



Datenstrukturen, Algorithmen und Programmierung 2 (DAP2)

Datenstrukturen

Problem

- Gegeben sind n Objekte O_1, \dots, O_n mit zugehörigen Schlüsseln $s(O_i)$

Operationen

- **Suche(x)**; Ausgabe O mit Schlüssel $s(O) = x$;
nil, falls kein Objekt mit Schlüssel x in Datenbank
- **Einfügen(O)**; Einfügen von Objekt O in Datenbank
- **Löschen(O)**; Löschen von Objekt O aus der Datenbank

Datenstrukturen

AVL-Bäume

- Balanzierte Binärbäume
- Suchen, Einfügen, Löschen, Min, Max, Nachfolger in $O(\log n)$ Zeit

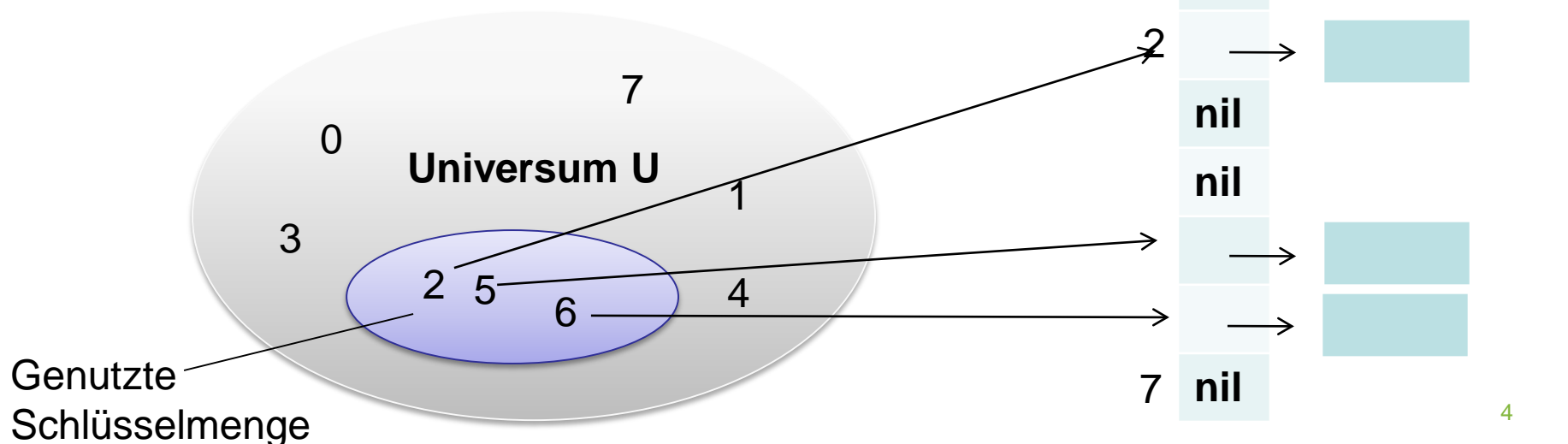
Frage

- Gibt es effizientere Datenstruktur für das Datenbank Problem als AVL-Bäume?

Datenstrukturen

Felder mit direkter Adressierung

- Schlüsselmenge aus Universum $U=\{0,\dots,m-1\}$
- Keine zwei Elemente haben denselben Schlüssel
- Feld $T[0,\dots,m-1]$
- Position i in T ist für Schlüssel i reserviert



Datenstrukturen

Operationen

DirectAddressSearch(k)

1. **return** T[k]

DirectAddressInsert(x)

1. $T[\text{key}[x]] \leftarrow x$

DirectAddressDelete(k)

1. $T[k] \leftarrow \text{nil}$

Datenstrukturen

Operationen

DirectAddressSearch(k)

1. **return** T[k]

DirectAddressInsert(x)

1. T[key[x]] \leftarrow x

DirectAddressDelete(k)

1. T[k] \leftarrow **nil**

Laufzeiten:
O(1)

Datenstrukturen

Zusammenfassung (direkte Adressierung)

- Einfügen, Löschen, Suchen in $O(1)$
- Min, Max $O(|U|)$
- Speicherbedarf $O(|U|)$
- Schlecht, wenn Universum groß ist (normaler Fall)

Datenstrukturen

Hashing

- Ziel: Speicherbedarf soll unabhängig von Universumsgröße sein
- Wollen nur die Suchzeit optimieren
- (Insert und Delete werden auch unter bestimmten Annahmen effizient sein)

Eingabe

- n Schlüssel aus Universum $U=\{0,\dots,m-1\}$


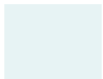
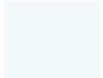
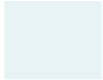
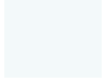
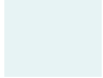

Aufgabe

- Finde Datenstruktur mit $O(n)$ Speicherbedarf, die Suche in $O(1)$ Zeit erlaubt

Datenstrukturen

Erste Idee

- Fasse Blöcke von r Elementen zusammen und bilde sie auf dieselbe Adresse ab

	Feld T
0-9	
10-19	
20-29	
30-39	
40-49	
50-59	
60-69	


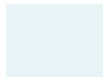
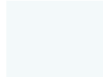
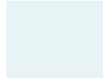
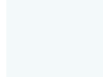
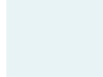
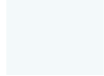
Datenstrukturen

