Informatik II - SS 2014

(Algorithmen & Datenstrukturen)

Vorlesung 7 (21.5.2014)

Binäre Suche, Hashtabellen I

UNI FREIBURG

Fabian Kuhn

Algorithmen und Komplexität

Abstrakte Datentypen : Dictionary

UNI FREIBURG

Dictionary: (auch: Maps, assoziative Arrays, Symbol Table)

- Verwaltet eine Kollektion von Elementen, wo bei jedes Element durch einen eindeutigen Schlüssel (key) repräsentiert wird

Operationen:

- *create* : erzeugt einen leeren Dictionary
- *D.insert(key, value)* : fügt neues *(key,value)*-Paar hinzu
  - falls schon ein Eintrag für *key* besteht, wird er ersetzt
- *D.find(key)* : gibt Eintrag zu Schlüssel *key* zurück
  - falls ein Eintrag vorhanden (gibt sonst einen Default-Wert zurück)
- *D.delete(key)* : löscht Eintrag zu Schlüssel *key*

Dictionary

UNI FREIBURG

- Wir kümmern uns in einer ersten Phase nur um die Basisoperationen *insert, find, delete* (und *create*)

Dictionary Beispiele:

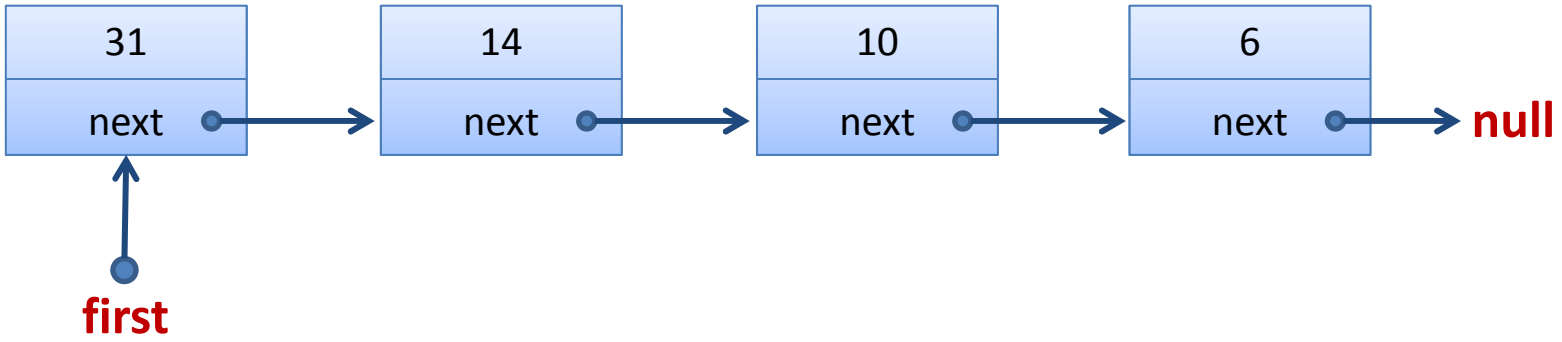
- Wörterbuch (key: Wort, value: Definition / Übersetzung)
- Telefonbuch (key: Name, value: Telefonnummer)
- DNS Server (key: URL, value: IP-Adresse)
- Python Interpreter (key: Variablenname, value: Wert der Variable)
  Java/C++ Compiler (key: Variablenname, value: Typinformation)

In all diesen Fällen ist insbesondere eine schnelle *find*-Op. wichig!

