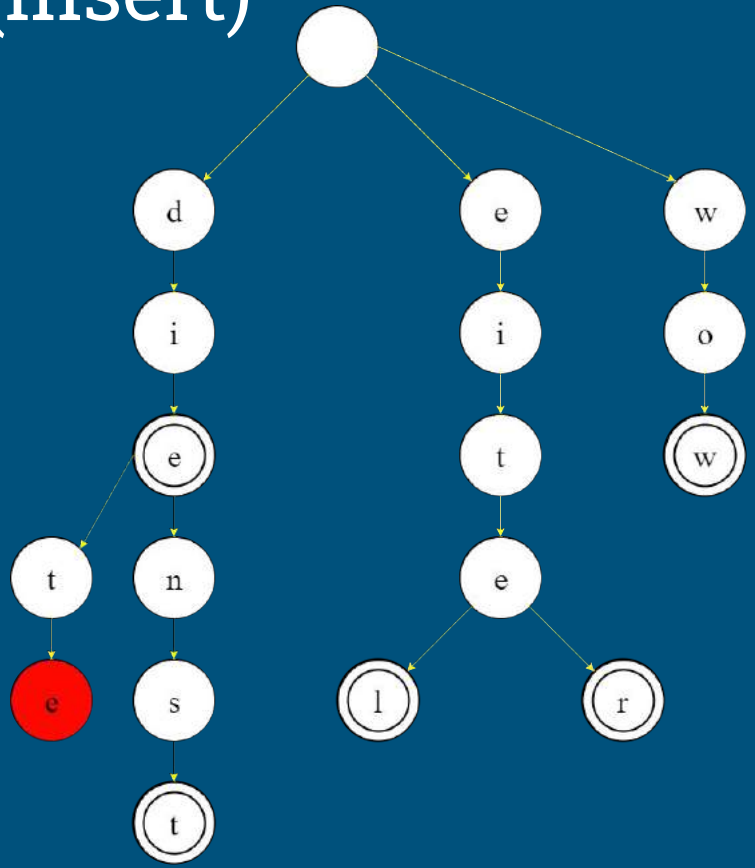
Enthält Knoten ‘d’ Kind ‘i’?

Enthält Knoten ‘i’ Kind ‘e’?

Enthält Knoten ‘e’ Kind ‘t’?

Erzeuge neuen Knoten ‘t’

Erzeuge neuen Knoten ‘e’



Trie Tree - Einfügen (insert)

Einfügen von “dieter”

Enthält Wurzel Kind ‘d’?

Enthält Knoten ‘d’ Kind ‘i’?

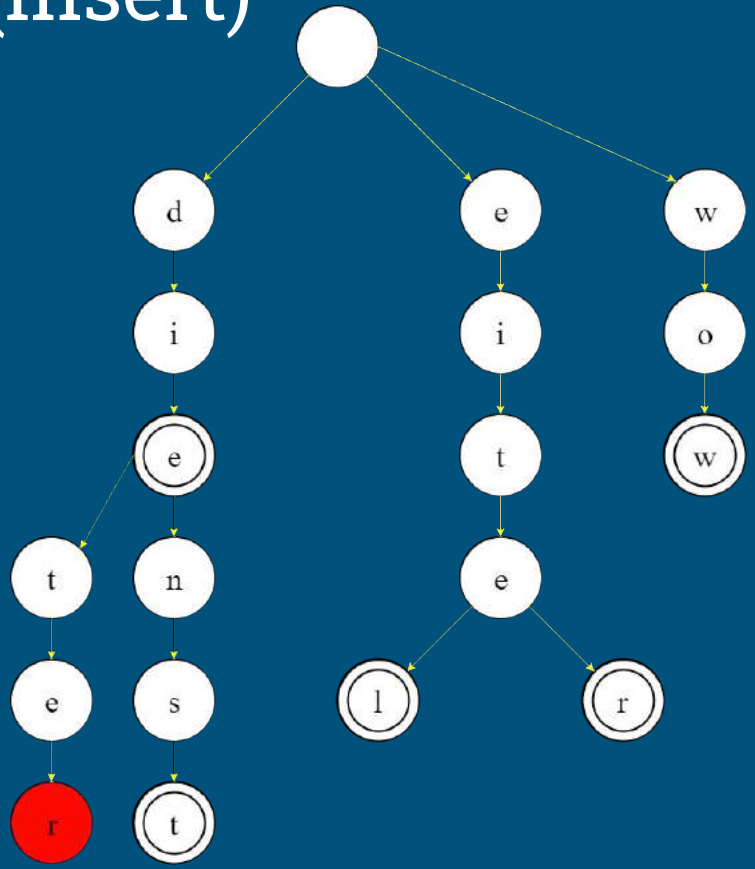
Enthält Knoten ‘i’ Kind ‘e’?

Enthält Knoten ‘e’ Kind ‘t’?

Erzeuge neuen Knoten ‘t’

Erzeuge neuen Knoten ‘e’

Erzeuge neuen Knoten ‘r’



Trie Tree - Einfügen (insert)

Einfügen von “dieter”

Enthält Wurzel Kind ‘d’?

Enthält Knoten ‘d’ Kind ‘i’?

Enthält Knoten ‘i’ Kind ‘e’?

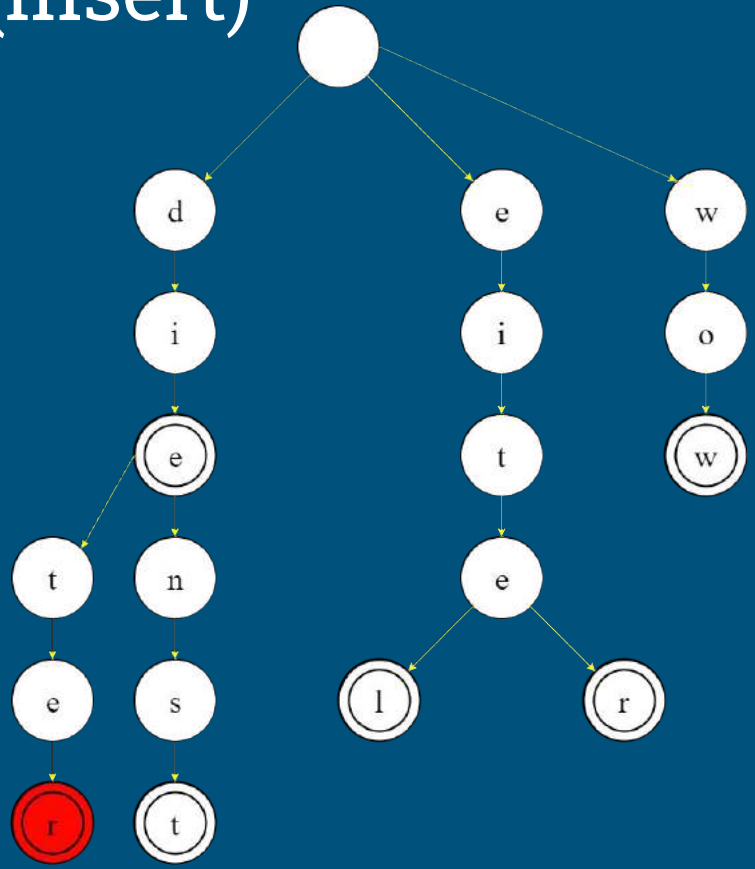
Enthält Knoten ‘e’ Kind ‘t’?

Erzeuge neuen Knoten ‘t’

Erzeuge neuen Knoten ‘e’

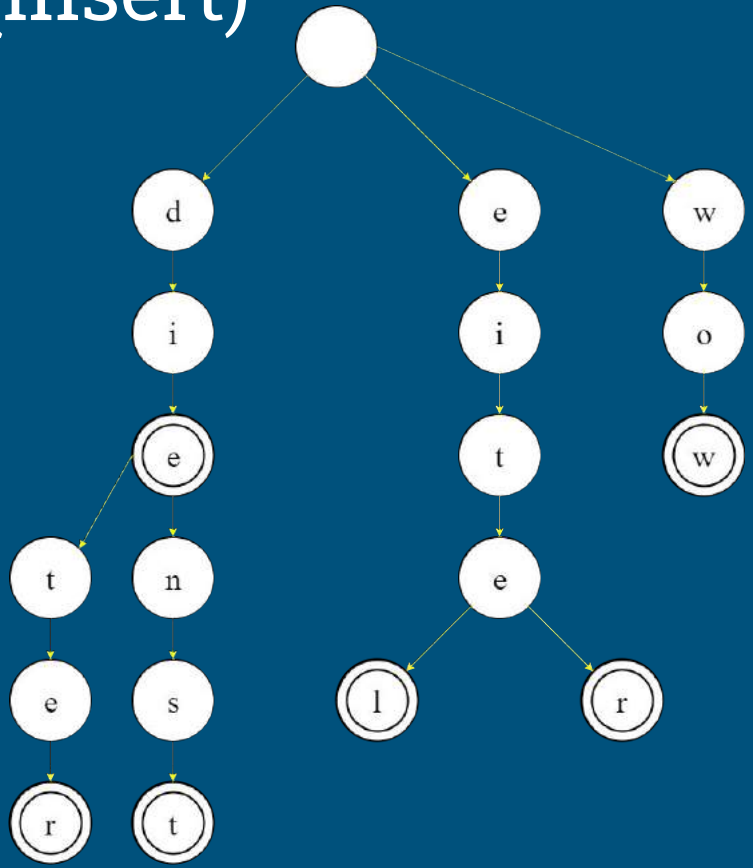
Erzeuge neuen Knoten ‘r’

Setze boolean



Trie Tree - Einfügen (insert)

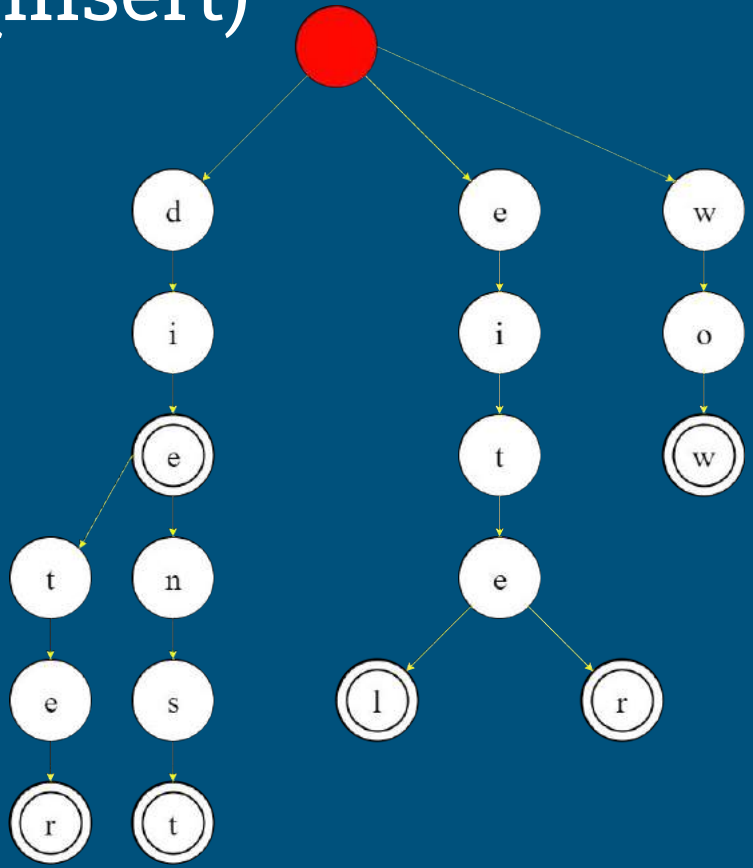
Einfügen von “ei”



Trie Tree - Einfügen (insert)

Einfügen von “ei”

Enthält Wurzel Kind ‘e’?

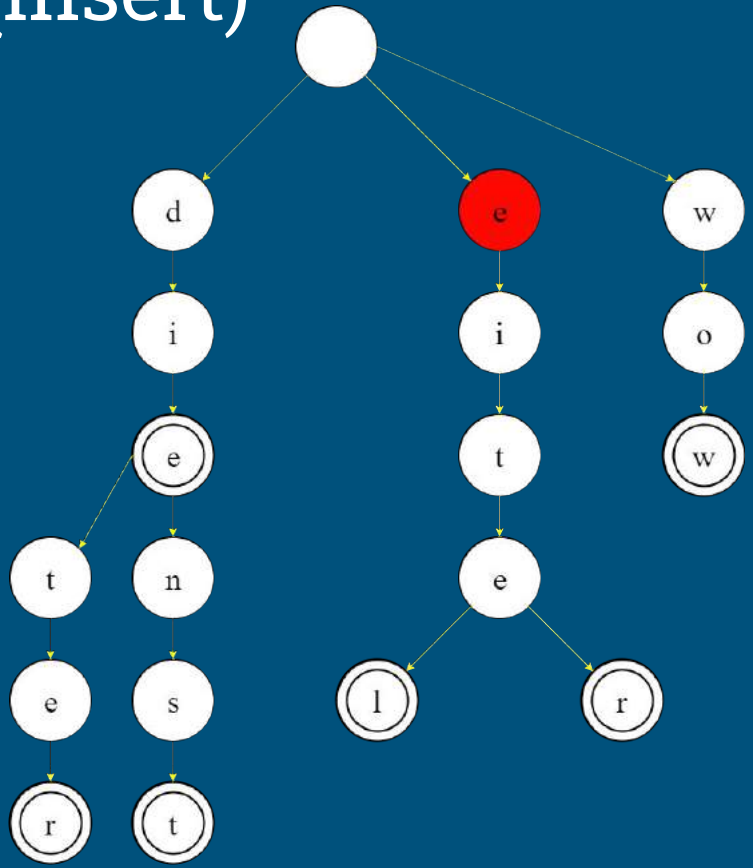


Trie Tree - Einfügen (insert)

Einfügen von “ei”

Enthält Wurzel Kind ‘e’?

Enthält Knoten ‘e’ Kind ‘i’?



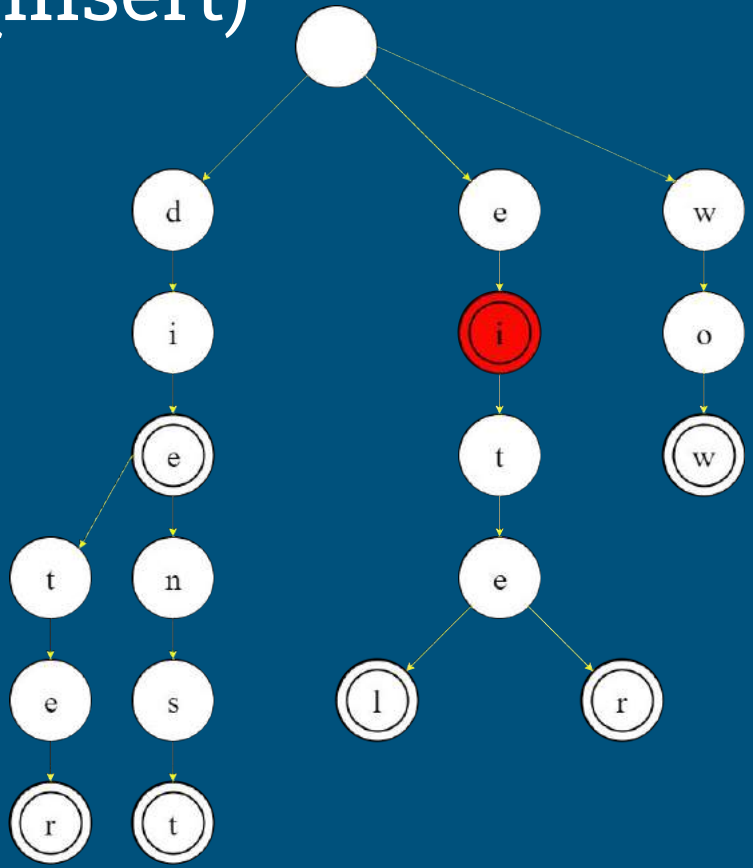
Trie Tree - Einfügen (insert)

Einfügen von “ei”

Enthält Wurzel Kind ‘e’?

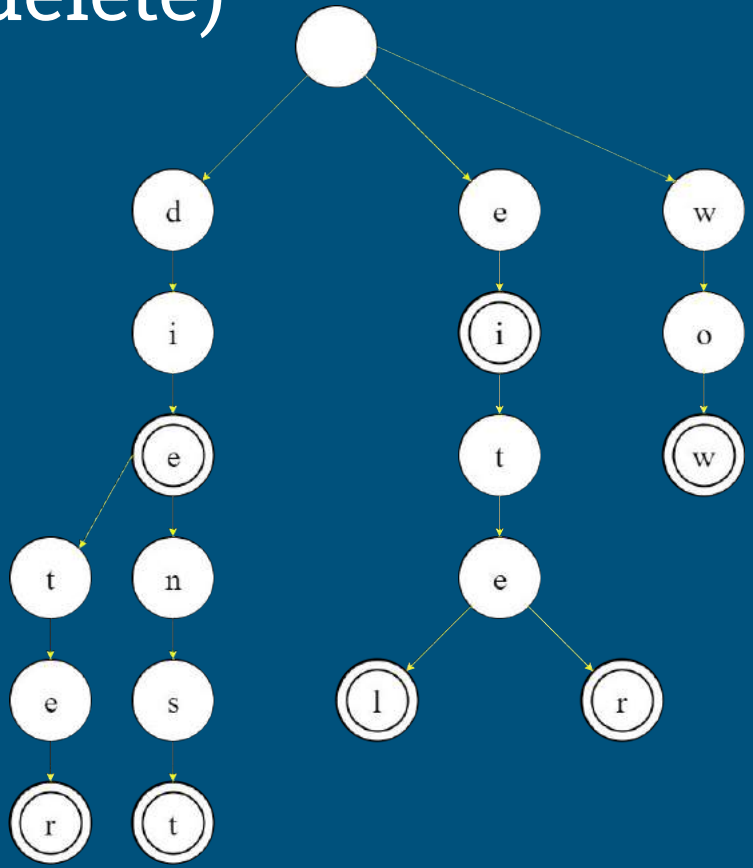
Enthält Knoten ‘e’ Kind ‘i’?

Setze boolean



Trie Tree - Löschen (delete)

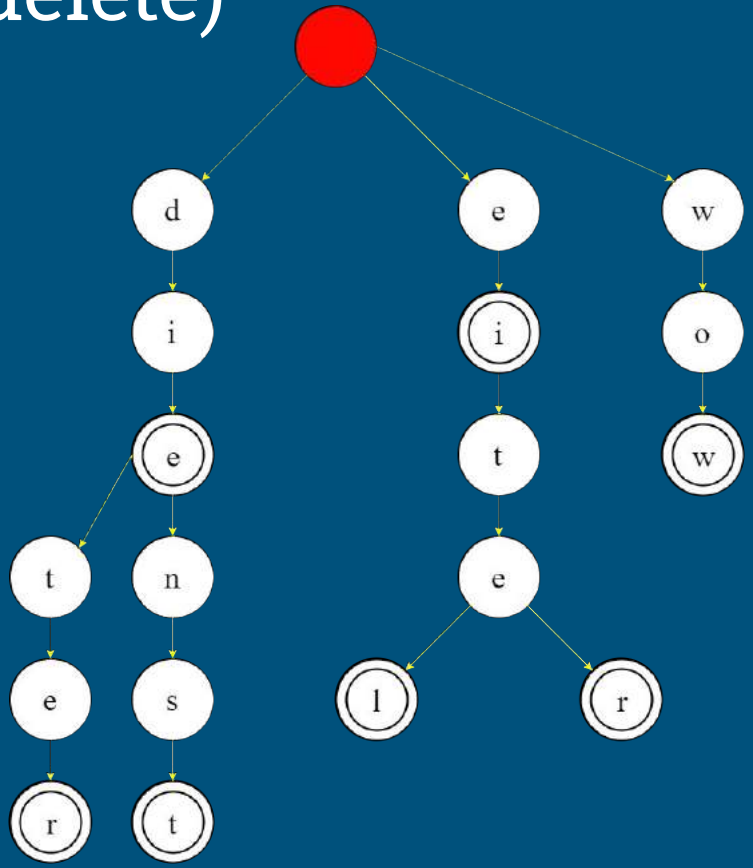
Löschen von “ei”



Trie Tree - Löschen (delete)

Löschen von “ei”

Enthält Wurzel Kind ‘e’?

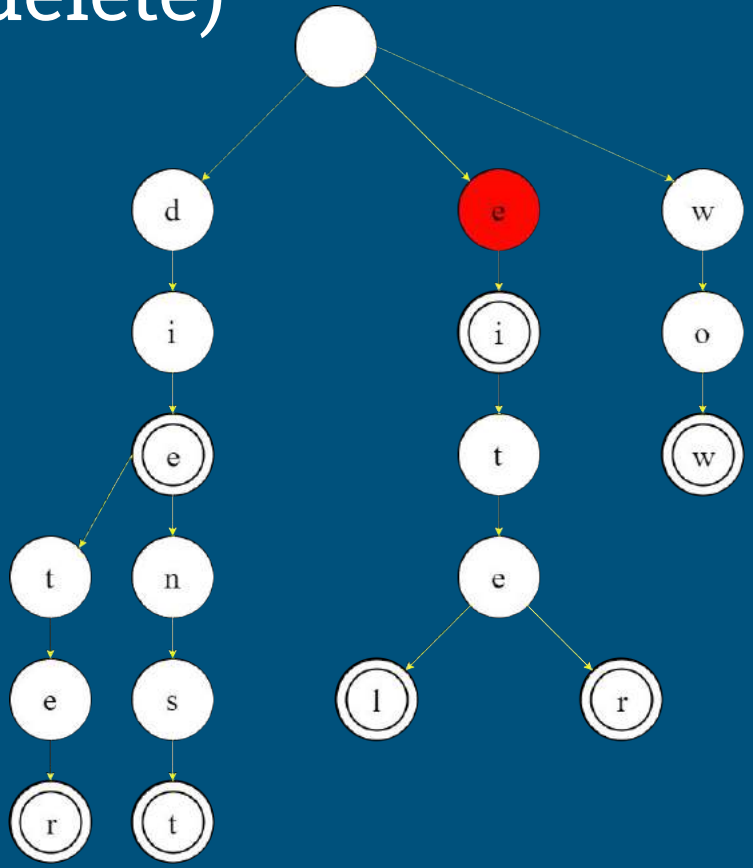


Trie Tree - Löschen (delete)

Löschen von “ei”

Enthält Wurzel Kind ‘e’?

Enthält Knoten ‘e’ Kind ‘i’?



