Algorithmen und Datenstrukturen

Vorlesung 12:
Hashing

Alexander Wolff

Wintersemester 2025

Hashing

Wörterbuch.

- Spezialfall einer dynamischen Menge
- Anwendung: im Compiler Symboltabelle für Schlüsselwörter

Abstrakter Datentyp.

stellt folgende Operationen bereit: INSERT, DELETE, SEARCH

Implementierung.

heute: Hashing

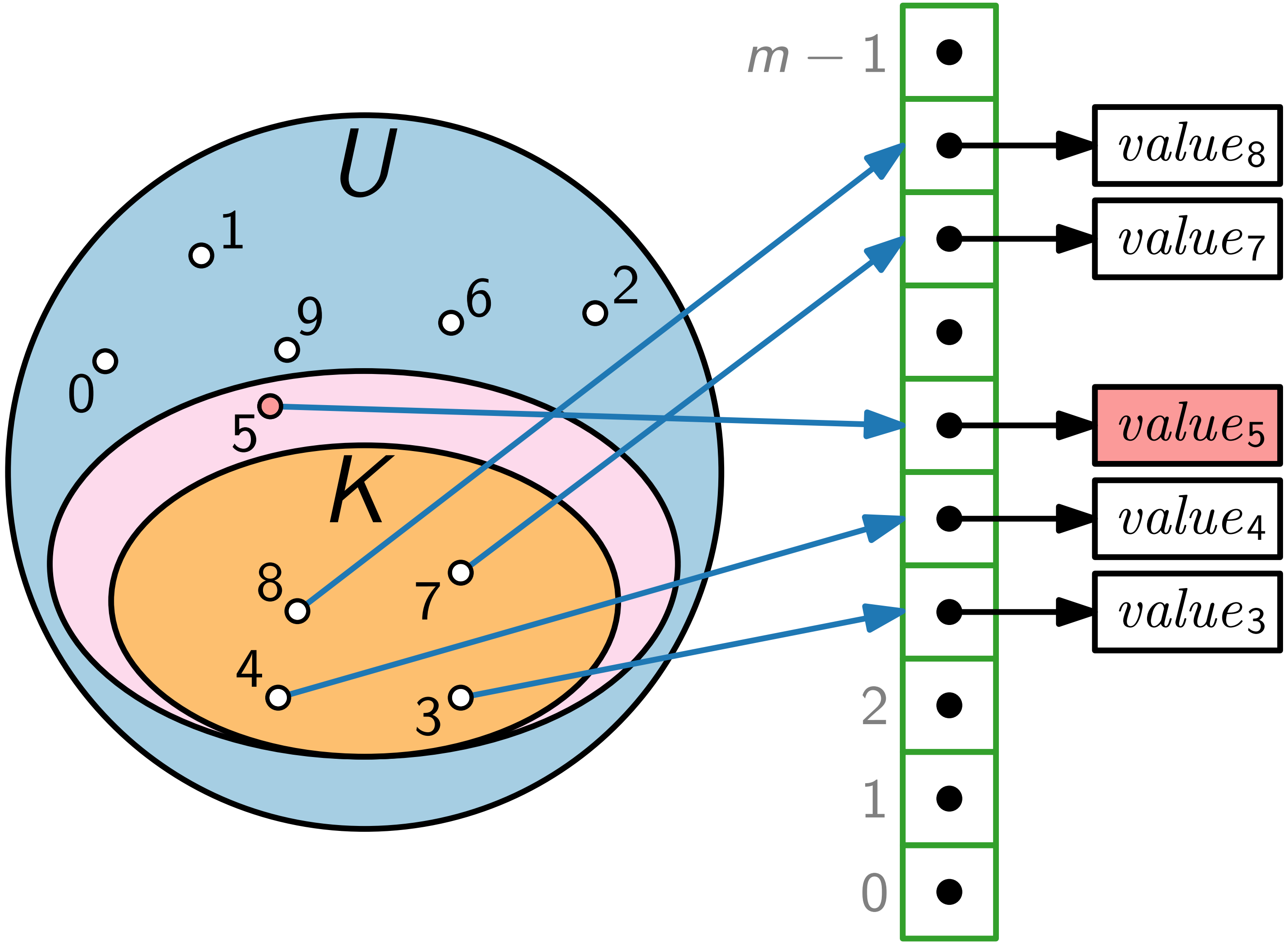
engl. to hash = zerhacken, kleinschneiden

Suchzeit:
- im schlechtesten Fall $\Theta(n)$,
- erwartet $\mathcal{O}(1)$ unter akzeptablen Annahmen

Direkte Adressierung

Annahmen.
- Schlüssel aus kleinem **Universum** $U = \{0, \ldots, m-1\}$
- Schlüssel paarweise verschieden

dyn. Menge!



Direkte Adressierung

Annahmen.
- Schlüssel aus kleinem **Universum** $U = \{0, \ldots, m-1\}$
- Schlüssel paarweise verschieden

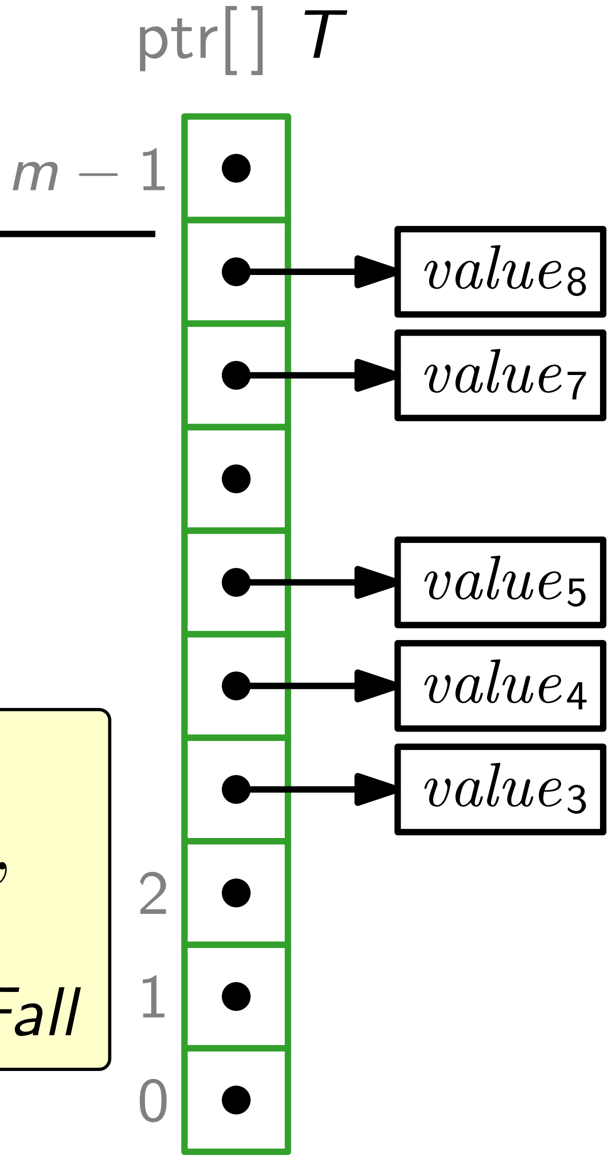
dyn. Menge!