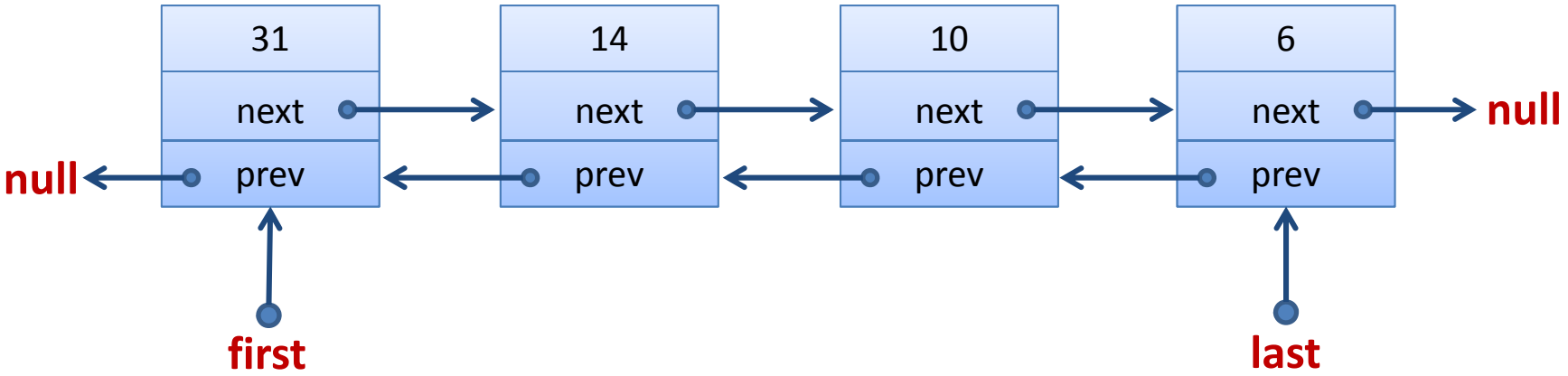
Verkettete Listen: Struktur

UNI FREIBURG

Einfach verkettete Liste (Singly Linked List):



Doppelt verkettete Liste (Doubly Linked List):

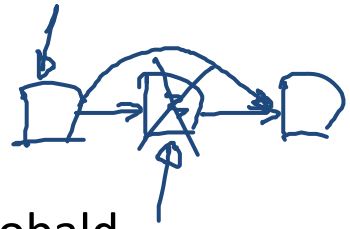


Dictionary mit verketteten Listen

UNI FREIBURG

Operationen:

- *create*:
  - lege neue leere Liste an
- *D.insert(key, value)*:
  - füge neues Element vorne ein
  - Annahme: Es gibt noch keinen Eintrag mit dem Schlüssel *key* (notwendig?)
- *D.find(key)*:
  - gehe von vorne durch die Liste
- *D.delete(key)*:
  - suche zuerst das Listenelement (wie in *find*)
  - lösche Element dann aus der Liste
  - Bei einfach verketten Listen muss man stoppen, sobald *current.next.key == key* ist!



Dictionary mit verketteten Listen

UNI FREIBURG

Laufzeiten:

create: $O(1)$

insert: $O(1)$

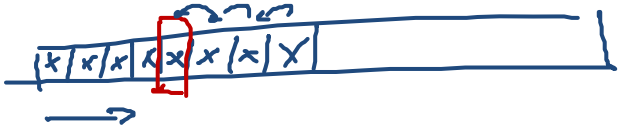
find: $O(n)$

delete: $O(n)$

Ist das gut?

Dictionary mit Array

Operationen:



- *create*:
  - lege neues Array der Länge *NMAX* an
- *D.insert(key, value)*:
  - füge neues Element hinten an (falls es noch Platz hat)
  - Annahme: Es gibt noch keinen Eintrag mit dem Schlüssel *key*
- *D.find(key)*:
  - gehe von vorne (oder hinten) durch die Elemente
- *D.delete(key)*:
  - suche zuerst nach dem *key*
  - lösche Element dann aus dem Array:

    Man muss alles dahinter um eins nach vorne schieben!

Dictionary mit Array

UNI FREIBURG

Laufzeiten:

create: $O(1)$

insert: $O(1)$

find: $O(n)$

delete: $O(n)$

Bessere Ideen?

Benutze sortiertes Array?

