Algorithmen und Datenstrukturen

D1. Sortieren und Suchen in Strings

Gabi Röger und Marcel Lüthi

Universität Basel

20. Mai 2020

Algorithmen und Datenstrukturen
20. Mai 2020 — D1. Sortieren und Suchen in Strings

String algorithmen oder generische Algorithmen?

- Alle Algorithmen zum Sortieren / Suchen wurden über beliebige Schlüssel definiert.
  - Können direkt auf Strings angewendet werden.
- Preis der Abstraktion / Allgemeinheit: Vorhandene Struktur der Schlüssel wird nicht ausgenutzt.

Frage

Können wir Eigenschaften von Strings ausnutzen um noch effizientere Algorithmen zu entwickeln?

Sortieralgorithmen

Algorithmus	Laufzeit $O(\cdot)$	Speicherbedarf $O(\cdot)$	stabil
	best/avg./worst	best/avg./worst	
Selectionsort	n^2	1	nein
Insertionsort	$n/n^2/n^2$	1	ja
Mergesort	$n \log n$	n	ja
Quicksort	$n \log n/n \log n/n^2$	$\log n/\log n/n$	nein
Heapsort	$n \log n$	1	nein

$O(n \log n)$ ist beweisbar der lower bound für allgemeine, vergleichsbasierte Sortierverfahren. Geht es besser mit Strings?

Heutiges Programm

- Motivation
- Abstraktion: Alphabet
- LSD Sortierverfahren
- Quicksort für Strings
- Tries

Repetition und Erweiterung bereits bekannter Konzepte

D1.1 Strings

Strings als fundamentale Abstraktion

Strings / Text ist in vielen Bereichen grundlegende Repräsentation von Informationen

- Programmcode
- Datenrepräsentation im Web (HTML / Json / CSS)
- Kommunikation (E-Mail, Textmessages)
- Gensequenzen

Strings

String

Endliche Folge von Zeichen (Character)

- Strings sind unveränderlich (immutable). Einmal erzeugt können Strings nicht mehr verändert werden.
  - Ideale Schlüssel für Symboltabellen
- Intern häufig als Array von Zeichen implementiert.

0	1	2	3	4	5	6	7	8	9	10	11
A	T	T	A	C	K	A	T	D	A	W	N

Characters

Früher:

- 7 Bit Zeichensatz (ASCII)
- 8 Bit Zeichensatz (extended ASCII)

Heute:

- 8 oder 16 bit Unicode Zeichensatz (UTF-8, UTF-16)

Unterschied Java / Python

- Java Character entspricht 16 bit Unicode Zeichen (UTF-16)
- Python kennt keinen Charactertyp. Ausdruck s[i] ist (UTF-8) String der Länge 1.

Abstraktion: Alphabet

- Unicode umfasst 1'112'064 Zeichen.
- Kleineres Alphabet reicht für viele Anwendungen aus

Name	Radix (R)	Bits ($\log_2(R)$)	Zeichen
BINARY	2	1	0 1
DNA	4	2	A C G T
LOWERCASE	26	5	a - z
UPPERCASE	26	5	A-Z
ASCII	128	7	ASCII Characters
EXTENDED_ASCII	256	8	EXTENDED_ASCII
UNICODE	1'114'112	21	UNICODE

Alphabet

Abstraktion Alphabet erlaubt uns Code unabhängig vom benutzten Alphabet zu schreiben.

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

D1.2 LSD-Sortierverfahren

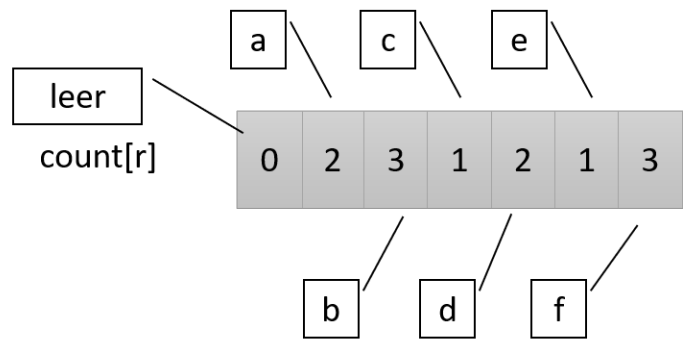
LSD-Sortierverfahren (1 Zeichen)

- Input: Array a, Output: Sortiertes array aux

```
N = len(a)  # Anzahl zu sortierender Zeichen
count = [0] * (alphabet.radix() + 1)
aux = [None] * N

# Zeichen zaehlen
for i in range(0, N):
    indexOfchar = alphabet.toIndex(a[i])
    count[indexOfchar + 1] += 1

# Kummulative Summe
for r in range(0, alphabet.radix()):
    count[r+1] += count[r]

# Verteilen
for i in range(0, N):
    indexOfchar = alphabet.toIndex(a[i])
    countForChar = count[indexOfchar]
    aux[countForChar] = a[i]
    count[indexOfchar] += 1
```

