Grundlagen der Algorithmen und Datenstrukturen

Kapitel 4

Christian Scheideler + Helmut Seidl

SS 2009

Wörterbuch-Datenstruktur

S: Menge von Elementen

Jedes Element e identifiziert über key(e).

Operationen:

- S.insert(Elem e): $S=S \cup \{e\}$;
- S.remove(Key k): $S=S\setminus\{e\}$; wobei e das Element ist mit key(e)==k
- S.find(Key k): Falls es ein $e \in S$ gibt mit key(e)==k, dann gib e aus, sonst null.

Wörterbuch-Datenstruktur

- Realisierung als(sortierte) Liste:
  - insert, remove und find auf Menge S der Größe n kosten im worst case Θ (n) Zeit
- Realisierung als (sortiertes) Feld:
  - insert und remove kosten im worst case Θ (n) Zeit
  - find kann so realisiert werden, dass es im worst case nur O(log n) Zeit benötigt ($\rightarrow$ binäre Suche!)

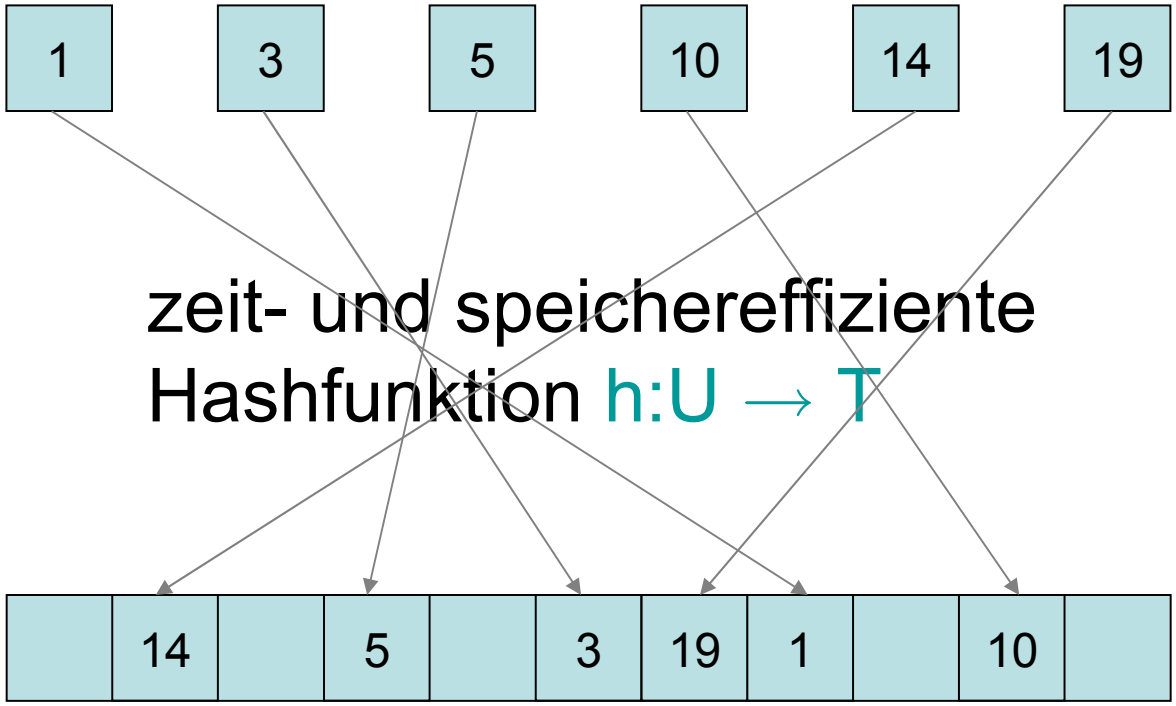
Wörterbuch-Datenstruktur

Kann man insert und remove besser mit einem Feld realisieren?

- folge Beispiel der Bibliothek!

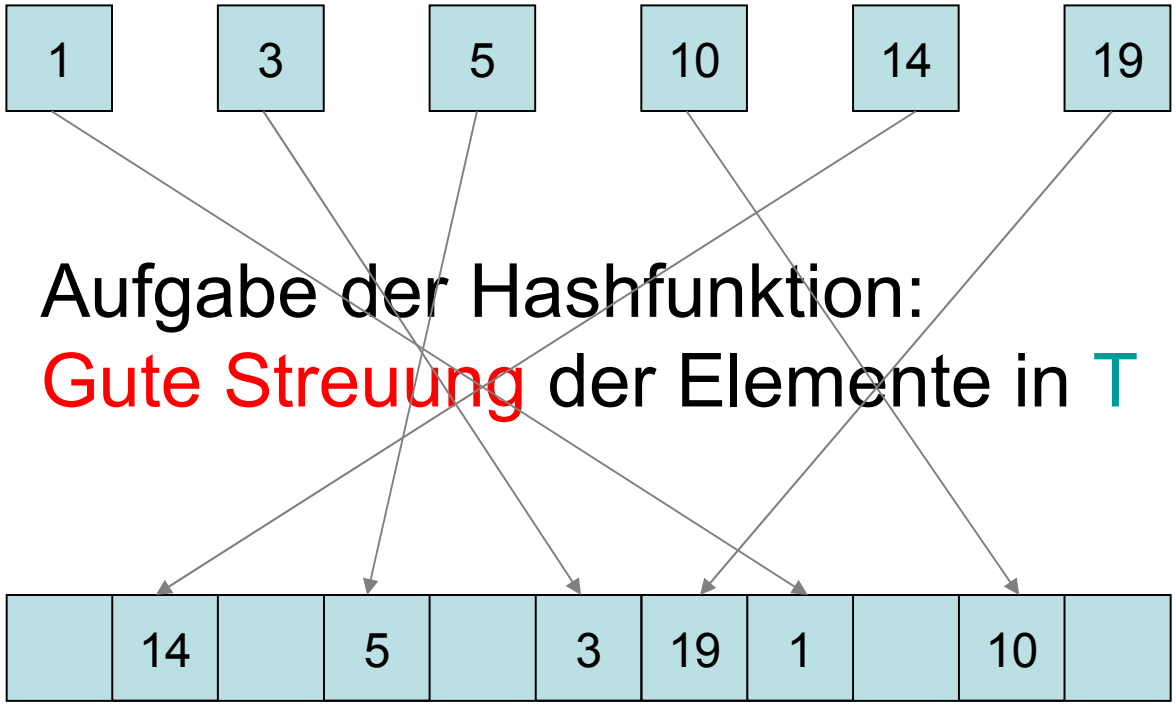
- verwende Hashtabellen (Ziel von Kapitel 4)