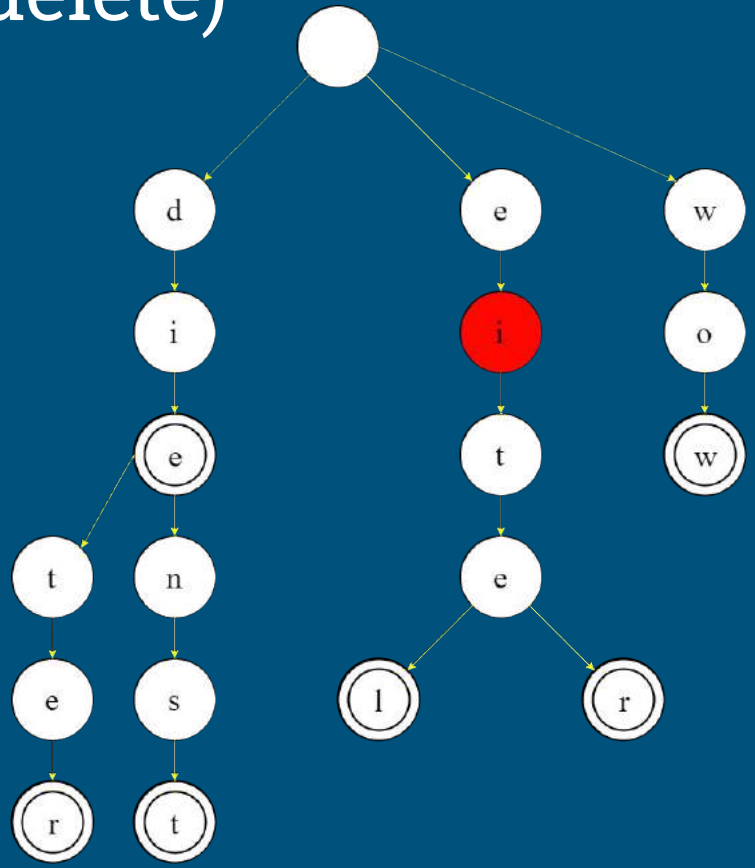
Trie Tree - Löschen (delete)

Löschen von “ei”

Enthält Wurzel Kind ‘e’?

Enthält Knoten ‘e’ Kind ‘i’?

Entferne boolean



Trie Tree - Löschen (delete)

Löschen von “ei”

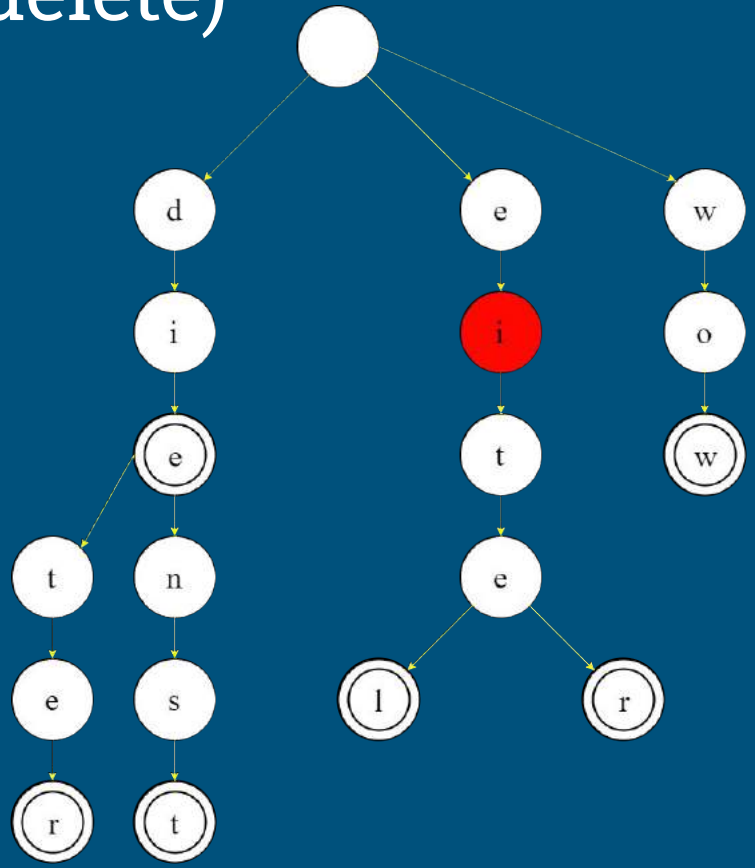
Enthält Wurzel Kind ‘e’?

Enthält Knoten ‘e’ Kind ‘i’?

Entferne boolean

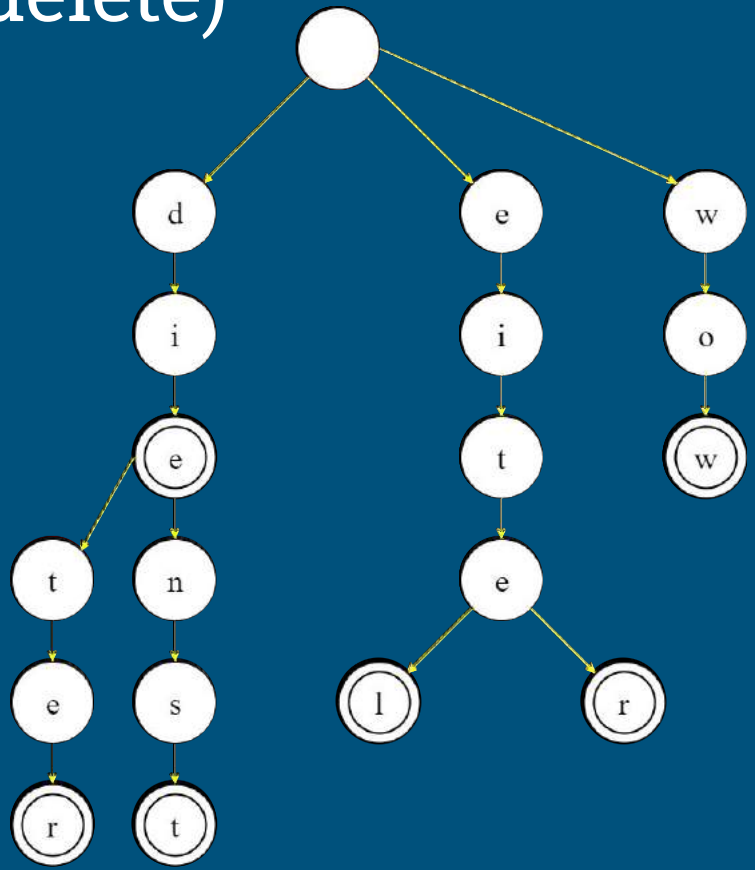
Hat Knoten ‘i’ Kinder?

Beenden



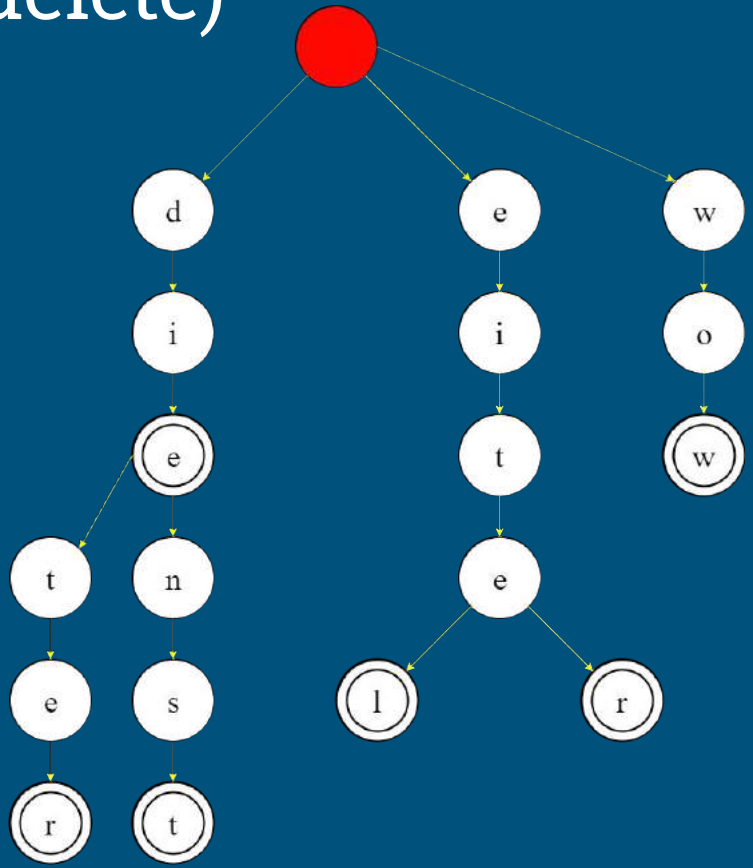
Trie Tree - Löschen (delete)

Löschen von “dieter”



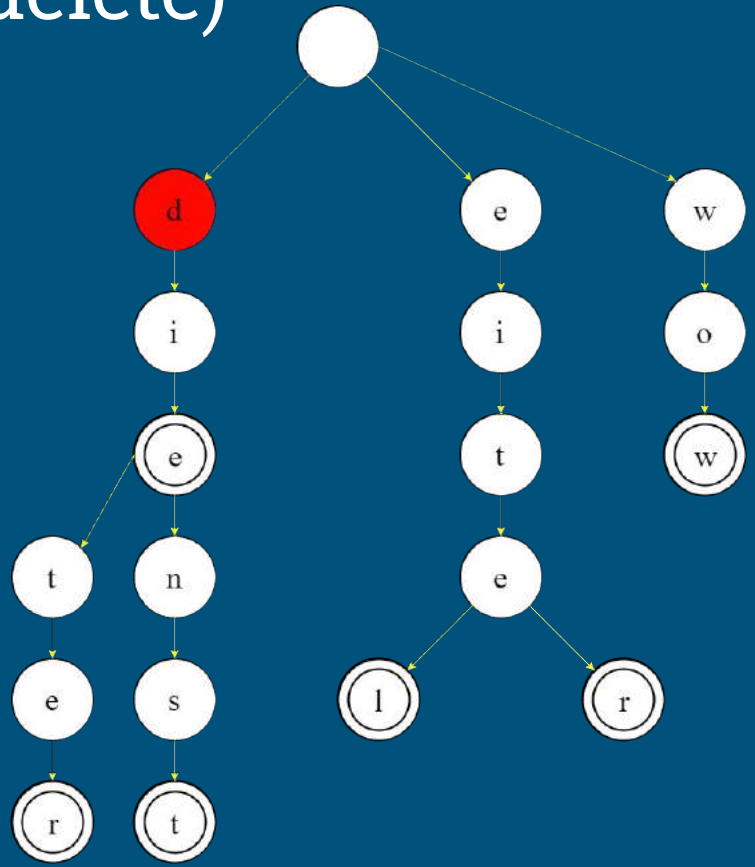
Trie Tree - Löschen (delete)

Löschen von “dieter”
Enthält Wurzel Kind ‘d’?



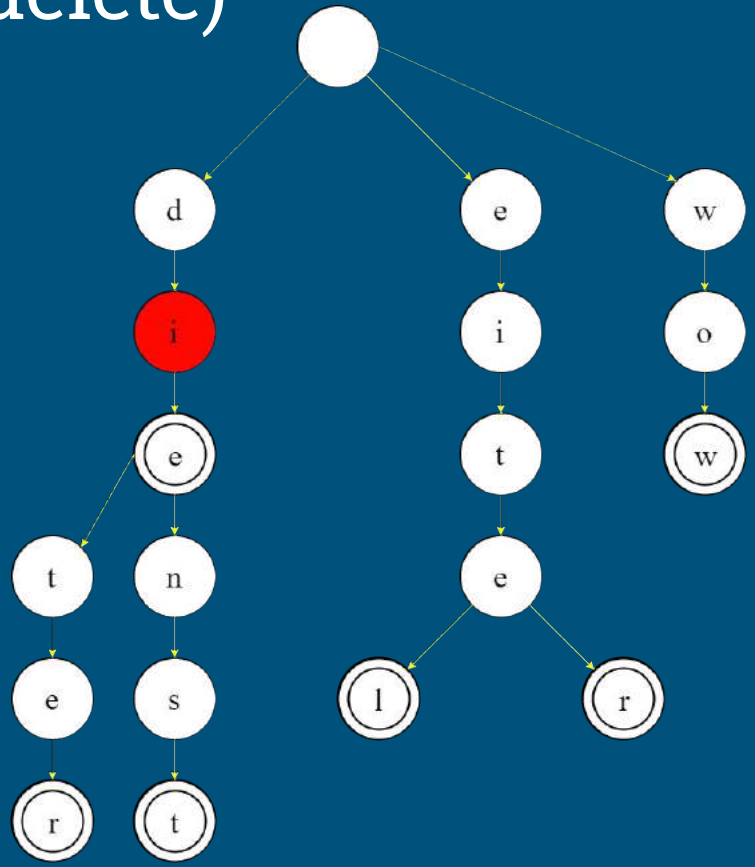
Trie Tree - Löschen (delete)

Löschen von “dieter”
Enthält Wurzel Kind ‘d’?
Enthält Knoten ‘d’ Kind ‘i’?



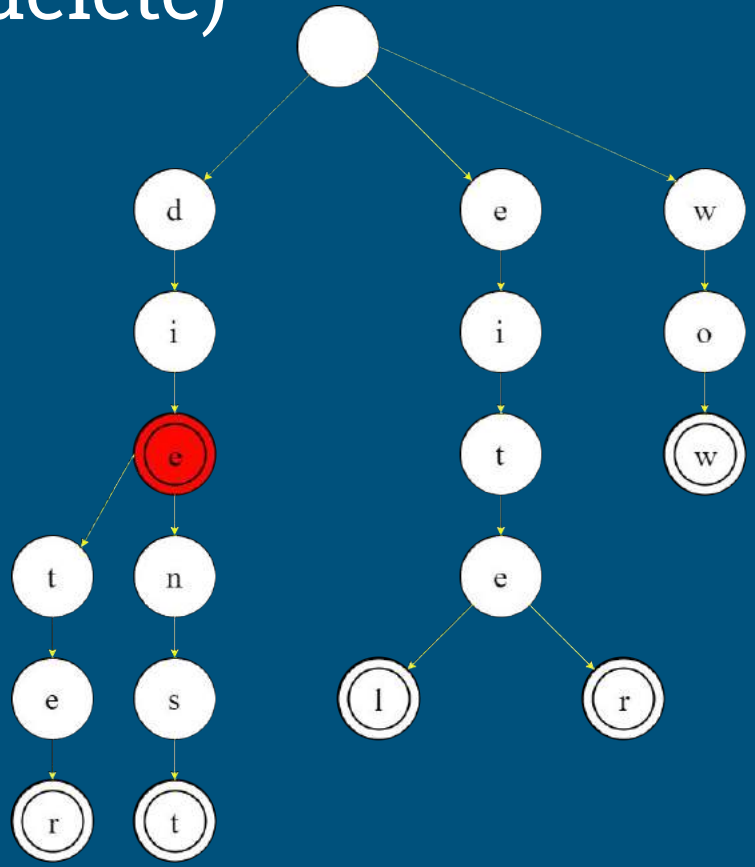
Trie Tree - Löschen (delete)

Löschen von “dieter”
Enthält Wurzel Kind ‘d’?
Enthält Knoten ‘d’ Kind ‘i’?
Enthält Knoten ‘i’ Kind ‘e’?



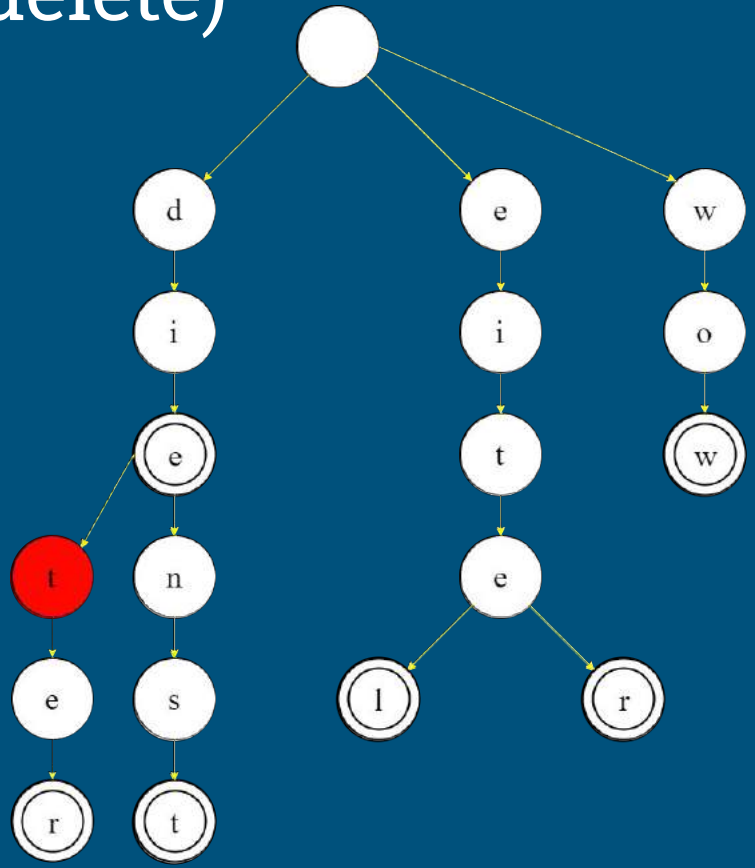
Trie Tree - Löschen (delete)

Löschen von “dieter”
Enthält Wurzel Kind ‘d’?
Enthält Knoten ‘d’ Kind ‘i’?
Enthält Knoten ‘i’ Kind ‘e’?
Enthält Knoten ‘e’ Kind ‘t’?



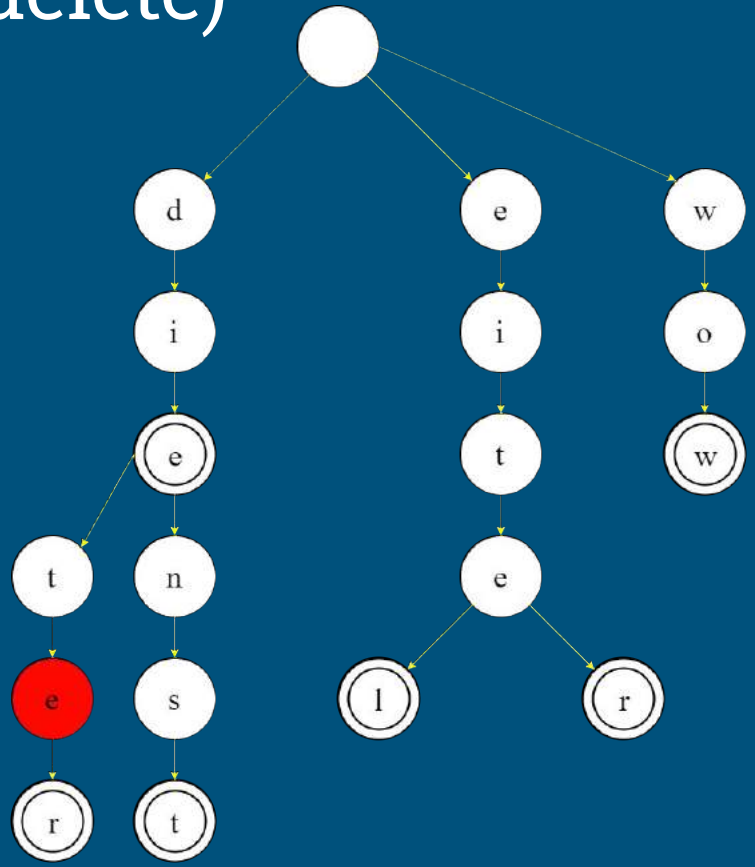
Trie Tree - Löschen (delete)

Löschen von “dieter”
Enthält Wurzel Kind ‘d’?
Enthält Knoten ‘d’ Kind ‘i’?
Enthält Knoten ‘i’ Kind ‘e’?
Enthält Knoten ‘e’ Kind ‘t’?
Enthält Knoten ‘t’ Kind ‘e’?



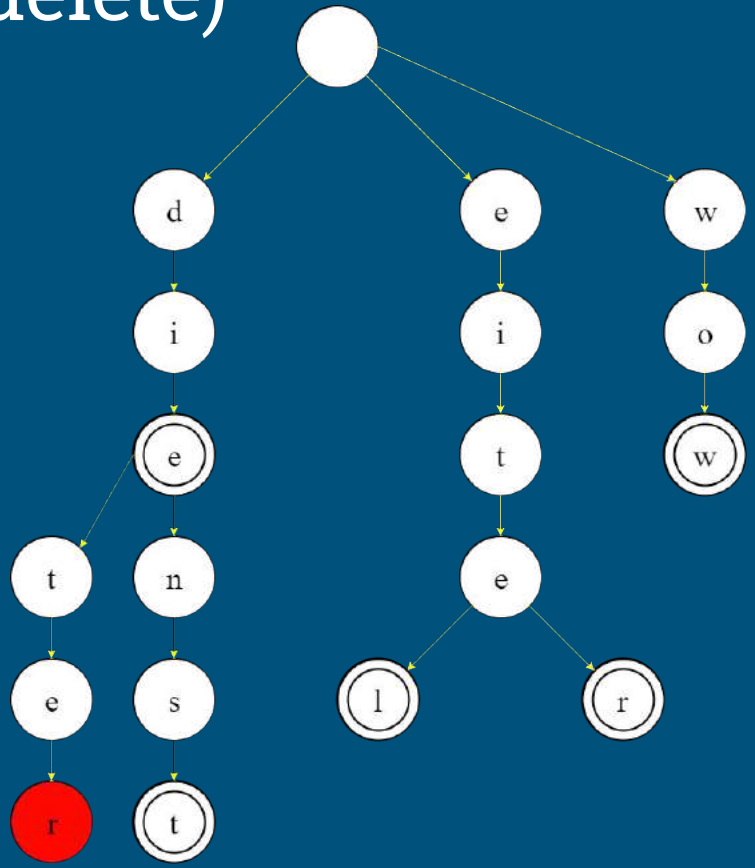
Trie Tree - Löschen (delete)

Löschen von “dieter”
Enthält Wurzel Kind ‘d’?
Enthält Knoten ‘d’ Kind ‘i’?
Enthält Knoten ‘i’ Kind ‘e’?
Enthält Knoten ‘e’ Kind ‘t’?
Enthält Knoten ‘t’ Kind ‘e’?
Enthält Knoten ‘e’ Kind ‘r’?