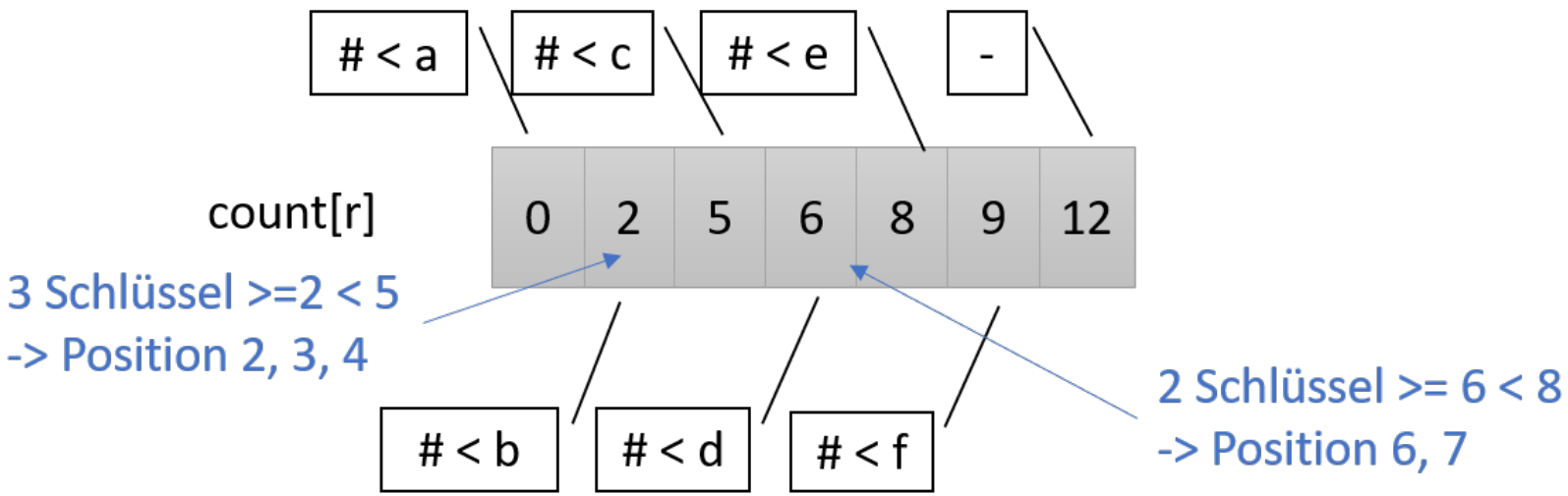
LSD-Sortierverfahren (1 Zeichen)

```
N = len(a)  # Anzahl zu sortierender Zeichen in array a
count = [0] * (alphabet.radix() + 1)

# Zeichen Zaehlen
for i in range(0, N):
    indexOfchar = alphabet.toIndex(a[i])
    count[indexOfchar + 1] += 1
```



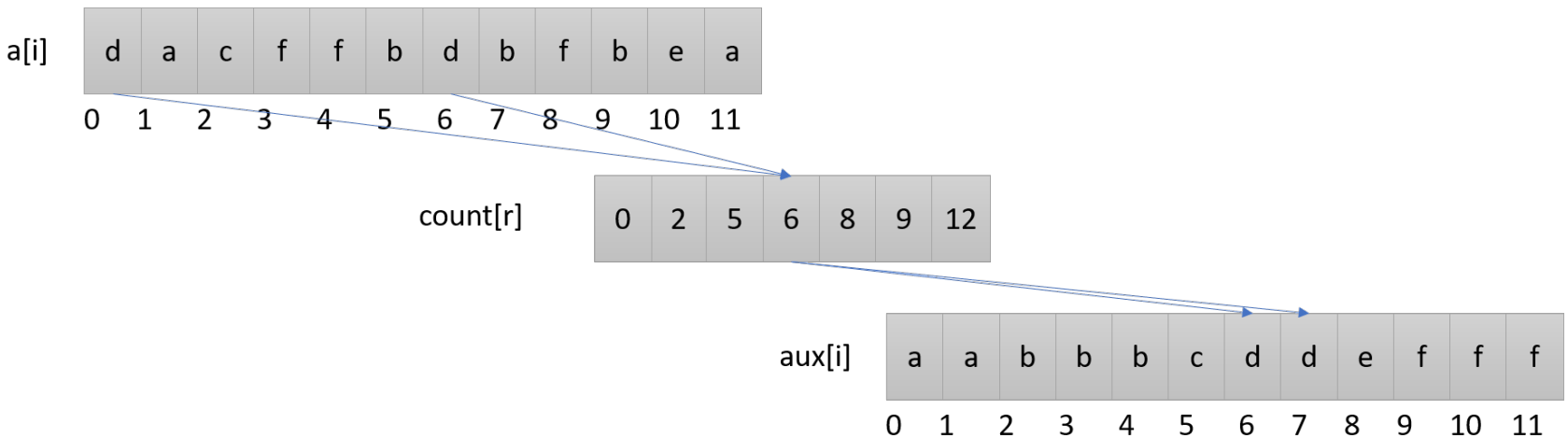
LSD-Sortierverfahren (1 Zeichen)

```
# Kummulative Summe
for r in range(0, alphabet.radix()):
    count[r+1] += count[r]
```



LSD-Sortierverfahren (1 Zeichen)

