Algorithmen und Datenstrukturen

D2. Suche in Strings[1]

Marcel Lüthi

Universität Basel

22. Mai 2019

[1] Folien angelehnt an Vorlesungsfolien von Sedgewick & Wayne
https://algs4.cs.princeton.edu/lectures/52Tries-2x2.pdf

Algorithmen und Datenstrukturen

22. Mai 2019 — D2. Suche in Strings[a]

[a] Folien angelehnt an Vorlesungsfolien von Sedgewick & Wayne
https://algs4.cs.princeton.edu/lectures/52Tries-2x2.pdf

D2.1 Einführung

Erinnerung: Symboltabellen

Abstraktion für Schlüssel/Werte Paar

Grundlegende Operationen

- Speichere Schlüssel mit dazugehörendem Wert.
- Suche zu Schlüssel gehörenden Wert.
- Schlüssel und Wert löschen.

Typische Beispiele

- DNS - Suche IP-Adresse zu Domainnamen
- Telefonbuch - Suche Telefonnummer zu Person / Adresse
- Wörterbuch - Suche Übersetzungen für Wort

Übersicht

	Worst-case		Average-case	
Implementation	suchen	einfügen	suchen (hit)	einfügen
Rot-Schwarz Bäume	$2\log_2(N)$	$2\log_2(N)$	$1\log_2(N)$	$1\log_2(N)$
Hashtabellen	N	N	1	1

- Frage: Geht es noch schneller?
- Antwort: Ja, wenn wir nicht ganzen String vergleichen müssen.

Symboltabelle für Strings

```
class StringST[Value]:

    def StringST()

    def put(key : String, value : Value) -> None

    def get(key : String) -> Value

    def delete(key : String) -> None

    def keys() -> Iterator[String]
```

Normale Symboltabellen Operationen, aber mit fixem Typ String als Schlüssel

Symboltabelle für Strings

```
class StringST[Value]:

    def StringST()

    def put(key : String, value : Value) -> None

    def get(key : String) -> Value

    def delete(key : String) -> None

    def keys() -> Iterator[String]

    def keysWithPrefix(s : String) -> Iterator[String]

    def keysThatMatch(s : String) -> Iterator[String]

    def longestPrefixOf(s : String) -> String
```

Mittels Tries lassen sich viele nützliche, zeichenbasierte Suchoperationen definieren.

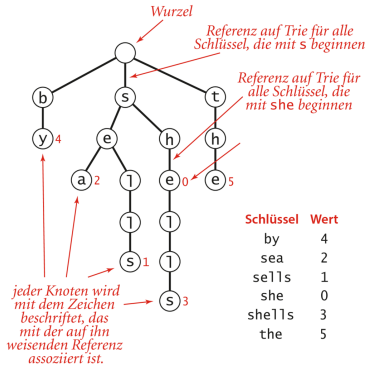
D2.2 Tries

Tries

Trie Von Retrieval.

- Ausgesprochen wie try

- Zeichen (nicht Schlüssel werden in Knoten gespeichert)
- Jeder Knoten hat R Knoten (also einen pro möglichem Zeichen)



Quelle: Sedgewick & Wayne, Algorithmen, Abbildung 5.19

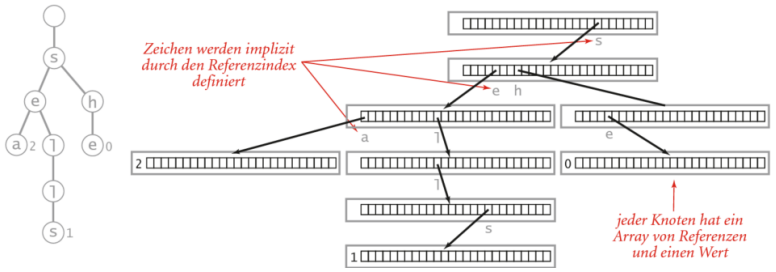
Erinnerung: Alphabet

Abstraktion Alphabet macht Code unabhängig von Alphabet.

```
class Alphabet:
    def __init__(s : List[char])
    def toChar(index : Int) -> char
    def toIndex(c : Char) -> int
    def contains(c : Char) -> boolean
    def R() -> int   # Radix
```

