

Algorithmen und Datenstrukturen

Vorlesung 12: Hashing

Hashing

Wörterbuch.

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

Abstrakter Datentyp.

stellt folgende Operationen bereit: INSERT, DELETE, SEARCH

Implementierung.

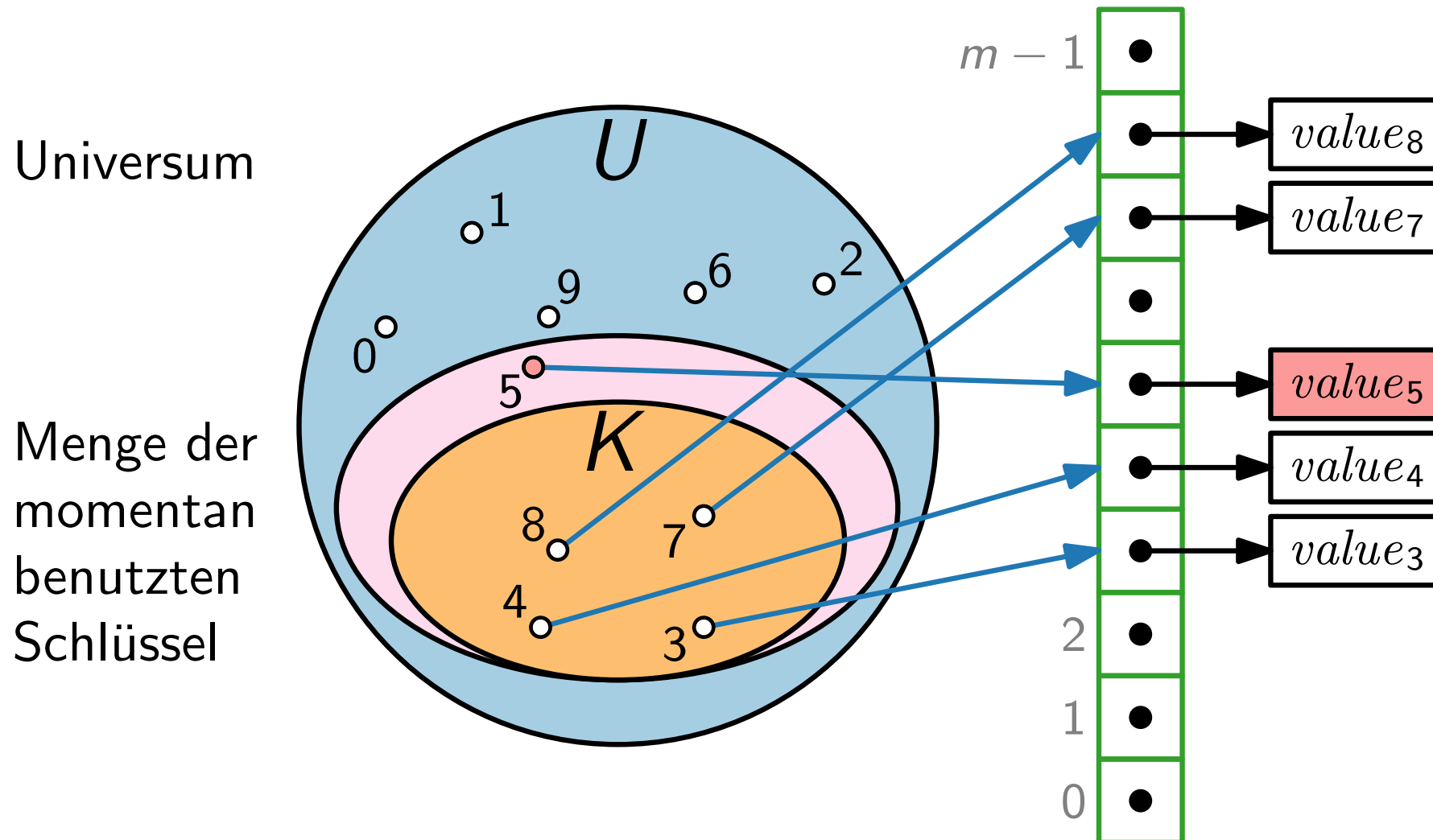
heute: Hashing

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** $U = \{0, \dots, m - 1\}$
 - Schlüssel paarweise verschieden dyn. Menge!