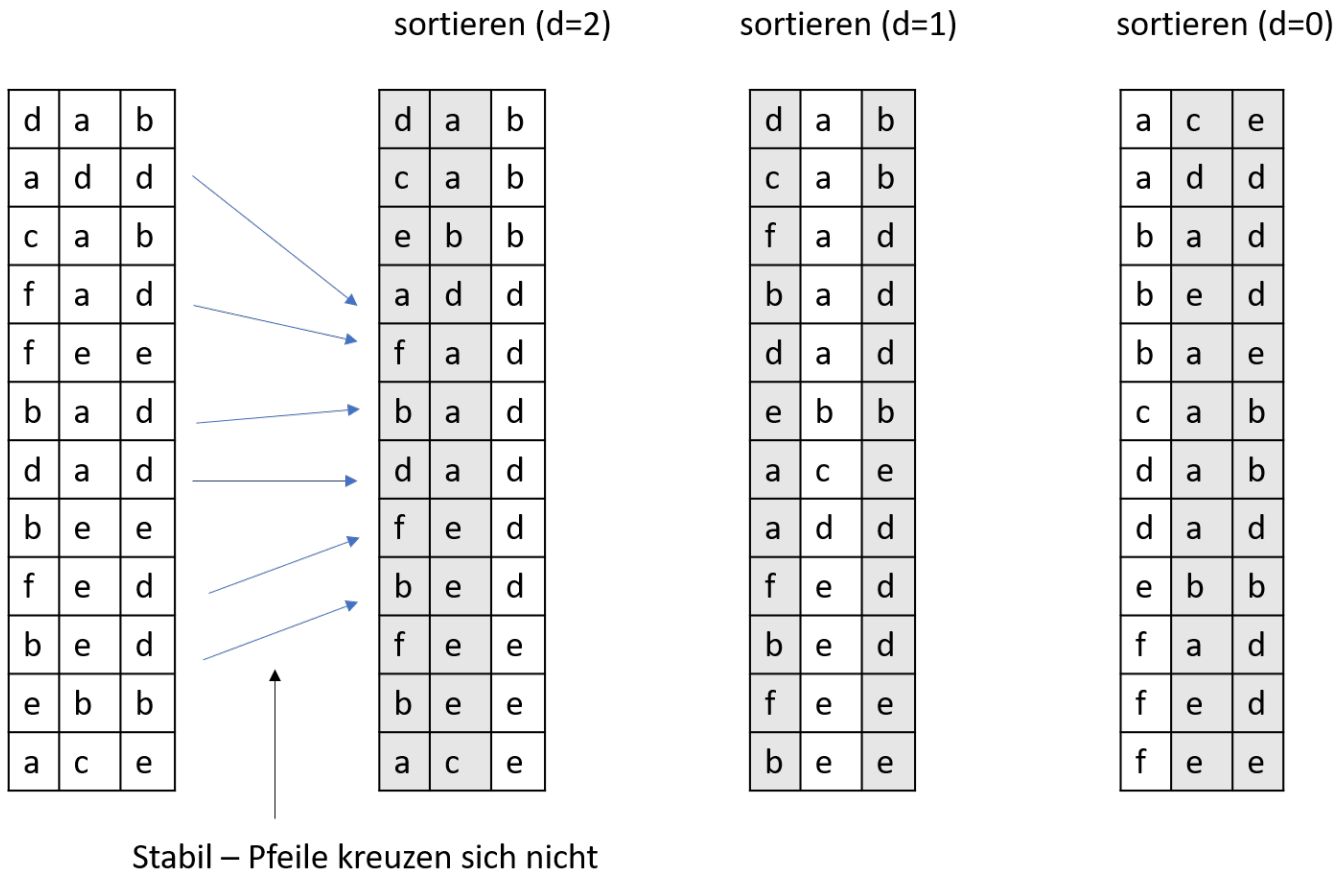
```
# Verteilen
for i in range(0, N):
    indexOfchar = alphabet.toIndex(a[i])
    countForChar = count[indexOfchar]
    aux[countForChar] = a[i]
    count[indexOfchar] += 1
```



LSD-Sortierverfahren (1 Zeichen)

- Verfahren ist stabil
- Zeitaufwand: Proportional zu $N + R$, wobei R Grösse des Alphabets ist
- Speicher: Proportional zu $N + R$ (aux-Array und count Array)

LSD-Sortierverfahren



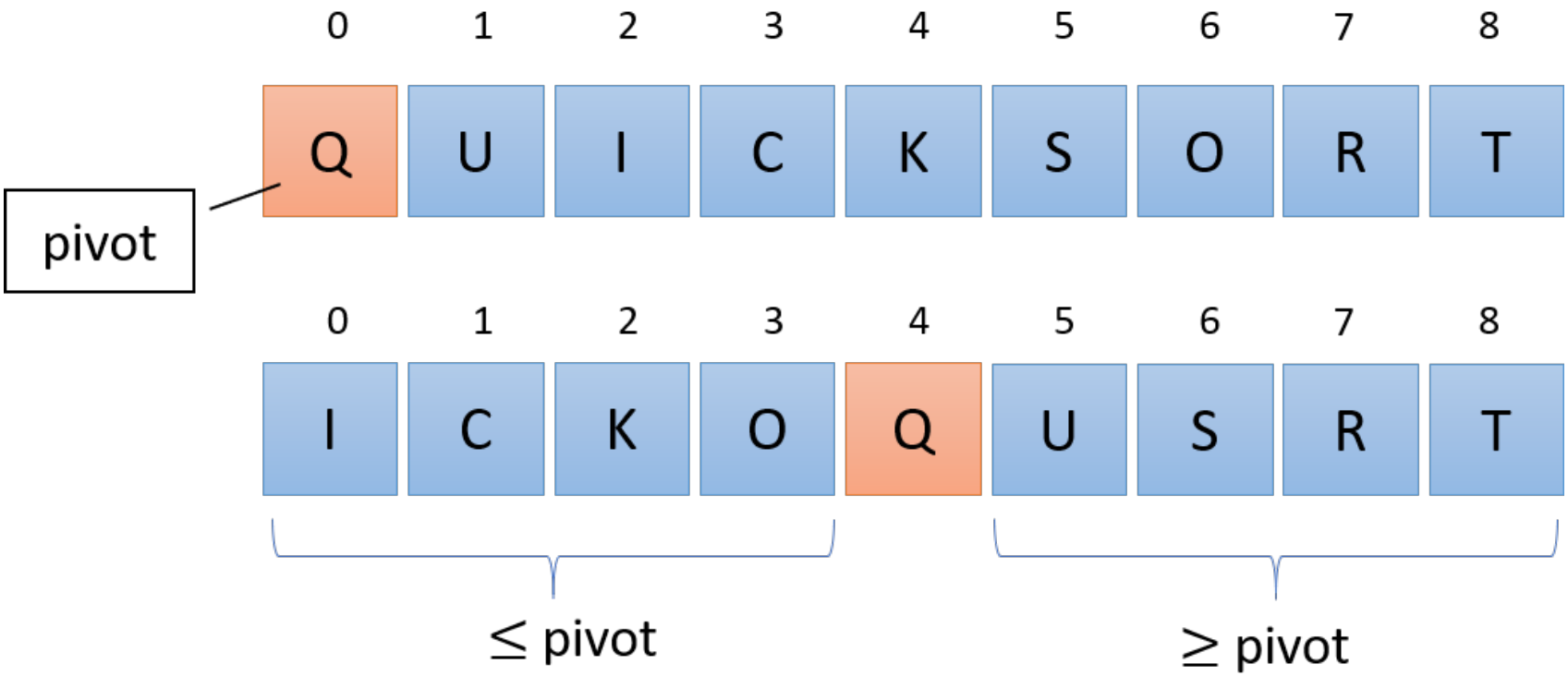
- Sortiere jedes Zeichen einzeln
  beginnend mit letztem (least significant digit)
- Funktioniert, da Sortierung stabil ist