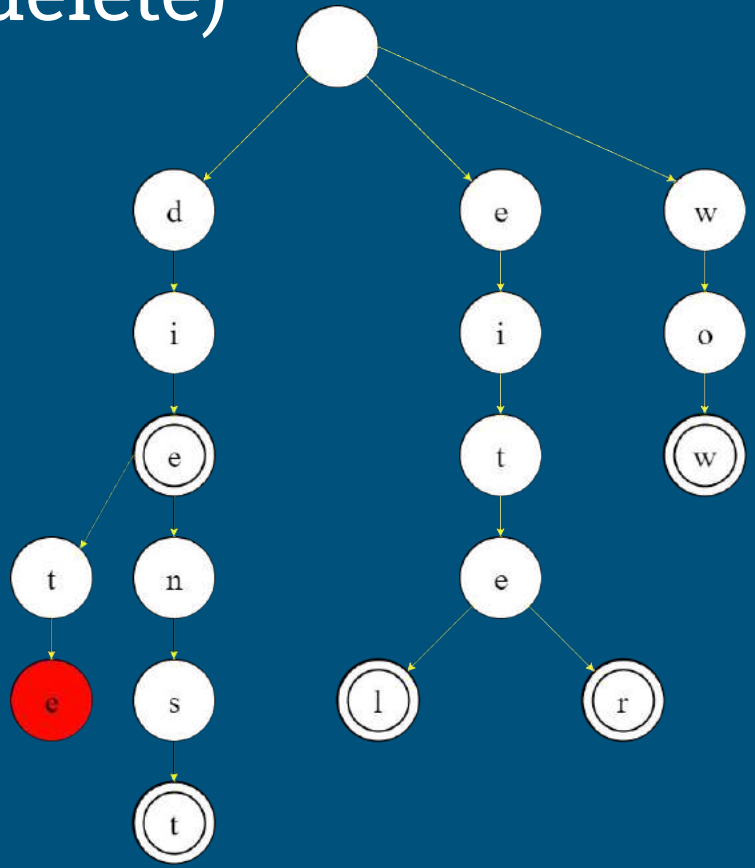
Trie Tree - Löschen (delete)

Löschen von “dieter”
Enthält Wurzel Kind ‘d’?
Enthält Knoten ‘d’ Kind ‘i’?
Enthält Knoten ‘i’ Kind ‘e’?
Enthält Knoten ‘e’ Kind ‘t’?
Enthält Knoten ‘t’ Kind ‘e’?
Enthält Knoten ‘e’ Kind ‘r’?
Entferne boolean
Hat ‘r’ Kinder?



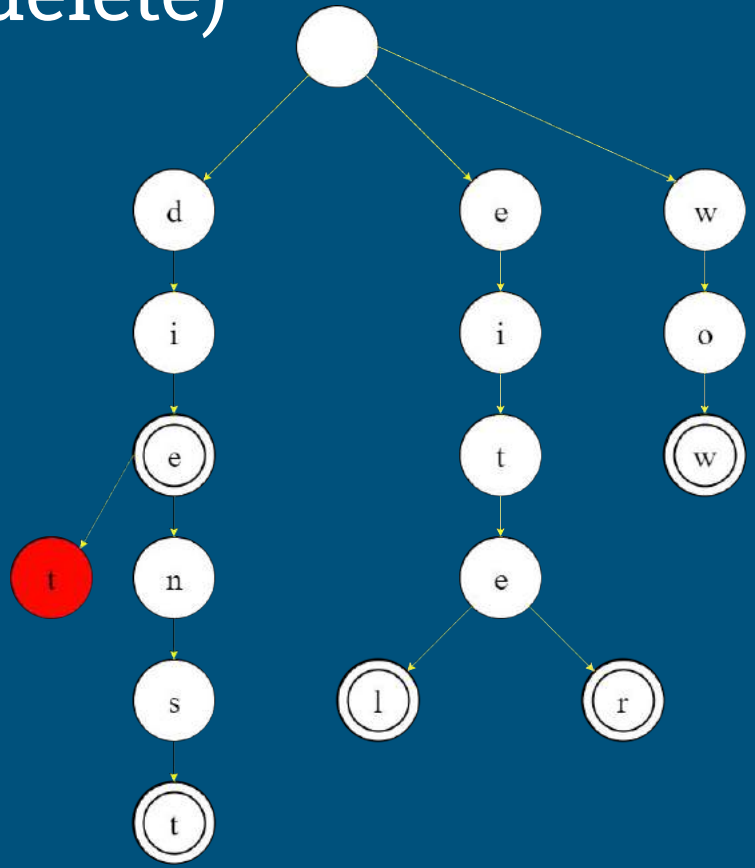
Trie Tree - Löschen (delete)

Löschen von “dieter”
Enthält Wurzel Kind ‘d’?
Enthält Knoten ‘d’ Kind ‘i’?
Enthält Knoten ‘i’ Kind ‘e’?
Enthält Knoten ‘e’ Kind ‘t’?
Enthält Knoten ‘t’ Kind ‘e’?
Enthält Knoten ‘e’ Kind ‘r’?
Entferne boolean
Hat ‘r’ Kinder?
Entferne ‘r’
‘e’ Ende eines Wortes, Wurzel oder Kinder?



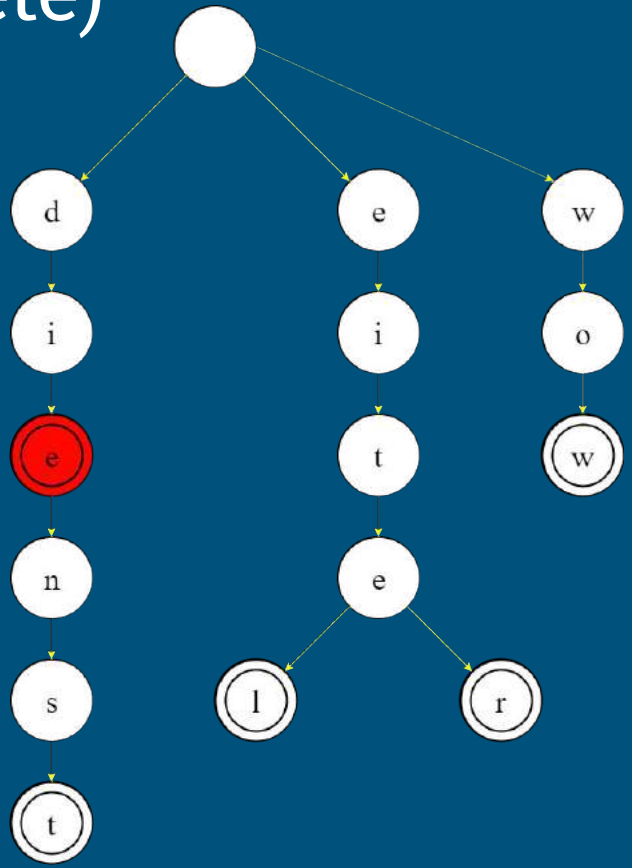
Trie Tree - Löschen (delete)

Löschen von “dieter”
Enthält Wurzel Kind ‘d’?
Enthält Knoten ‘d’ Kind ‘i’?
Enthält Knoten ‘i’ Kind ‘e’?
Enthält Knoten ‘e’ Kind ‘t’?
Enthält Knoten ‘t’ Kind ‘e’?
Enthält Knoten ‘e’ Kind ‘r’?
Entferne boolean
Hat ‘r’ Kinder?
Entferne ‘r’
‘e’ Ende eines Wortes, Wurzel oder Kinder?
Entferne ‘e’
‘t’ Ende eines Wortes, Wurzel oder Kinder?



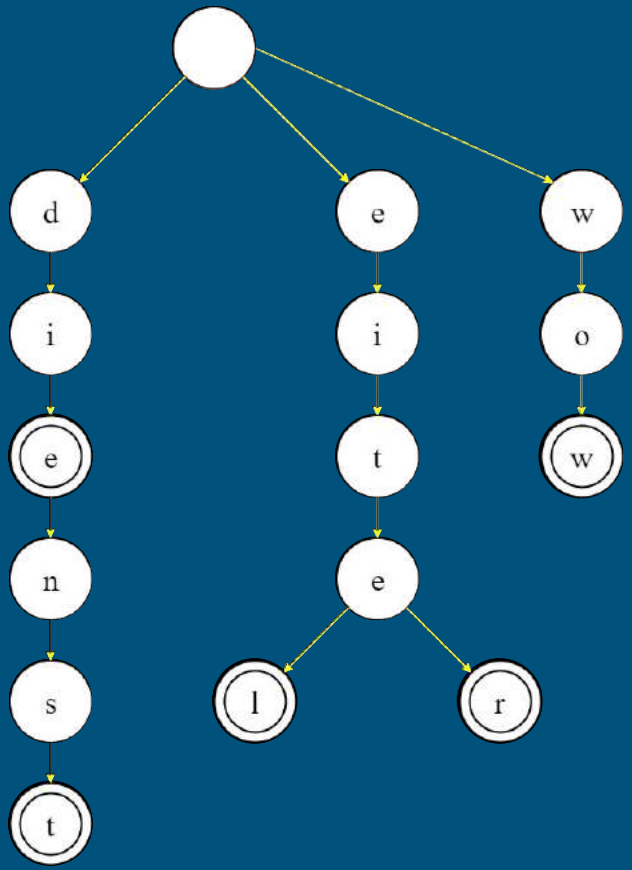
Trie Tree - Löschen (delete)

Löschen von “dieter”
Enthält Wurzel Kind ‘d’?
Enthält Knoten ‘d’ Kind ‘i’?
Enthält Knoten ‘i’ Kind ‘e’?
Enthält Knoten ‘e’ Kind ‘t’?
Enthält Knoten ‘t’ Kind ‘e’?
Enthält Knoten ‘e’ Kind ‘r’?
Entferne boolean
Hat ‘r’ Kinder?
Entferne ‘r’
‘e’ Ende eines Wortes, Wurzel oder Kinder?
Entferne ‘e’
‘t’ Ende eines Wortes, Wurzel oder Kinder?
Entferne ‘t’
‘e’ Ende eines Wortes, Wurzel oder Kinder?
Beenden



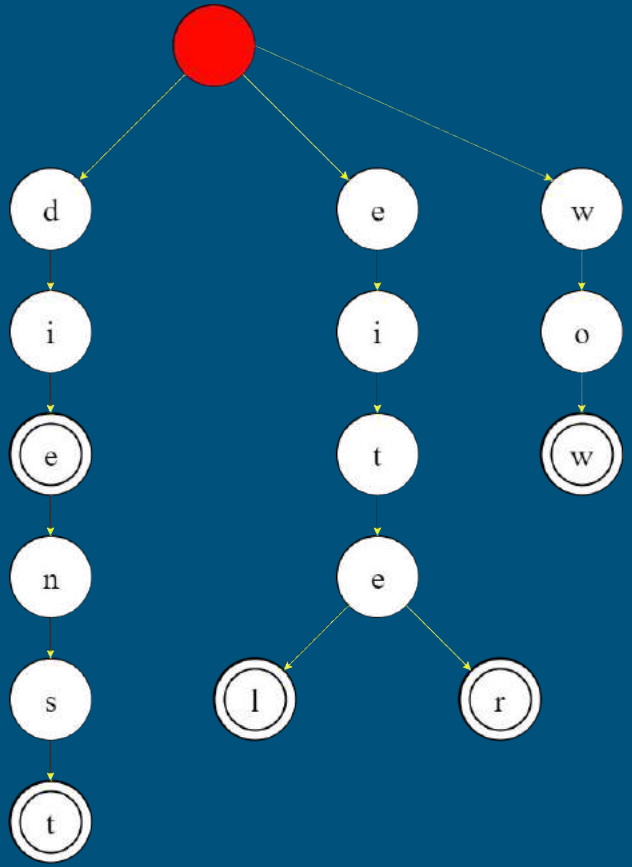
Trie Tree - Löschen (delete)

Löschen von “wow”



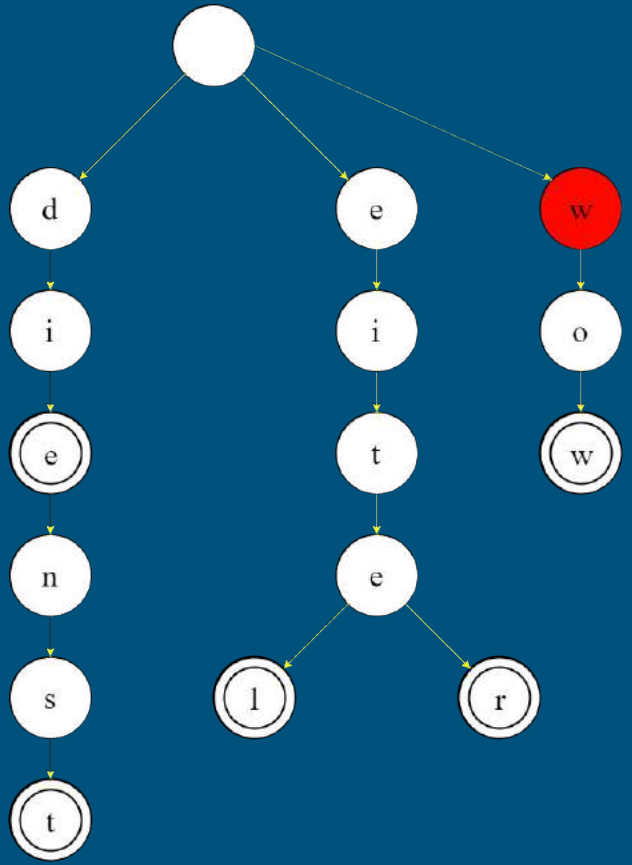
Trie Tree - Löschen (delete)

Löschen von “wow”
Enthält Wurzel Kind ‘w’?



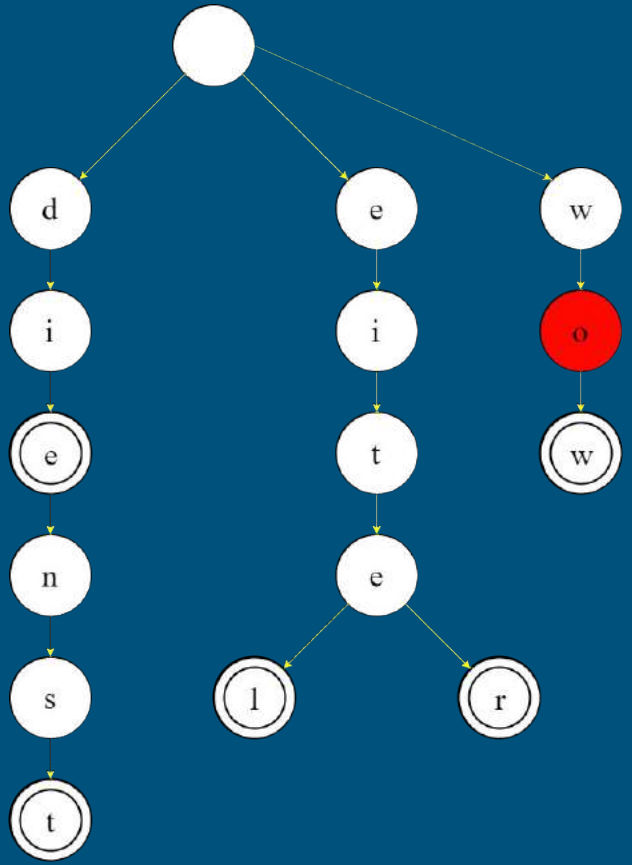
Trie Tree - Löschen (delete)

Löschen von “wow”
Enthält Wurzel Kind ‘w’?
Enthält Knoten ‘w’ Kind ‘o’?



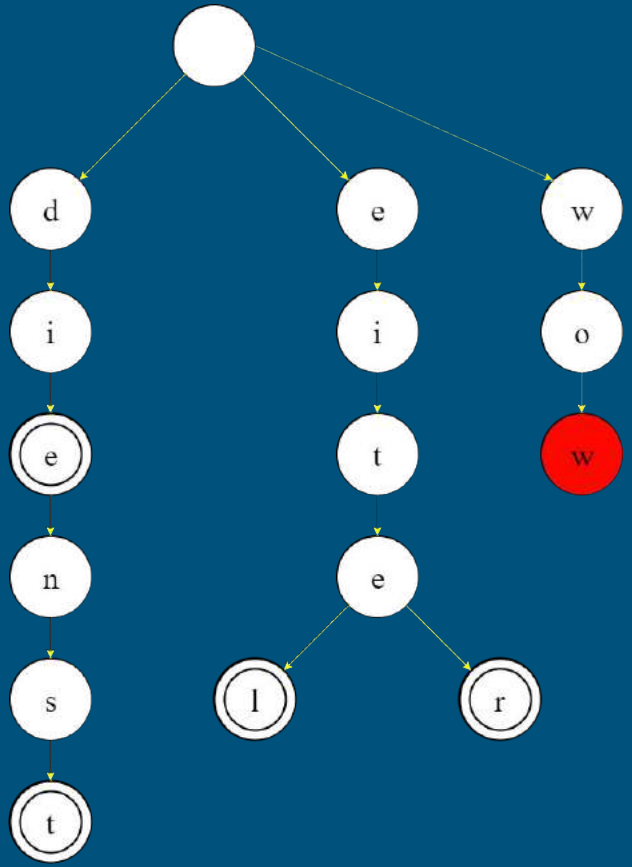
Trie Tree - Löschen (delete)

Löschen von “wow”
Enthält Wurzel Kind ‘w’?
Enthält Knoten ‘w’ Kind ‘o’?
Enthält Knoten ‘o’ Kind ‘w’?



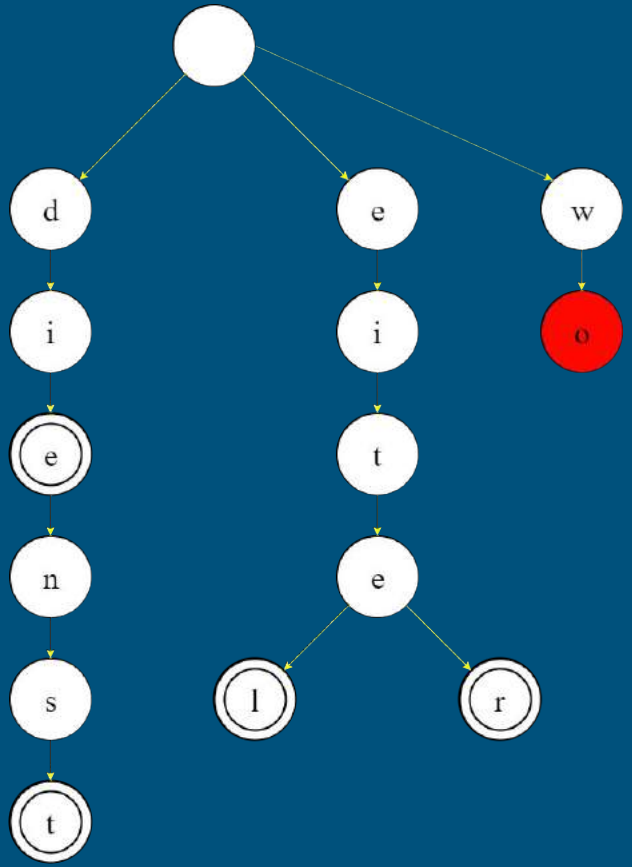
Trie Tree - Löschen (delete)

Löschen von “wow”
Enthält Wurzel Kind ‘w’?
Enthält Knoten ‘w’ Kind ‘o’?
Enthält Knoten ‘o’ Kind ‘w’?
Entferne boolean
Hat ‘w’ Kinder?



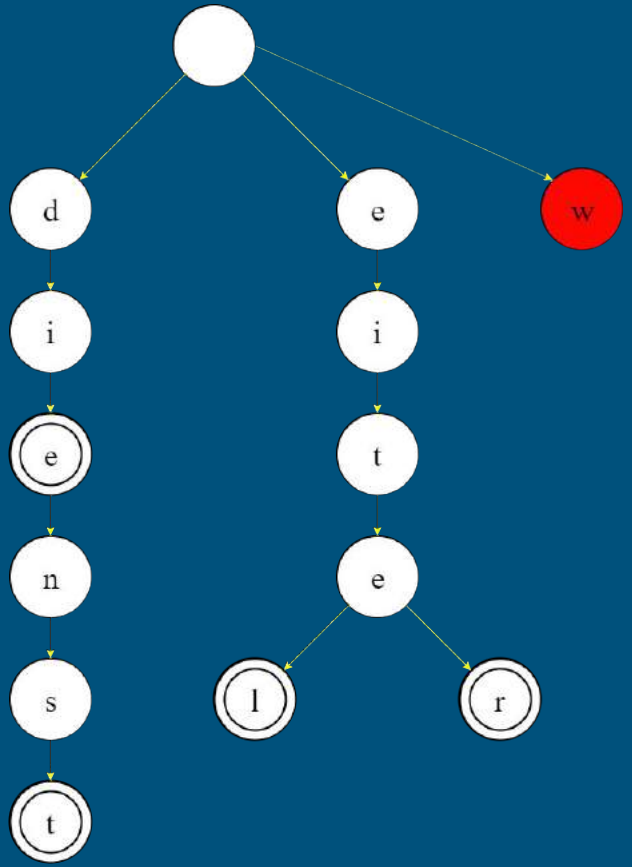
Trie Tree - Löschen (delete)

Löschen von “wow”
Enthält Wurzel Kind ‘w’?
Enthält Knoten ‘w’ Kind ‘o’?
Enthält Knoten ‘o’ Kind ‘w’?
Entferne boolean
Hat ‘w’ Kinder?
Entferne ‘w’
‘o’ Ende eines Wortes, Wurzel oder Kinder?



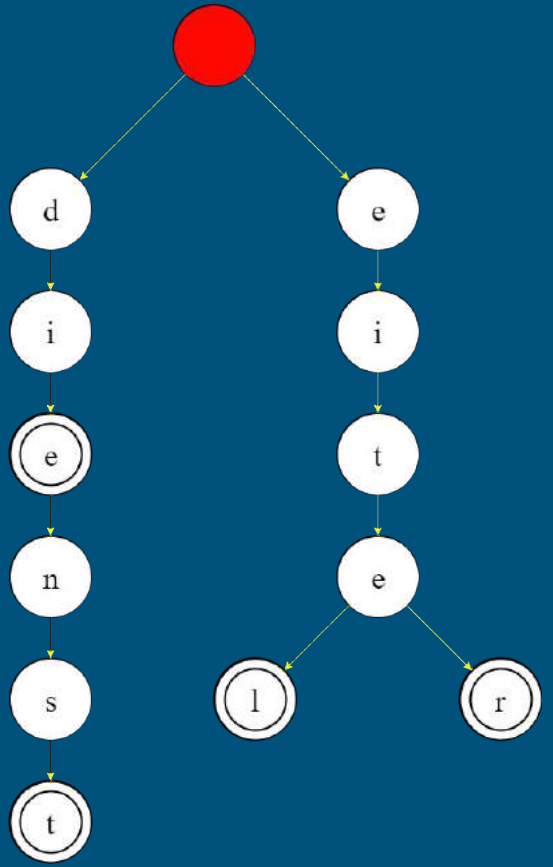
Trie Tree - Löschen (delete)

Löschen von “wow”
Enthält Wurzel Kind ‘w’?
Enthält Knoten ‘w’ Kind ‘o’?
Enthält Knoten ‘o’ Kind ‘w’?
Entferne boolean
Hat ‘w’ Kinder?
Entferne ‘w’
‘o’ Ende eines Wortes, Wurzel oder Kinder?
Entferne ‘o’
‘w’ Ende eines Wortes, Wurzel oder Kinder?