Direkte Adressierung

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

Operation	Implementierung	
<code>HASHDA(int m)</code>	<pre> $T = \text{new value}[0 \dots m - 1]$ for $j = 0$ to $m - 1$ do $T[j] = \text{nil}$ // $T[j] = \text{Zeiger auf } j. \text{ Datensatz}$ </pre>	$m - 1$ • • • • • • • • • • $\text{ptr}[] \ T$
<code>ptr INSERT(key k, value v)</code>	<pre> // lege neuen Datensatz an // und initialisiere ihn mit v $T[k] = \text{new value}(v)$ </pre>	8 value₈ 7 value₇ 5 value₅ 4 value₄ 3 value₃
<code>DELETE(key k)</code>	<pre> // Speicher freigeben, auf $T[k] = \text{nil}$ </pre>	
<code>ptr SEARCH(key k)</code>	<pre> return $T[k]$ </pre>	

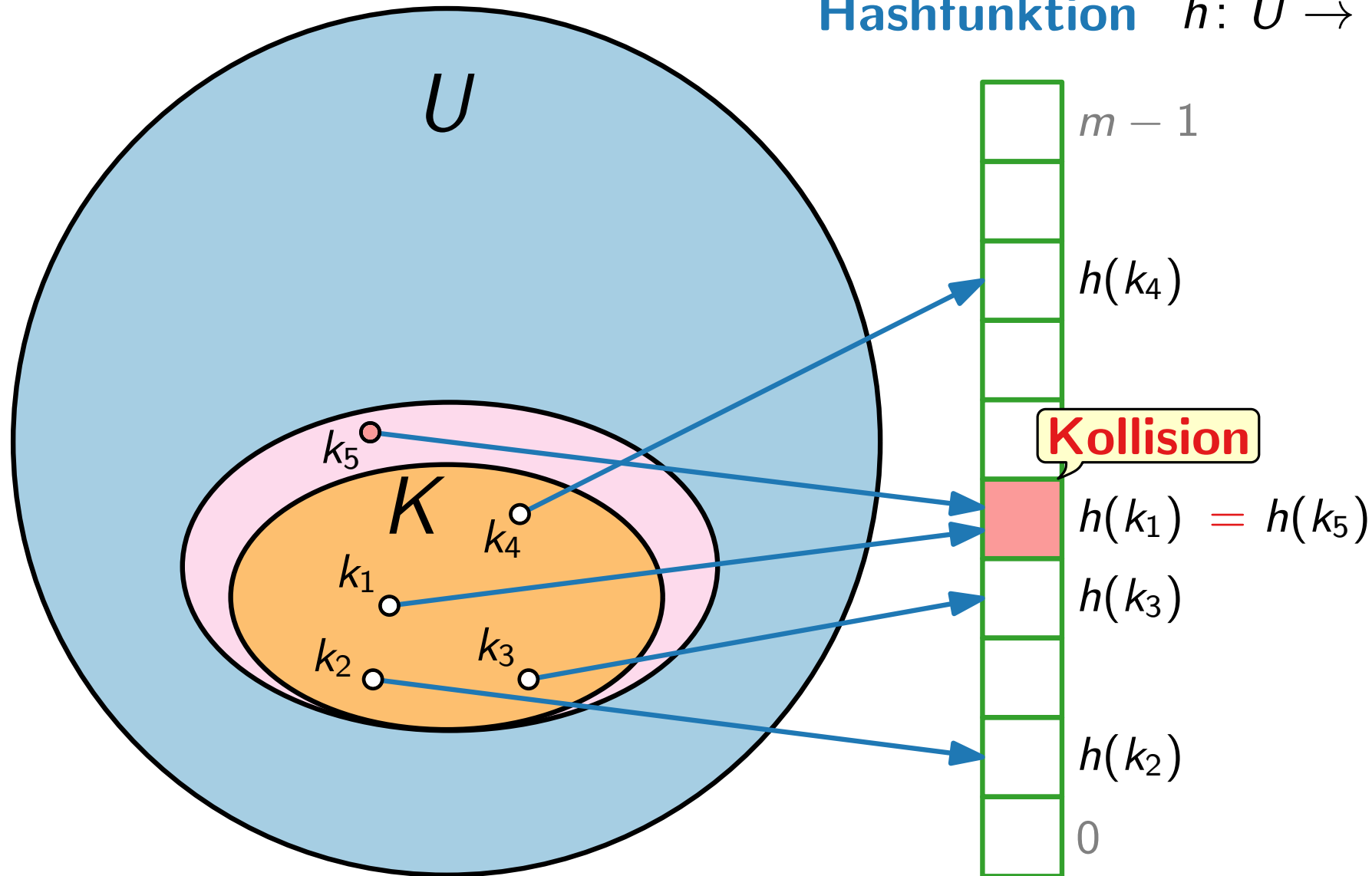
Laufzeiten?