LSD-Sortierverfahren

```
N = len(a);  aux = [None] * N ;   d = numDigits - 1
while d >= 0:
    count = [0] * (alphabet.radix() + 1)

    for i in range(0, N):
        indexOfcharAtPosdInA = alphabet.toIndex(a[i][d])
        count[indexOfcharAtPosdInA + 1] += 1

    for r in range(0, alphabet.radix()):
        count[r+1] += count[r]

    for i in range(0, N):
        indexOfCharAtPosdInA = alphabet.toIndex(a[i][d])
        countForChar = count[indexOfCharAtPosdInA]
        aux[countForChar] = a[i]
        count[indexOfCharAtPosdInA] += 1

    for i in range(0, N):
        a[i] = aux[i]
    d -= 1
```

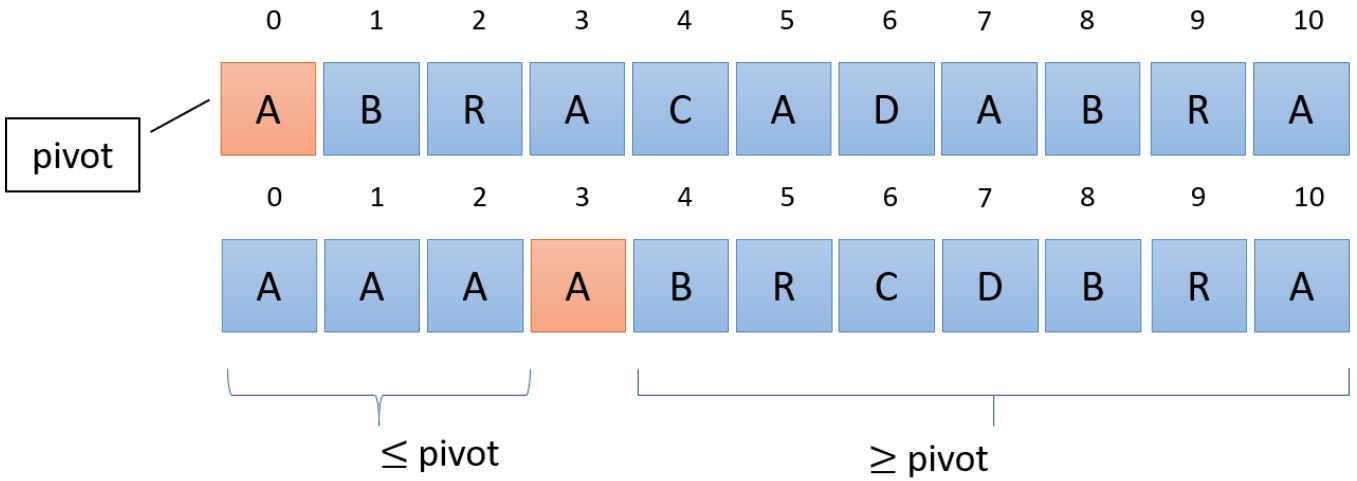
D1.3 Quicksort

Erinnerung: Quicksort



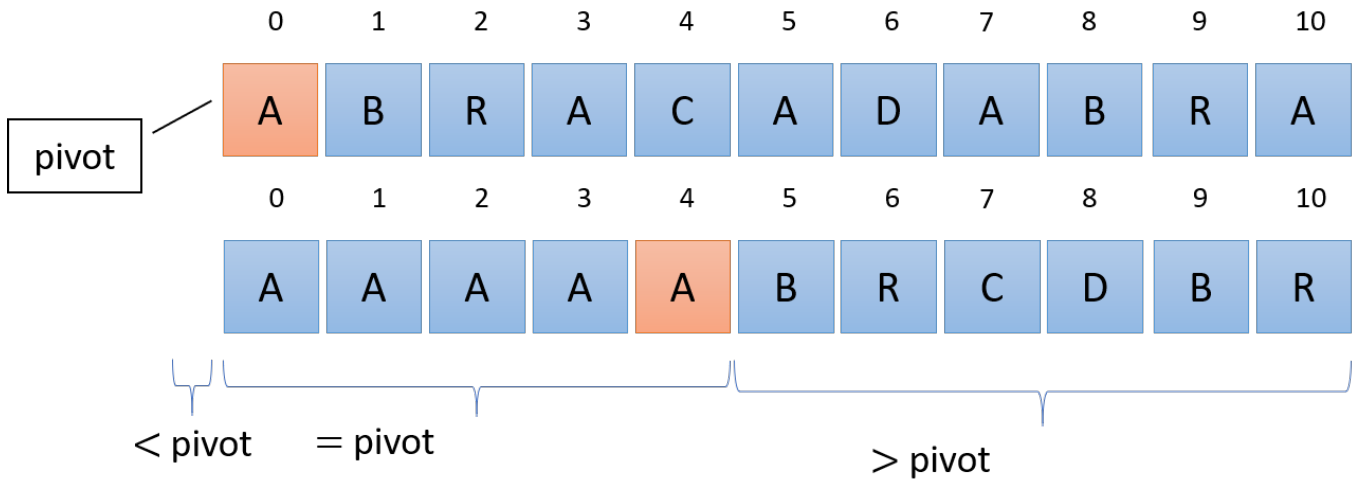
- Wähle Pivot Element
- Partitioniere Array
- Rekursion auf linkes und rechtes Teilarray

Quicksort: Gleiche Schlüssel



- Was passiert bei vielen gleichen Schlüsseln?
- Unnötige Partitionierung von gleichen Schlüsseln.