Operation	Implementierung
$\textsc{HashDA}(\text{int } m)$	T = **new** value$[0 \ldots m-1]$ **for** $j = 0$ **to** $m-1$ **do** $T[j] = nil$ // $T[j]$ = Zeiger auf j. Datensatz

ptr $\textsc{Insert}(\text{key } k, \text{value } v)$	// lege neuen Datensatz an
	// und initialisiere ihn mit v
	$T[k]$ = **new** value(v)

$\textsc{Delete}(\text{key } k)$	// Speicher freigeben, auf $T[k] = nil$
ptr $\textsc{Search}(\text{key } k)$	**return** $T[k]$

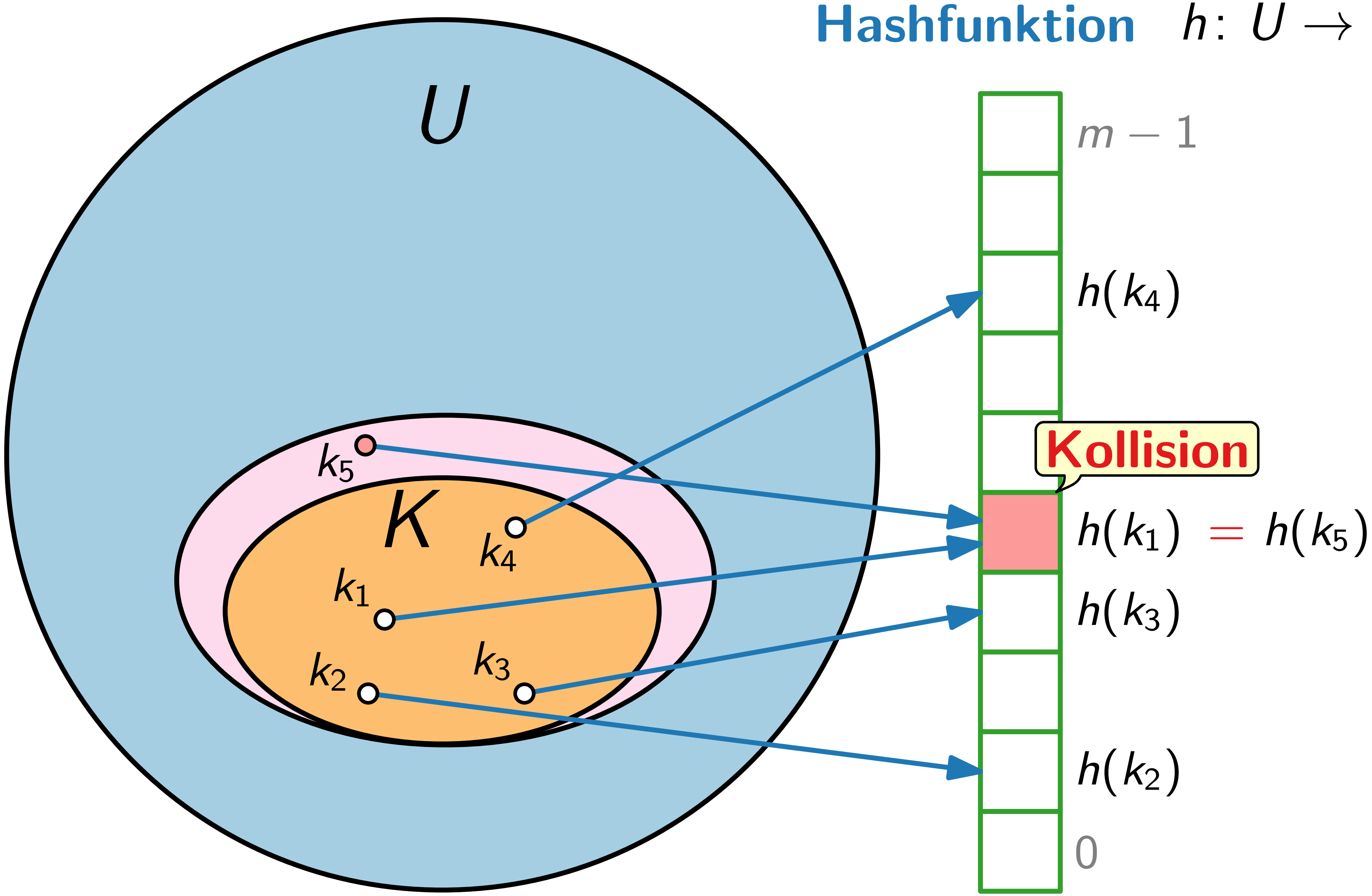
Laufzeiten?
$\textsc{Insert}$, $\textsc{Delete}$, $\textsc{Search}$ $\mathcal{O}(1)$
im *schlechtesten Fall*