INSERT, DELETE,
SEARCH $\mathcal{O}(1)$
im *schlechtesten Fall*

Hashing mit Verkettung

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

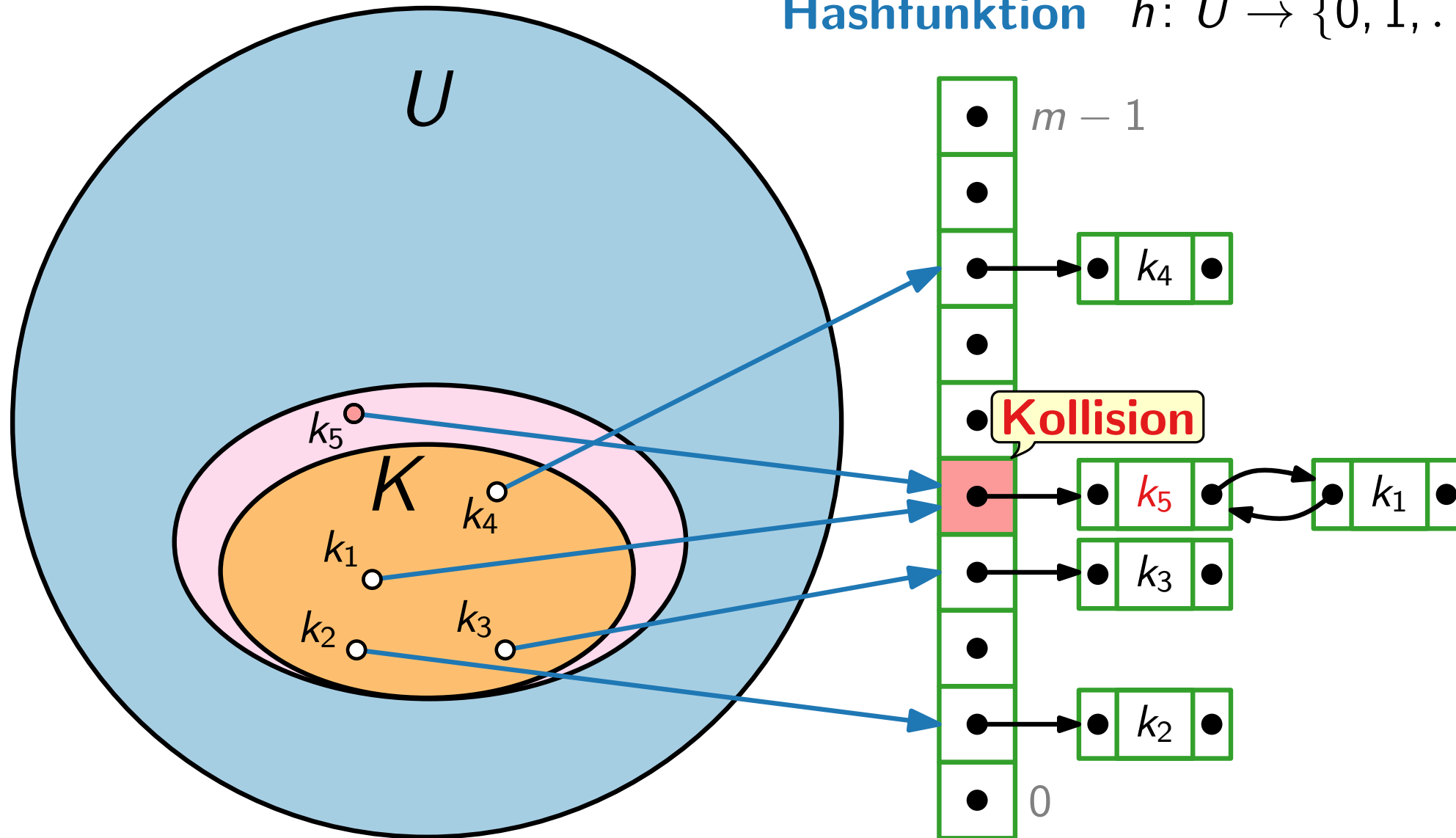
Hashfunktion $h: U \rightarrow \{0, 1, \dots, m-1\}$