3-Wege Quicksort

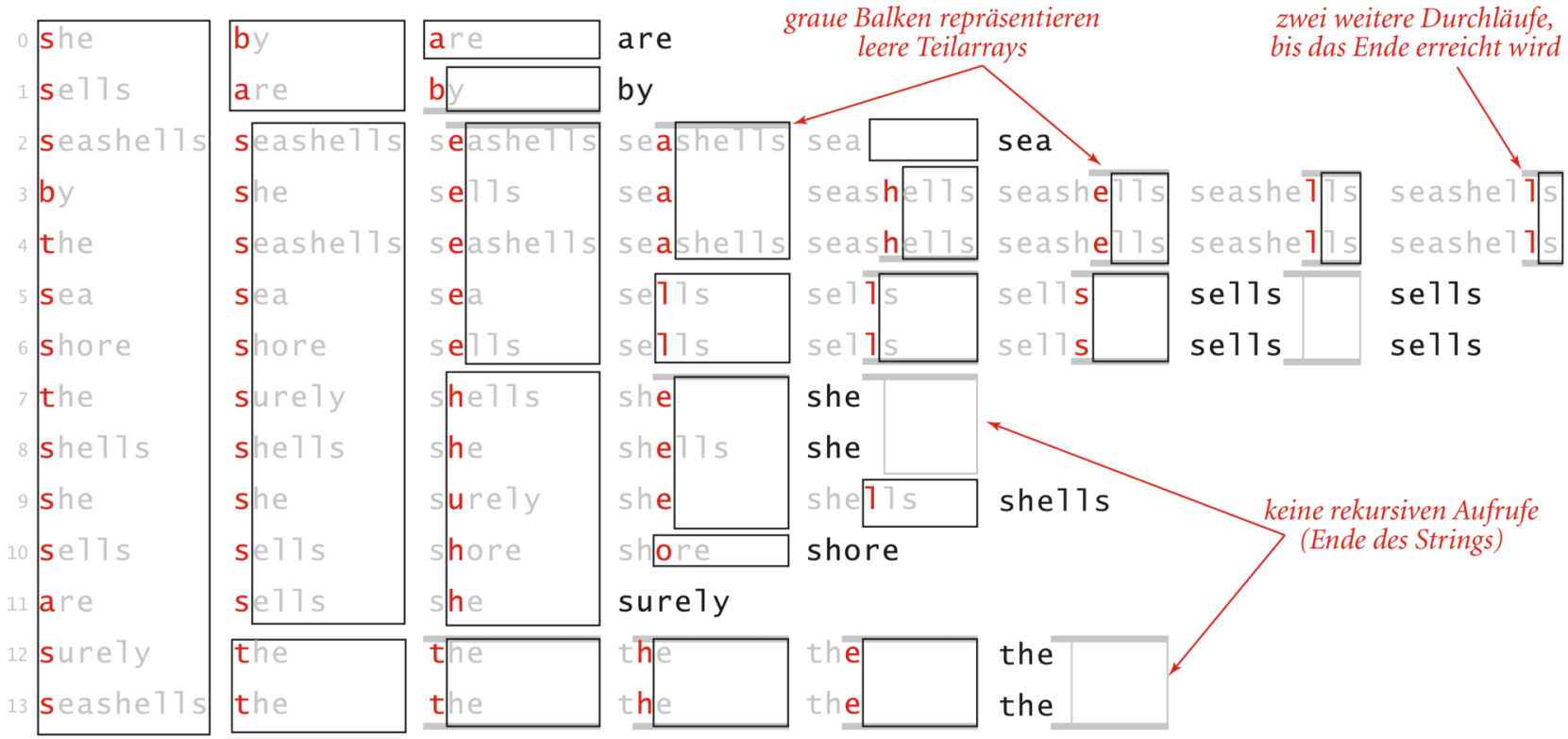


- Gleiche Schlüssel sind bereits sortiert.
  - Kein rekursiver Aufruf mehr nötig.

Quicksort für Strings

- 3-Wege Quicksort per Buchstabe
- Bei gleichen Anfangsbuchstaben, vergleiche nächsten Buchstaben.

Quicksort für Strings



Quelle: Sedgewick & Wayne, Algorithmen, Abbildung 5.18

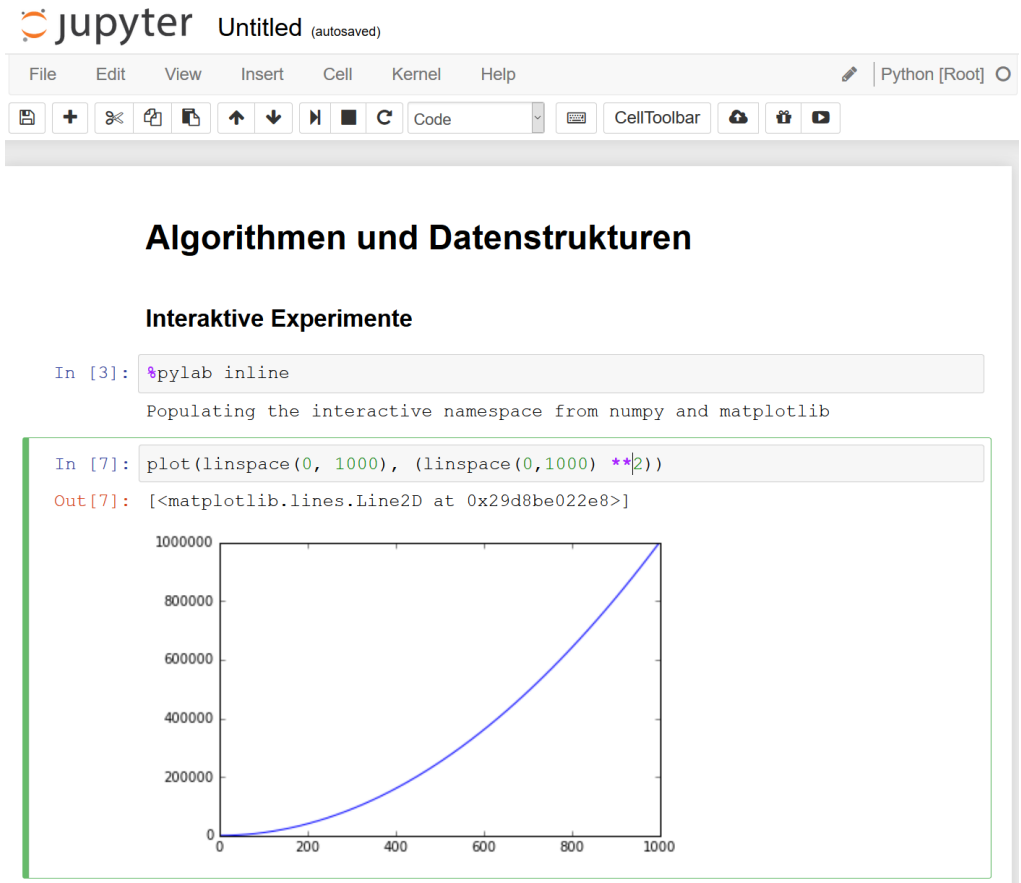
Laufzeit

Theorem

Um ein Array von N zufälligen Strings zu sortieren, benötigt der 3-Weg-Quicksort für Strings im Durchschnitt $\sim 2N\ln N$ Zeichenvergleiche.