Hashing



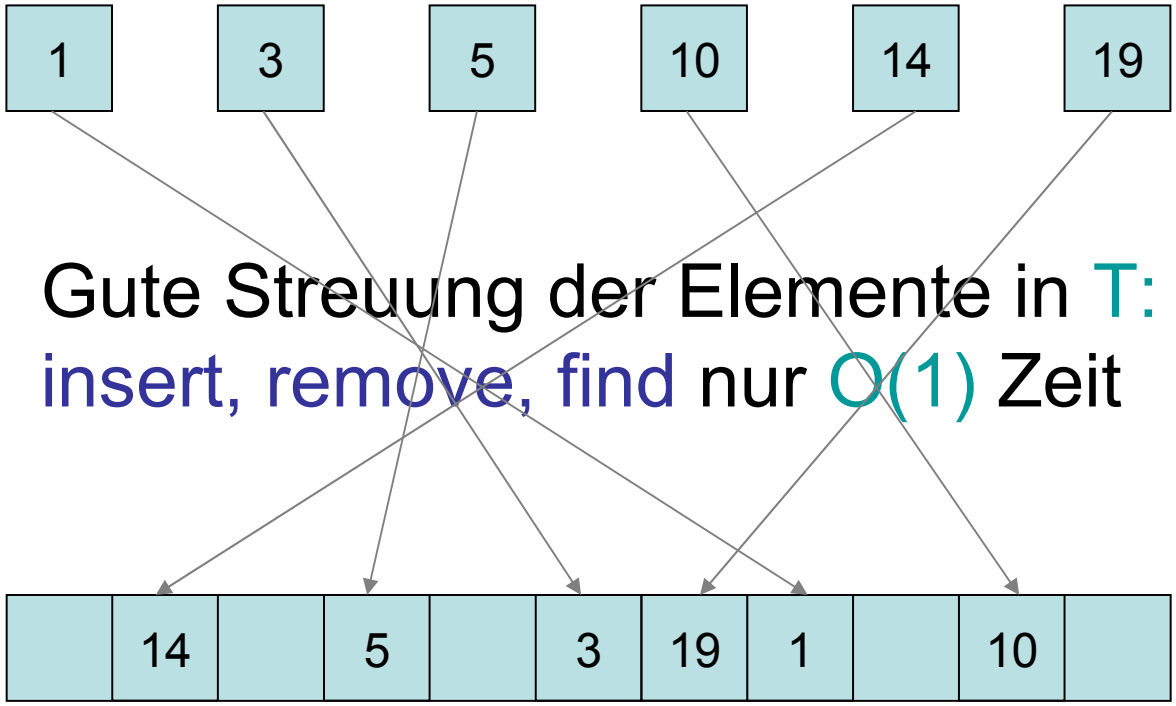
Hashtabelle T

Hashing



Hashtabelle T

Hashing



Hashtabelle T

Hashing (perfekte Streuung)

```
void insert(Elem e) {
   T[h(key(e))] = e;
}
void remove(Key k) {
   T[h(k)] = null;
}
Elem find(Key k) {
   return T[h(k)];
}
```

Hashing

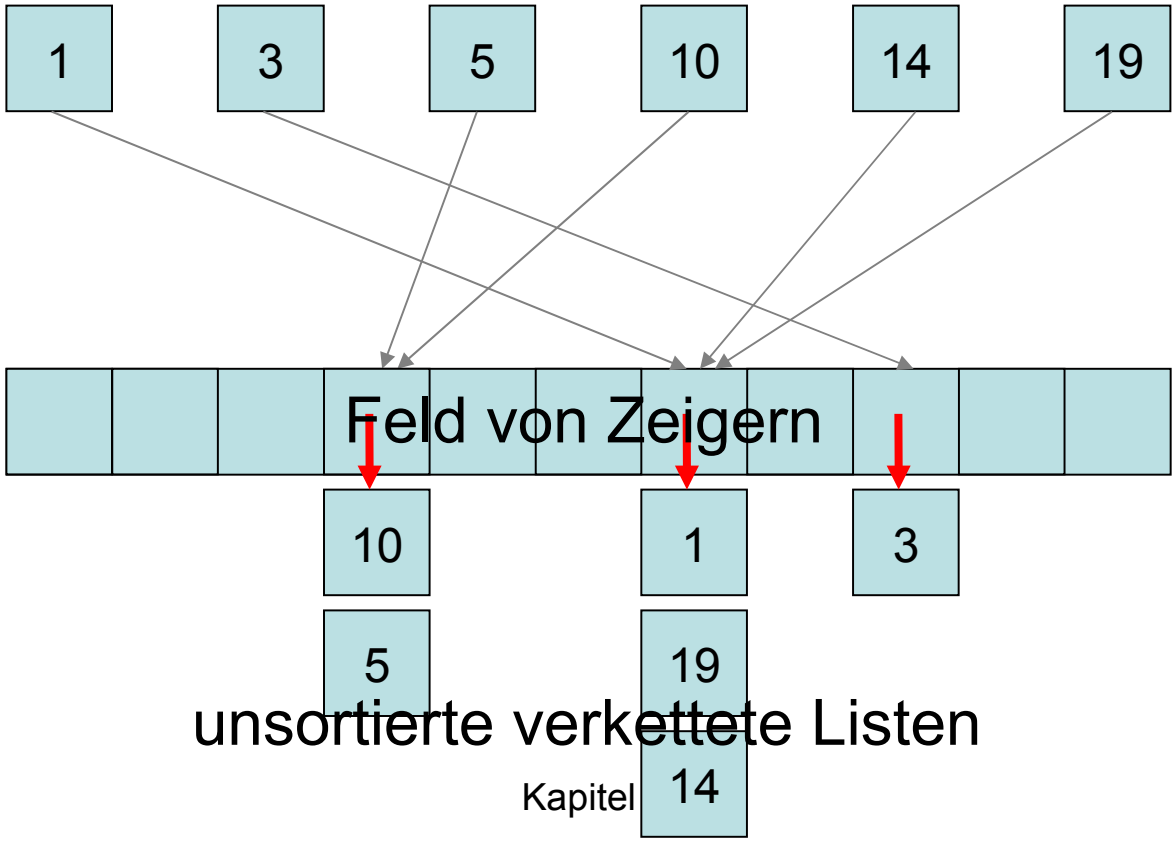
Problem: gute Streuung

Fälle:

- Statisches Wörterbuch: nur find
- Dynamisches Wörterbuch: insert, remove und find

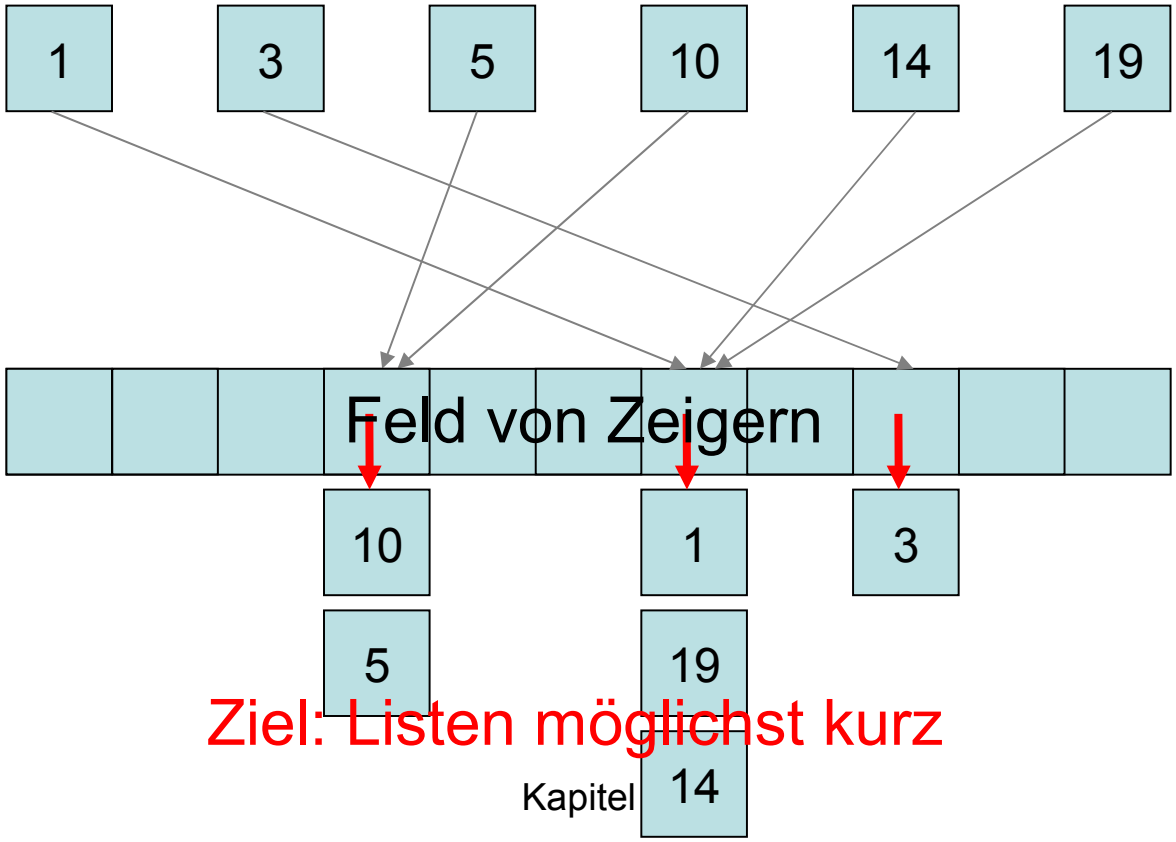
Dynamisches Wörterbuch

Hashing with Chaining:



Dynamisches Wörterbuch

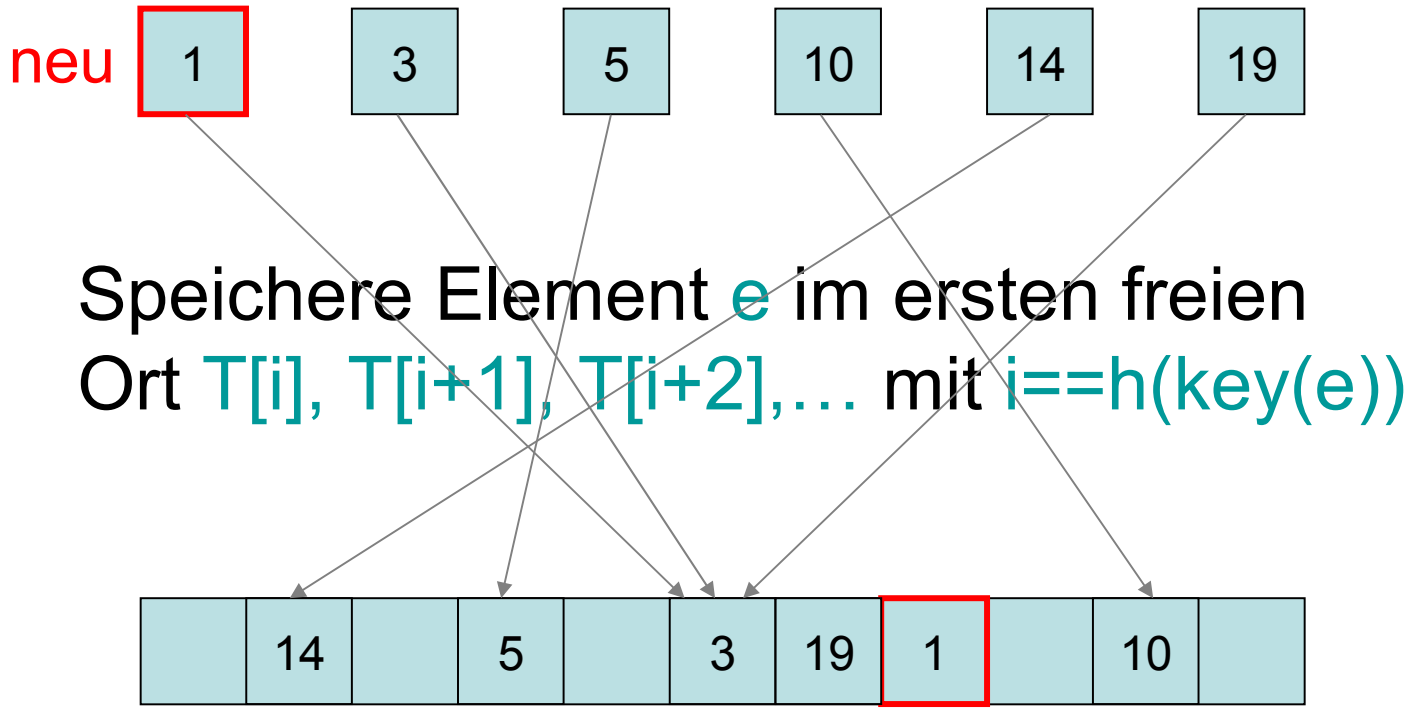
Hashing with Chaining:



Ziel: Listen möglichst kurz

Dynamisches Wörterbuch

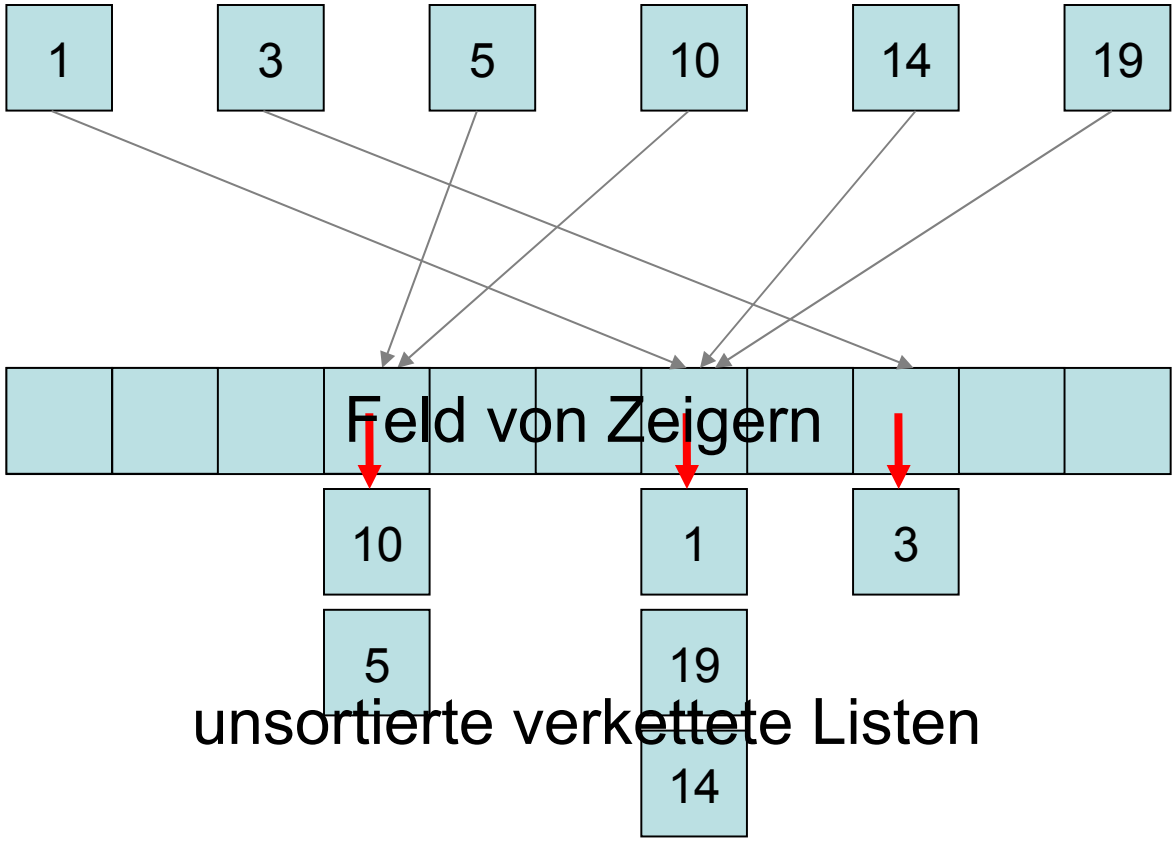
Hashing with Linear Probing:



Dynamisches Wörterbuch

Hashing with Linear Probing:

Hashing with Chaining



Hashing with Chaining

- List<Elem>[] T
- List<Elem>: DS mit Operationen insert, remove, find

```
void insert(Elem e) {
   T[h(key(e))].insert(e);
}
void remove(Key k) {
   T[h(k)].remove(e);
}
Elem find(Key k) {
   return T[h(k)].find(k);
}
```

Hashing with Chaining

Theorem 4.1: Falls n Elemente in einer Hash-tabelle T der Größe m mittels einer zufälligen Hashfunktion h gespeichert werden, dann ist für jedes $T[i]$ die erwartete Anzahl Elemente in $T[i]$ in $O(1+n/m)$.

Beweis:

- Betrachte feste Position $T[i]$
- Zufallsvariable $X_e \in \{0,1\}$ für jedes $e \in S$
- $X_e = 1 \Leftrightarrow h(key(e))=i$
- $X = \sum_e X_e$: Anzahl Elemente in $T[i]$
- $E[X] = \sum_e E[X_e] = \sum_e Pr[X_e=1] = \sum_e (1/m) = n/m$

Hashing with Chaining

Problem: Wie konstruiert man genügend zufällige Hashfunktionen?

Definition 4.2 [universelles Hashing]:
Sei c eine positive Konstante. Eine Familie H von Hashfunktionen auf {0,…,m-1} heißt c-universell falls für ein beliebiges Paar x≠y von Schlüsseln gilt, dass

$$|\{h \in H: h(x)=h(y)\}| \leq (c/m)\ |H|$$

Universelles Hashing

Theorem 4.3: Falls n Elemente in einer Hashtabelle T der Größe m mittels einer zufälligen Hashfunktion h aus einer c-universellen Familie gespeichert werden, dann ist für jedes $T[i]$ die erwartete Anzahl Elemente in $T[i]$ in $O(1+c \cdot n/m)$.