UNI FREIBURG

- Teure Operation bei Liste/Array, insbesondere ***find***
- Falls (sobald) sich die Einträge nicht zu sehr ändern, ist *find* die wichtigste Operation!
- Kann man in einem (nach Schlüsseln) sortierten Array schneller nach einem bestimmten Schlüssel suchen?
  - Beispiel: Suche Tel.-Nr. einer Person im Telefonbuch…

Ideen:

Gehe in die Mitte
und dann entweder links oder rechts

Binäre Suche

Benutze Divide and Conquer Idee!

Suche nach der Zahl (dem Key) 19:

2	3	4	6	9	12	15	16	17	18	19	20	24	27	29

$16 \overset{?}{\leq} 19$

$20 > 19$

Binäre Suche

2	3	4	6	9	12	15	16	17	18	19	20	24	27	29

Algorithmus (Array A der Länge n, Suche nach Schlüssel x):

```
ℓ = 0; r = n-1
while r > ℓ  do
    m = ⌊(ℓ+r)/2⌋
    if A[m] < x then    (x ist rechts von m)
        ℓ = m+1
    else if A[m] > x then    (x ist links von m)
        r = m-1
    else
        ℓ = m; r = m
```

$A[\ell] \stackrel{?}{=} x$

Binäre Suche

2	3	4	6	9	12	15	16	17	18	19	20	24	27	29

Algorithmus (Array A der Länge n, Suche nach Schlüssel x):

```
l = 0; r = n – 1;
while r > l do
    m = (l + r) / 2;
    if A[m] < x then
        l = m + 1
    else if A[m] > x then
        r = m – 1
    else
        l = m; r = m
```

Falls Schlüssel x im Array ist, dann gilt am Schluss $A[l] = x$

Ist der Algorithmus korrekt?

UNI FREIBURG

Wie überprüft man das?

- Empirisch: Unit Test oder auch systematischere Tests…
- Formal?
  - Korrektheit ist (meistens) noch wichtiger als Performance!

- **Vorbedingung**
  - Bedingung, welche am Anfang (der Methode / Schleife / …) gilt
- **Nachbedingung**
  - Bedingung, welche am Schluss (der Methode / Schleife / …) gilt
- **Schleifeninvariante**
  - Bedingung welche am Anfang / Ende jedes Schleifendurchlaufs gilt

Ist der Algorithmus korrekt?

```
l = 0; r = n - 1;
while r > l do
    m = (l + r) / 2;
    if A[m] < x then l = m + 1
    else if A[m] > x then r = m - 1
    else l = m; r = m
```

Vorbedingung

- *Array ist am Anfang sortiert, Array hat Länge n*

Nachbedingung

- *Falls x im Array ist, dann gilt $A[l] = x$*

Schleifeninvariante

- *Falls x im Array ist, dann gilt $A[l] \leq x \leq A[r]$*

Ist der Algorithmus korrekt?

while (r>l) do ...

Vorbedingung

- *Array ist am Anfang sortiert, Array hat Länge n*

```
l = 0; r = n - 1;
```

Schleifeninvariante

$A[0] \leq x \leq A[n-1]$

- *Falls x im Array ist, dann gilt $A[l] \leq x \leq A[r]$*
- Vorbedingung und Zuweisung zu l und r → Schleifeninvariante
  - Invariante gilt am Anfang des ersten Schleifendurchlaufs

Nachbedingung

$A[r] \leq A[l] \leq x \leq A[r]$

A ist sortiert

- *Falls x im Array ist, dann gilt $A[l] = x$*
- Abbruchbedingung while-Schleife → $l \geq r$ und damit $A[l] \geq A[r]$
- Falls x im Array ist, dann folgt aus der Schleifeninvariante und da A sortiert ist, dass $A[l] = A[r]$ und damit $A[l] = x$

Ist der Algorithmus korrekt?

```
l = 0; r = n - 1;
while r > l do
    m = (l + r) / 2;
    if A[m] < x then l = m + 1
    else if A[m] > x then r = m - 1
    else l = m; r = m
```