Hashing mit Verkettung

Annahme. **großes** Universum U, d.h. $|U| \gg |K|$

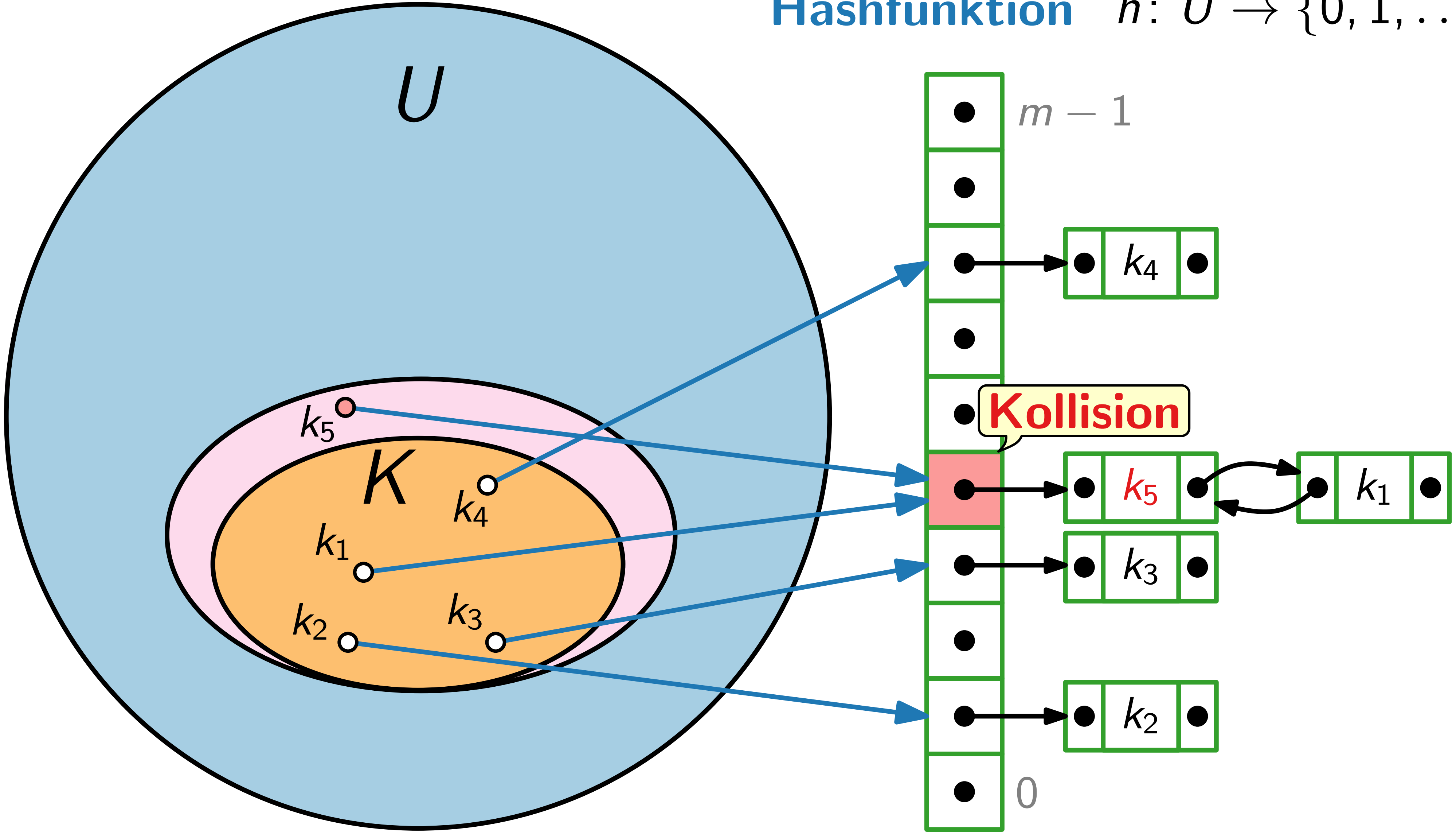
Hashfunktion $h\colon U \to \{0, 1, \dots, m-1\}$



Hashing mit Verkettung

Annahme. **großes** Universum U, d.h. $|U| \gg |K|$

Hashfunktion $h\colon U \to \{0, 1, \ldots, m-1\}$



Hashing mit Verkettung

Voraussetzungen:
- $|U| \gg |K|$
- Zugriff auf Hashfunktion h

Operation | **Implementierung**

HashChaining(int m)

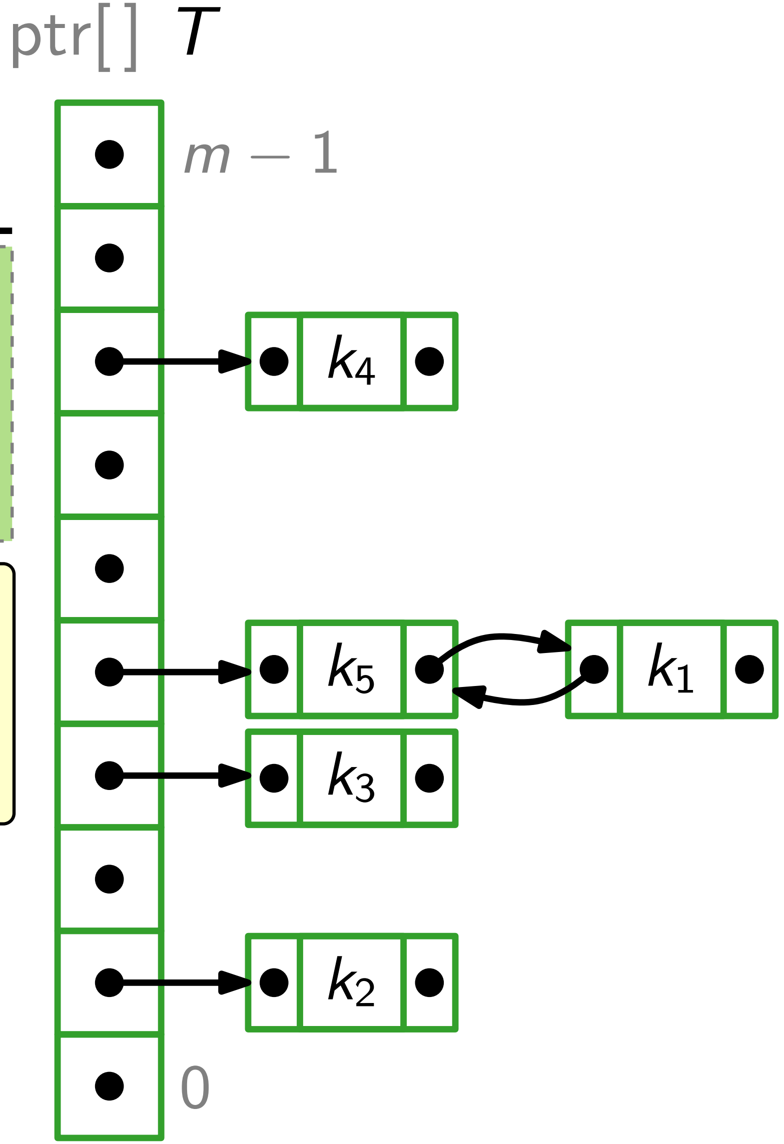
```
$T$ = new List[$0 \dots m-1$]
for $j = 0$ to $m-1$ do
    $T[j]$ = new List()
```

Aufgabe.
Schreiben Sie Insert, Delete & Search.
Verwenden Sie Methoden der DS List!

ptr Insert(key k)

Delete(ptr x)

ptr Search(key k)