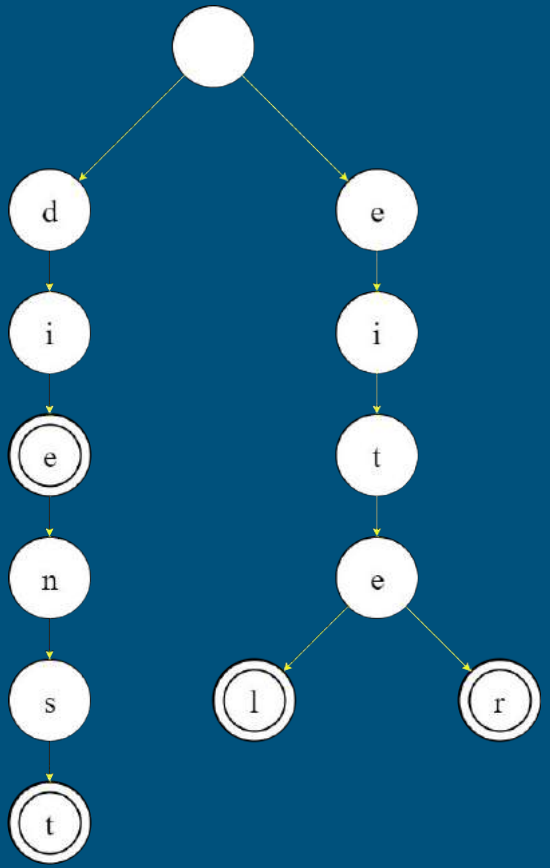
Trie Tree - Löschen (delete)

Löschen von “wow”
Enthält Wurzel Kind ‘w’?
Enthält Knoten ‘w’ Kind ‘o’?
Enthält Knoten ‘o’ Kind ‘w’?
Entferne boolean
Hat ‘w’ Kinder?
Entferne ‘w’
‘o’ Ende eines Wortes, Wurzel oder Kinder?
Entferne ‘o’
‘w’ Ende eines Wortes, Wurzel oder Kinder?
Entferne ‘w’
‘Wurzel’ Ende eines Wortes, Wurzel oder Kinder?
Beenden



Trie Tree - Finden (find)

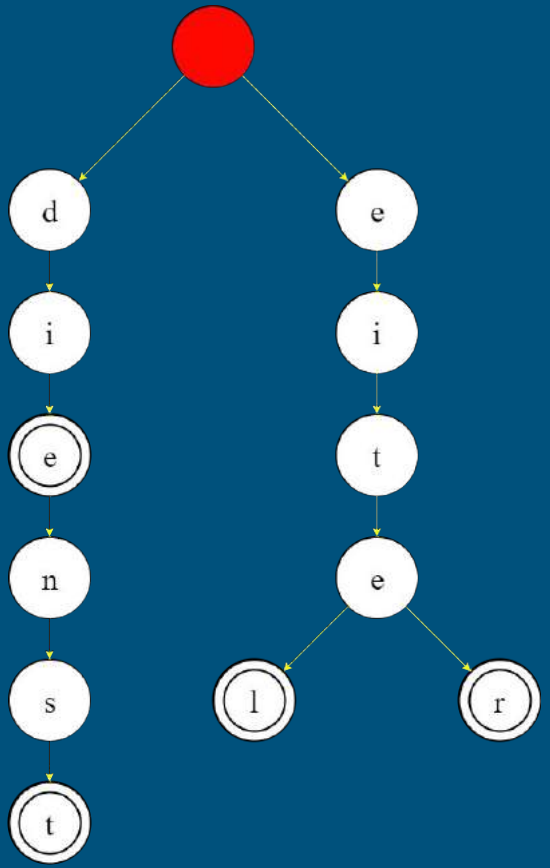
Finden von “die”



Trie Tree - Finden (find)

Finden von “die”

Enthält Wurzel Kind ‘d’?

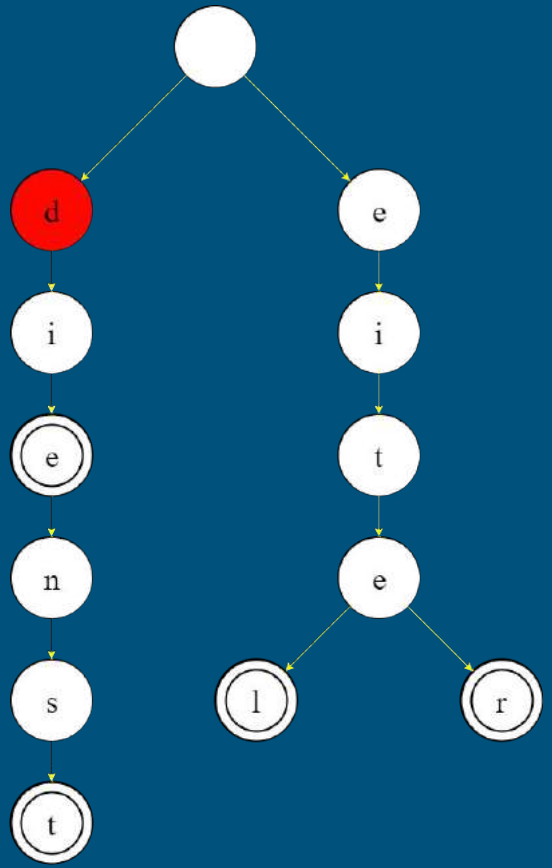


Trie Tree - Finden (find)

Finden von “die”

Enthält Wurzel Kind ‘d’?

Enthält Knoten ‘d’ Kind ‘i’?



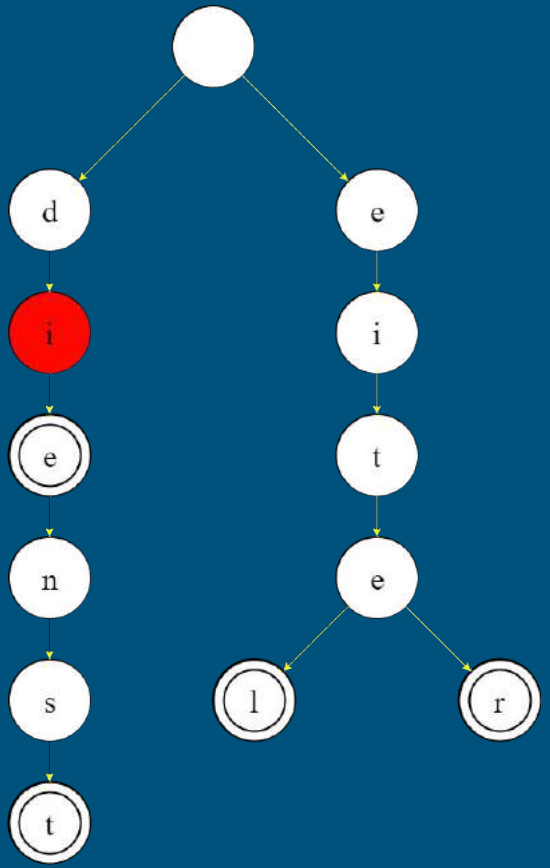
Trie Tree - Finden (find)

Finden von “die”

Enthält Wurzel Kind ‘d’?

Enthält Knoten ‘d’ Kind ‘i’?

Enthält Knoten ‘i’ Kind ‘e’?



Trie Tree - Finden (find)

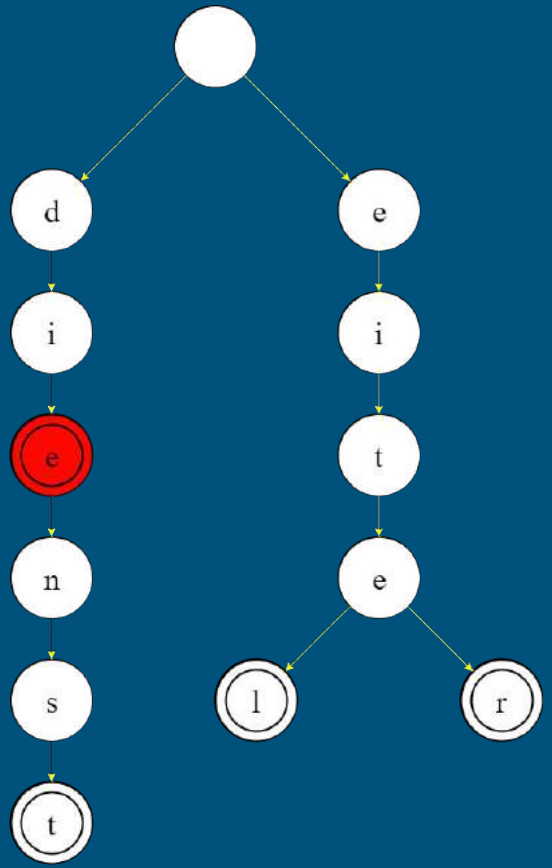
Finden von “die”

Enthält Wurzel Kind ‘d’?

Enthält Knoten ‘d’ Kind ‘i’?

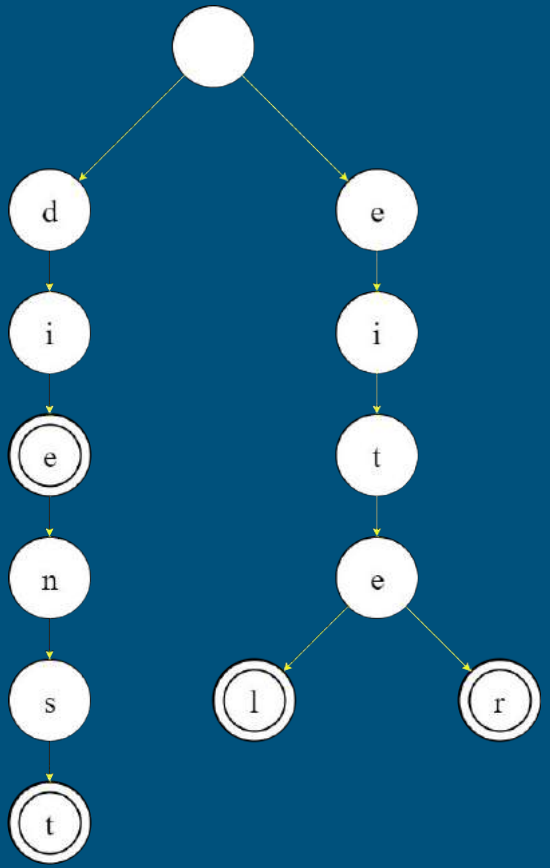
Enthält Knoten ‘i’ Kind ‘e’?

return boolean



Trie Tree - Finden (find)

Finden von “wow”

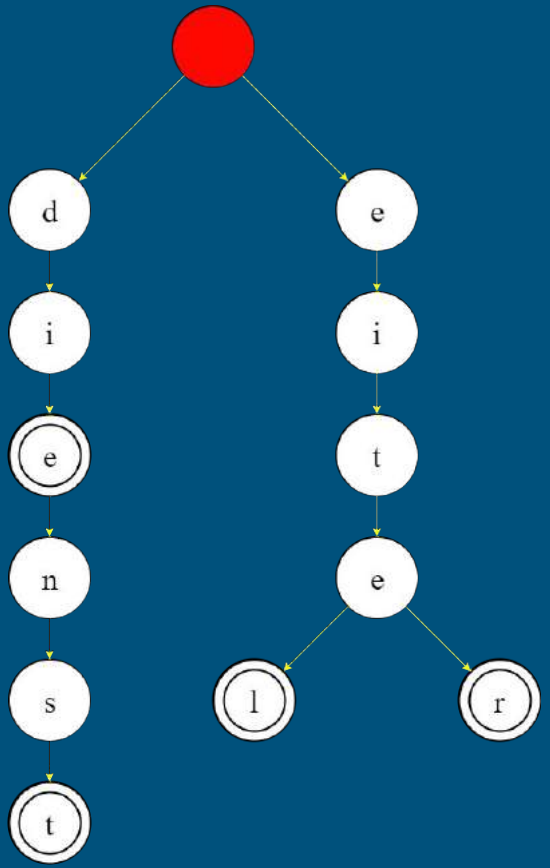


Trie Tree - Finden (find)

Finden von “wow”

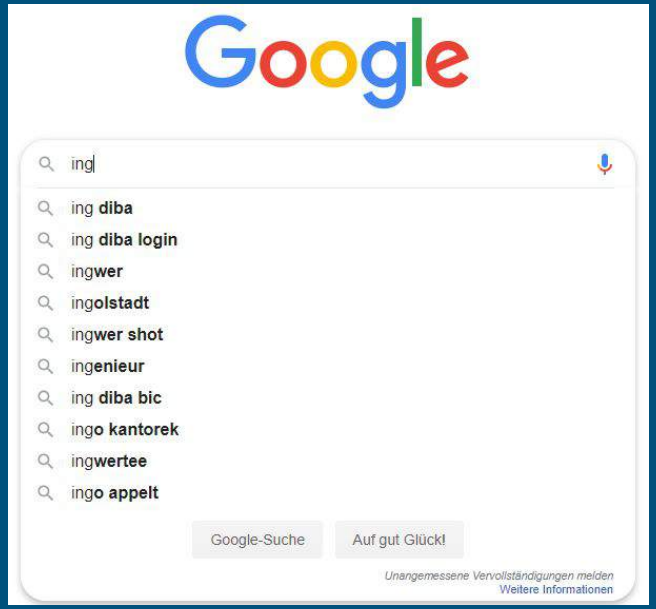
Enthält Wurzel Kind ‘w’?

return false



Trie Tree - Praktische Anwendungen

Autovervollständigung durch Präfixsuche



Traversieren bis zum Ende des Präfixes

Falls Ende eines Strings in Liste einfügen

Rekursiv Kinder durchlaufen und bei aktiviertem boolean in Liste einfügen

Trie Tree - Praktische Anwendungen

Zusätzliche Informationen (z.B. Duden)

- Statt boolean am Ende eines Wortes weitere Informationen
- Beispiel Duden:
  - Bedeutungen
  - Synonyme
  - Herkunft
  - etc.

Trie Tree - Praktische Anwendungen

Übersetzung

- string statt boolean am Ende eines Wortes
- strings beinhalten die Übersetzung
- Beispiel: Trie.find(“wort”) => “word”

Trie Tree - Optimierung: Patricia-Trie

Practical Algorithm to Retrieve Information Coded in Alphanumeric

1968 von Donald R. Morrison veröffentlicht

Knoten speichern keine character sondern strings

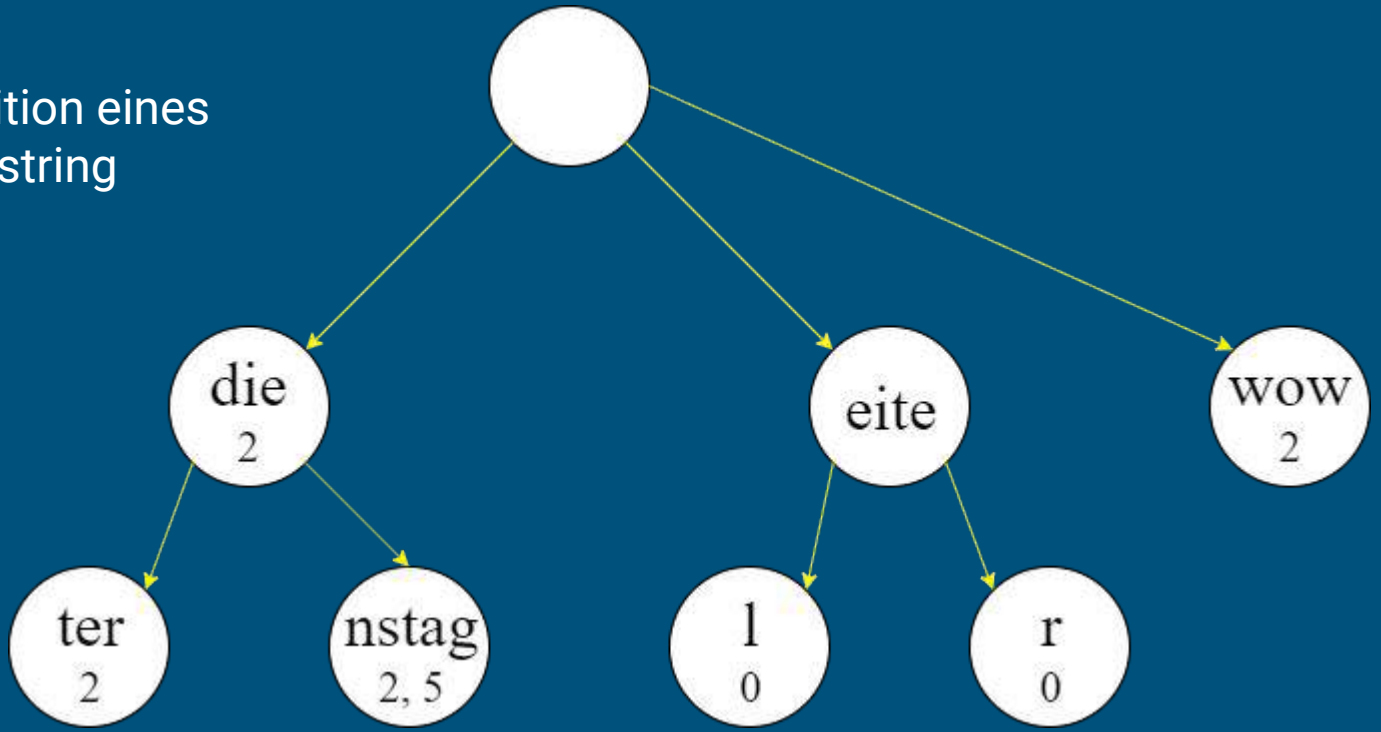
Knoten ohne Verzweigungen werden zusammengefasst

Stärke: Schnellere Suche und geringerer Platzbedarf

Schwäche: Komplexeres Einfügen und Löschen

Trie Tree - Optimierung: Patricia-Trie

Speichern der Position eines Wortendes im Substring



Trie Tree - Implementierungen

Laufzeit der Wörterbuchoperationen und Platzbedarf stark von Implementierung der Speicherung ausgehender Kanten abhängig:

Implementation	Laufzeit	Platzbedarf
Einfach verkettete Liste	O(k * a)	O(n)
Sortiertes Array	O(k * log a)	O(n)
Array in Alphabetsgröße	O(k)	O(n * a)
Hashtabelle	amortisiert O(k)	O(n)

k: Länge des Strings

n: Gesamtlänge aller Strings

a: Größe des Alphabets

C# - Dictionary - Allgemeines

namespace: System.Collections.Generic

Dictionary<TKey, TValue> -> Beliebiger Datentyp als Schlüssel und Wert

Als Hash-Tabelle implementiert

C# - Dictionary - Wörterbuchoperationen

Einfügen:

- Add(TKey, TValue): Fügt neues Schlüssel-Wert-Paar hinzu (Exception, falls Schlüssel bereits existiert)
- dictionary[TKey] = TValue: Fügt neues Schlüssel-Wert-Paar hinzu (Überschreibt Wert falls Schlüssel schon vorhanden)

Löschen:

- Remove(TKey): Entfernt Schlüssel-Wert-Paar

C# - Dictionary - Wörterbuchoperationen

Finden:

- Dictionary[TKey]:
  Liefert Wert zum Schlüssel (Exception, falls Schlüssel nicht im Dictionary)

- TryGetValue(TKey, out TValue):
  Speichert den Wert zum Schlüssel in TValue (Falls Schlüssel nicht im Dictionary: TValue == Standardwert des Typs (z.B. string -> null))
  Gibt bool zurück, ob Schlüssel enthalten ist

C# - Dictionary - Zusätzliche Operationen

- Clear(): Entfernt jedes Schlüssel-Wert-Paar
- ContainsKey(TKey): Überprüft, ob ein Schlüssel enthalten ist
- ContainsValue(TValue): Überprüft, ob ein Wert enthalten ist
- GetEnumerator(): Liefert einen Enumerator, der das Dictionary durchläuft

C# - Dictionary - Hashing

Speichern der Daten in einem Array

Indexposition im Array wird durch Hashfunktion berechnet

Vergleiche bei Suchoperationen obsolet

ideale Hashtabelle: Jeder Schlüssel erzeugt anderen Index (unrealistisch, da Array extrem groß sein muss)

C# - Dictionary - realistische Hashtabelle

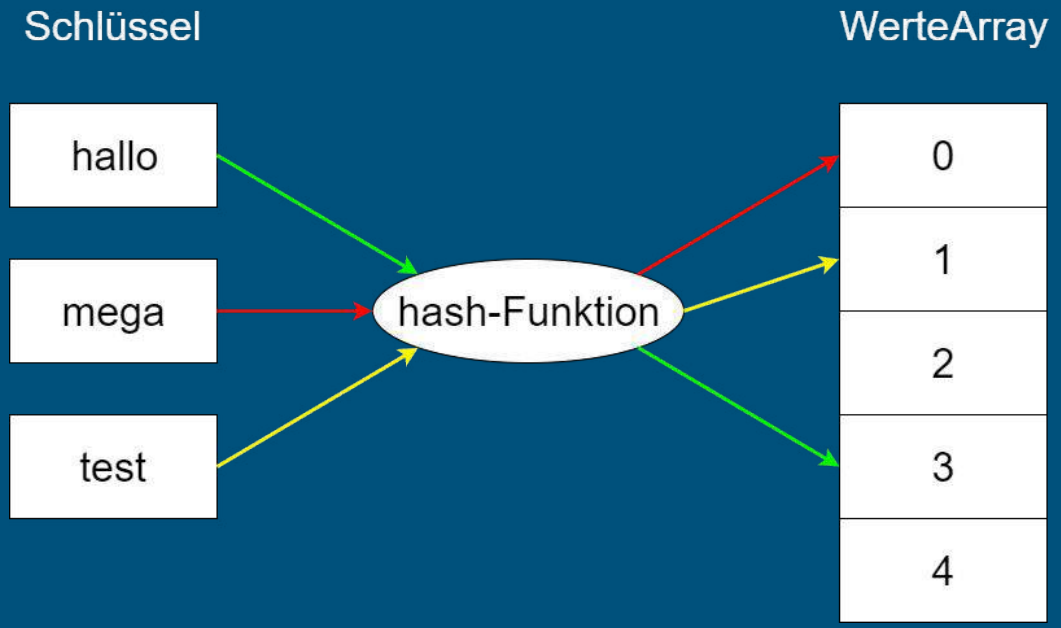
Mehrere Schlüssel erzeugen gleichen Index -> Kollisionen treten auf