- Gleiche Anzahl Vergleiche wie standard (3-Wege) Quicksort
- Aber: Wir haben Zeichenvergleiche und nicht Schlüsselvergleiche

Implementation



Jupyter Notebooks: Stringsort.ipynb

D1.4 Tries

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
```

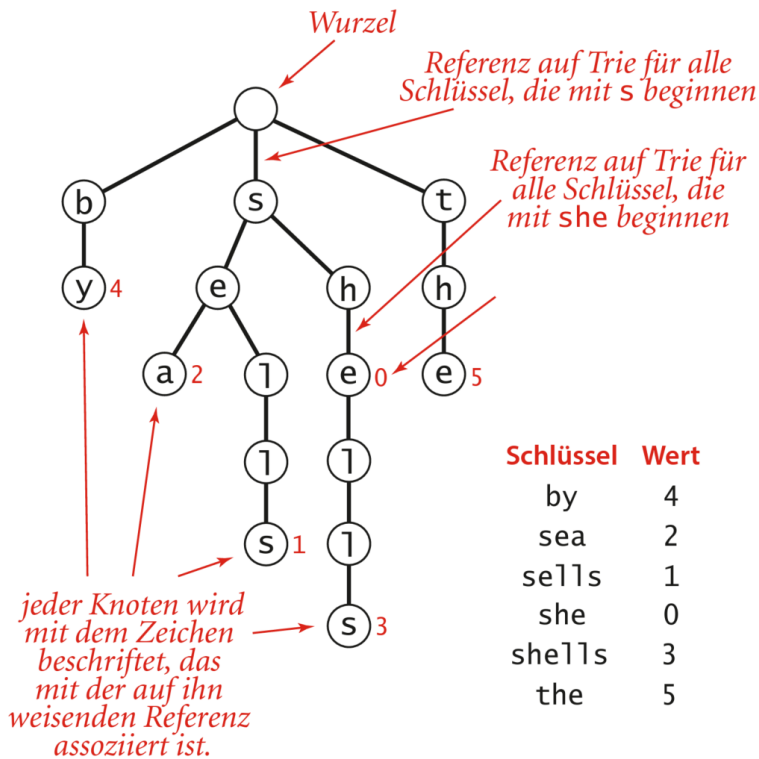
Mittels Tries lassen sich viele nützliche, zeichenbasierte Suchoperationen definieren.

Tries

Trie Von Retrieval.

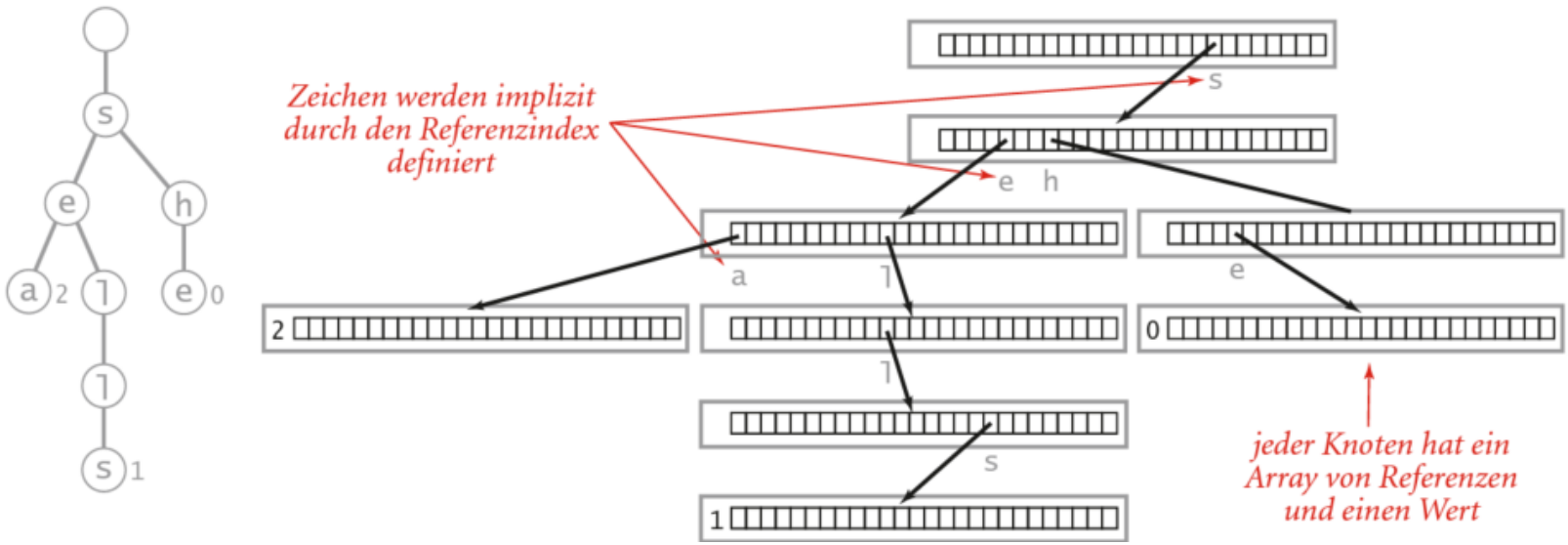
- Ausgesprochen wie try

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



Quelle: Sedgewick & Wayne, Algorithmen, Abbildung 5.19

Repräsentation der Knoten



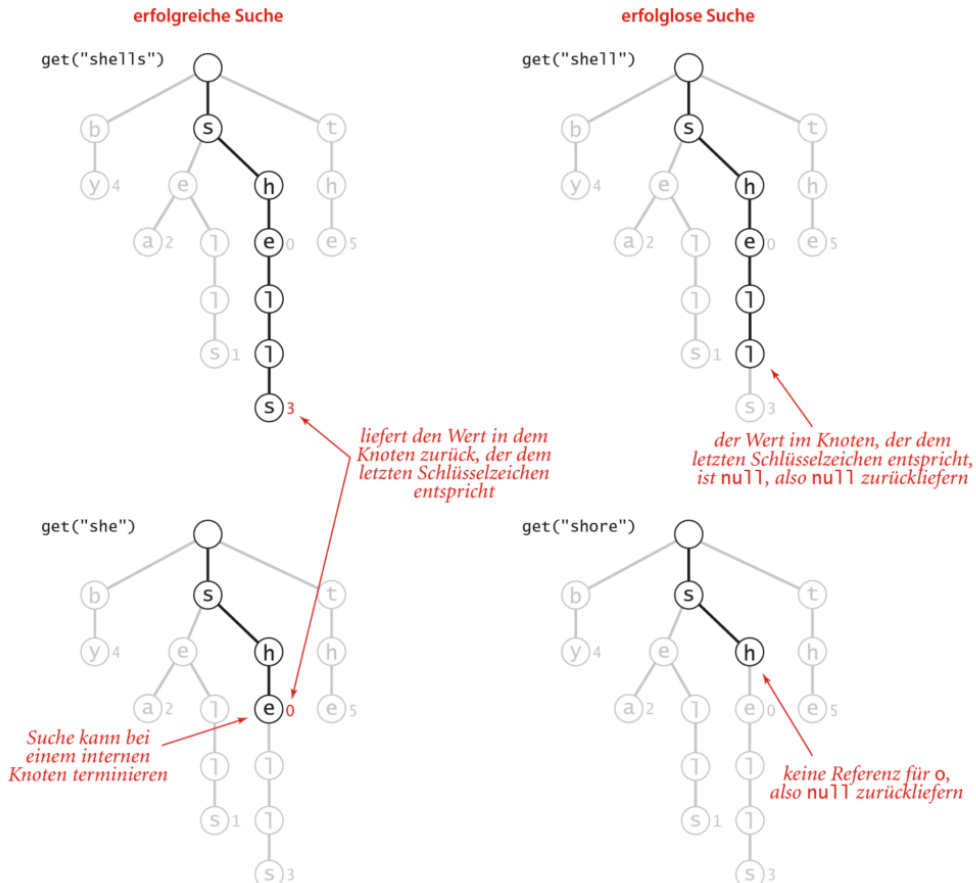
Quelle: Sedgewick & Wayne, Algorithmen, Abbildung 5.21