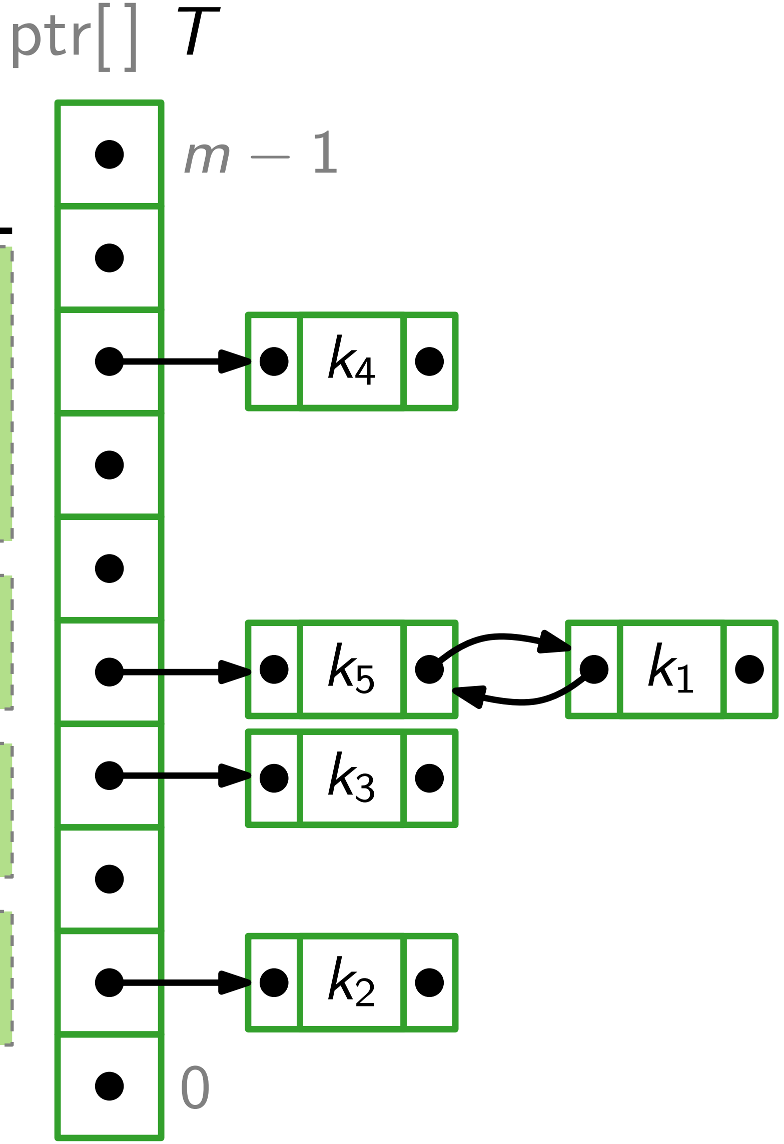


Hashing mit Verkettung

Voraussetzungen:

- $|U| \gg |K|$
- Zugriff auf Hashfunktion h

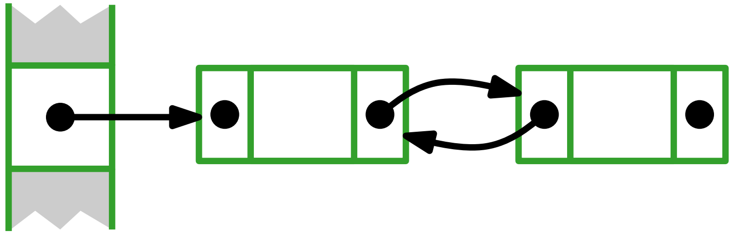
Operation	**Implementierung**
$\textsc{HashChaining}(\text{int } m)$	$T = \textbf{new } \textsc{List}[0 \dots m-1]$ **for** $j = 0$ **to** $m-1$ **do** $\lfloor \quad T[j] = \textbf{new } \textsc{List}()$
$\text{ptr } \textsc{Insert}(\text{key } k)$	**return** $T[h(k)].\textsc{Insert}(k)$
$\textsc{Delete}(\text{ptr } x)$	$T[h(x.key)].\textsc{Delete}(x)$
$\text{ptr } \textsc{Search}(\text{key } k)$	**return** $T[h(k)].\textsc{Search}(k)$



Analyse

Definition. Die **Auslastung** α einer Hashtabelle sei n/m, also der Quotient der Anzahl der gespeicherten Elemente und der Tabellengröße.

Anmerkung. α ist die durchschnittliche Länge einer **Kette**.



Laufzeit: $\Theta(n)$ im schlimmsten Fall.

z.B. $h(k) = 0 \quad \forall k \in K$.

Annahme: **Einfaches uniformes Hashing:**

jedes Element von U wird mit gleicher Wahrscheinlichkeit in jeden der m Einträge der Tabelle gehasht – unabhängig von anderen Elementen.

d.h. $\mathbf{Pr}[h(k) = i] = 1/m$

Suche

Fälle:

1) erfolglose Suche
2) erfolgreiche Suche

Notation: $n_j = T[j].\mathit{length}$ für $j = 0, 1, \ldots, m-1$.

Dann gilt: $n = n_0 + n_1 + \cdots + n_{m-1}$.

$$\mathbf{E}[n_j] = n/m = \alpha$$

Satz. Unter der Annahme des einfachen uniformen Hashings durchsucht eine **erfolglose** Suche erwartet α Elemente.

Beweis. Wenn die Suche nach einem Schlüssel k erfolglos ist, muss $T[h(k)]$ komplett durchsucht werden.

$$\mathbf{E}\big[T[h(k)].\mathit{length}\big] = \mathbf{E}[n_{h(k)}] = \alpha.$$

Erfolgreiche Suche

Satz. Unter der Annahme des einfachen uniformen Hashings durchsucht eine **erfolgreiche** Suche erwartet höchstens $1 + \alpha/2$ Elemente.

Beweis. Noch eine Annahme:

Jedes der n Elemente in T ist mit gleicher Wahrscheinlichkeit das gesuchte Element x.

räumlich

$\#$ durchsuchte Elemente $=$ $\#$ Elemente **vor** x in $T[h(x)]$ $+ 1$

X $=$ $\#$ Elemente, die **nach** x in $T[h(x)]$ eingefügt wurden $+ 1$

zeitlich

Sei $x_1, x_2, \ldots, x_n$ die Folge der Schlüssel in der Reihenfolge des Einfügens.

Definiere Indikator-Zufallsvariable: $X_{ij} = 1$ falls $h(x_i) = h(x_j)$.

Klar: $\mathbf{E}[X_{ij}] = \mathbf{Pr}[X_{ij} = 1] = 1/m$

einfaches uniformes Hashing!

Erfolgreiche Suche

$$X \quad = \#\ \text{Elemente, die } \textbf{nach } x \text{ in } T[h(x)] \text{ eingefügt wurden } + 1$$

$$\begin{aligned}\mathbf{E}[X] &= \frac{1}{n}\sum_{i=1}^{n} \mathbf{E}\left[1 + \#\ \text{Elemente, die nach } x_i \text{ in } T[h(x_i)] \text{ eingefügt wurden}\right] \\ &= \frac{1}{n}\sum_{i=1}^{n} 1 + \frac{1}{n}\sum_{i=1}^{n} \mathbf{E}\,\underbrace{[\#\ \text{Elemente } x_j \text{ mit } j > i \text{ und } h(x_i) = h(x_j)]}_{\overset{\|}{\sum_{j=i+1}^{n} X_{ij}}}\end{aligned}$$

Definiere Indikator-Zufallsvariable: $X_{ij} = 1$ falls $h(x_i) = h(x_j)$.