Beweis:

- Betrachte festes Element e_0 mit Position $T[i]$
- Zufallsvariable $X_e \in \{0,1\}$ für jedes $e \in S \setminus \{e_0\}$
- $X_e = 1 \Leftrightarrow h(key(e))=i$
- $X = \sum_e X_e$: Anzahl Elemente in Position $T[i]$ von e_0
- $E[X] = \sum_e E[X_e] = \sum_e Pr[X_e=1] \leq \sum_e (c/m) = (n-1)c/m$

Universelles Hashing

Betrachte die Familie der Hashfunktionen

$$h_a(x) = a \cdot x \,\%\, m$$

mit $a,x \in \{0,\ldots,m-1\}^k$. ($a \cdot x = \sum_i a_i x_i$)

Theorem 4.4: $H=\{ h_a: a \in \{0,\ldots,m-1\}^k \}$ ist eine 1-universelle Familie von Hashfunktionen, falls m prim ist.

Universelles Hashing

Beweis:

Universelles Hashing

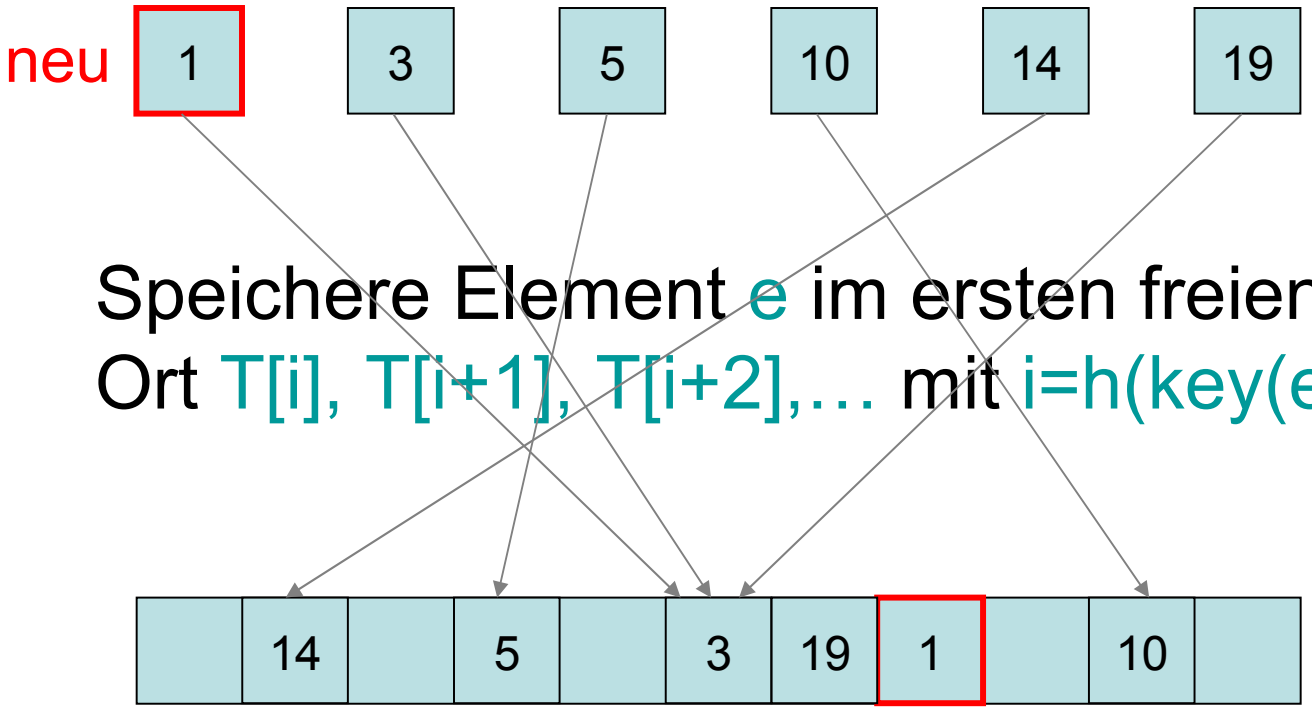
Betrachte die Familie der Hashfunktionen

$$h'_a(x) = \sum_{i=0}^{k-1} a^i x_i \,\%\, m$$

mit $a, x_i \in \{0,\ldots,m-1\}^k$

Theorem 4.5: $H=\{ h'_a : a \in \{0,\ldots,m-1\} \}$ ist eine k-universelle Familie von Hashfunktionen, falls m prim ist.

Hashing with Linear Probing



Speichere Element e im ersten freien Ort T[i], T[i+1], T[i+2],… mit i=h(key(e))

Hashing with Linear Probing

```
Elem [] T;   // Feld sollte genügend groß sein!

void insert(Elem e) {
    i = h(key(e));
    while (T[i] != null && T[i] != e) i = i+1 % m;
    T[i] = e;
}
Elem find(Key k) {
    i = h(k);
    while (T[i] != null && key(T[i]) != k)
          i=i+1 % m;
    return T[i];
}
```