Name	Radix (R)	Bits ($\log_2(R)$)	Zeichen
BINARY	2	1	0 1
DNA	4	2	A C G T
ASCII	128	7	ASCII Characters
EXTENDED_ASCII	256	8	EXTENDED_ASCII
UNICODE	1'114'112	21	UNICODE

Repräsentation der Knoten



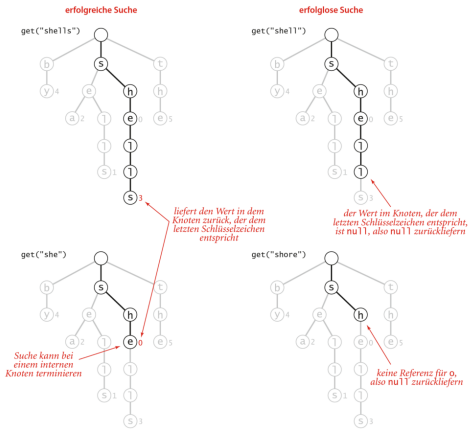
Quelle: Sedgewick & Wayne, Algorithmen, Abbildung 5.21

```
class Node:
    value = None
    children = [None] * R # R: Radix von Alphabet
```


Suche in Trie

Dem Zeichen entsprechenden Link folgen

- Erfolgreiche Suche: Endet an Knoten mit definiertem Wert
- Erfolglose Suche: Endet an Knoten mit undefiniertem Wert (null)



Quelle: Sedgewick & Wayne, Algorithmen, Abbildung 5.20

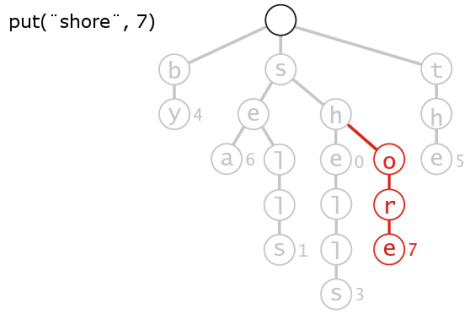
Suche in Tries

```
def get(node, key, d):
    if (node == None):
        return None
    if d == len(key):
        return node
    c = alphabet.toIndex(key[d])
    return get(node.children[c], key, d + 1)
```


Einfügen in Trie

Dem Zeichen entsprechenden Link folgen

- Erfolgreiche Suche: Wert neu setzten
- Erfolglose Suche: Neuen Knoten erzeugen.



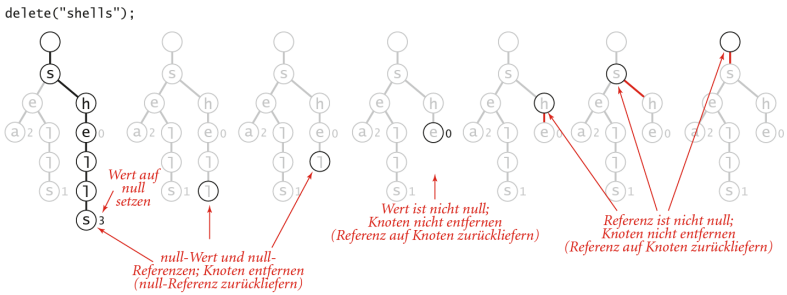
Quelle: Sedgewick & Wayne, Algorithmen, Abbildung 5.22

Einfügen in Trie

```
def put(node, key, value, d):
    if node == None:
        node = Node(alphabet.radix())
    if d == len(key):
        node.value = value
        return node
    c = alphabet.toIndex(key[d])
    node.children[c] = put(node.children[c], key, value, d + 1)
    return node
```


Löschen von Schlüsseln

- Schlüssel finden und Knoten löschen.
- Rekursiv alle Knoten mit nur null-Werten und null-links löschen