Lösung -> Array speichert verkettete Listen mit eingegebenen Werten

Zu volle Liste -> Erweiterung des Arrays, Anpassung der Hashfunktion und rehashing

C# - Dictionary - Einfügen

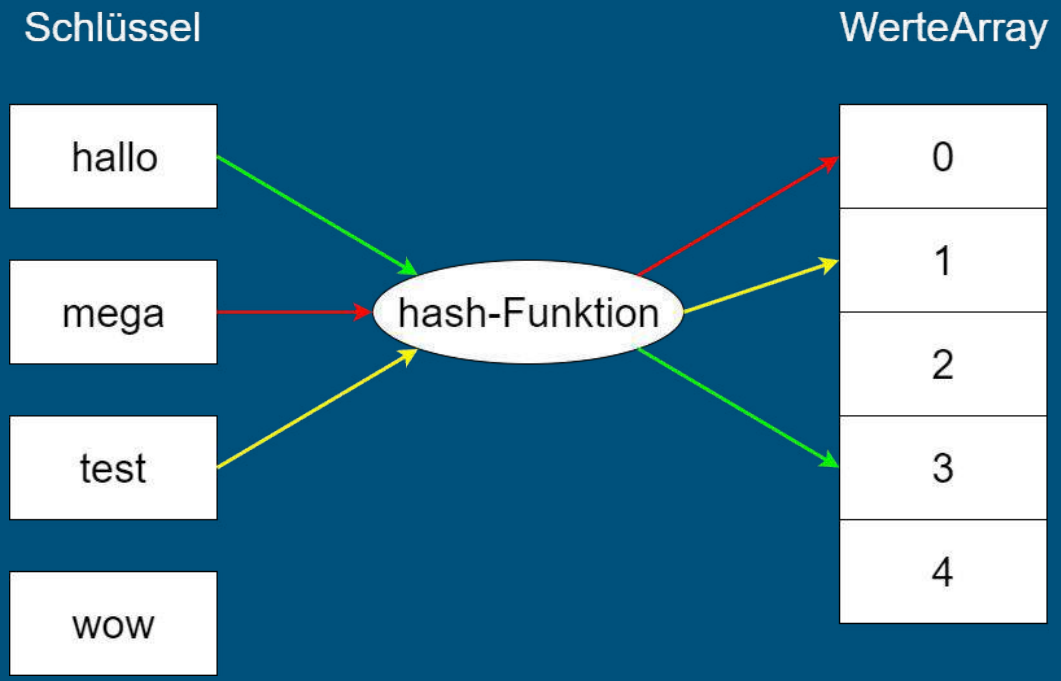
Einfügen von “wow”



C# - Dictionary - Einfügen

Einfügen von “wow”

Schlüssel “wow” hinzufügen

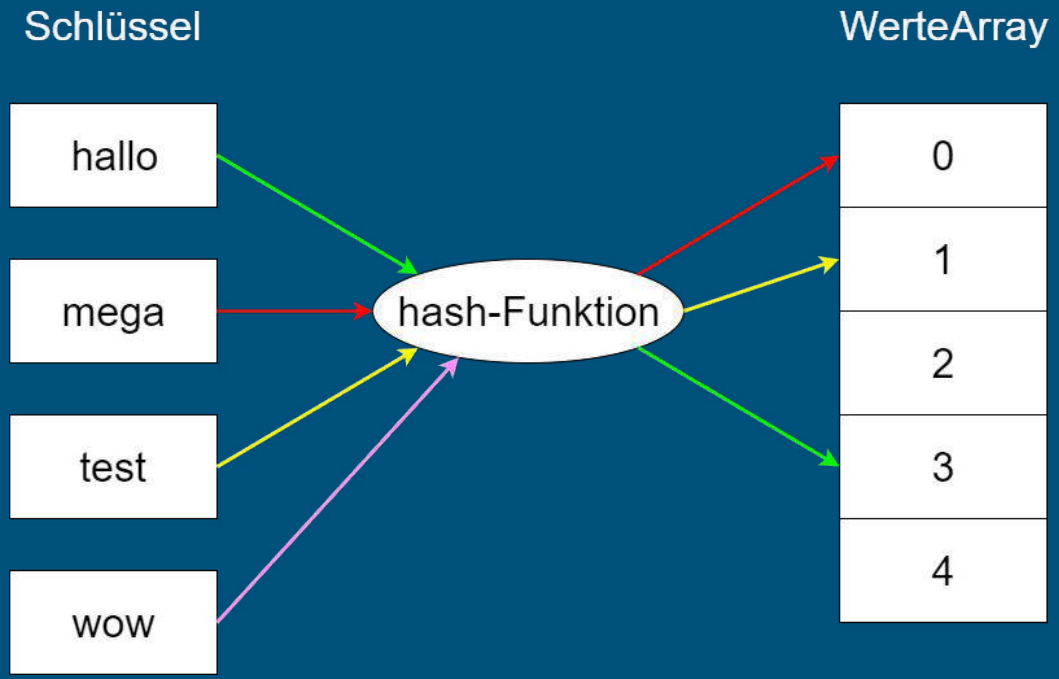


C# - Dictionary - Einfügen

Einfügen von “wow”

Schlüssel “wow” hinzufügen

“wow” -> hash-Funktion



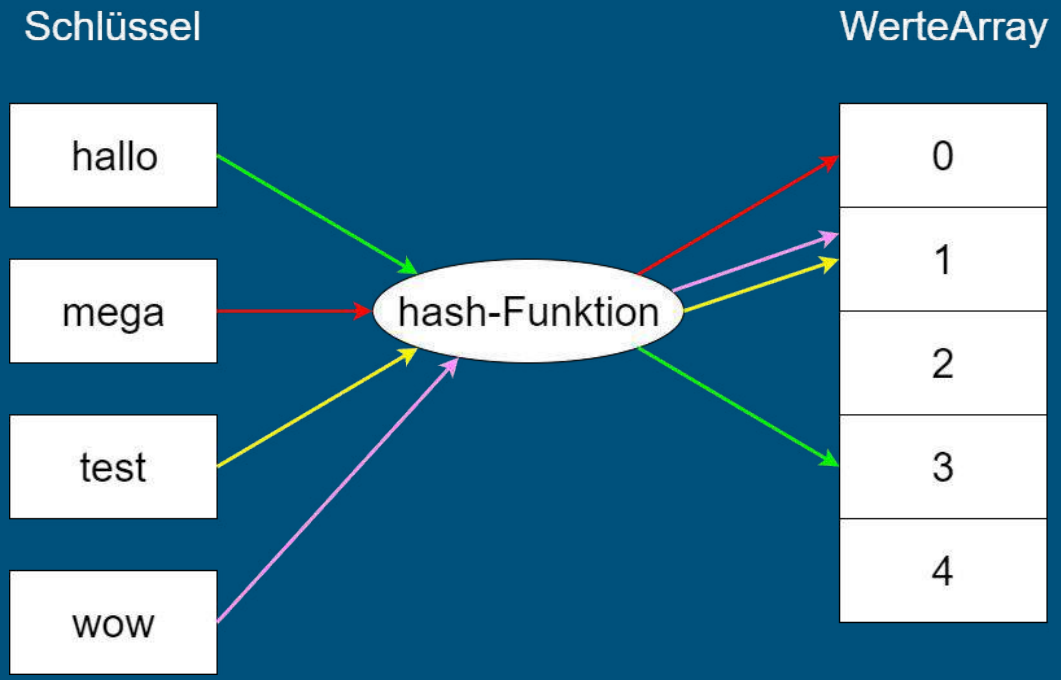
C# - Dictionary - Einfügen

Einfügen von “wow”

Schlüssel “wow” hinzufügen

“wow” -> hash-Funktion

Wert zu “wow” an ermittelte Liste anhängen



C# - Dictionary - Einfügen

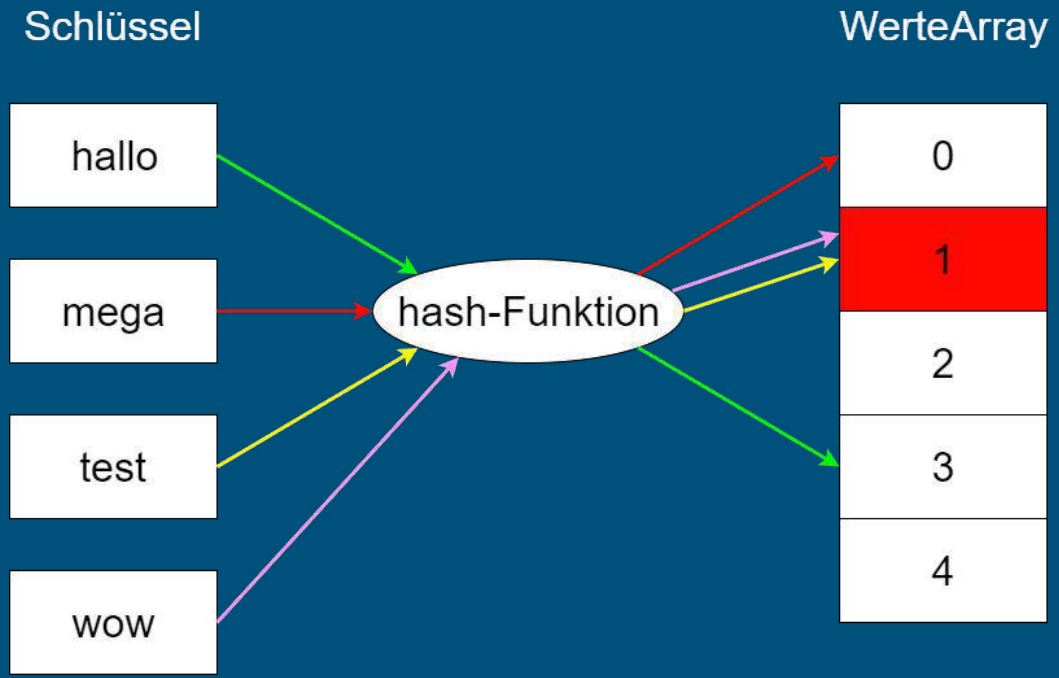
Einfügen von “wow”

Schlüssel “wow” hinzufügen

“wow” -> hash-Funktion

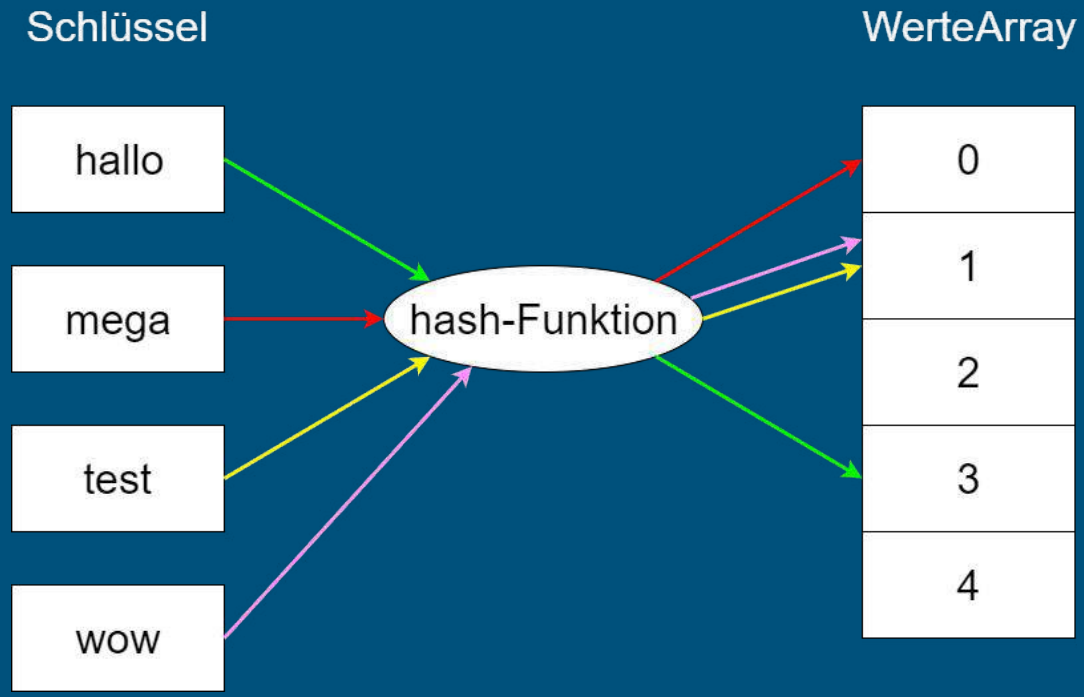
Wert zu “wow” an ermittelte Liste anhängen

Prüfen, ob rehashing notwendig ist (z.B. Liste zu lang)



C# - Dictionary - Löschen

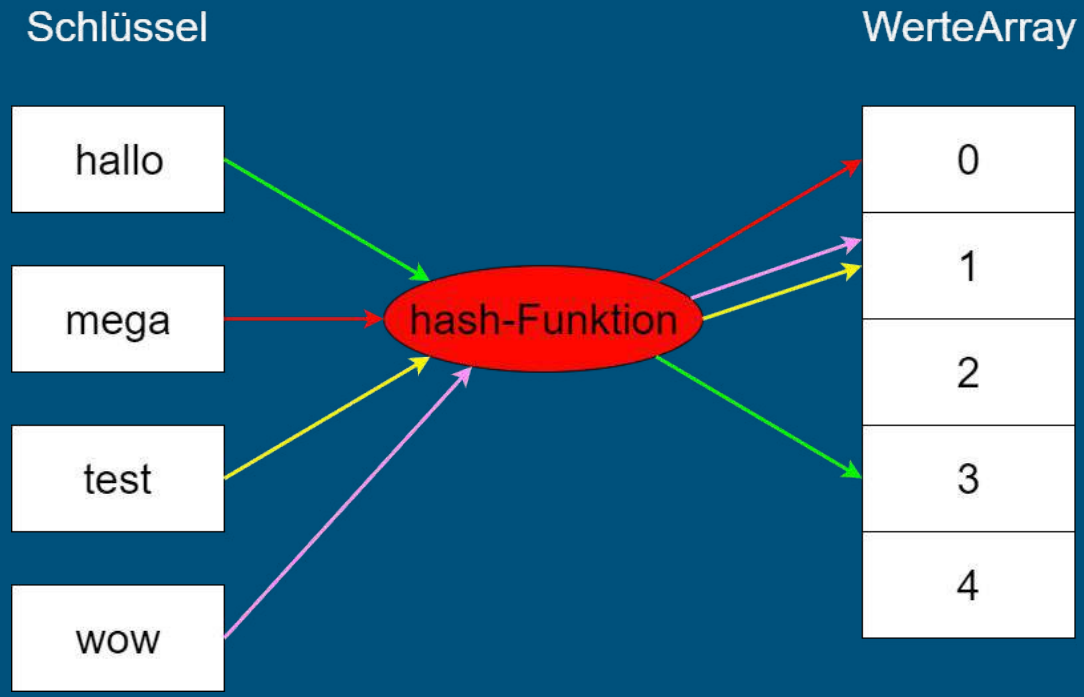
Löschen von “wow”



C# - Dictionary - Löschen

Löschen von “wow”

“wow” -> hash-Funktion

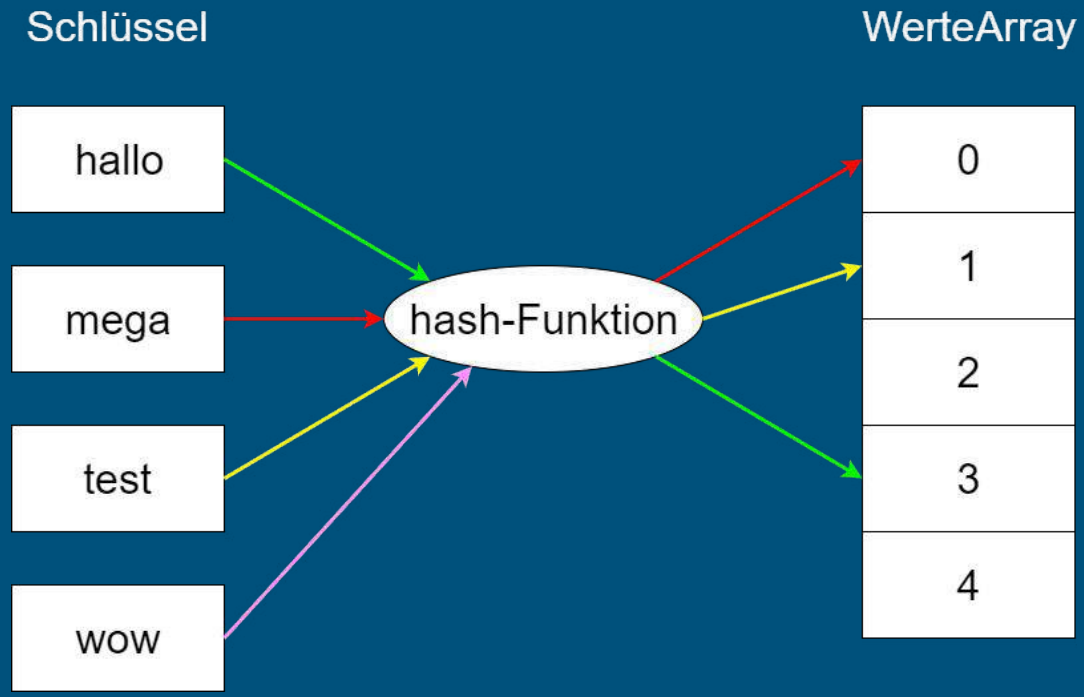


C# - Dictionary - Löschen

Löschen von “wow”

“wow” -> hash-Funktion

“wow” aus ermittelter Liste löschen



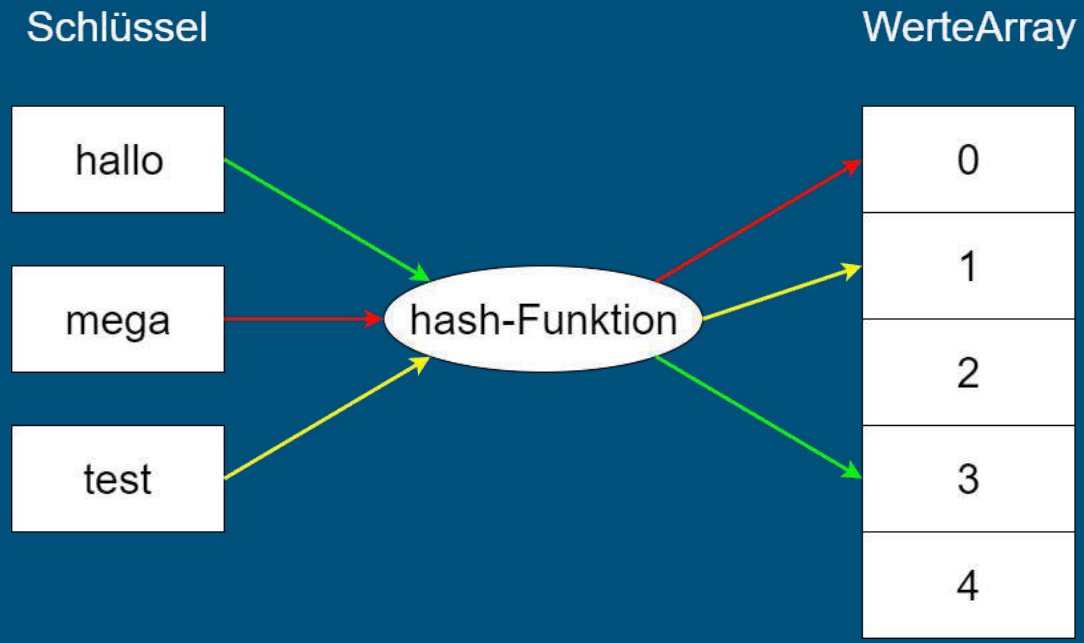
C# - Dictionary - Löschen

Löschen von “wow”