Klar: $\mathbf{E}[X_{ij}] = \mathbf{Pr}[X_{ij} = 1] = 1/m$

Erfolgreiche Suche

X = # Elemente, die **nach** x in $T[h(x)]$ eingefügt wurden $+ 1$

$$\mathbf{E}[X] = \frac{1}{n}\sum_{i=1}^{n} \mathbf{E}\Big[1 + \#\text{ Elemente, die nach } x_i \text{ in } T[h(x_i)] \text{ eingefügt wurden}\Big]$$

$$= \frac{1}{n}\sum_{i=1}^{n} 1 + \frac{1}{n}\sum_{i=1}^{n} \underbrace{\mathbf{E}\left[\#\text{ Elemente } x_j \text{ mit } j > i \text{ und } h(x_i) = h(x_j)\right]}_{\|}$$

$$= 1 + \frac{1}{n}\sum_{i=1}^{n} \mathbf{E}\left[\sum_{j=i+1}^{n} X_{ij}\right]$$

$n - i$

$$= 1 + \frac{1}{n}\sum_{i=1}^{n}\sum_{j=i+1}^{n} \mathbf{E}[X_{ij}] = 1 + \frac{1}{nm}\sum_{i=1}^{n}\sum_{j=i+1}^{n} 1 = 1 + \frac{1}{nm}\sum_{i=1}^{n} n - i$$

$1/m$

Setze $k = n - i$

$$= 1 + \frac{1}{nm}\sum_{k=0}^{n-1} k = 1 + \frac{n(n-1)}{2nm} = 1 + \frac{n-1}{2m} < 1 + \frac{\alpha}{2}$$

$\alpha = n/m$

1) $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ **arithmetische Reihe**

Zusammenfassung Ergebnisse

Satz. Unter der Annahme des einfachen uniformen Hashings durchsucht beim Hashing mit Verkettung eine

- **erfolgreiche** Suche erwartet höch. $1 + \alpha/2$ Elemente
- **erfolglose** Suche erwartet α Elemente.

Und **Einfügen**? Und **Löschen**?

Satz. Unter der Annahme des einfachen uniformen Hashings laufen **alle** Wörterbuch-Operationen in (erwartet) konstanter Zeit, falls $n \in \mathcal{O}(m)$.

$\Rightarrow \frac{n}{m}$ konstant

Was ist eine gute Hashfunktion?

1. so „zufällig“ wie möglich – um der Annahme des einfachen uniformen Hashings möglichst nahe zu kommen.
    - Hashfunktion sollte die Schlüssel aus dem Universum U möglichst gleichmäßig über die m Plätze der Hashtabelle verteilen.
    - Hashfunktion sollte Muster in der Schlüsselmenge K gut **auflösen**.

Beispiel: $U =$ Zeichenketten, $K =$ Wörter der deutschen Sprache

h : nimm die ersten drei Buchstaben → Zahl

schlecht:
- viele Wörter fangen mit „sch“ an
  $\Rightarrow$ selber Hashwert
- andere Buchstaben haben keinen Einfluss

2. einfach zu berechnen!

Annahme: Alle Schlüssel sind (natürliche) Zahlen.

Suche also Hashfunktionen: $\mathbb{N} \rightarrow \{0, \ldots, m-1\}$

Rechtfertigung:

33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	’	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o		
48	0	64	@	80	P	96	‘	112	p		

American
Standard
Code of
Information
Interchange

Zum Beispiel:

$$\text{AW} \rightarrow (65,87)_{10} = (1000001, 1010111)_2 \rightarrow 1000001\,1010111_2 = 65 \cdot 128 + 87 = 8407_{10}$$

Divisionsmethode

Hashfunktion $h\colon \mathbb{N} \to \{0, \ldots, m-1\}$

$$k \mapsto k \bmod m$$

Beispiel: $h(k) \; = \; k \bmod 1024$

10 niedrigwertigste Stellen

$h(1026) = 2 \qquad 1026_{10} = 010000000010_2$

$h(2050) = 2 \qquad 2050_{10} = 100000000010_2$

d.h. die 2 höherwertigsten Stellen werden von h ignoriert

Moral: vermeide $m =$ Zweierpotenz

Strategie: wähle für m eine Primzahl, entfernt von Zweierpotenz

↓

löst Muster gut auf

Multiplikationsmethode

Hashfunktion $h\colon \mathbb{N} \to \{0, \ldots, m-1\}$

$$k \mapsto \lfloor m \cdot (\underbrace{kA \bmod 1}_{\text{gebrochener Anteil von } kA}) \rfloor, \quad \text{wobei } 0 < A < 1.$$

d.h. $kA - \lfloor kA \rfloor$

- Verschiedene Werte von A „funktionieren“ verschieden gut.

  gut: z.B. $A \approx \frac{\sqrt{5}-1}{2}$ [Knuth: The Art of Computer Programming III, ’73]

- Vorteil gegenüber Divisionsmethode: Wahl von m relativ beliebig.

  Insbesondere $m =$ Zweierpotenz möglich.

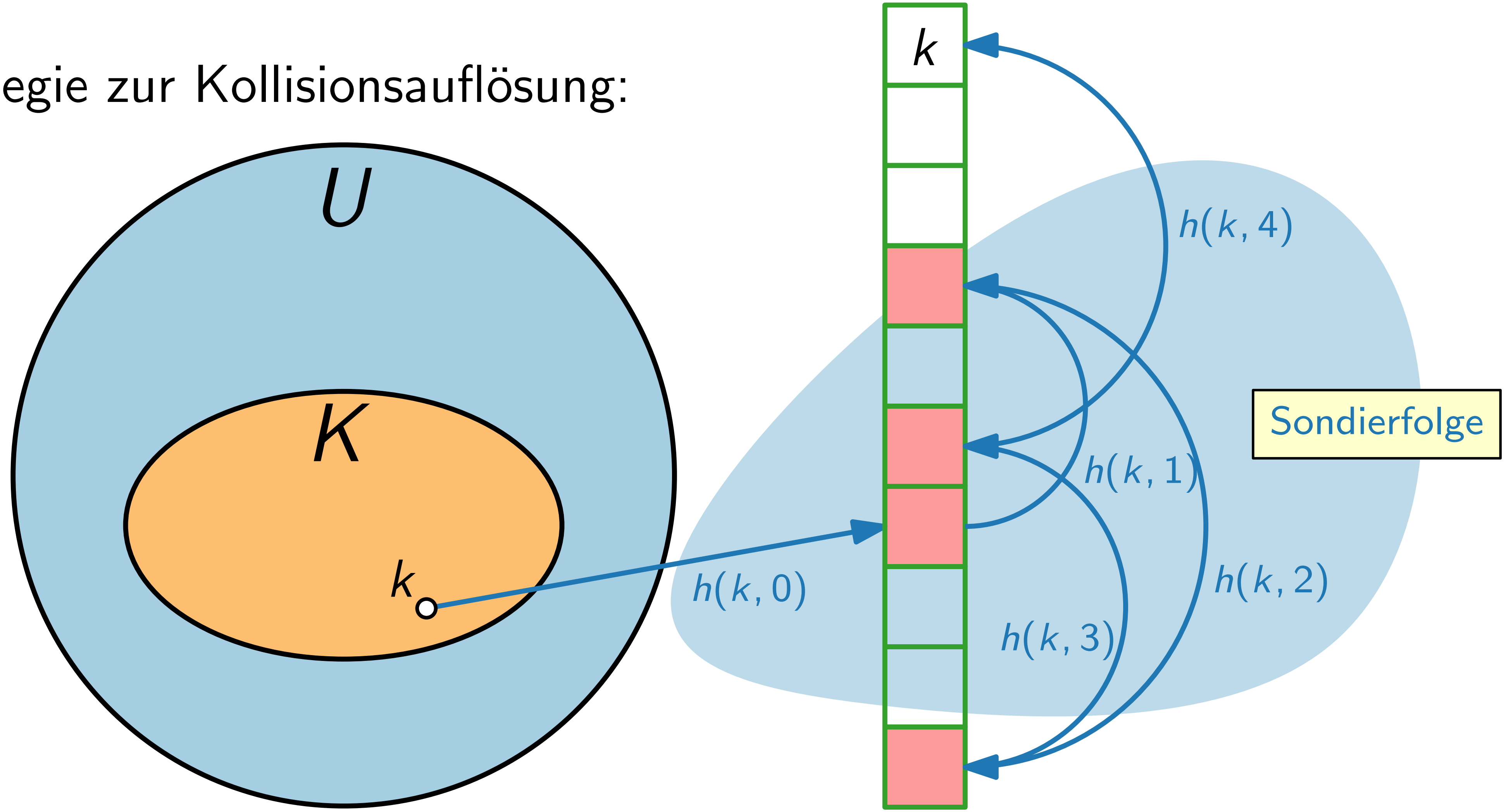
  ⇒ schnell berechenbar (in Java verschiebt `a << s` die Dualzahlendarstellung von a um s Stellen nach links)