Hashing mit Verkettung

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

Hashfunktion $h: U \rightarrow \{0, 1, \dots, m-1\}$



Hashing mit Verkettung

Voraussetzungen:

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

Operation

Implementierung

`HASHCHAINING(int  $m$ )`

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

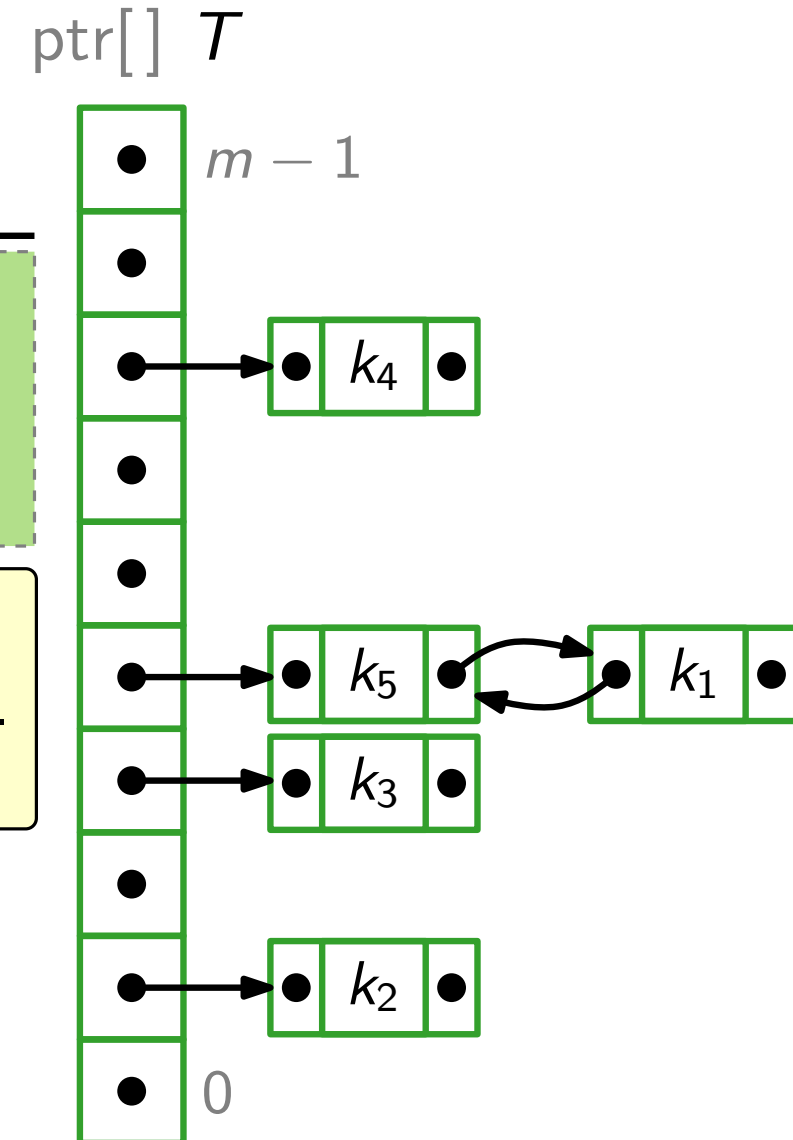
`ptr INSERT(key  $k$ )`

`DELETE(ptr  $x$ )`

`ptr SEARCH(key  $k$ )`

Aufgabe.

Schreiben Sie INSERT, DELETE & SEARCH.
Verwenden Sie Methoden der DS LIST!