Hashing with Linear Probing

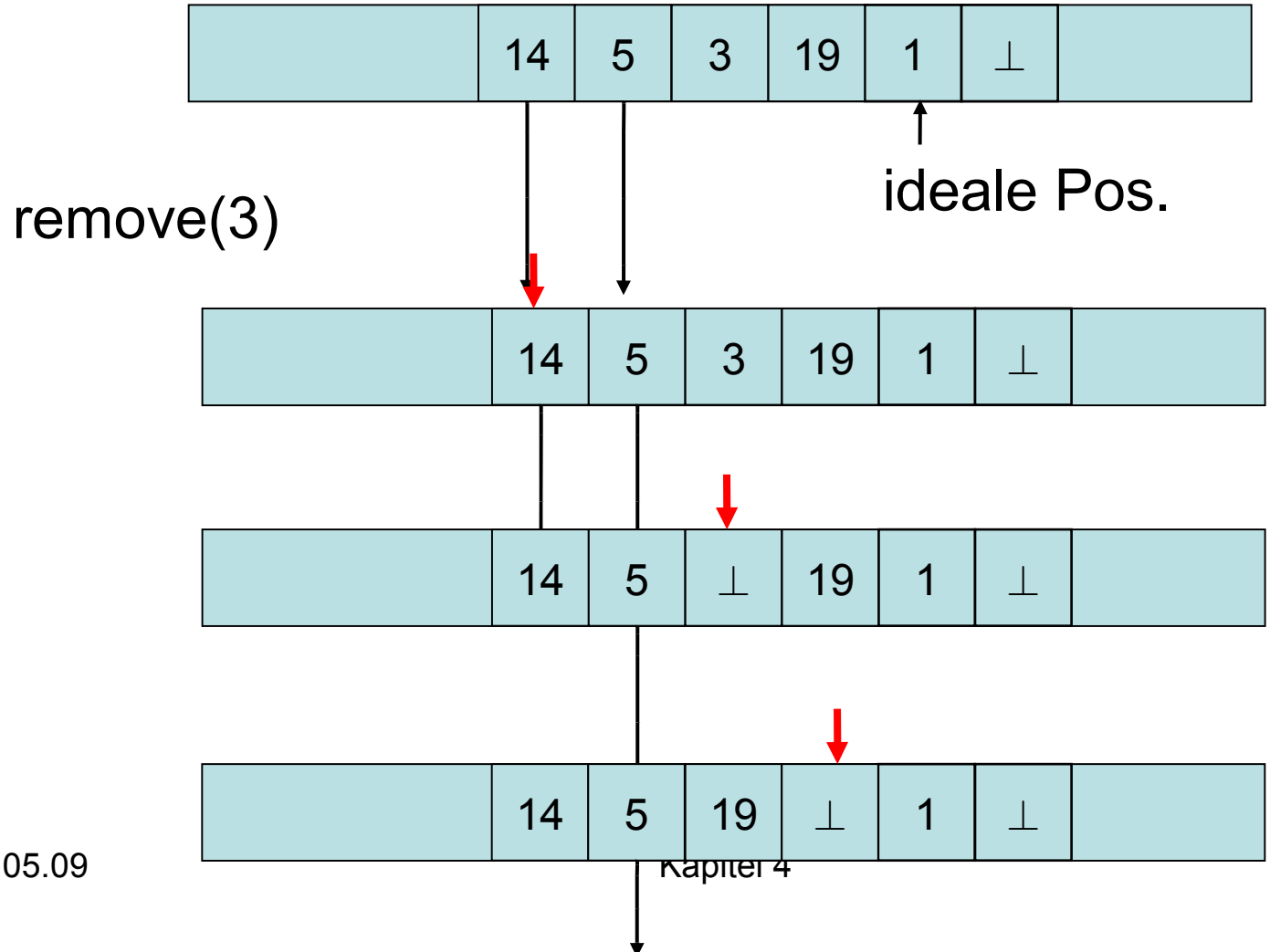
Problem: Löschen von Elementen

Lösungen:

1. Verbiete Löschungen
2. Markiere Position als gelöscht mit speziellem Zeichen (ungleich null)
3. Stelle die folgende Invariante sicher:
   Für jedes $e \in S$ mit idealer Position $i=h(key(e))$ und aktueller Position j gilt

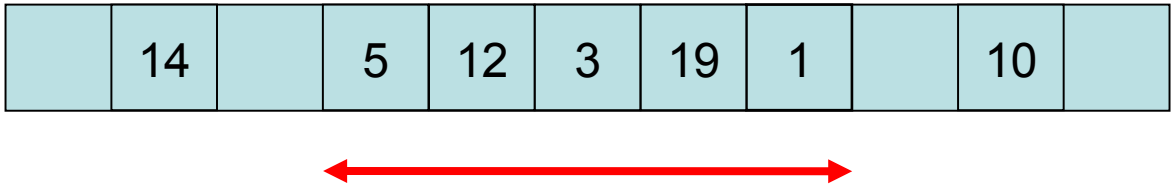
   $T[i], T[i+1] \ldots, T[j]$ sind besetzt

Beispiel für remove Operation



Hashing with Linear Probing

Lauf:



Lauf der Länge 5

Theorem 4.1: Falls n Elemente in einer Hashtabelle T der Größe m>2en mittels einer zufälligen Hashfunktion h gespeichert werden, dann ist für jedes T[i] die erwartete Länge eines Laufes in T, der T[i] enthält, O(1). (e: Eulersche Zahl)

Hashing with Linear Probing

Beweis:

n: Anzahl Elemente, m>2en: Größe der Tabelle

Anzahl Möglichkeiten zur Wahl von k Elementen: $\binom{n}{k}$

Wahrscheinlichkeit, dass Hashwerte in Lauf: $(k/m)^k$

Pr[T[i] in Lauf der Länge k] < $(en/k)^k$ $k(k/m)^k$ < $k(1/2)^k$

Hashing with Linear Probing

- p_k := Pr[T[i] in Lauf der Länge k] < k $(1/2)^k$

E[Länge des Laufs über T[i]]
= $\sum_{k \geq 0} k \cdot p_k < \sum_{k \geq 0} k^2 (1/2)^k = O(1)$

Also erwartet konstanter Aufwand für Operationen insert, remove und find.