Hashing mit offener Adressierung

Alle Elemente werden direkt in der Hashtabelle gespeichert.

$\Rightarrow$ Tabelle kann volllaufen $\Rightarrow$ $\alpha \leq 1$

Strategie zur Kollisionsauflösung:



Hashing mit offener Adressierung

Operation | **Implementierung**

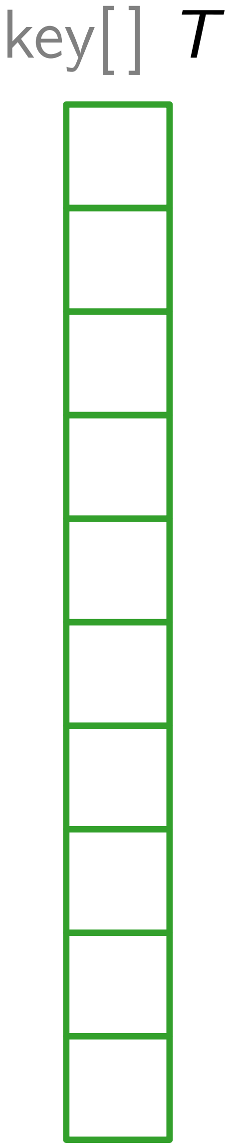
HashOA(int m)

```
$T = \textbf{new}\ \text{key}[0 \dots m - 1]$
for $j = 0$ to $m - 1$ do $T[j] = -1$
```

int Insert(key k)

```
$i = 0$
repeat
    Unterschied zu while?
    Abbruchbedingung nach Ausführung des Schleifeninhalts
    → Schleifeninhalt wird mindestens einmal ausgeführt
until $i == m$
```

int Search(key k)



Hashing mit offener Adressierung

Operation | **Implementierung**

HASHOA(int m)

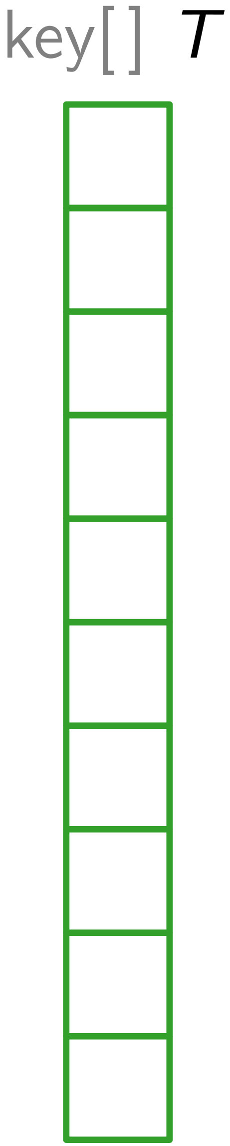
```
$T$ = new key$[0 \dots m-1]$
for $j = 0$ to $m-1$ do $T[j] = -1$
```

int INSERT(key k)

```
$i = 0$
repeat
    $j = h(k, i)$
    if $T[j] == -1$ then
        $T[j] = k$
        return $j$
    else $i = i + 1$
until $i == m$
error "table overflow"
```

int SEARCH(key k)

Aufgabe:
Schreiben Sie SEARCH mit **repeat**-Schleife!



Hashing mit offener Adressierung

Operation

HASHOA(int m)

int ~~INSERT~~(key k)
SEARCH

...und DELETE()?

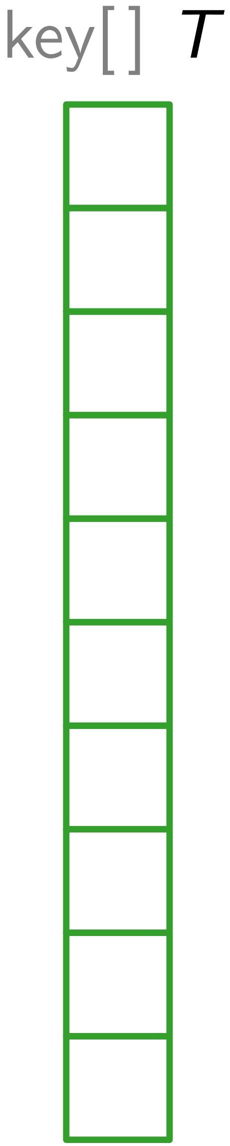
Umständlich; z.B. indem man **Grabsteine** hinterlässt, die einem sagen, dass man bei SEARCH weitersuchen soll.

int SEARCH(key k)

Implementierung

```
$T$ = new key$[0 \dots m-1]$
for $j = 0$ to $m-1$ do $T[j] = -1$
```

```
$i = 0$
repeat
    $j = h(k, i)$
    if $T[j] == \overset{k}{\cancel{-1}}$ then
        ~~$T[j] = k$~~
        return $j$
    else $i = i + 1$
until $i == m$ or $T[j] == -1$
~~error "table overflow"~~ return $-1$
```



Berechnung von Sondierfolgen

Hashfunktion für offene Adressierung $h: \quad U \times \{0, \dots, m-1\} \to \{0, \dots, m-1\}$

$\langle h(k,0), h(k,1), \dots, h(k,m-1)\rangle$ heißt **Sondierfolge** (für k).