$A[l] \leq x \leq A[r]$

INV

INV

Schleifeninvariante

- *Falls x im Array ist, dann gilt $A[l] \leq x \leq A[r]$*

$l = m+1$: $l \; m \; r$, $A[m] < x$

$r = m-1$: analog

Terminiert der Algorithmus?

```
l = 0; r = n - 1;
while r > l do
    m = (l + r) / 2;
    if A[m] < x then l = m + 1
    else if A[m] > x then r = m - 1
    else l = m; r = m
```

$\frac{l+r}{2} - \frac{1}{2} \leq m \leq \frac{l+r}{2}$

- Veränderung der Anz. Elemente $(r - l + 1)$ pro Schleifendurchlauf?
  - $l = m + 1$: (neues l)

$$r - (m+1) + 1 \leq r - \left(\frac{l+r}{2} + \frac{1}{2}\right) + 1 = \frac{r-l+1}{2}$$

  - $r = m - 1$:

$$(m-1) - l + 1 \leq \frac{l+r}{2} - 1 - l + 1 = \frac{r-l}{2} < \frac{r-l+1}{2}$$

  - Sonst wird x gefunden und $r - l + 1$ wird 1

Laufzeit

Terminiert der Algorithmus?

- In jedem Schleifendurchlauf wird die Anzahl der Elemente mindestens halbiert.
- Der Algorithmus terminiert!

Laufzeit?

$$T(n) \leq T(\lceil n/2 \rceil) + c, \qquad T(1) \leq c$$

$$\begin{aligned} T(n) &\leq T(n/2) + c \\ &\leq T(n/4) + 2c \\ &\leq T(n/8) + 3c \\ &\vdots \\ &\leq T(n/2^k) + k \cdot c \\ &\vdots \\ &\leq T(1) + c \cdot \log_2 n \leq c(1 + \log n) \end{aligned}$$

Laufzeit Binäre Suche

Der Algorithmus terminiert in Zeit $O(\log n)$.

$T(n) \leq T(\lfloor n/2 \rfloor) + c$ $\qquad$ $T(n) \leq c(1+\log n)$

Verankerung: $T(1) \leq c(1+\log 1) = c$ ✓

Schritt: $T(n) \leq T(\lfloor n/2 \rfloor) + c$

$\leq c(1+\log \lfloor n/2 \rfloor) + c$

$\leq c(1+\log(n/2)) + c$

$= c(1+\log_2 n)$ ✓

$\log_2(n/2) = \log_2 n - 1$

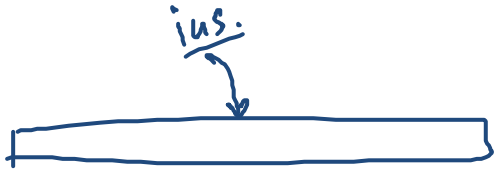
sind in Zeit $O(\log n)$

Dictionary mit sortiertem Array

UNI FREIBURG

Operationen:

- *create*:
  - lege neues Array der Länge *NMAX* an
- *D.find(key)*:
  - **Suche nach *key* mit binärer Suche**
- *D.insert(key, value)*:
  - suche nach *key* und füge neues Element an der richtigen Stelle ein
  - Einfügen: alles dahinter muss um eins nach hinten geschoben werden!



- *D.delete(key)*:
  - suche zuerst nach dem *key* und lösche den Eintrag
  - Löschen: alles dahinter muss um eins nach vorne geschoben werden!

Dictionary mit sortiertem Array

UNI FREIBURG

Laufzeiten:

create: $O(1)$

insert: $O(n)$

find: $O(\log n)$

delete: $O(n)$

Können wir alle Operationen schnell machen?

- und das *find* noch schneller?

Direkte Adressierung

Mit einem Array können alles schnell machen,

...falls das Array gross genug ist.