Dynamisches Wörterbuch

Problem: Hashtabelle ist zu groß oder zu klein (sollte nur um konstanten Faktor abweichen von der Anzahl der Elemente)

Lösung: Reallokation

- Wähle geeignete Tabellengröße
- Wähle neue Hashfunktion
- Übertrage Elemente auf die neue Tabelle

Dynamisches Wörterbuch

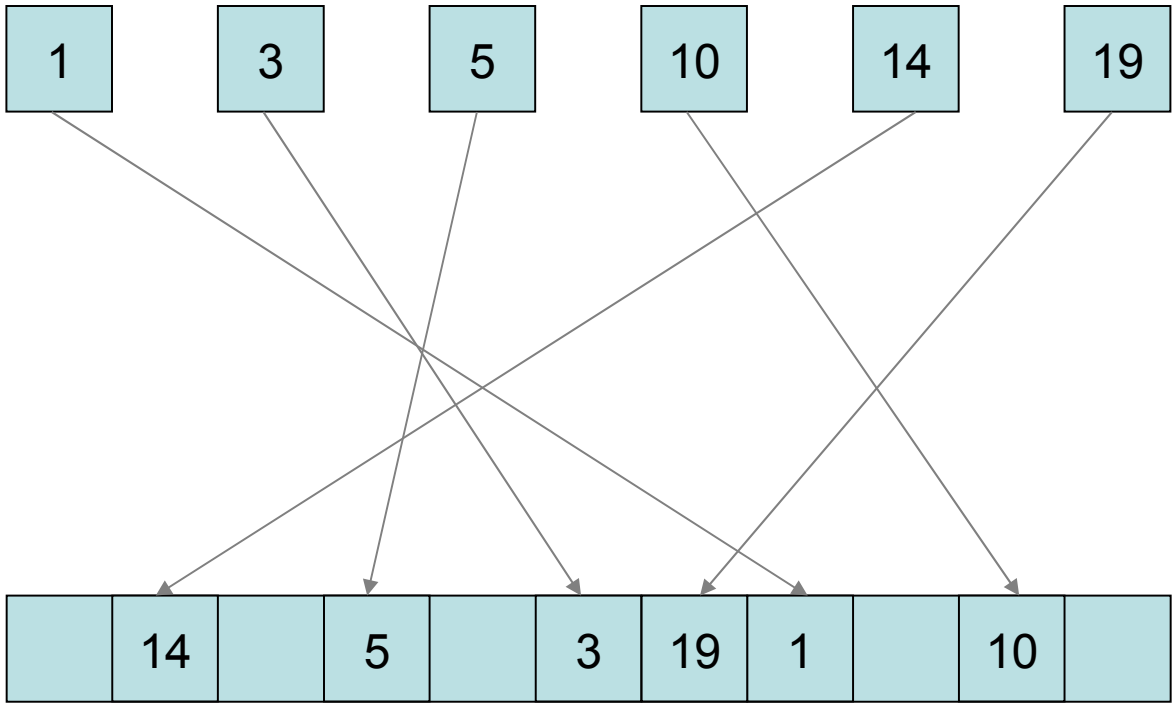
Problem: Tabellengröße m sollte prim sein (für gute Verteilung der Schlüssel)

Lösung:

- Für jedes k gibt es Primzahl in [k,2k]
- Wähle primes m, so dass $m \in [k,2k]$
- Jede nichtprime Zahl in [k,2k] muss Teiler $< \sqrt{2k}$ haben
  $\rightarrow$ erlaubt effiziente Primzahlfindung

Statisches Wörterbuch

Ziel: perfekte Hashtabelle

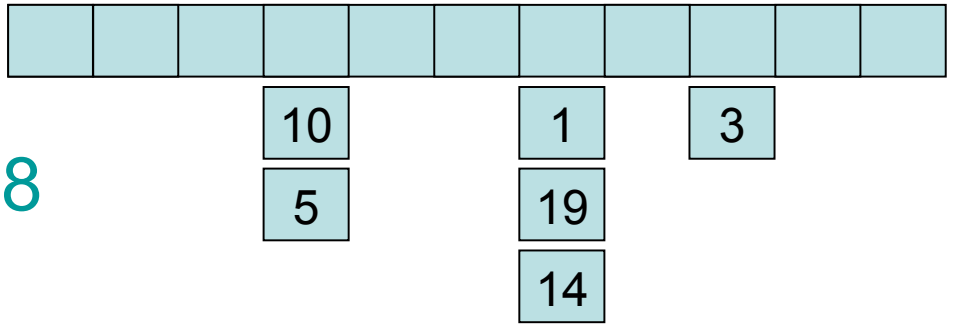


Statisches Wörterbuch

- S: feste Menge an Elementen
- H_m: Familie c-universeller Hashfunktionen auf {0, …,m-1}
- C(h) für ein $h \in H_m$: Anzahl Kollisionen zwischen Elementen in S für h, d.h.

$$C(h) = |\{(x,y): x,y \in S, x \neq y, h(x)=h(y)\}|$$

C(h) = 2+6 = 8



Statisches Wörterbuch

Lemma 4.5: $E[C(h)] \leq c \cdot n(n-1)/m$ und für $\geq 1/2$ der $h \in H_m$ ist $C(h) \leq 2c \cdot n(n-1)/m$.

Statisches Wörterbuch

…für $\geq 1/2$ der $h \in H_m$ ist $C(h) \leq 2c \cdot n(n-1)/m$.

Statisches Wörterbuch

b_i^h: Anzahl Elemente, für die $h(key(x))=i$ ist

Lemma 4.6: $C(h) = \sum_i b_i^h(b_i^h-1)$.