```
class Node:
    value = None
    children = [None] * alphabet.radix()
```

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

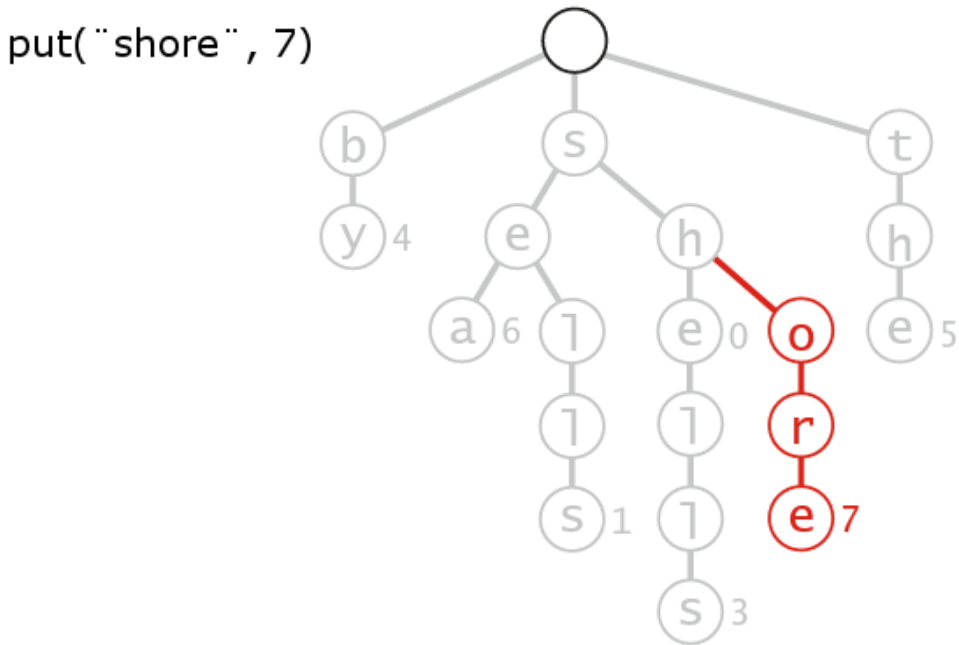
Suche in Tries

```
def get(key):
    node = get_rec(root, key, 0)
    if (node == None):
        return None
    else:
        return node.value