Annahme: Schlüssel sind ganze Zahlen zwischen 0 und $M-1$

1	null
2	null
3	**Value 1**
4	null
5	null
6	null
7	**Sebastian**
8	**Value 3**
9	null
⋮	⋮
M	null

find(3) → *"Value 1"*

insert(7, "Sebastian")

delete(5)

alles $O(1)$

Direkte Adressierung : Probleme

UNI FREIBURG

1. Direkte Adressierung benötigt zu viel Platz!

– Falls Schlüssel ein beliebiger *int* (32 bit) sein kann:

Wir benötigen ein Array der Grösse $2^{32} \approx 4 \cdot 10^9$.

Bei 64 bit Integers sind's sogar schon mehr als 10^{19}…

2. Was tun, wenn die Schlüssel keine ganzen Zahlen sind?

– Wo kommt das *(key,value)*-Paar *("Sebastian", "Assistent")* hin?

– Wo soll der Schlüssel 3.14159 gespeichert werden?

– Pythagoras: "Alles ist Zahl"

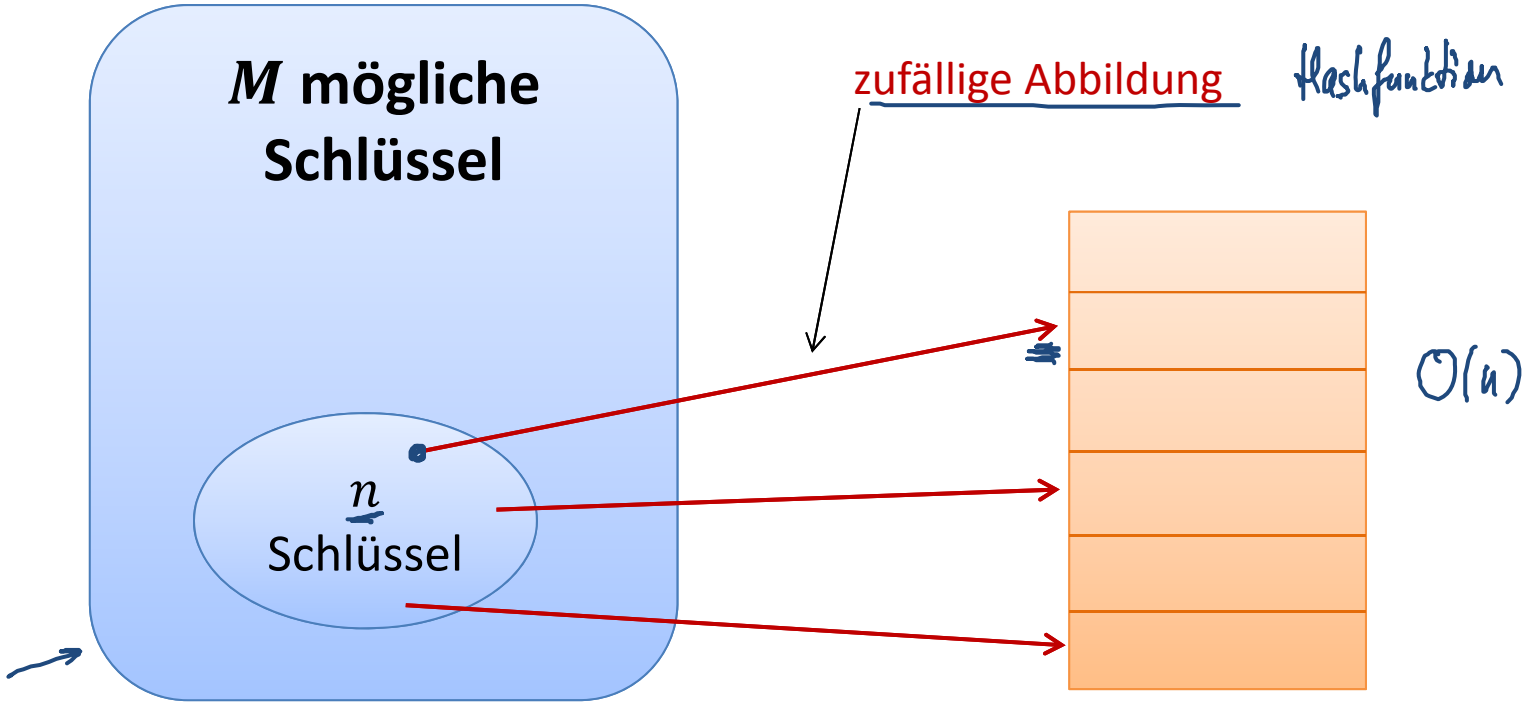
"Alles" kann als Folge von Bits abgespeichert werden:
Interpretiere Bit-Folge als ganze Zahl