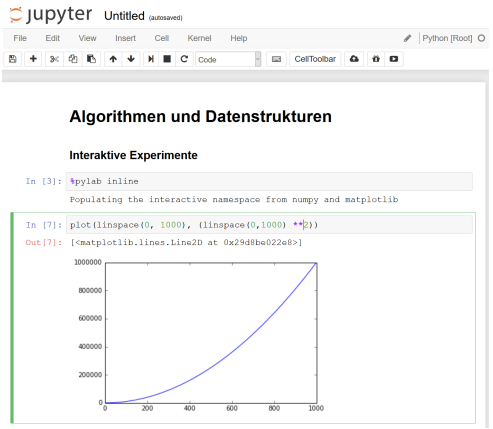


Quelle: Sedgewick & Wayne, Algorithmen, Abbildung 5.26

Löschen von Schlüsseln

```
def delete(node, key, d):
    if node == None:
        return None
    if d == len(key):
        node.value = None
    else:
        c = alphabet.toIndex(key[d])
        node.children[c] = delete(node.children[c], key, d + 1)

    if node.value != None:
        return node

    nonNullChildren = [c for c in node.children if c != None]
    if len(nonNullChildren) > 0:
        return node
    else:
        return None
```


Implementation und Beispielanwendung



Jupyter Notebook: Tries.ipynb

Analyse: Form des Tries

Theorem

Die verkettete Struktur (Form) eines Trie ist nicht abhängig von der Schlüsselreihenfolge beim Löschen/Einfügen: Für jede gegebene Menge von Schlüsseln gibt es einen eindeutigen Trie.

Analyse: Einfügen

Theorem

Die Anzahl der Arrayzugriffe beim Suchen in einem Trie oder beim Einfügen eines Schlüssels in einen Trie ist höchstens 1 plus der Länge des Schlüssels.

Theorem

Die durchschnittliche Anzahl der untersuchten Knoten bei einer erfolglosen Suche in einem Trie, der aus N Zufallsschlüsseln über einem Alphabet der Grösse R erstellt wird, beträgt $\sim \log_R(N)$.

Take-Home Message

Auch in riesigen Datenmengen können wir mit wenigen Vergleichen jeden Wert finden.

Speichereffiziente Variante: Ternäre Suchtries

Ein Problem mit Tries

- In jedem Knoten wird ein Array der Grösse R gespeichert
- Beispiel Unicode (utf-16): $R = 2^{16} = 65536$.

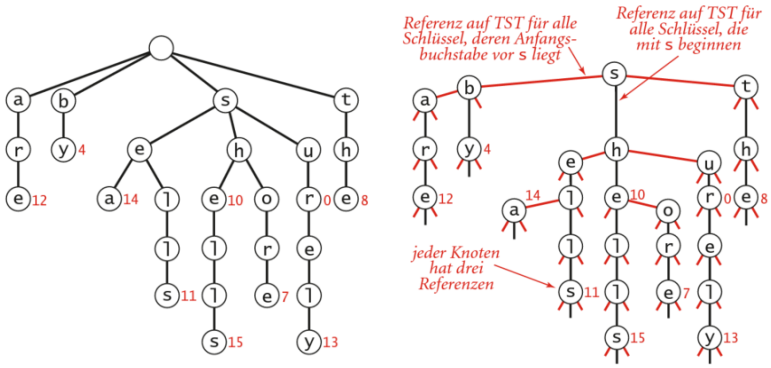
Lösung:

- Speichere Zeichen c sowie Wert im Knoten
- Jeder Knoten hat 3 Kinder:
  - Kleiner c (linker Teilbaum)
  - Gleich c (mittlerer Teilbaum)
  - Grösser c (rechter Teilbaum)

Bentley, Jon L., and Robert Sedgewick. "Fast algorithms for sorting and searching strings."1997.