“wow” -> hash-Funktion

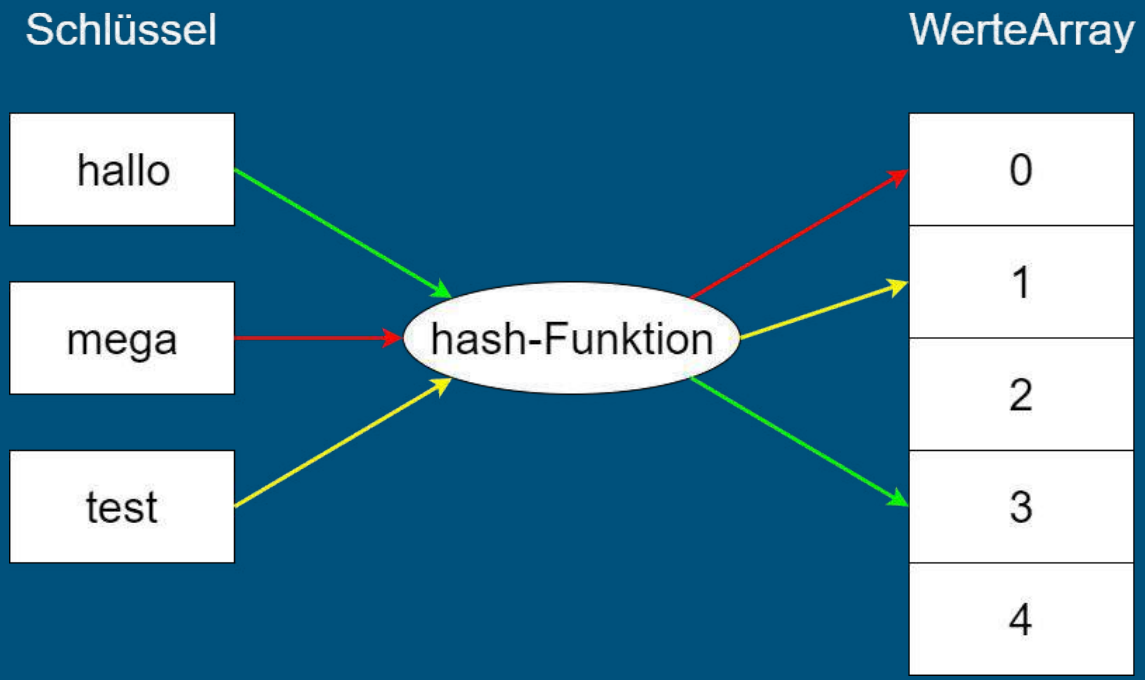
“wow” aus ermittelter Liste löschen

“wow” aus Schlüsseln löschen



C# - Dictionary - Finden

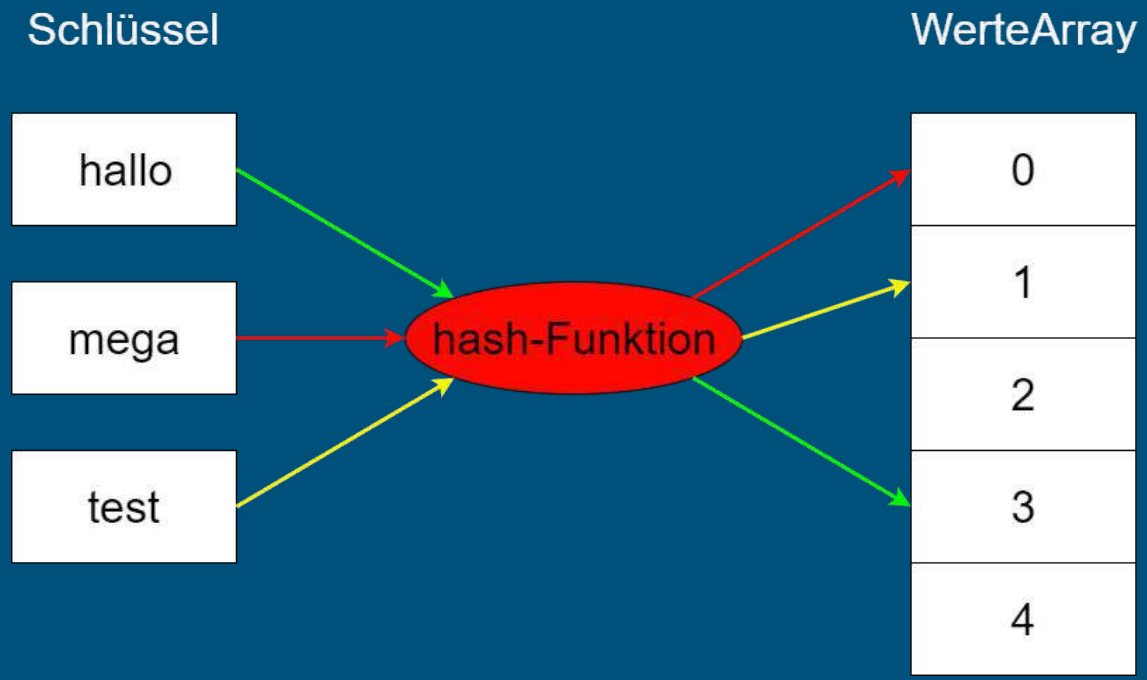
Finden von “mega”



C# - Dictionary - Finden

Finden von “mega”

“mega” -> hash-Funktion



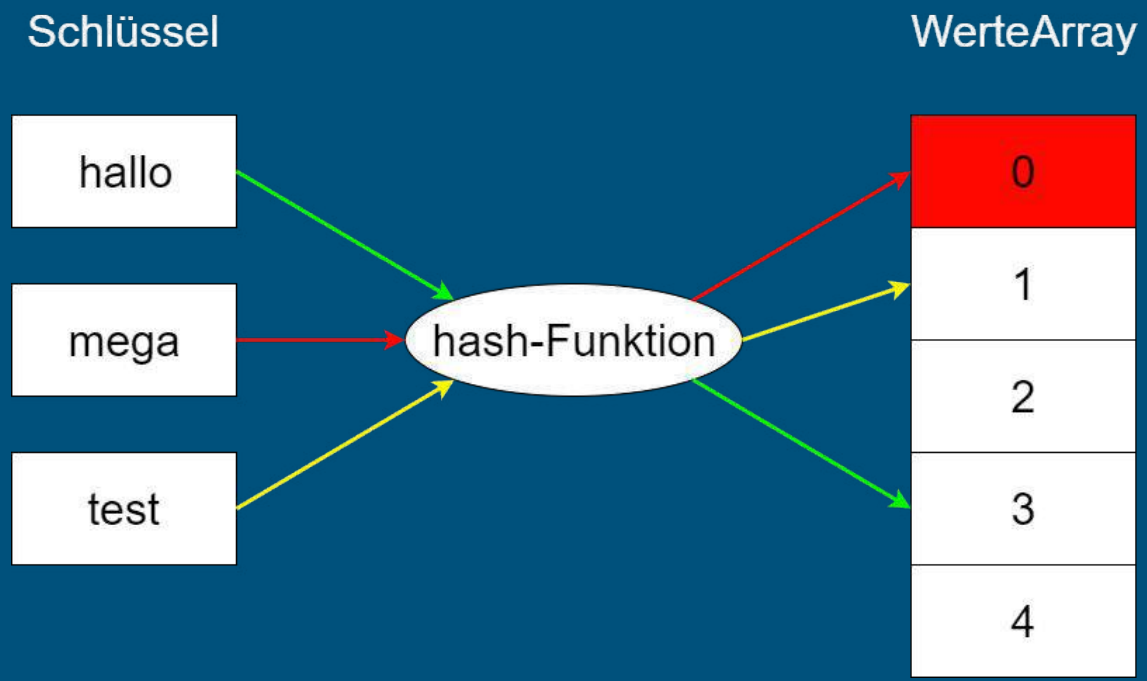
C# - Dictionary - Finden

Finden von “mega”

“mega” -> hash-Funktion

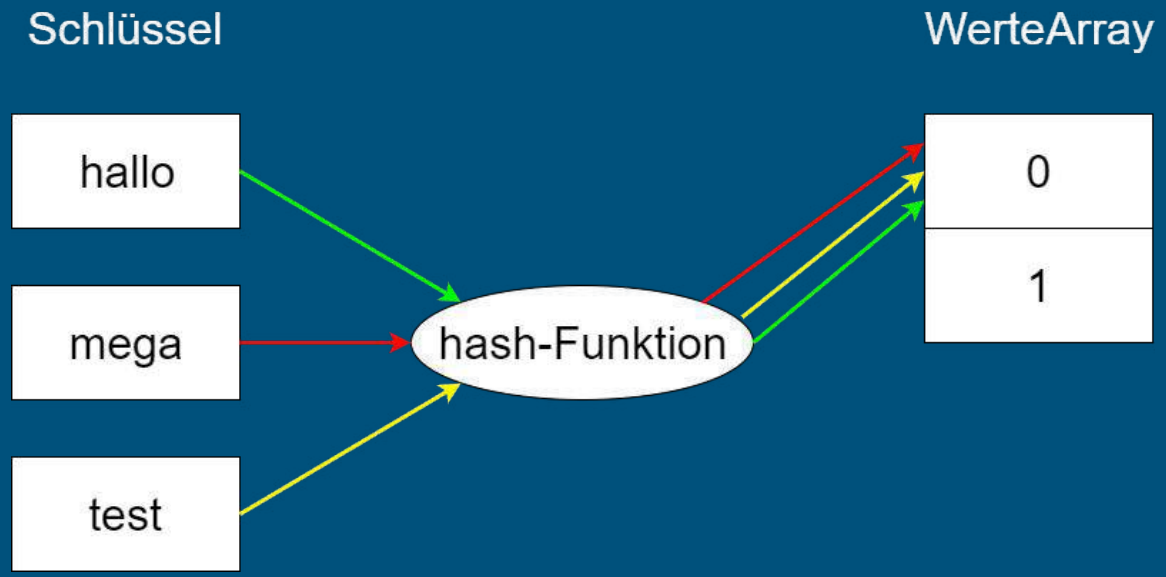
“mega” in ermittelter Liste suchen

Wert zu “mega” auslesen



C# - Dictionary - Rehashing

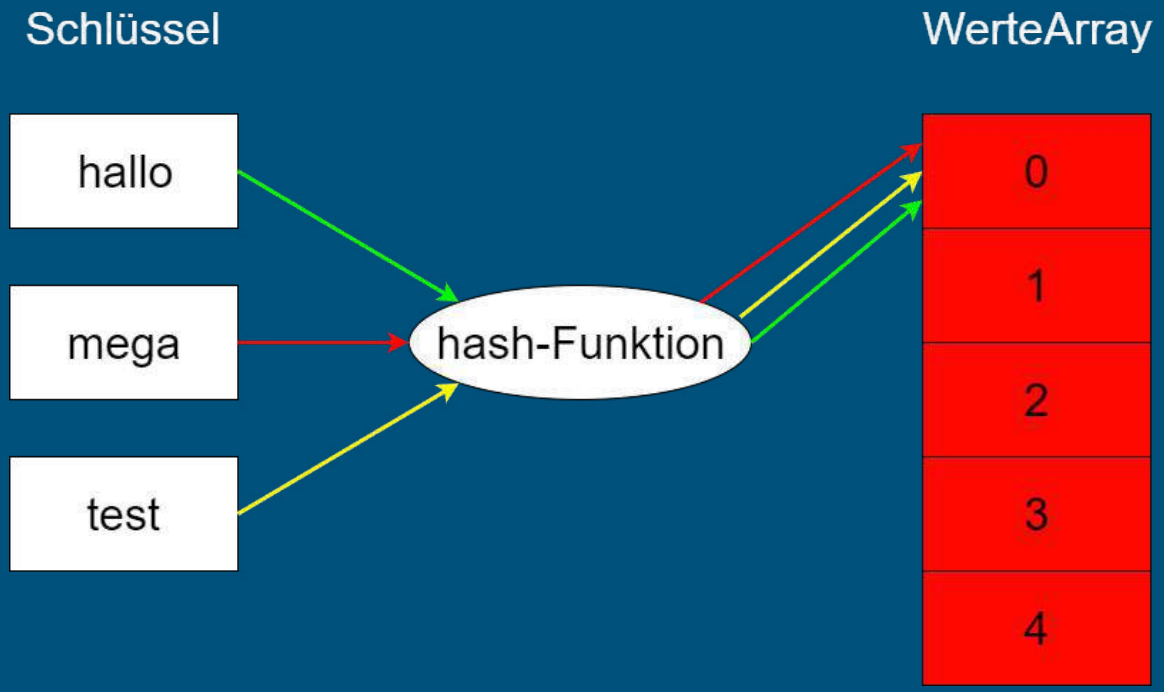
Liste 0 zu voll



C# - Dictionary - Rehashing

Liste 0 zu voll

WerteArray erweitern

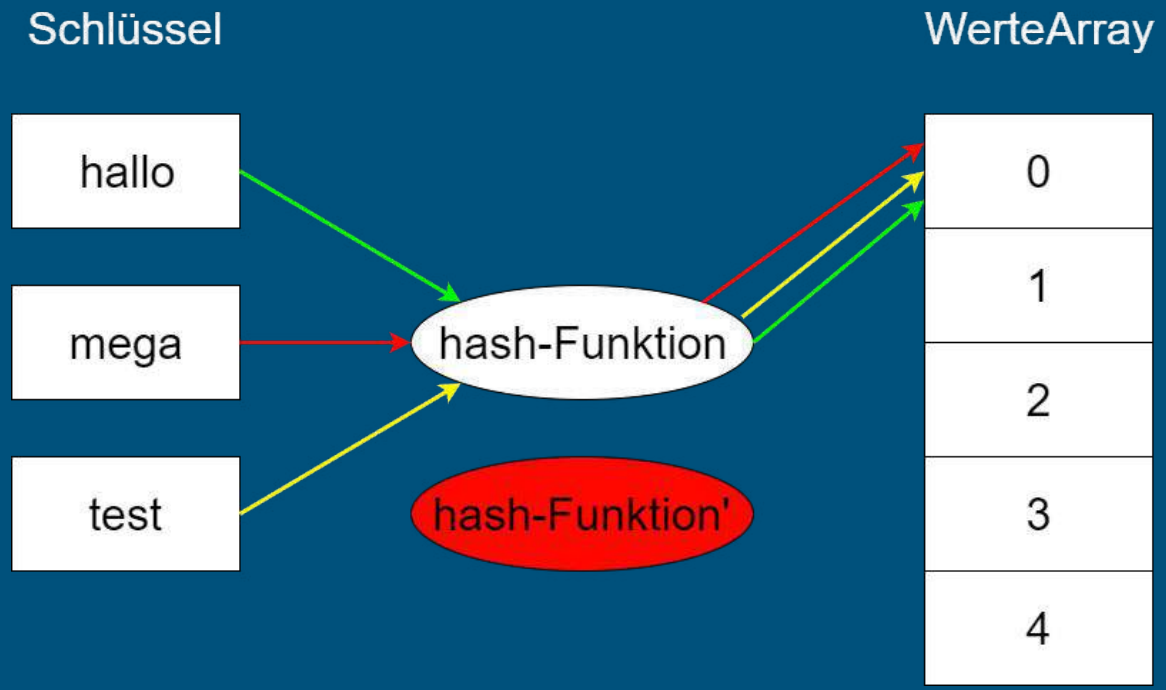


C# - Dictionary - Rehashing

Liste 0 zu voll

WerteArray erweitern

zweite hash-Funktion erzeugen



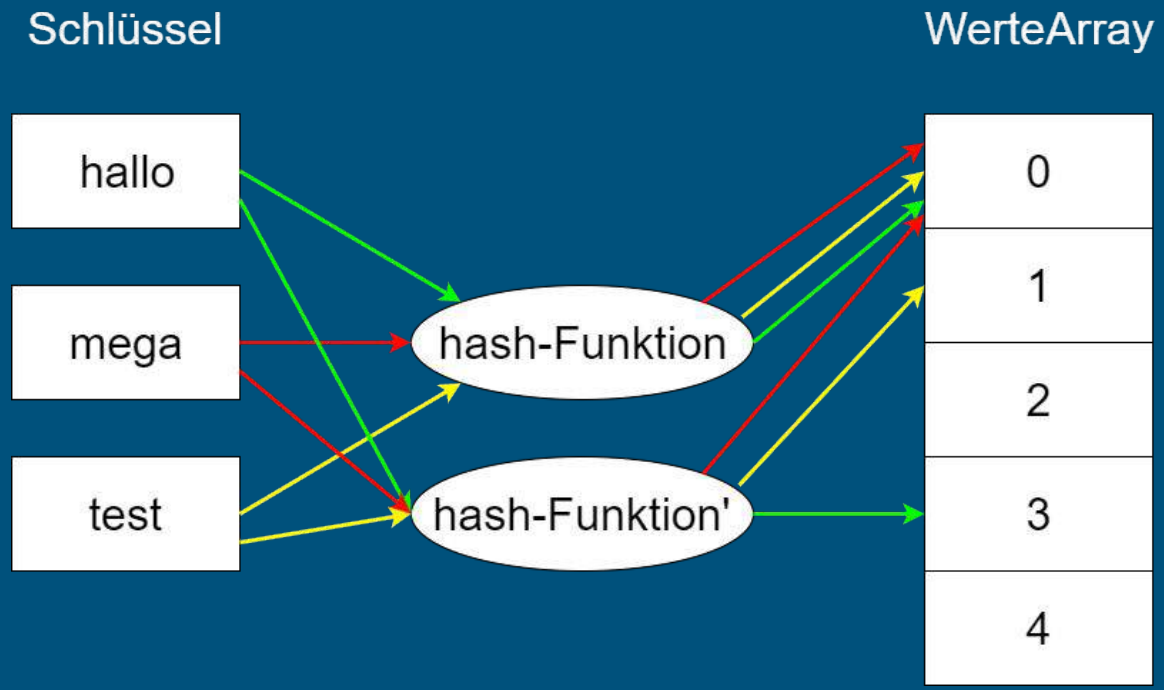
C# - Dictionary - Rehashing

Liste 0 zu voll

WerteArray erweitern

zweite hash-Funktion erzeugen

erste hash-Funktion zum Auslesen, zweite zum Schreiben nutzen



C# - Dictionary - Rehashing

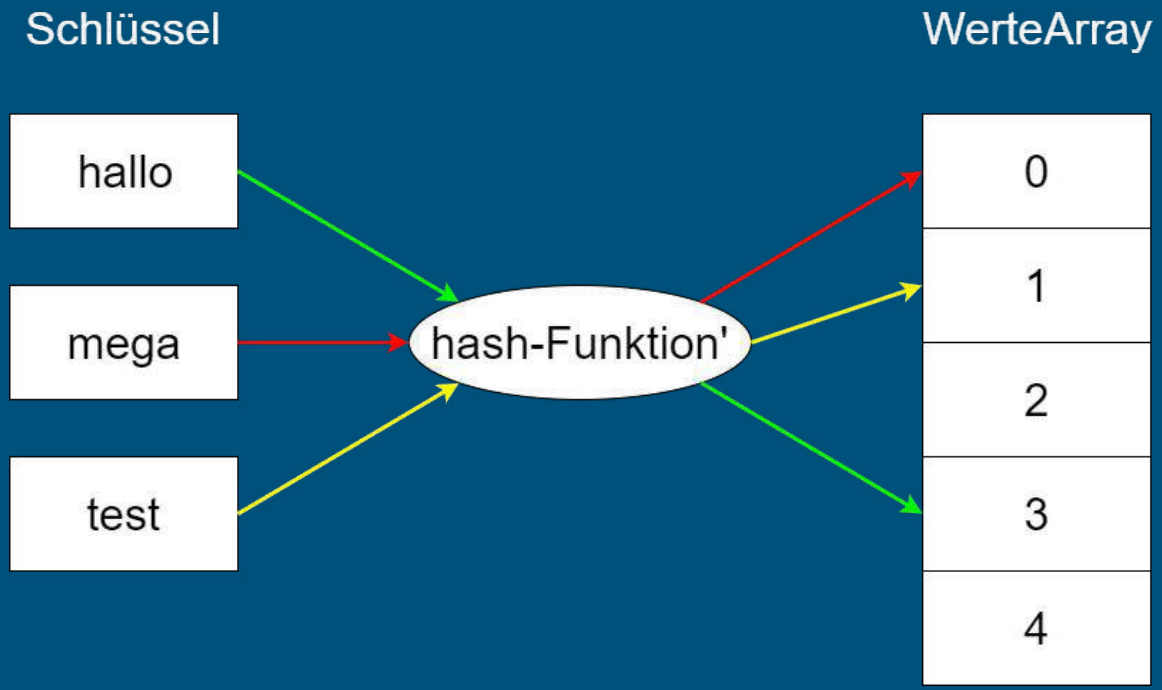
Liste 0 zu voll

WerteArray erweitern

zweite hash-Funktion erzeugen

erste hash-Funktion zum Auslesen, zweite zum Schreiben nutzen

erste hash-Funktion verwerfen



C# - Dictionary - Effizienz

Stark abhängig von verwendeter Hashfunktion

Wörterbuchoperationen bei guter Hashfunktion: amortisiert O(1)

Im schlimmsten Fall (Hashfunktion liefert immer gleichen Wert) O(n)

Platzbedarf: O(m + n)

n: Einträge in der Hashtabelle

m: Positionen im Array

Balancierte Binäre Suchbäume

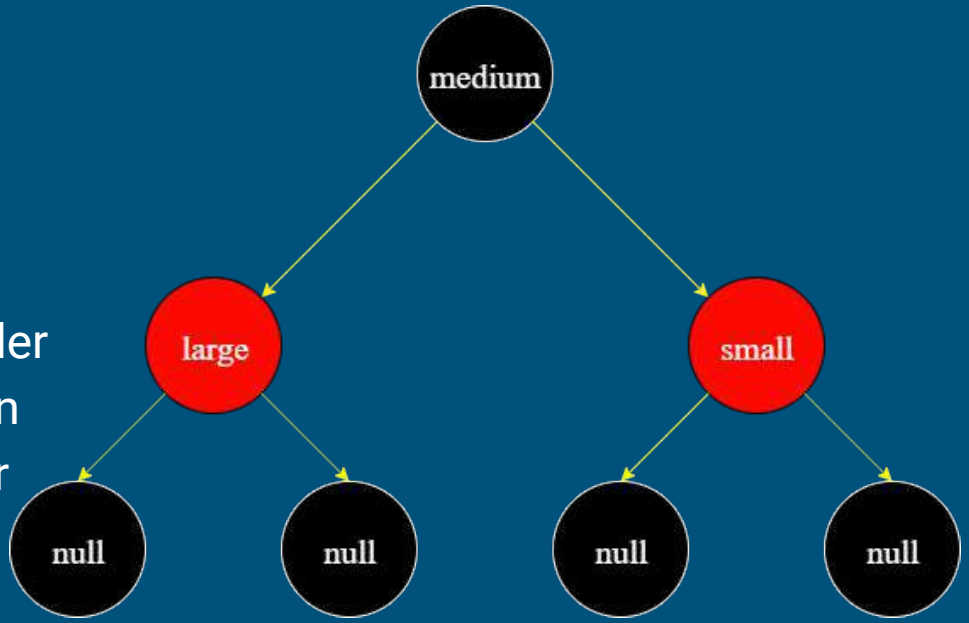
Rot-Schwarz-Baum - Aufbau

- Alle Knoten sind rot oder schwarz
- Wurzel ist schwarz
- Blätter sind schwarz
- Rote Knoten haben keine roten Kinder
- Alle Pfade eines Knotens zu Blättern haben die gleiche Anzahl schwarzer Knoten
- Knoten werden initial als rot eingefügt

Balancierte Binäre Suchbäume

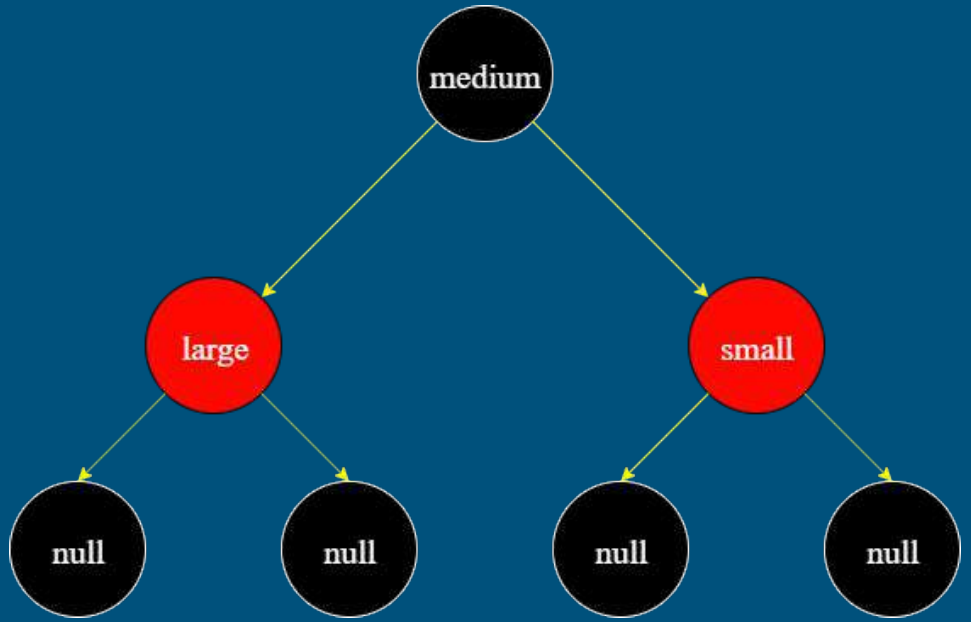
Rot-Schwarz-Baum - Beispiel

- Alle Knoten sind rot oder schwarz
- Wurzel ist schwarz
- Blätter sind schwarz
- Rote Knoten haben keine roten Kinder
- Alle Pfade eines Knotens zu Blättern haben die gleiche Anzahl schwarzer Knoten



Balancierte Binäre Suchbäume

Rot-Schwarz-Baum - Einfügen (“die”)

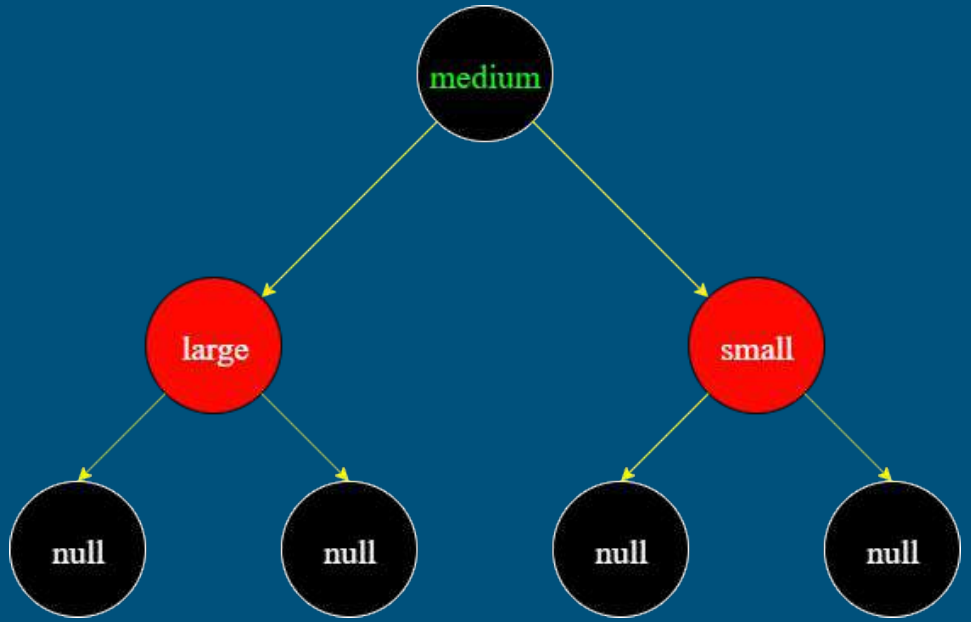


Balancierte Binäre Suchbäume

Rot-Schwarz-Baum - Einfügen (“die”)

string.Compare(“die”, “medium”);

“medium” linkes Kind vorhanden?