Hashing mit Verkettung

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

Operation

Implementierung

`HASHCHAINING(int  $m$ )`

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

`ptr INSERT(key  $k$ )`

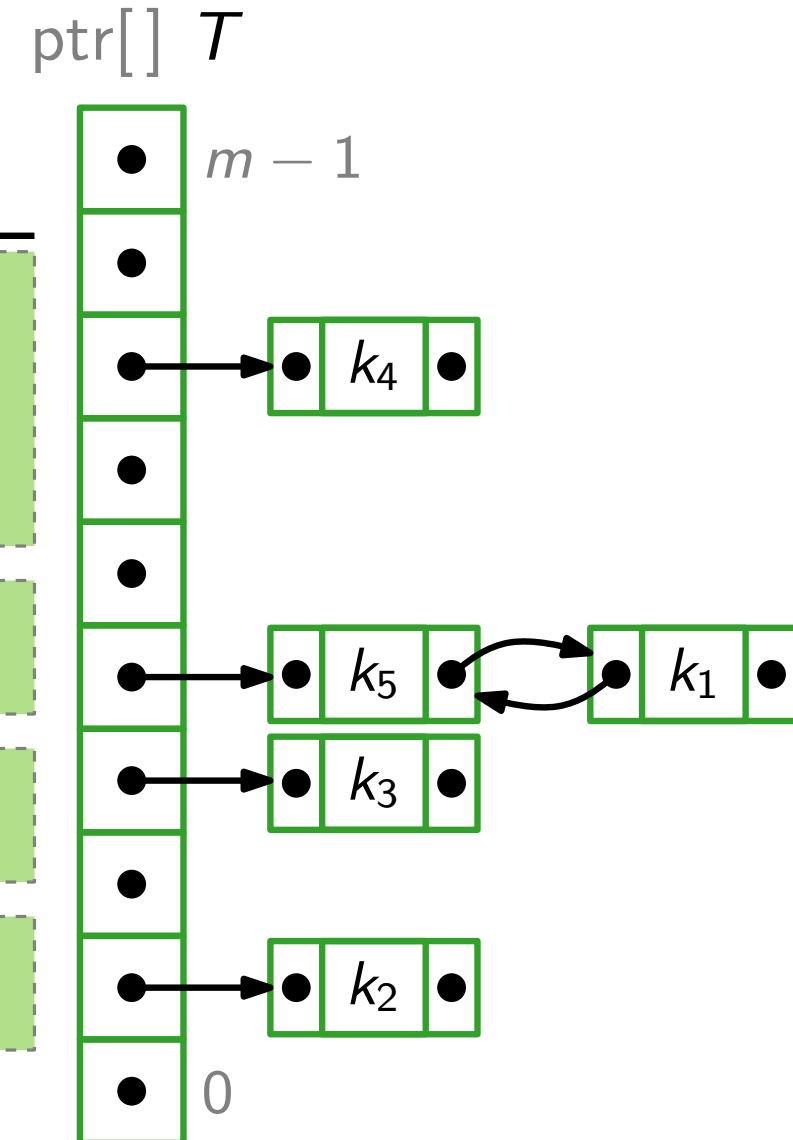
```
return  $T[h(k)].\text{INSERT}(k)$ 
```

`DELETE(ptr  $x$ )`

```
 $T[h(x.\text{key})].\text{DELETE}(x)$ 
```

`ptr SEARCH(key  $k$ )`

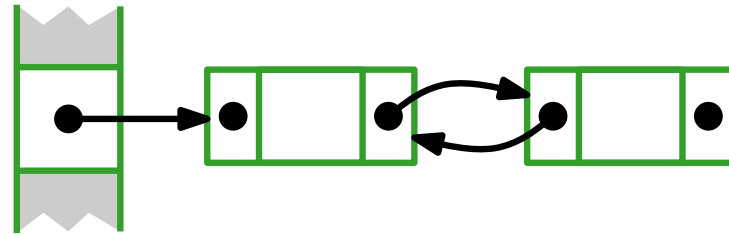
```
return  $T[h(k)].\text{SEARCH}(k)$ 
```



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.

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

Suche

Fälle:

- 1) erfolglose Suche
- 2) erfolgreiche Suche

Notation: $n_j = T[j].length$ für $j = 0, 1, \dots, m - 1$.

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

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

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.

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

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 .

durchsuchte Elemente = # Elemente **vor** ^{räumlich} x in $T[h(x)]$ + 1

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

Sei $x_1, x_2, \dots, x_n$ 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 = # Elemente, die **nach** x in $T[h(x)]$ eingefügt wurden **+ 1**

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

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

$$\sum_{j=i+1}^n X_{ij}$$

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**

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

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

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

$$= 1 + \frac{1}{n} \sum_{i=1}^n \sum_{j=i+1}^n \mathbf{E}[X_{ij}] \quad \text{[} 1/m \text{]} = 1 + \frac{1}{nm} \sum_{i=1}^n \sum_{j=i+1}^n 1 = 1 + \frac{1}{nm} \sum_{i=1}^n n - i \quad \text{[Setze } k = n - i \text{]}$$

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

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

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 **aauflösen**.