Voraussetzungen:

- eine Sondierfolge ist eine Permutation von $\langle 0, 1, \dots, m-1\rangle$
  (Sonst durchläuft die Folge nicht alle Tabelleneinträge genau einmal!)
- Existenz von „gewöhnlicher“ Hashfunktion $h_0 \colon U \to \{0, \dots, m-1\}$

Verschiedene Typen von Sondierfolgen:

- Lineares Sondieren: $h(k,i) = (h_0(k) + i) \bmod m$
- Quadratisches Sondieren: $h(k,i) = (h_0(k) + c_1 i + c_2 i^2) \bmod m$
- Doppeltes Hashing: $h(k,i) = (h_0(k) + i \cdot h_1(k)) \bmod m$

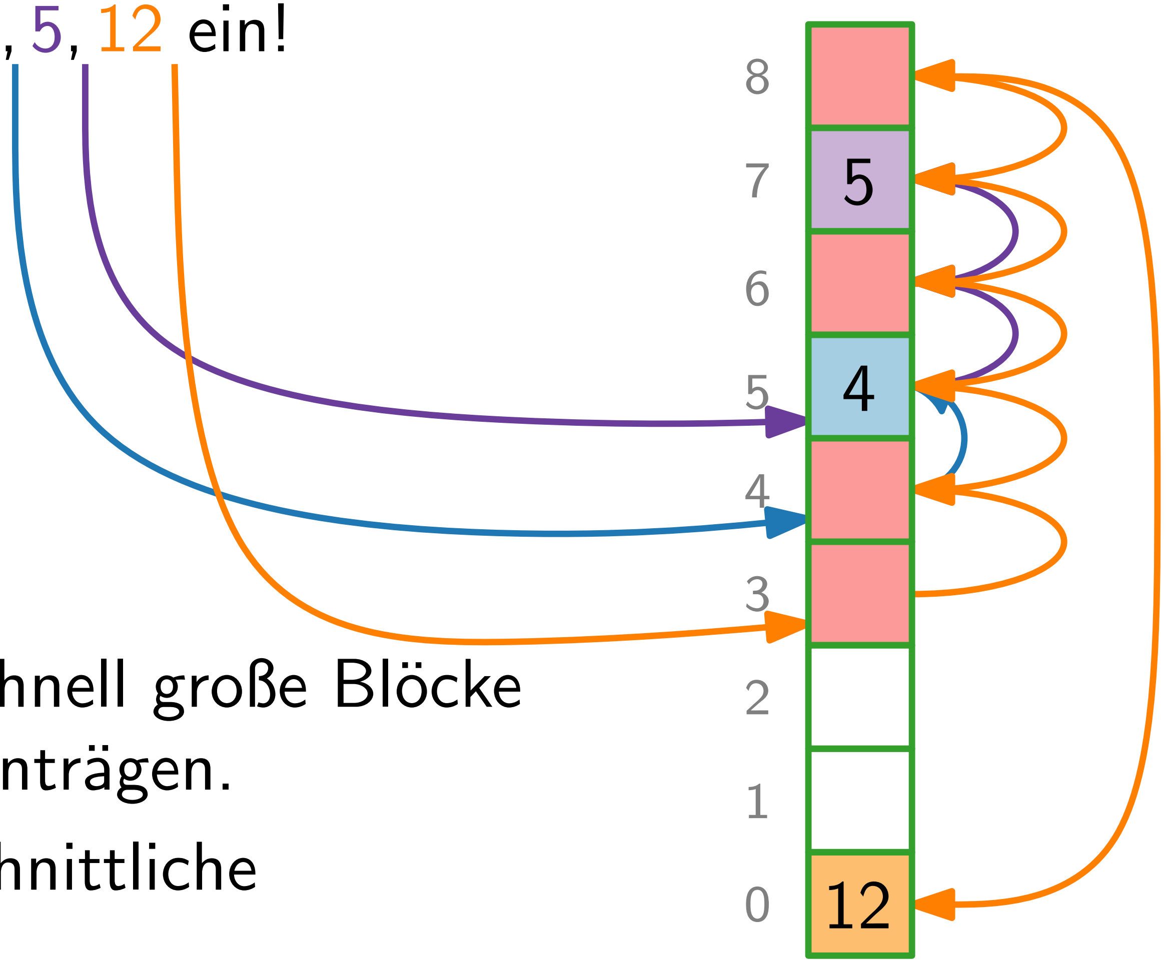
Lineares Sondieren

Demo.
`https://algo.uni-trier.de/demos/hashing.html`

Sondierfolge: $h(k, i) = (h_0(k) + i) \bmod m$

Beispiel: $h_0(k) = k \bmod 9$ und $m = 9$

Füge Schlüssel 4, 5, 12 ein!



Problem: Es bilden sich schnell große Blöcke von besetzten Einträgen.

⇒ hohe durchschnittliche Suchzeit!