– **Verschärft das Platz-Problem noch zusätzlich!**

Hashing : Idee

Problem

- Riesiger Raum S an möglichen Schlüsseln
- Anzahl n der wirklich benutzten Schlüssel ist **viel** kleiner
  - Wir möchten nur Arrays der Grösse $\approx n$ (resp. $O(n)$) verwenden…
- Wie können wir M Schlüssel auf $O(n)$ Array-Positionen abbilden?



Hashfunktionen

UNI FREIBURG

Schlüsselraum S, $|S| = M$ (alle möglichen Schlüssel)

Arraygrösse m ($\approx$ Anz. Schlüssel, welche wir max. speichern wollen)

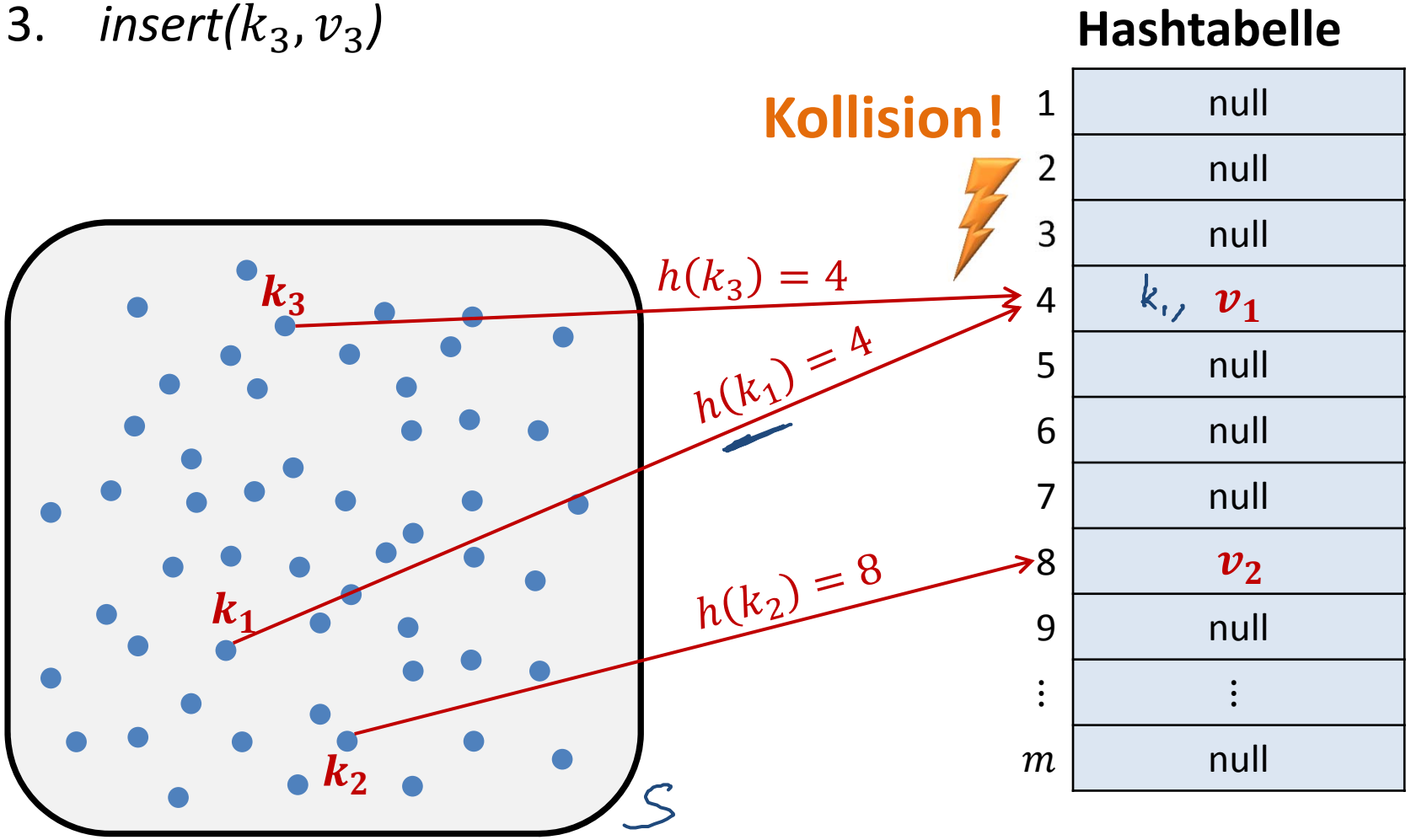
Hashfunktion

$$h: S \to \{0, \dots, m-1\}$$

- Bildet Schlüssel vom Schlüsselraum S in Arraypositionen ab
- h sollte möglichst nahe bei einer zufälligen Funktion sein
  - alle Elemente in $\{0, \dots, m-1\}$ etwa gleich vielen Schlüsseln zugewiesen sein
  - ähnliche Schlüssel sollten auf verschiedene Positionen abgebildet
- h sollte möglichst schnell berechnet werden können
  - Wenn möglich in Zeit $O(1)$
  - Wir betrachten es im folgenden als Grundoperation (Kosten = 1)

Funktionsweise Hashtabellen

1. $insert(k_1, v_1)$
2. $insert(k_2, v_2)$
3. $insert(k_3, v_3)$



Hashtabellen : Kollisionen

UNI FREIBURG

Kollision:

Zwei Schlüssel k_1, k_2 kollidieren, falls $h(k_1) = h(k_2)$.

Was tun bei einer Kollision?

- Können wir Hashfunktionen wählen, bei welchen es keine Kollisionen gibt? geht nicht
  wenn wir die Schlüssel kennen geht's
- Eine andere Hashfunktion nehmen? Größe m
  müssten alles neu einfügen
- Weitere Ideen?