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

h : nimm die ersten drei Buchstaben $\rightarrow$ 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: $\mathbb{N} \rightarrow \{0, \dots, m - 1\}$

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:

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

Divisionsmethode

Hashfunktion $h: \mathbb{N} \rightarrow \{0, \dots, m-1\}$

$$k \mapsto k \bmod m$$

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

$$h(1026) = 2 \quad 1026_{10} = 01\text{00000000}10_2$$

$$h(2050) = 2 \quad 2050_{10} = 10\text{00000000}10_2$$

10 niedrigwertigste Stellen

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

Moral: vermeide $m = \text{Zweierpotenz}$

Strategie: wähle für m eine Primzahl, entfernt von Zweierpotenz



löst Muster gut auf

Multiplikationsmethode

Hashfunktion $h: \mathbb{N} \rightarrow \{0, \dots, m-1\}$

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

gebrochener Anteil von kA

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 $m = \text{Zweierpotenz}$ möglich.

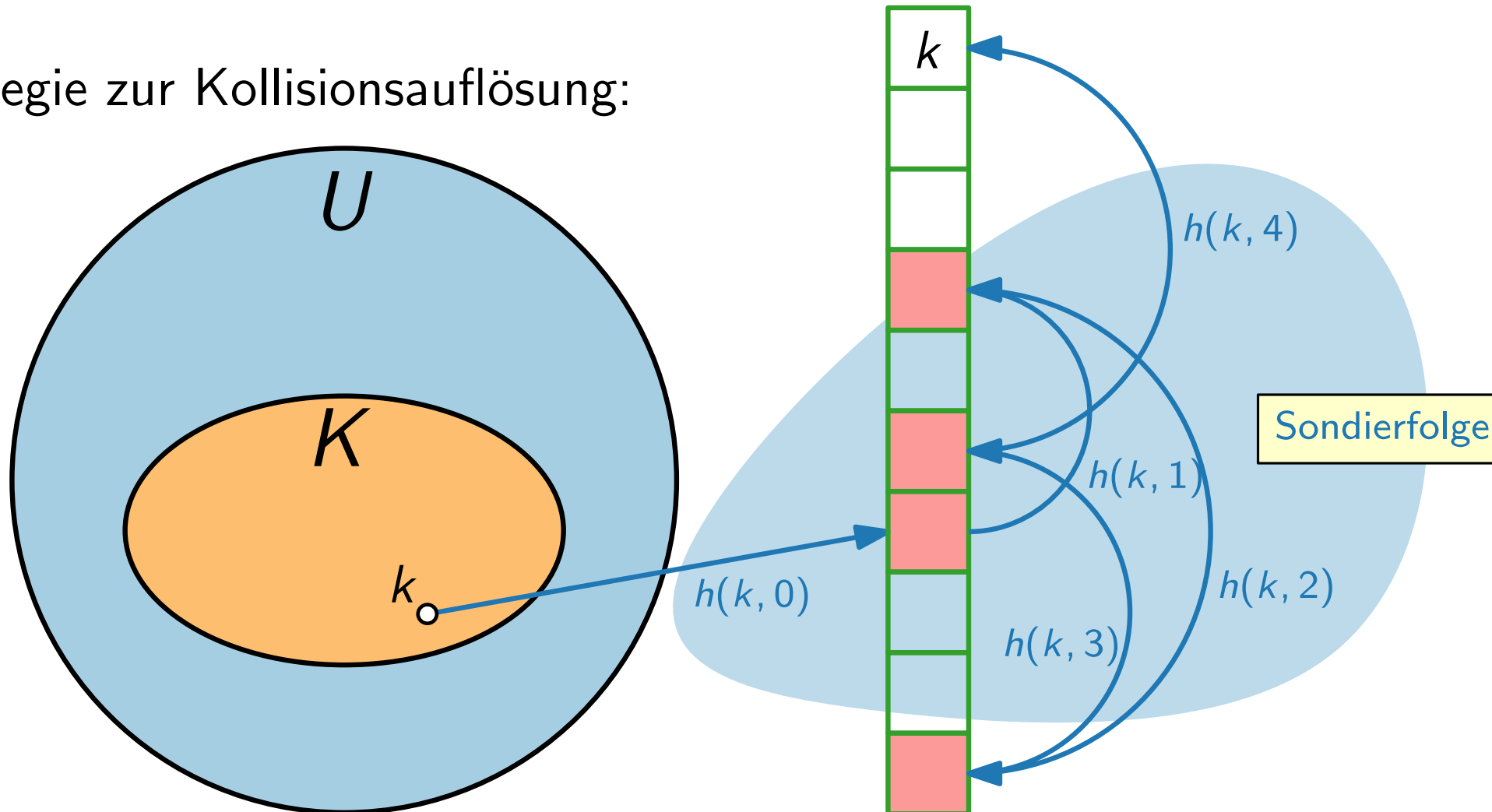
$\Rightarrow$ schnell berechenbar (in Java verschiebt `a << s` die Dualzahlendarstellung von a um s Stellen nach links)