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

Einfügen in Trie

Dem Zeichen entsprechenden Link folgen

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



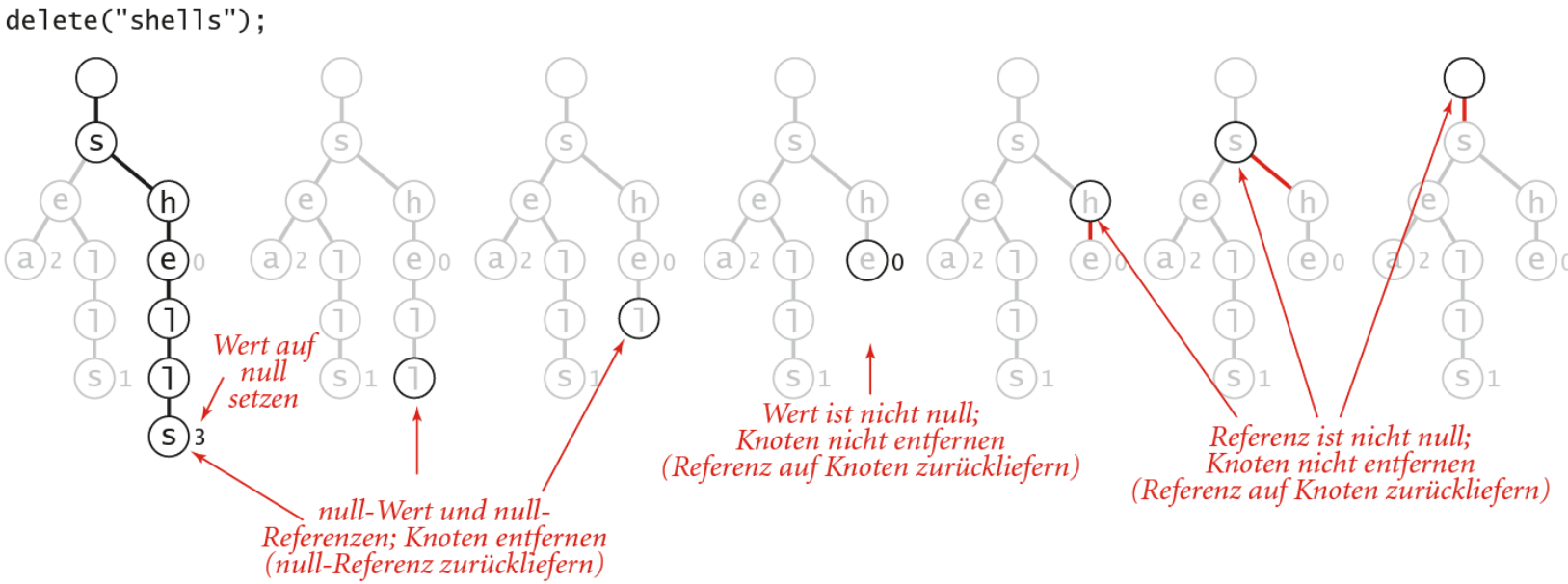
Quelle: Sedgewick & Wayne, Algorithmen, Abbildung 5.22

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

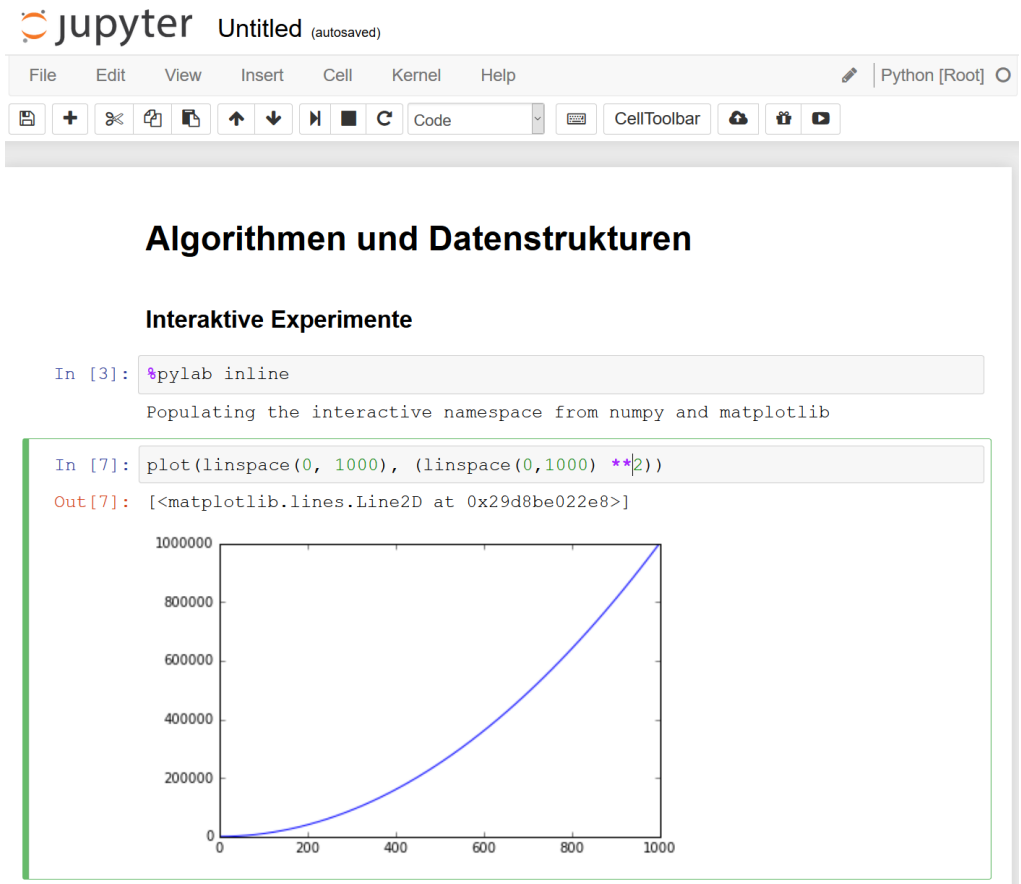


Quelle: Sedgewick & Wayne, Algorithmen, Abbildung 5.26

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 $\sim \log_R(N)$.

Tries: Take-Home Message

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

Zusammenfassung

- String-Algorithmen nutzen Struktur von Strings aus
  - Vorteil: Erlaubt Implementation von schnelleren Algorithmen
  - Nachteil: Algorithmen nur für Schlüssel mit Typ String anwendbar.
- Strategie und Konzepte von bestehenden Algorithmen wurde übernommen.

- LSD-Sort: Radixsort angewendet auf Zeichen
- Quicksort für Strings: Macht Vergleich für jedes Zeichen
- Tries: Suchbaum der durch Zeichen indiziert ist