Statisches Wörterbuch

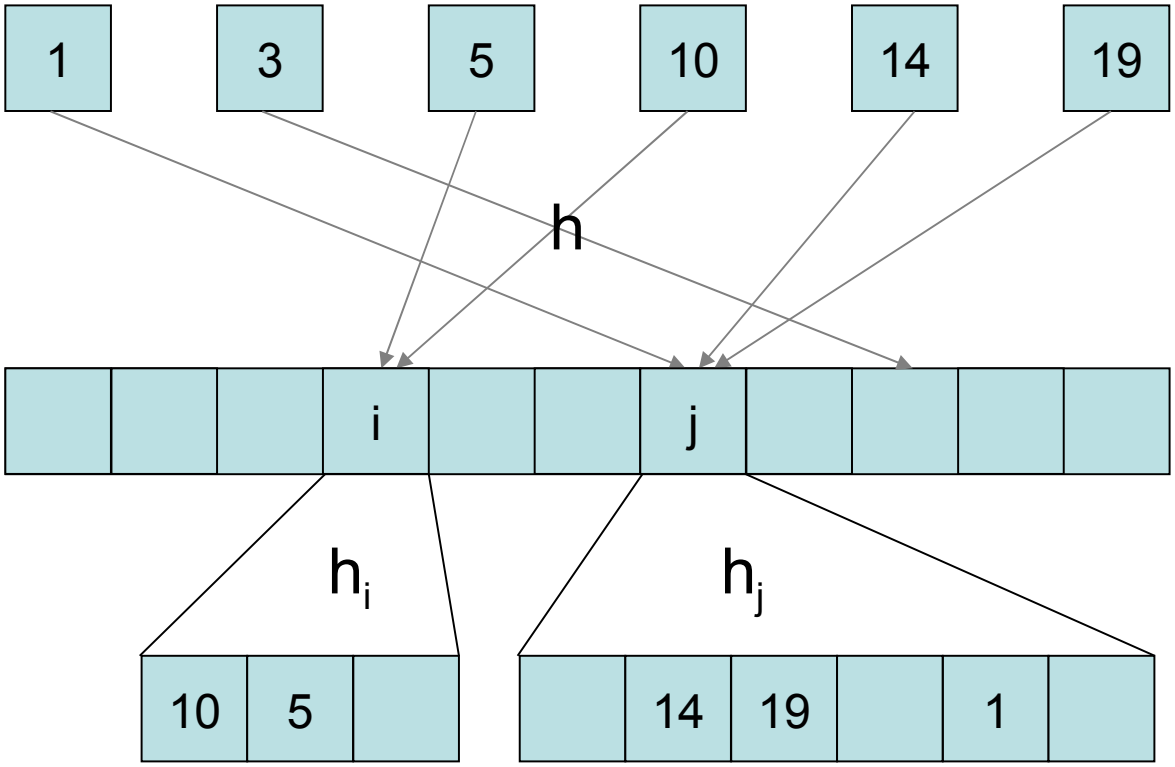
Konstruktion der Hashtabelle:

Wähle eine Funktion $h \in H_{\alpha n}$ mit $C(h) \leq 2cn(n-1)/(\alpha\, n) = 2cn/\alpha$, α konstant
(Lemma 4.6: zufällige Wahl mit Wahrscheinlichkeit ≥ 1/2 erfolgreich)

Für jede Position i sei $m_i = 2cb_i(b_i-1)+1$.

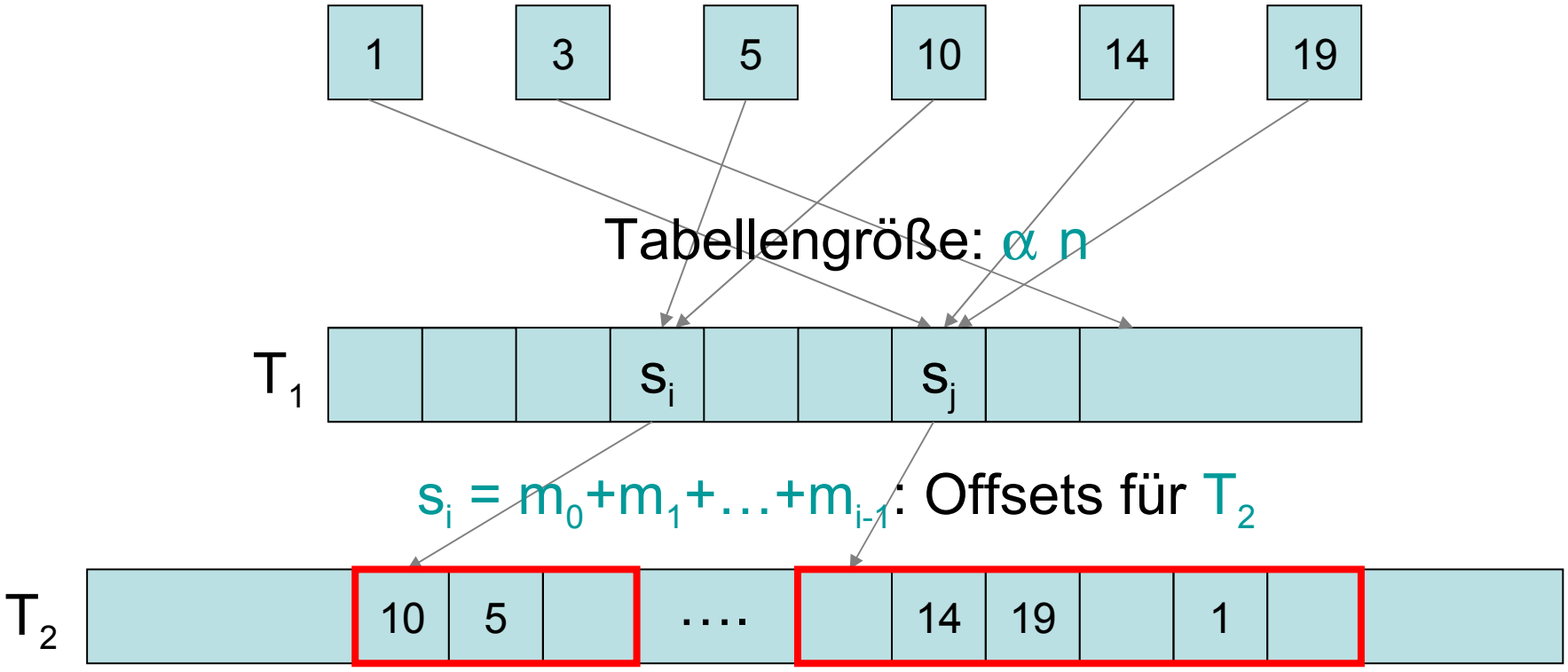
- Wähle eine Funktion $h_i \in H_{m_i}$ mit $C(h_i)<1$ (Lemma 4.6: zufällige Wahl mit Wahrscheinlichkeit ≥ 1/2 erfolgreich)

Statisches Wörterbuch



Keine Kollisionen in Subtabellen

Statisches Wörterbuch



Tabellengröße: $\sum_i m_i \leq \sum_i (2c\, b_i(b_i\text{-}1)+1) \leq 4c\, n/\alpha + \alpha\, n$

Statisches Wörterbuch

Theorem 4.7: Für jede Menge von n Schlüsseln gibt es eine perfekte Hash-funktion der Größe $\Theta(n)$, die in erwarteter Zeit $\Theta(n)$ konstruiert werden kann.

Perfekte Hashfunktionen sind auch dynamisch konstruierbar →

Cukoo Hashing

Nächstes Thema

Weiter mit Kapitel 5: Suchen und Sortieren