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

`HASHOA(int  $m$ )`

`int INSERT(key  $k$ )`

`int SEARCH(key  $k$ )`

Implementierung

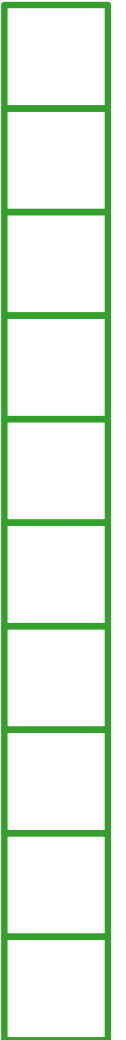
```
 $T = \text{new key}[0 \dots m - 1]$ 
for  $j = 0$  to  $m - 1$  do  $T[j] = -1$ 
```

```
 $i = 0$ 
repeat
    

Unterschied zu while?


    Abbruchbedingung nach
    Ausführung des Schleifeninhalts
    → Schleifeninhalt wird
    mindestens einmal ausgeführt
until  $i == m$ 
```

key[] T



Hashing mit offener Adressierung

Operation

`HASHOA(int  $m$ )`

`int INSERT(key  $k$ )`

Aufgabe:

Schreiben Sie `SEARCH` mit **repeat**-Schleife!

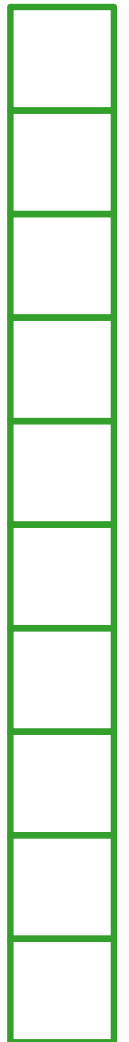
`int SEARCH(key  $k$ )`

Implementierung

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

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

key[] T



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.

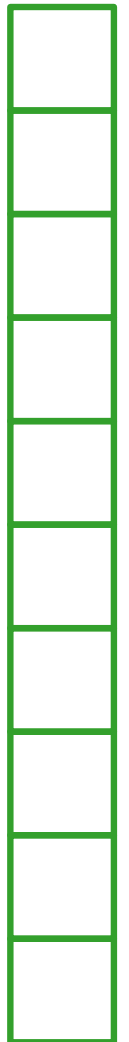
int SEARCH(key k)

Implementierung

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

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

key[] T



Berechnung von Sondierfolgen

Hashfunktion für offene Adressierung $h : U \times \{0, \dots, m-1\} \rightarrow \{0, \dots, m-1\}$
 $\langle h(k, 0), h(k, 1), \dots, h(k, m-1) \rangle$ 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 $h_0 : U \rightarrow \{0, \dots, m-1\}$

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 $m = 9$

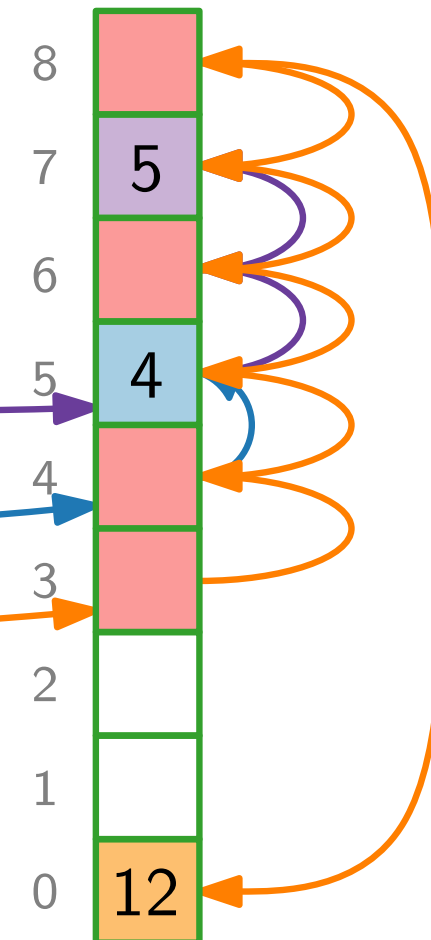
Füge Schlüssel 4, 5, 12 ein!