Speichereffiziente Variante: Ternäre Suchtries

Beispiel:



Quelle: Sedgewick & Wayne, Algorithmen, Abbildung 5.28

- Bei Übereinstimmung wird Suche mit nächstem Zeichen im mittleren Teilbaum weitergeführt.
- Suche ist erfolgreich, wenn Wert in Endknoten der Suche gespeichert ist.

Tries versus Hashing

Hashing:

- Muss ganzen Schlüssel anschauen
- Etwa gleiche Kosten für erfolgreiche und erfolglose Suche
- Performance hängt von Hashfuktion ab
- Keine ordnungsbasierten Operationen

Tries:

- Nur für Strings geeignet
- Macht nur so viele Vergleiche wie gerade benötigt werden
- Erfolglose Suche benötigt nur ein paar Zeichenvergleichen
- Flexible zeichenbasierte Operationen werden unterstützt

D2.3 Zeichenbasierte Operationen

Zeichenbasierte Operationen

Schlüssel	Wert
by	4
sea	6
sells	1
she	0
shells	3
shore	7
the	5

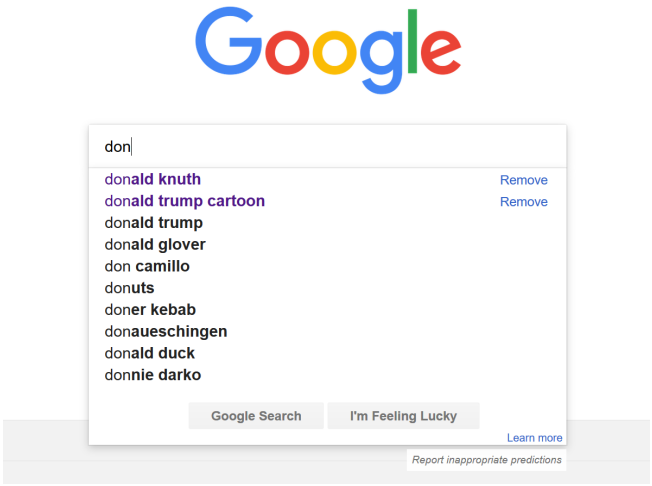
Präfix matching Präfix: `sh`: Schlüssel, `she`, `shells`, `shore`

Wildcard matching `.he`: Schlüssel, `she`, `the`

Längstes Präfix Anfrage: `shellsort` Schlüssel: `shells`

Präfixübereinstimmung

Beispielanwendung: Autocomplete



Zeichenbasierte Operationen

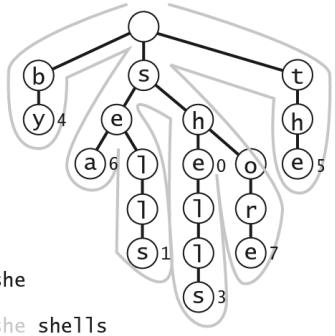
```
class StringST:

    def StringST()

    def put(key : String, value : Value) -> None

    def get(key : String) -> Value

    def delete(key : String) -> None

    def keys() -> Iterator[String]

    def keysWithPrefix(s : String) -> Iterator[String]

    def keysThatMatch(s : String) -> Iterator[String]

    def longestPrefixOf(s : String) ->  String
```

Warmup: Alle Schlüssel zurückgeben

- Traversieren des Baumes
- Bei jedem Knoten mit Wert $\neq$ null, Schlüssel merken

```
keysWithPrefix("");
```

Schlüssel	q
b	
by	by
s	
se	
sea	by sea
sel	
sell	
sells	by sea sells
sh	
she	by sea sells she
shell	
shells	by sea sells she shells
sho	
shor	
shore	by sea sells she shells shore
t	
th	
the	by sea sells she shells shore the