primäres Clustering

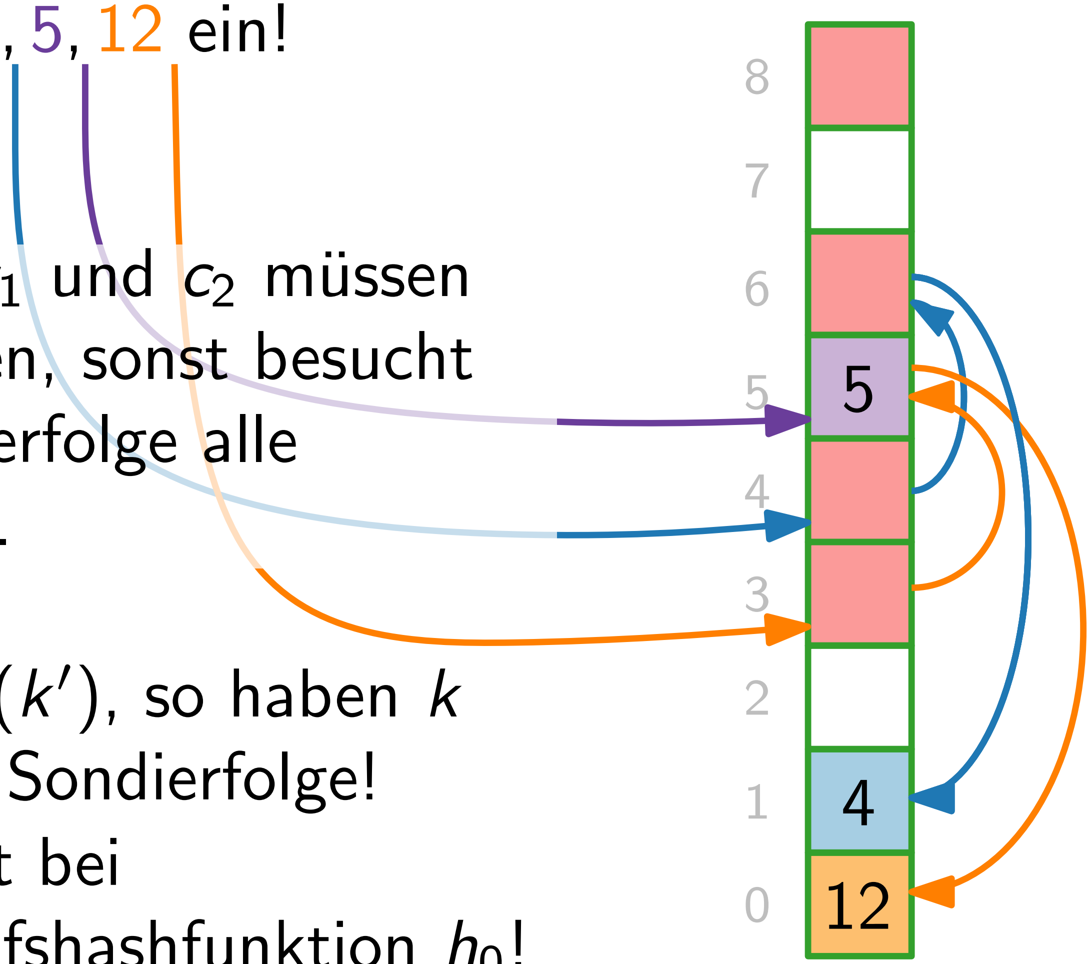
Quadratisches Sondieren

Demo.
`https://algo.uni-trier.de/demos/hashing.html`

Sondierfolge: $h(k, i) = (h_0(k) + c_1 i + c_2 i^2) \bmod m$

Beispiel: $h_0(k) = k \bmod 9$ und $m = 9$ und $c_1 = c_2 = 1$

Füge Schlüssel 4, 5, 12 ein!



Problem: Die Größen m, c_1 und c_2 müssen zueinander passen, sonst besucht nicht jede Sondierfolge alle Tabelleneinträge.

Problem: Falls $h_0(k) = h_0(k')$, so haben k und k' **dieselbe** Sondierfolge!
$\Rightarrow$ hohe Suchzeit bei schlechter Hilfshashfunktion h_0!

sekundäres Clustering

Doppeltes Hashing

Demo.
`https://algo.uni-trier.de/demos/hashing.html`

Sondierfolge: $h(k, i) = (h_0(k) + i \cdot h_1(k)) \bmod m$

Vorteile:

- Sondierfolge hängt zweifach vom Schlüssel k ab!
- potentiell m^2 verschiedene Sondierfolgen möglich
  (bei linearem & quadratischem Sondieren nur m.)

Frage: Was muss gelten, damit eine Sondierfolge alle Tabelleneinträge durchläuft?

Antwort: $k' = h_1(k)$ und m müssen **teilerfremd** sein ,
d.h. $\mathrm{ggT}(k', m) = 1$.

$\mathrm{ggT}(a, b) =_{\text{Def.}} \max\{t \colon t | a \text{ und } t | b\}$

Also: z.B. $m =$ Zweierpotenz und h_1 immer ungerade.
oder $m =$ prim und $0 < h_1(k) < m$ für alle k.

Uniformes Hashing

kein neues Hashverfahren, sondern eine (idealisierte) Annahme...

Annahme: Die Sondierfolge jedes Schlüssels ist gleich wahrscheinlich eine der $m!$ Permutationen von $\langle 0, 1, \ldots, m-1 \rangle$.

Satz. Unter der Annahme von uniformem Hashing ist die erwartete Anzahl der versuchten Tabellenzugriffe bei offener Adressierung und

- erfolgloser Suche $\leq \dfrac{1}{1-\alpha}$
- erfolgreicher Suche $\leq \dfrac{1}{\alpha} \ln \dfrac{1}{1-\alpha}$

d.h. Suche dauert erwartet $\mathcal{O}(1)$ Zeit, falls α konst.

(Beweis siehe [CLRS 11.4])

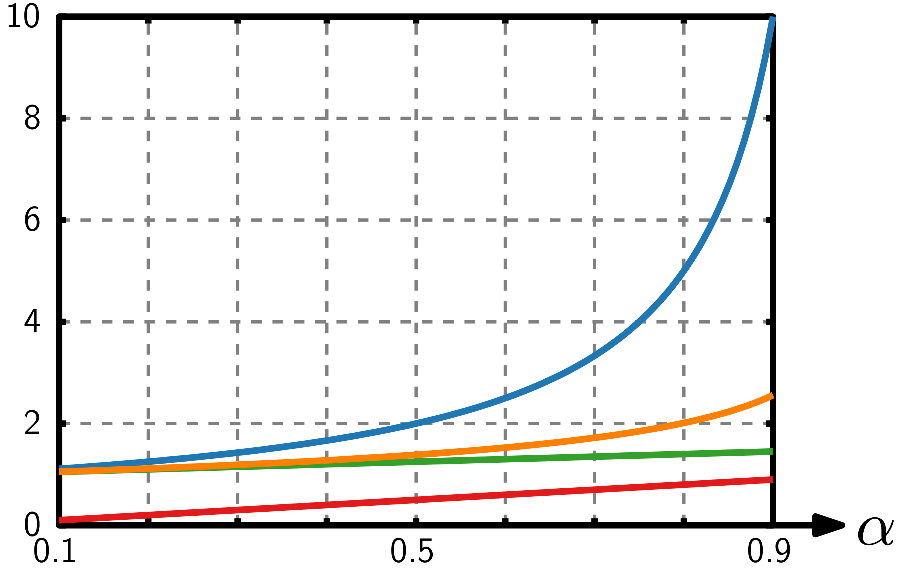
Zusammenfassung

mit Verkettung

- ⊕ funktioniert für $n \in \mathcal{O}(m)$
- ⊕ gute erwartete Suchzeit:
  erfolglos: $\boxed{\alpha}$ $\boxed{= n/m}$
  erfolgreich: $\boxed{1 + \frac{\alpha}{2}}$

[Modell: einfaches uniformes Hashing]

- ⊖ Listenoperationen langsam



mit offener Adressierung

- ⊖ funktioniert nur für $n \leq m$
- ⊖ langsam, wenn $n \approx m$

Sondiermethoden:

- lineares Sondieren
- quadratisches Sondieren
- doppeltes Hashing

- ⊕ gute erwartete Suchzeit:
  erfolglos: $\boxed{\frac{1}{1-\alpha}}$
  erfolgreich: $\boxed{\frac{1}{\alpha} \ln \frac{1}{1-\alpha}}$

[Modell: uniformes Hashing]