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

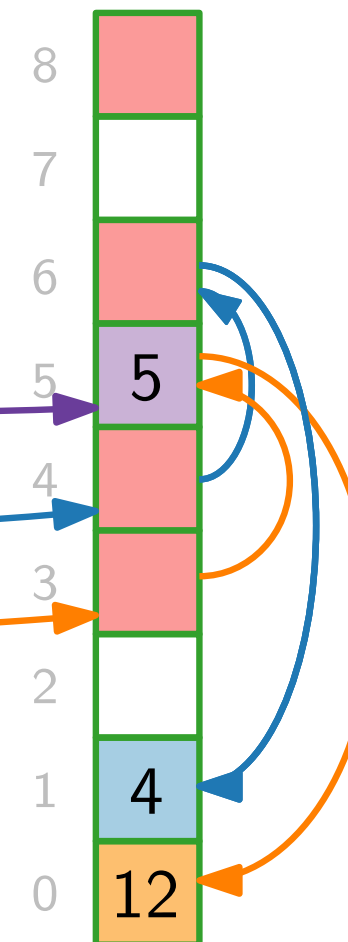
Beispiel: $h_0(k) = k \bmod 9$ und $m = 9$ und $c_1 = c_2 = 1$

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!
⇒ 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. $\text{ggT}(k', m) = 1$.

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

Also: z.B. $m = \text{Zweierpotenz}$ und h_1 immer ungerade.
oder $m = \text{prim}$ 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 $\langle 0, 1, \dots, m - 1 \rangle$.

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

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

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

(Beweis siehe [CLRS 11.4])

Zusammenfassung

mit Verkettung

⊕ funktioniert für $n \in \mathcal{O}(m)$

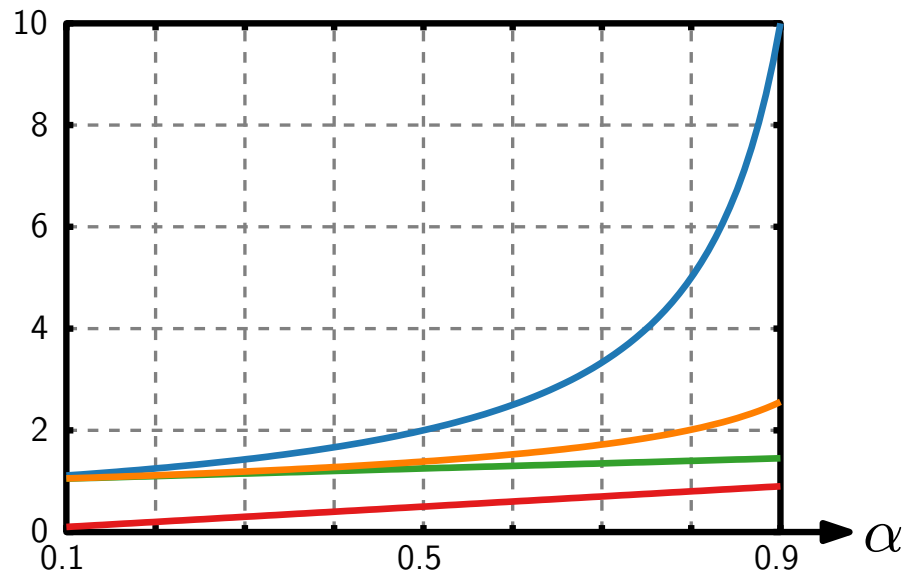
⊕ gute erwartete Suchzeit:

erfolglos: $\alpha = n/m$

erfolgreich: $1 + \frac{\alpha}{2}$

[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: $\frac{1}{1-\alpha}$

erfolgreich: $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$

[Modell: uniformes Hashing]