Hashtabellen : Kollisionen

UNI FREIBURG

Kollisionen Lösungsansätze

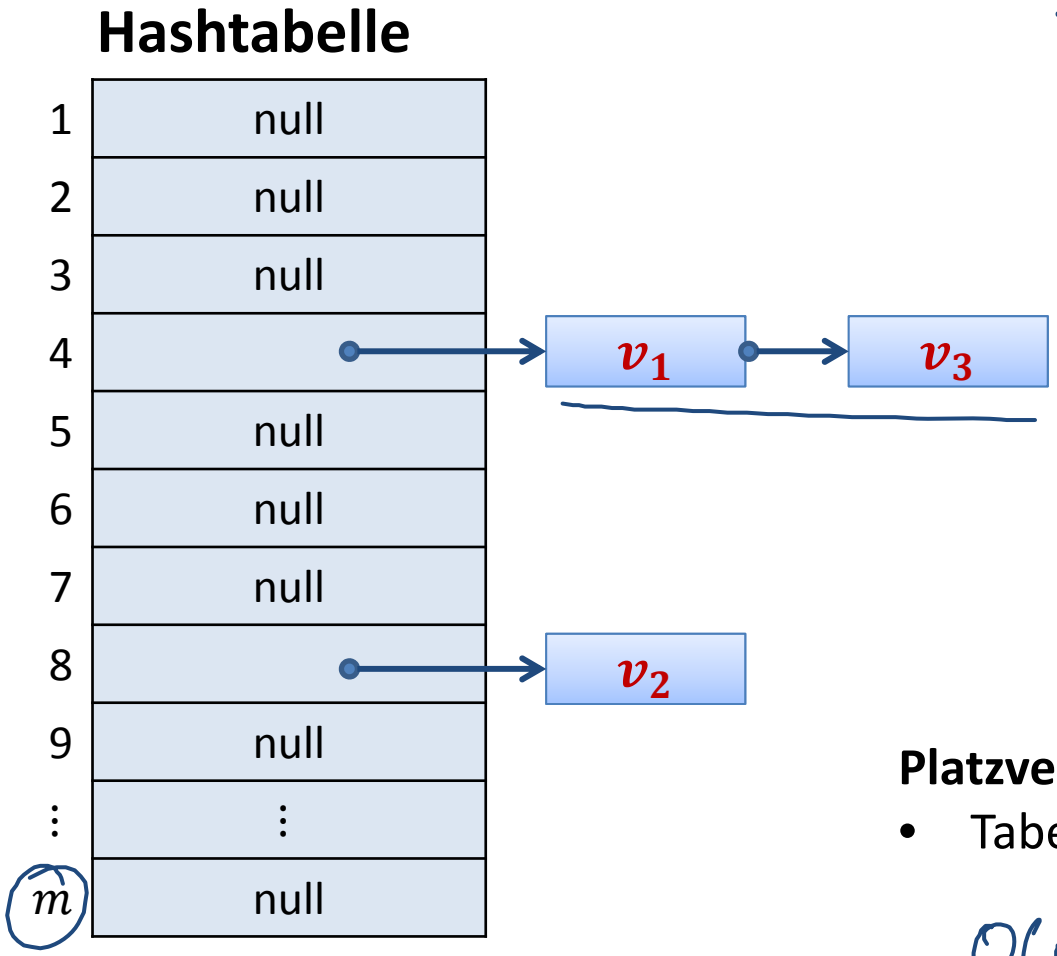
- Annahme: Schlüssel k_1 und k_2 kollidieren

1. Speichere beide (key,value)-Paare an die gleiche Stelle
   - Die Hashtabelle muss an jeder Position Platz für mehrere Elemente bieten
   - Wir wollen die Hashtabelle aber nicht einfach vergrössern (dann könnten wir gleich mit einer grösseren Tabelle starten…)
   - **Lösung: Verwende verkettete Listen**

2. Speichere zweiten Schlüssel an eine andere Stelle
   - Kann man zum Beispiel mit einer zweiten Hashfunktion erreichen
   - Problem: An der alternativen Stelle könnte wieder eine Kollision auftreten
   - Es gibt mehrere Lösungen
   - Eine Lösung: Verwende viele mögliche neue Stellen (Man sollte sicherstellen, dass man die meistens nicht braucht…)

Hashtabellen mit Chaining

- Jede Stelle in der Hashtabelle zeigt auf eine verkette Liste



insert: key x

gehe zur Stelle $h(x)$

find:

gehe zur Stelle $h(x)$

suche in Liste

delete:

analog

Platzverbrauch:

- Tabellengrösse m, Anz. Elemente n

$O(m+n)$

Laufzeit Hashtabellen-Operationen

UNI FREIBURG

Zuerst, um's einfach zu machen, für den Fall ohne Kollisionen…

create: $O(1)$

insert: $O(1)$

find: $O(1)$

delete: $O(1)$

- Solange keine Kollisionen auftreten, sind Hashtabellen extrem schnell (falls die Hashfunktion schnell ausgewertet werden kann)
- Wir werden sehen, dass dies auch mit Kollisionen gilt…

Laufzeit mit Chaining

Verkettete Listen an allen Positionen der Hashtabelle

create: $O(1)$

insert: $O(1)$

find: $O(1 + \text{"Länge der Liste an Stelle } h(x)\text{"})$

worst case: $O(n)$

delete: "

Funktionsweise Hashtabellen

Schlechtester Fall bei Hashing mit Chaining

- Alle Schlüssel, welche vorkommen, haben den gleichen Hashwert
- Ergibt eine verkettete Liste der Länge n
- Wahrscheinlichkeit bei zufälligem h: $\left(\frac{1}{m}\right)^{n-1}$