Balancierte Binäre Suchbäume

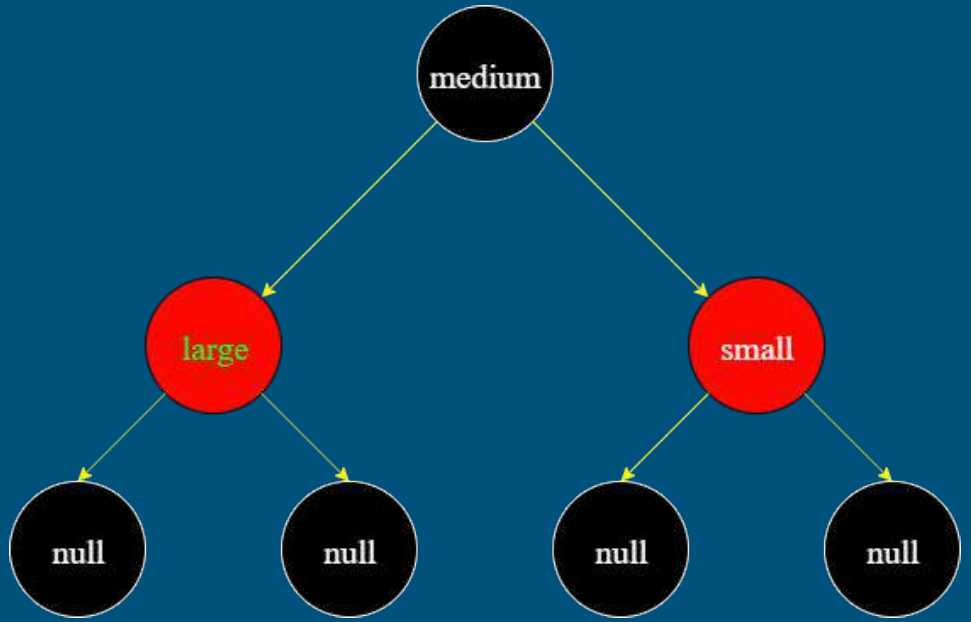
Rot-Schwarz-Baum - Einfügen (“die”)

string.Compare(“die”, “medium”);

“medium” linkes Kind vorhanden?

string.Compare(“die”, “large”);

“large” linkes Kind vorhanden?



Balancierte Binäre Suchbäume

Rot-Schwarz-Baum - Einfügen (“die”)

string.Compare(“die”, “medium”);

“medium” linkes Kind vorhanden?

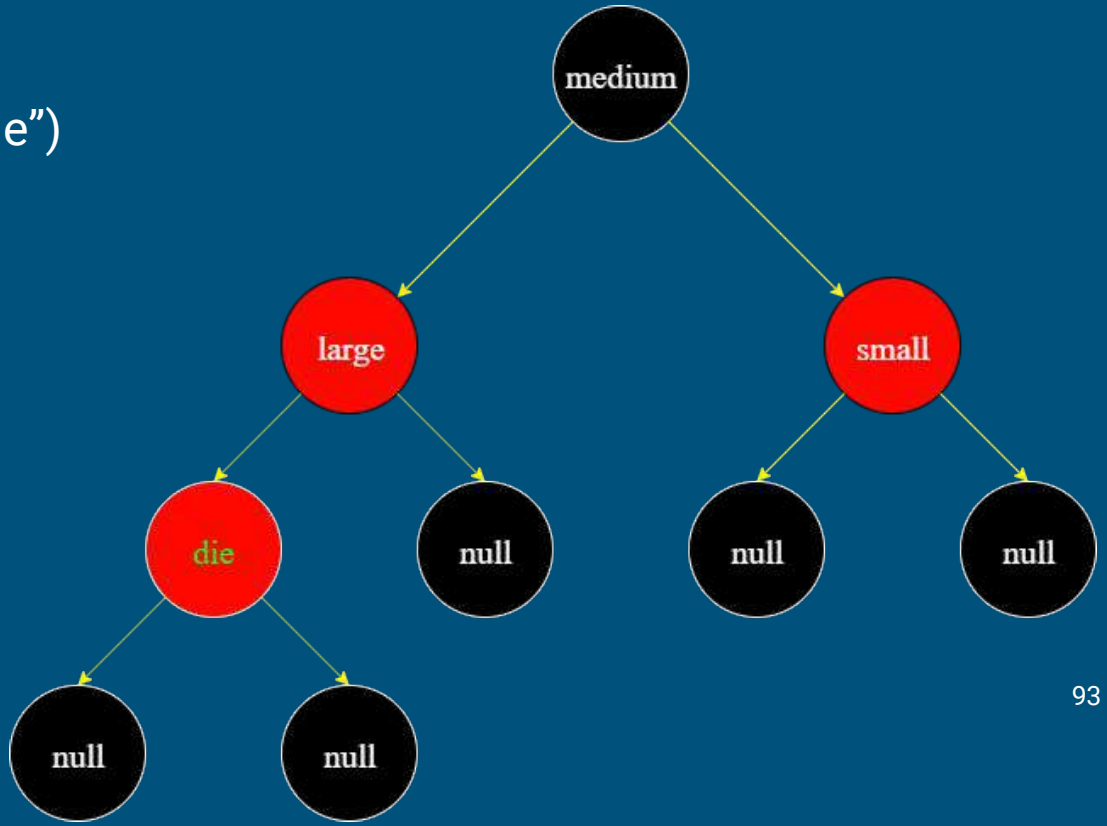
string.Compare(“die”, “large”);

“large” linkes Kind vorhanden?

“large” linkes Kind -> “die”

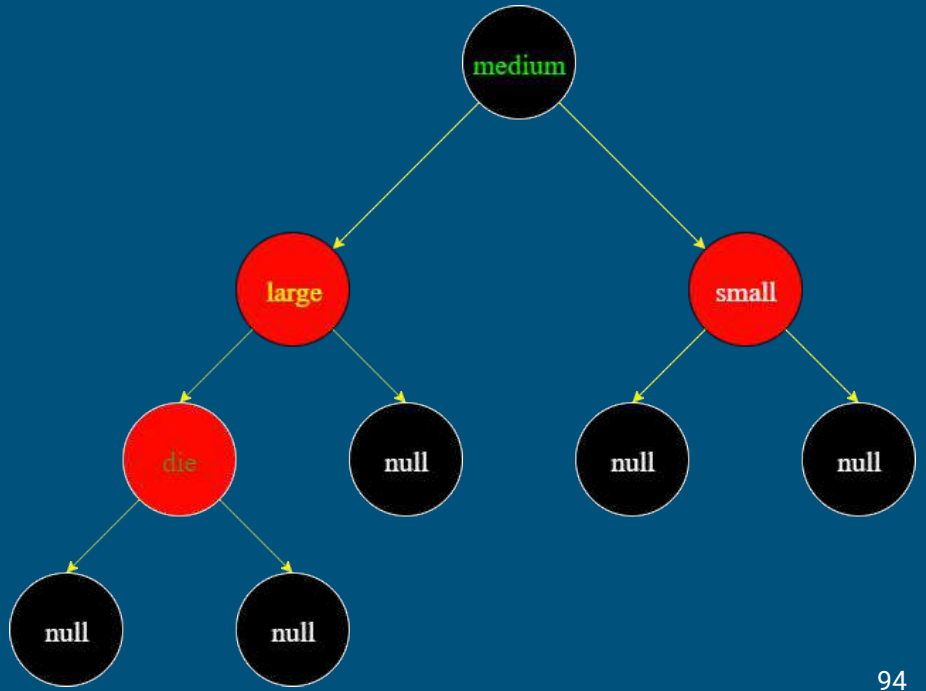
rot auf rot nicht möglich!

-> Umfärbung



Balancierte Binäre Suchbäume

loop bis Knoten == Wurzel oder Farbe des Elternknotens schwarz {



} loop Ende

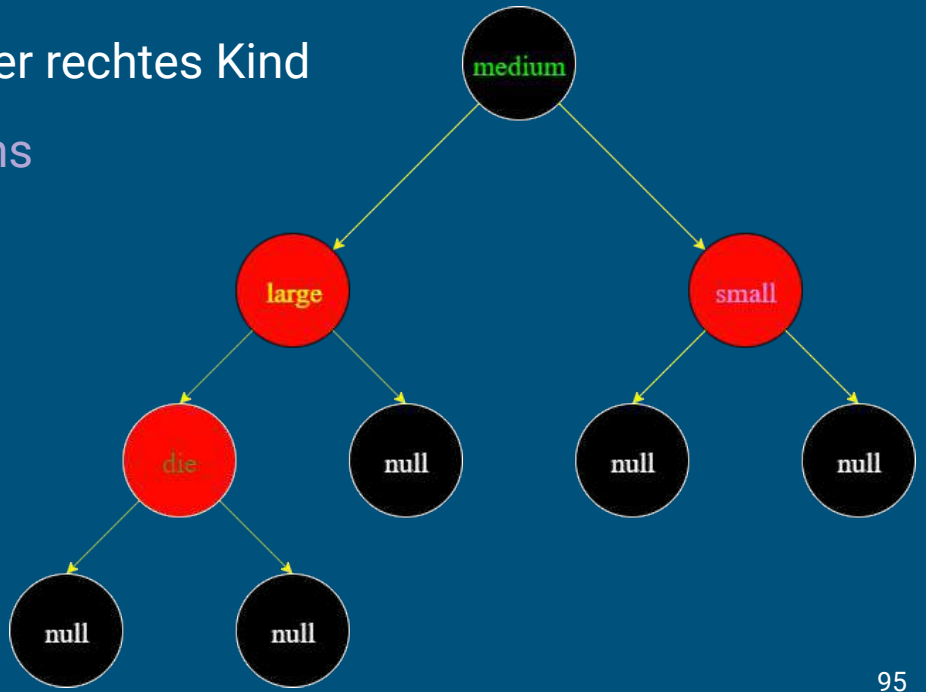
Balancierte Binäre Suchbäume

loop bis Knoten == Wurzel oder Farbe des Elternknotens schwarz {

Fallunterscheidung: Elternknoten linkes oder rechtes Kind

Fallunterscheidung: Farbe des Onkelknotens

} loop Ende



Balancierte Binäre Suchbäume

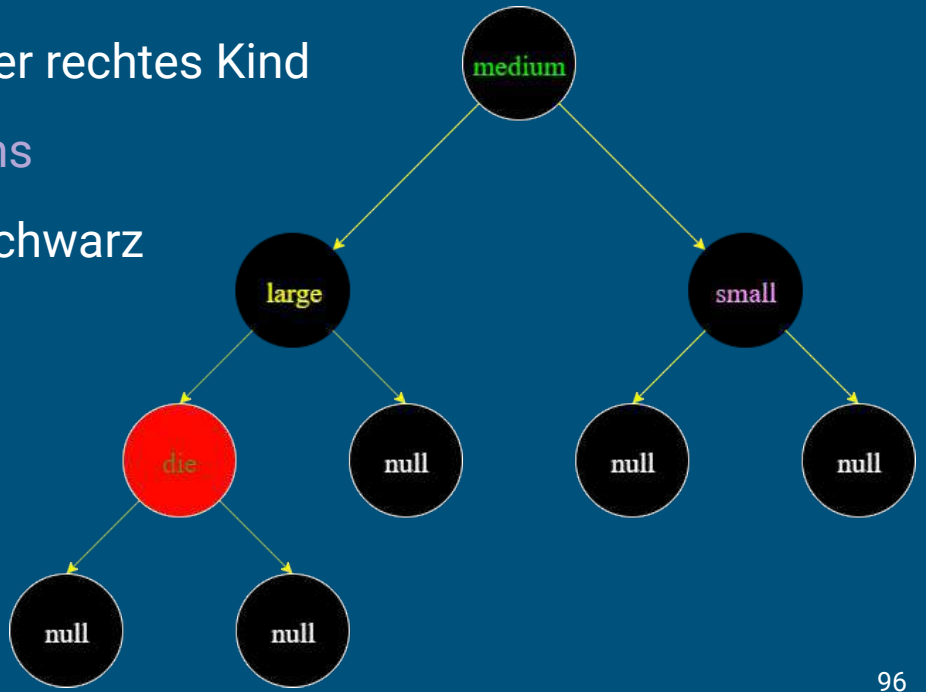
loop bis Knoten == Wurzel oder Farbe des Elternknotens schwarz {

Fallunterscheidung: Elternknoten linkes oder rechtes Kind

Fallunterscheidung: Farbe des Onkelknotens

Fall links, rot { Onkel- und Elternknoten -> schwarz

} loop Ende



Balancierte Binäre Suchbäume

loop bis Knoten == Wurzel oder Farbe des Elternknotens schwarz {

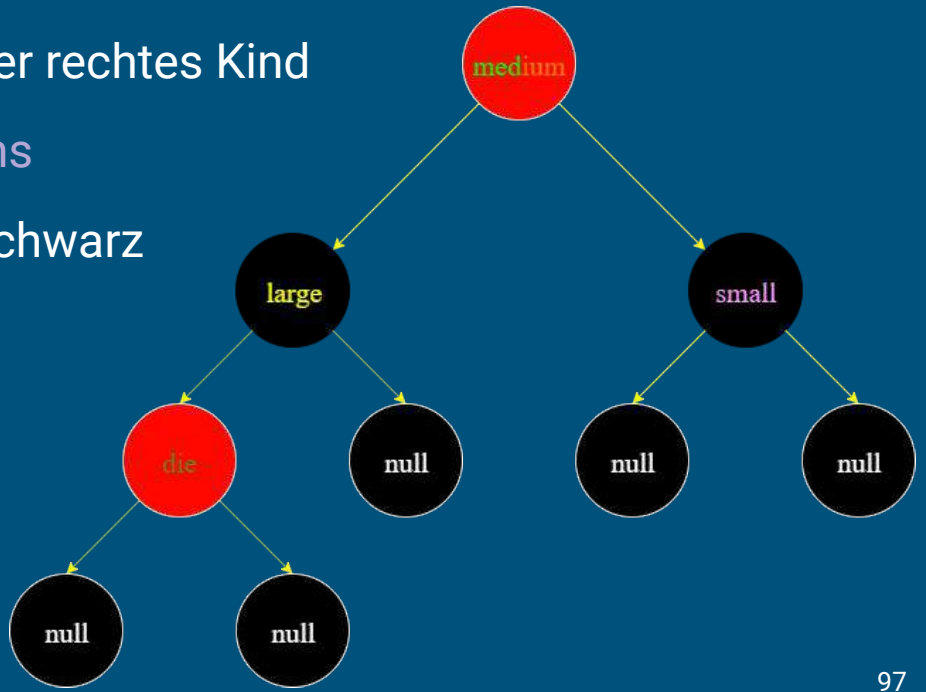
Fallunterscheidung: Elternknoten linkes oder rechtes Kind

Fallunterscheidung: Farbe des Onkelknotens

Fall links, rot { Onkel- und Elternknoten -> schwarz

Großeltern -> rot

} loop Ende



Balancierte Binäre Suchbäume

loop bis Knoten == Wurzel oder Farbe des Elternknotens schwarz {

Fallunterscheidung: Elternknoten linkes oder rechtes Kind

Fallunterscheidung: Farbe des Onkelknotens

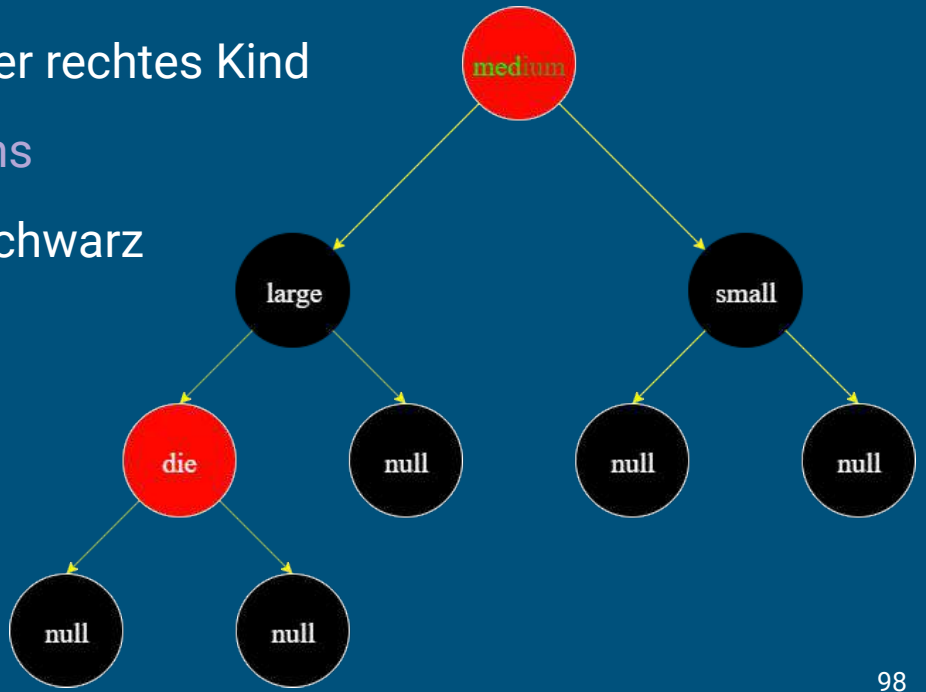
Fall links, rot { Onkel- und Elternknoten -> schwarz