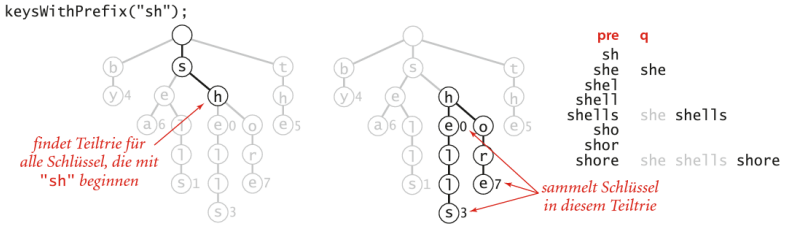


Quelle: Sedgewick & Wayne, Algorithmen, Abbildung 5.23

Präfixübereinstimmung

- Subtrie mit Präfix finden
- Alle Schlüssel von diesem Subtrie zurückgeben

```
keysWithPrefix("sh");
```



pre	q
sh	
she	she
shel	
shell	
shells	she shells
sho	
shor	
shore	she shells shore

Quelle: Sedgewick & Wayne, Algorithmen, Abbildung 5.24

Präfixübereinstimmung

```
def keys(node):
    return keysWithPrefix(node, "")

def keysWithPrefix(node, refix):
    queue = []
    collect(get(node, prefix, 0), prefix, queue)
    return queue


def collect(node, prefix, queue):
    if node == None:
        return
    if node.value != None:
        queue.append(prefix)

    labeledChildren = [(alphabet.toChar(p), child)
                       for (p, child) in enumerate(node.children)
                       if child != None]

    for (char, child) in labeledChildren:
        collect(child, prefix + char, queue)
```


Längstes Präfix

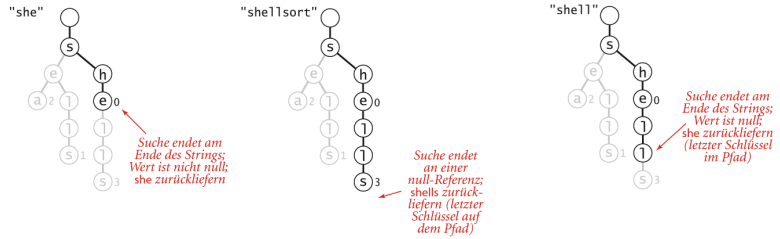
- Beispielanwendung: Routing, LZW-Kompression

"128"
"128.112"
"128.112.055"
"128.112.055.15"
"128.112.136"
"128.112.155.11"
"128.112.155.13"
"128.222"
"128.222.136"

Anfrage:
longestPrefixOf("128.112.136.11") = "128.112.136"
longestPrefixOf("128.112.100.16") = "128.112"
longestPrefixOf("128.166.123.45") = "128"

Längstes Präfix

- Nach Anfragestring suchen
- Längster Schlüssel auf dem Weg dahin speichern.



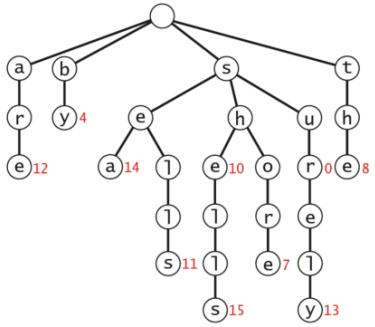
Quelle: Sedgewick & Wayne, Algorithmen, Abbildung 5.25

Längstes Präfix

```
def longestPrefixOf(node, s):
    l = search(node, s, 0, 0)
    return s[0:l]


def search(node, s, d, length):
    if node == None:
        return length
    if node.value != None:
        length = d
    if d == len(s):
        return length
    c = alphabet.toIndex(s[d])
    return search(node.children[c], s, d+1, length)
```


Quiz: Wildcard matching

- Wie implementieren wir Wildcard matching?



Quelle: Sedgewick & Wayne, Algorithmen, Abbildung 5.28

Zusammenfassung Symboltabellen

Wir haben viele sehr effiziente Implementation zur Auswahl.

Rot-Schwarz Bäume

- Garantie: $log(N)$ Schlüsselvergleiche
- Ordnungsbasierte Operationen

Hashtabellen

- Durchschnittliche Komplexität Suche, Einfügen $O(1)$
- Benötigt gute Hashfunktion

Tries

- Garantie: Komplexität um String der Länge N zu suchen: $log(N)$
- Mächtige zeichenbasierte Operationen