Großeltern -> rot

Knoten = Großelternknoten }

Weitere Fälle: siehe Beispielprogramm

} loop Ende



Balancierte Binäre Suchbäume

loop bis Knoten == Wurzel oder Farbe des Elternknotens schwarz {

Fallunterscheidung: Elternknoten linkes oder rechtes Kind

Fallunterscheidung: Farbe des Onkelknotens

Fall links, rot { Onkel- und Elternknoten -> schwarz

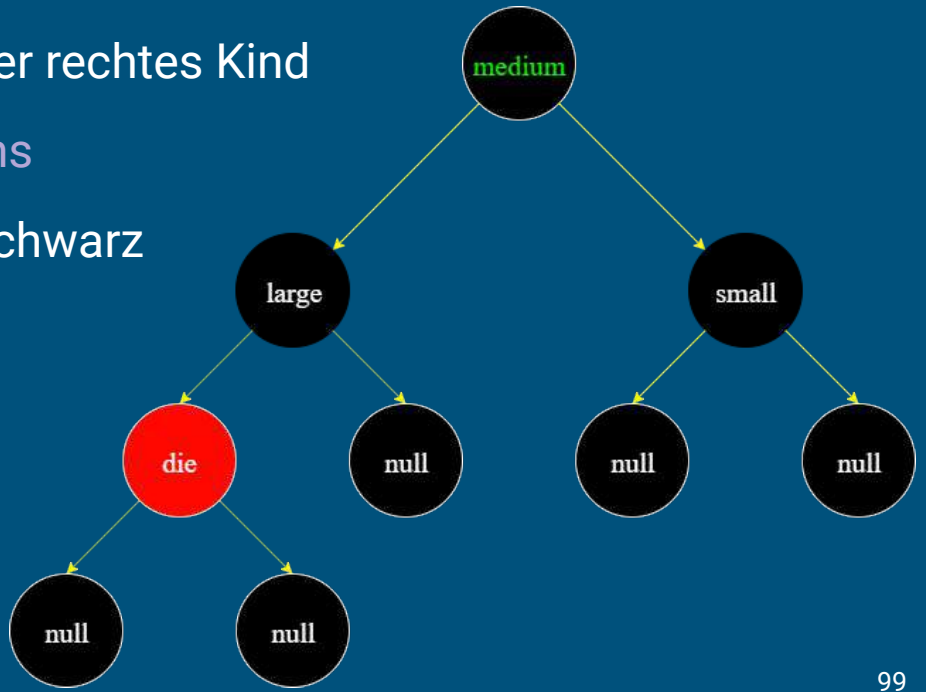
Großeltern -> rot

Knoten = Großelternknoten }

Weitere Fälle: siehe Beispielprogramm

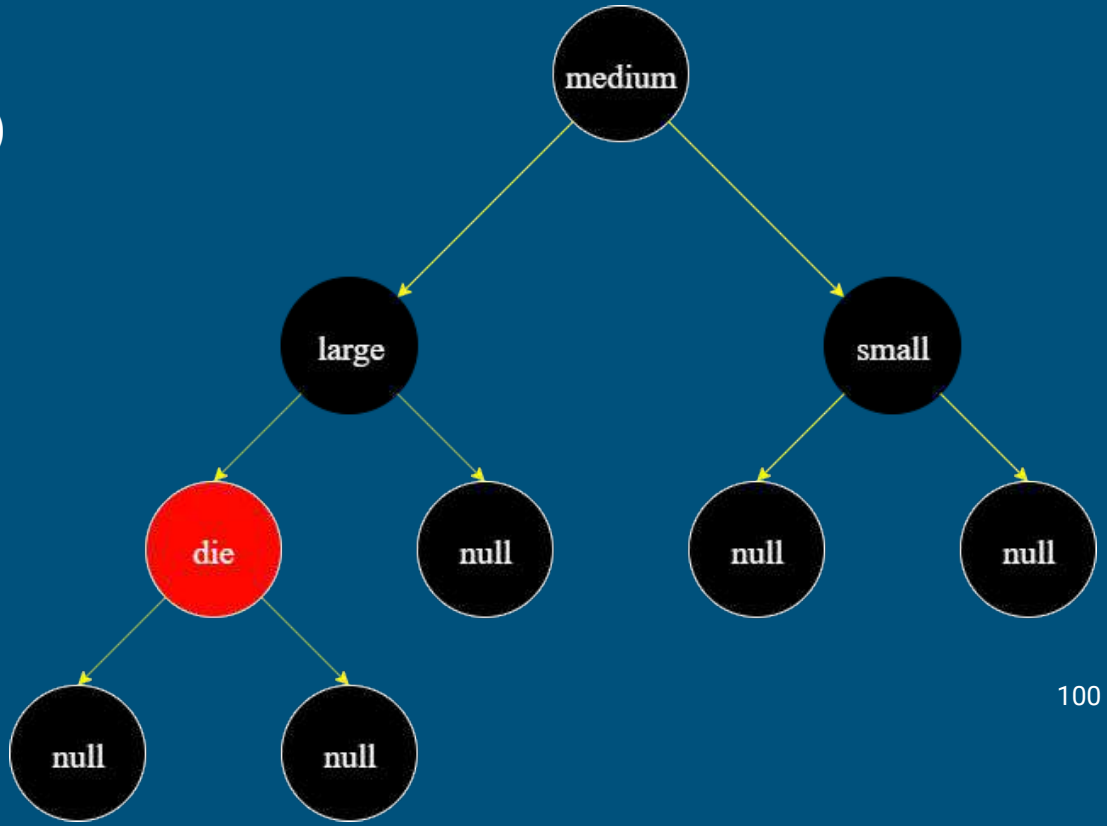
} loop Ende

Wurzel -> schwarz



Balancierte Binäre Suchbäume

Rot-Schwarz-Baum - Finden(“die”)

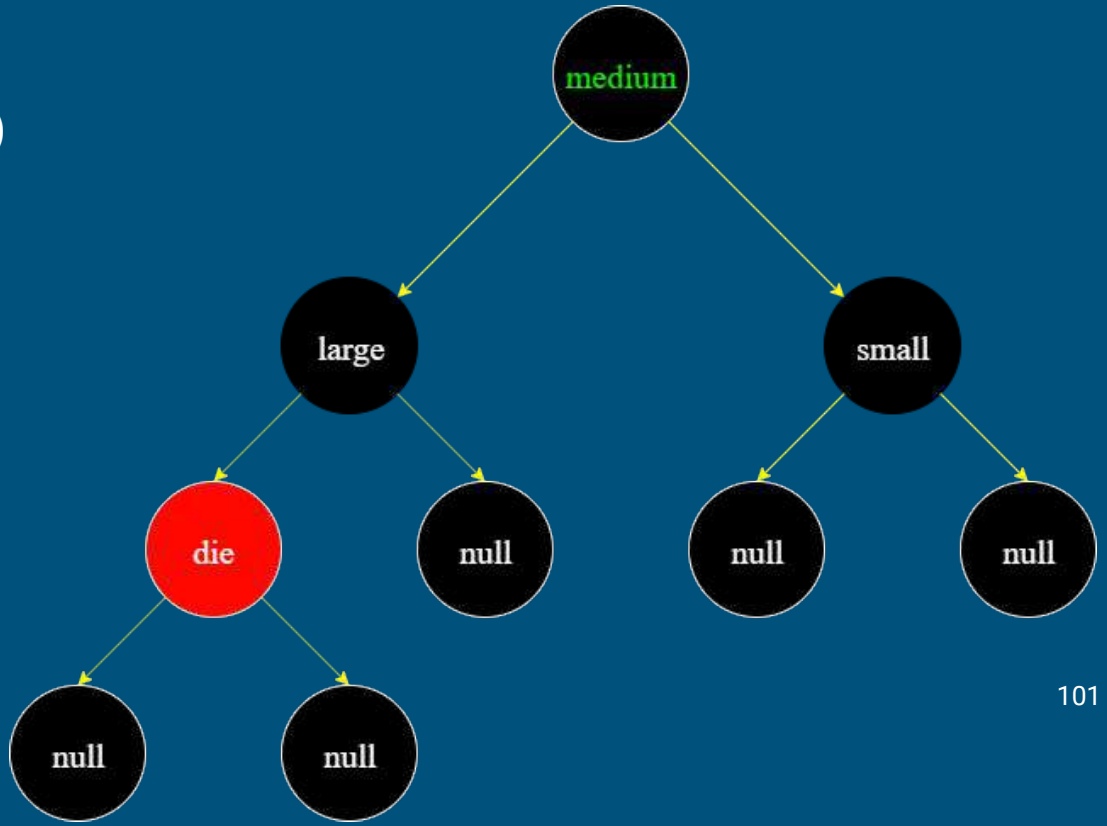


Balancierte Binäre Suchbäume

Rot-Schwarz-Baum - Finden(“die”)

string.Compare(“die”, “medium”);

“medium” linkes Kind vorhanden?



Balancierte Binäre Suchbäume

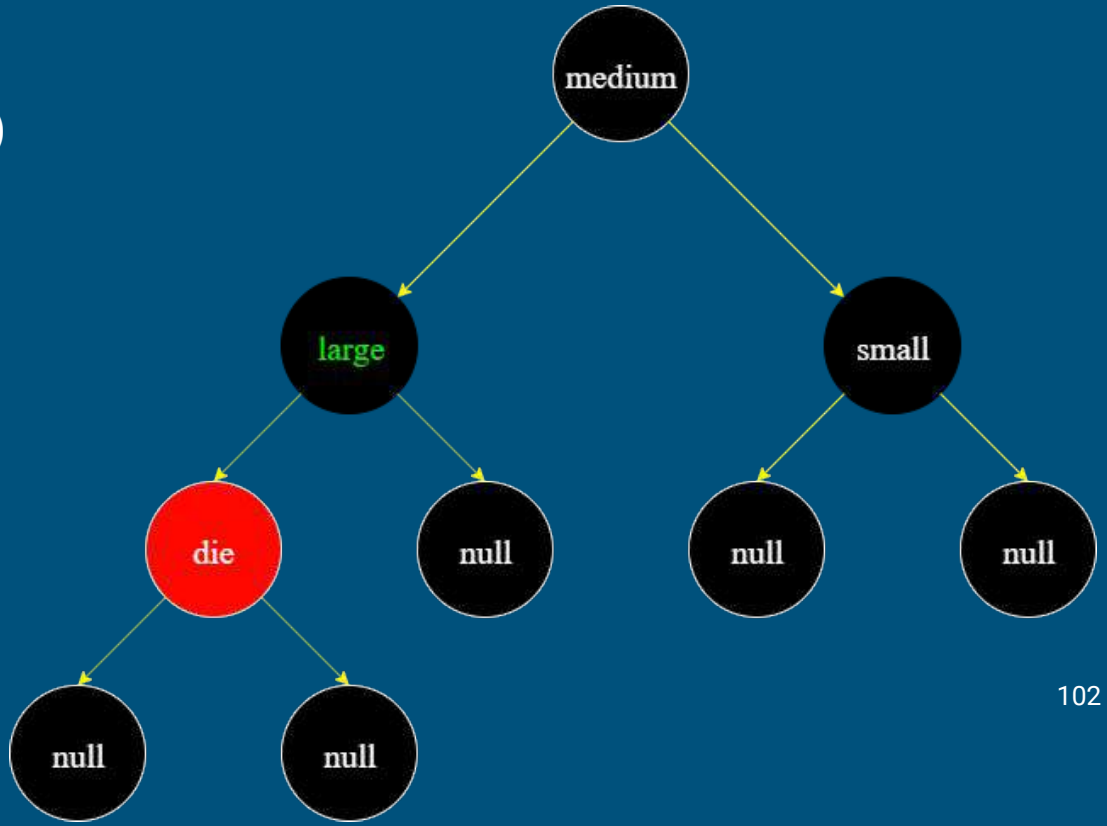
Rot-Schwarz-Baum - Finden(“die”)

string.Compare(“die”, “medium”);

“medium” linkes Kind vorhanden?

string.Compare(“die”, “large”);

“large” linkes Kind vorhanden?



Balancierte Binäre Suchbäume

Rot-Schwarz-Baum - Finden(“die”)

string.Compare(“die”, “medium”);

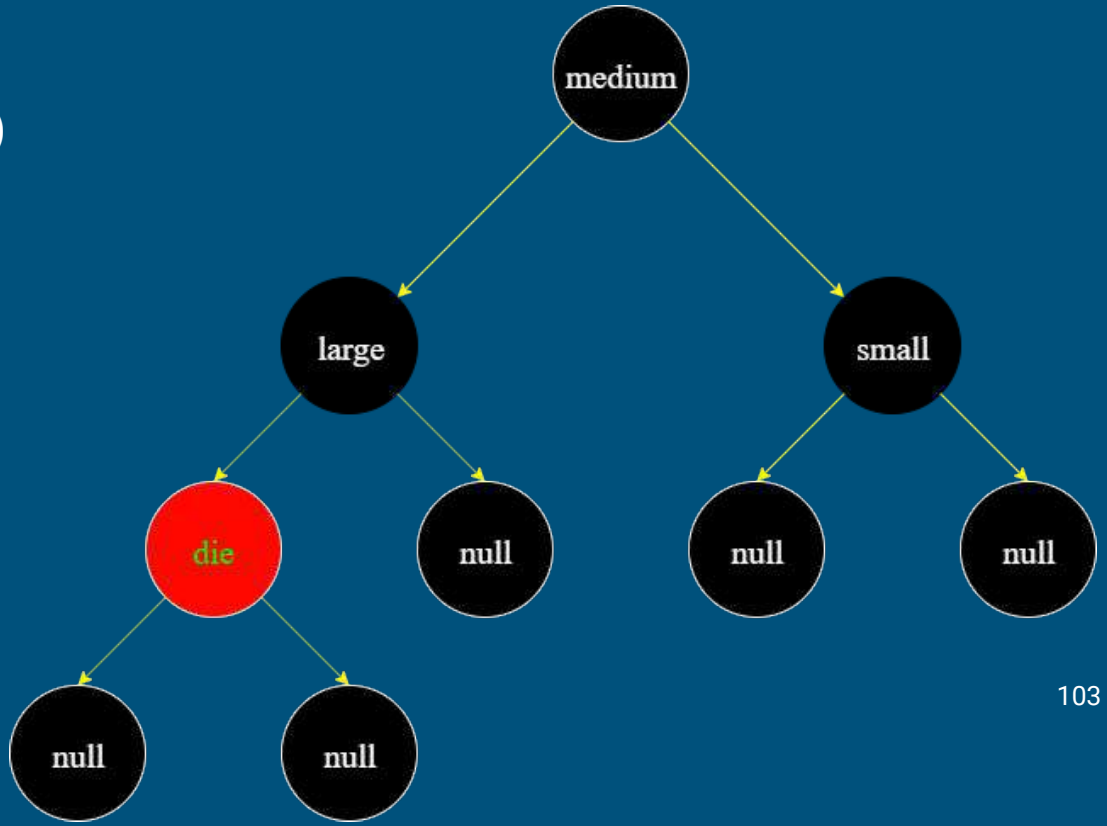
“medium” linkes Kind vorhanden?

string.Compare(“die”, “large”);

“large” linkes Kind vorhanden?

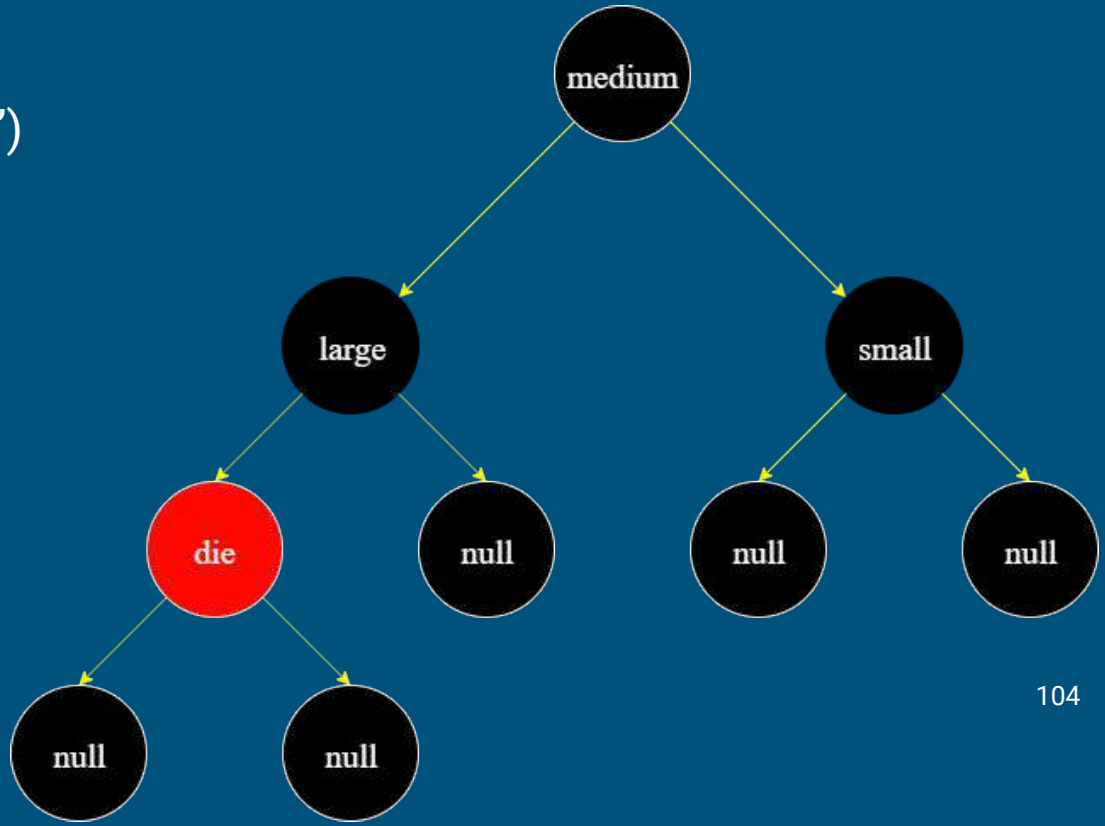
string.Compare(“die”, “die”);

gefunden



Balancierte Binäre Suchbäume

Rot-Schwarz-Baum - Löschen(“die”)



Balancierte Binäre Suchbäume

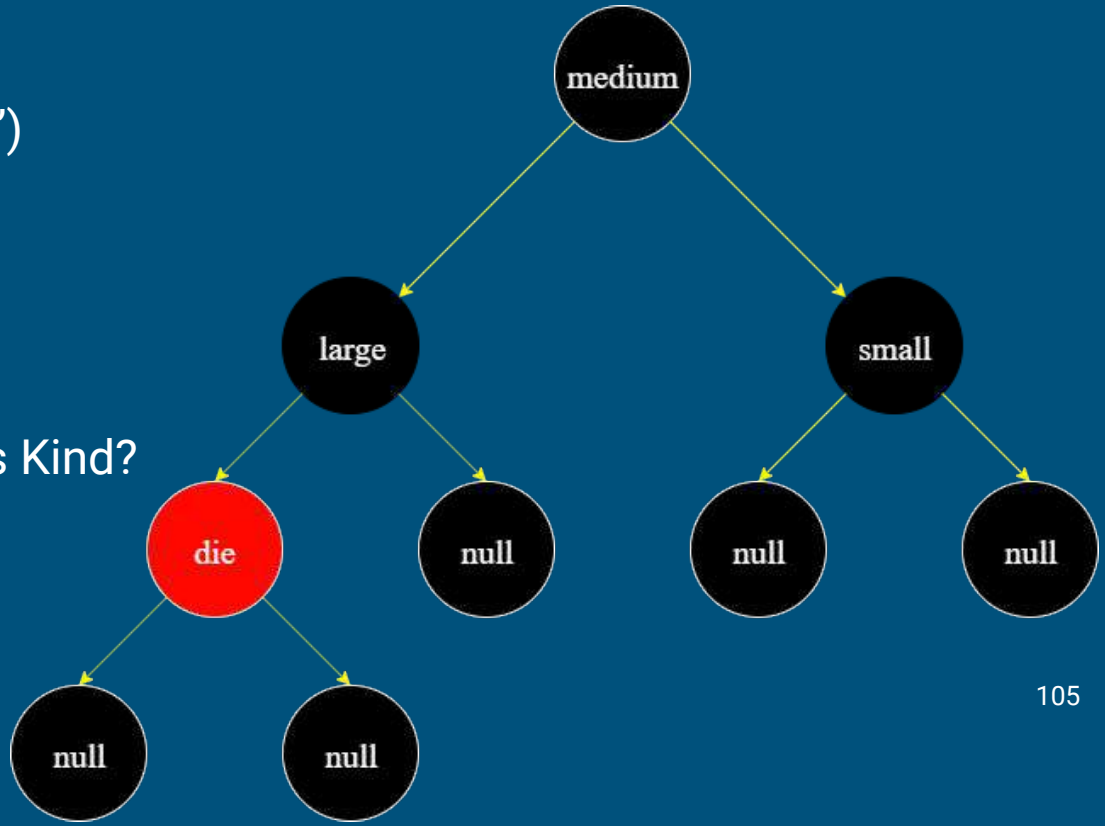
Rot-Schwarz-Baum - Löschen(“die”)

Knoten “die” finden

Welche Farbe hat Knoten “die”?

Hat Knoten “die” Kinder?

Ist Knoten “die” linkes oder rechtes Kind?



Balancierte Binäre Suchbäume

Rot-Schwarz-Baum - Löschen(“die”)

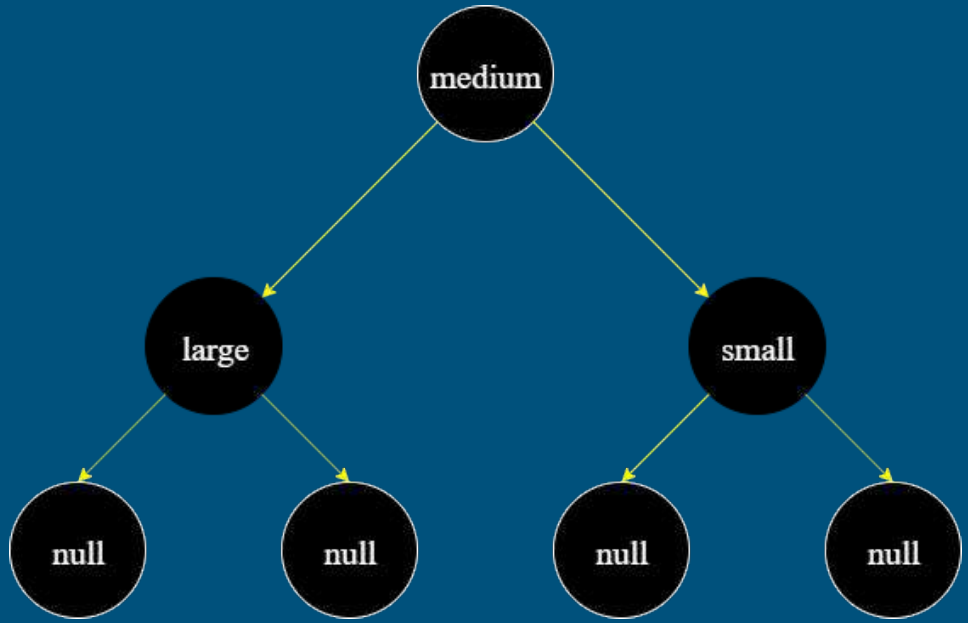
Knoten “die” finden

Welche Farbe hat Knoten “die”?

Hat Knoten “die” Kinder?

Ist Knoten “die” linkes oder rechtes Kind?

Lösche “large”s linkes Kind



Weiteres im Beispielprogramm

Balancierte Binäre Suchbäume

Rot-Schwarz-Baum - Effizienz

n: Anzahl der im Baum gespeicherten Elemente

Platzbedarf: O(n)

Wörterbuchoperationen: O(log n)

Balancierte Binäre Suchbäume

Weitere balancierte binäre Suchbäume

- AA - Baum
- AVL - Baum
- B - Baum
- 2 - 3 - Baum

Alle dieser Bäume haben die gleiche Komplexität, wie der Rot - Schwarz - Baum

Balancierte Binäre Suchbäume vs. Tries

Speichern von strings - Balancierte Binäre Suchbäume oder Tries?

	Trie (Hash-Implementation)	Balancierte Binäre Suchbäume
Laufzeit (worst-case)	amortisiert O(k)	O(log n)
Platz (worst-case)	O(n)	O(n)
Laufzeit (average-case)	amortisiert O(k)	O(log n)
Platz (average-case)	O(n / log k)	O(n)

k: Länge eines Strings
n: Anzahl Strings

Je mehr Strings gespeichert werden und Überschneidungen haben, desto besser eignet sich ein Trie.
Bei wenigen Strings ohne vielen Überschneidungen empfiehlt sich ein balancierter binärer Suchbaum.