Erste Idee

- Fasse Blöcke von r Elementen zusammen und bilde sie auf dieselbe Adresse ab

Beispiel

- Schlüsselmenge aus Universum $\{0, \dots, 69\}$
8, 13, 15, 30, 41, 56, 58

	Feld T
0-9	
10-19	
20-29	
30-39	
40-49	
50-59	
60-69	

Datenstrukturen

Erste Idee


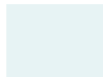
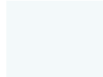
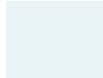
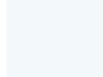
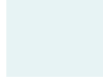
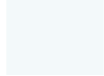
- Fasse Blöcke von r Elementen zusammen und bilde sie auf dieselbe Adresse ab

Beispiel

- Schlüsselmenge aus Universum $\{0, \dots, 69\}$
8, 13, 15, 30, 41, 56, 58

Problem

- 13, 15 und 56, 58 liegen im selben Bereich

Feld T	
0-9	
10-19	
20-29	
30-39	
40-49	
50-59	
60-69	

Datenstrukturen

Erste Idee

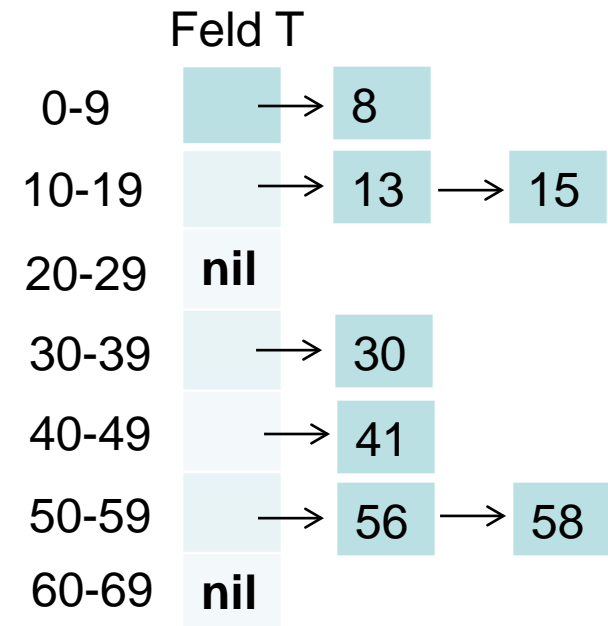
- Fasse Blöcke von r Elementen zusammen und bilde sie auf dieselbe Adresse ab

Beispiel

- Schlüsselmenge aus Universum $\{0, \dots, 69\}$
8, 13, 15, 30, 41, 56, 58

Problem

- 13, 15 und 56, 58 liegen im selben Bereich
- Auflösen durch Listen



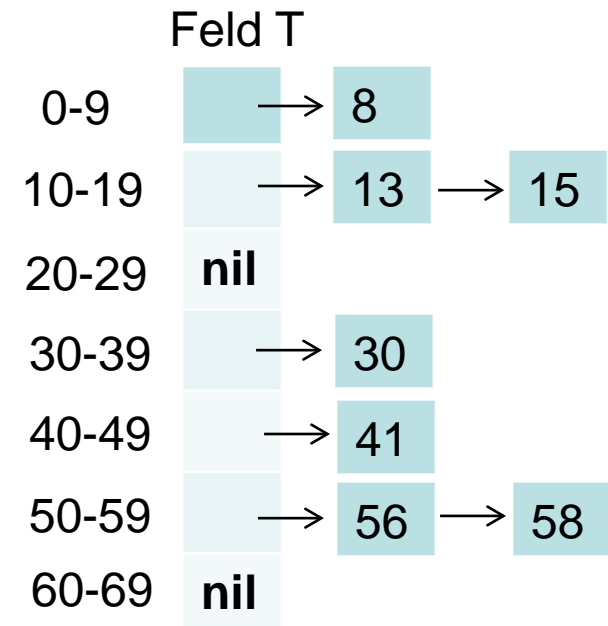
Datenstrukturen

Insert(x)

1. $p \leftarrow \lfloor \text{key}[x]/r \rfloor$
2. ListInsert($T[p], x$)

Laufzeit

- $O(1)$



Datenstrukturen

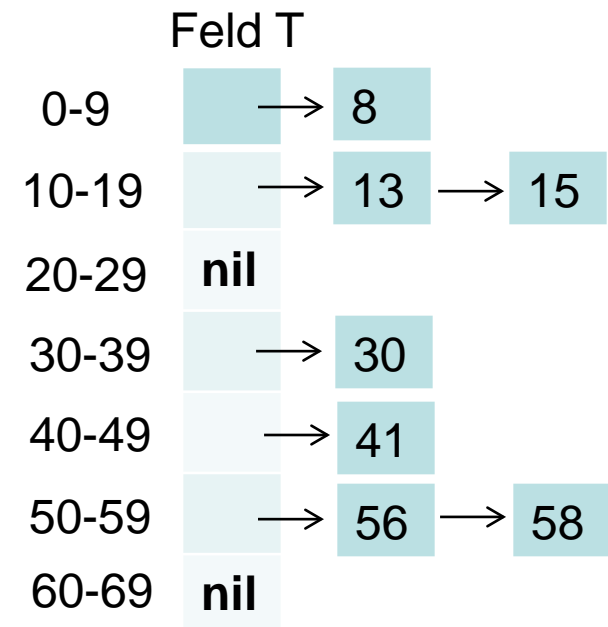
y ist Referenz auf
das zu löschende
Listenelemente

Delete(y)

1. ListDelete(y)

Laufzeit

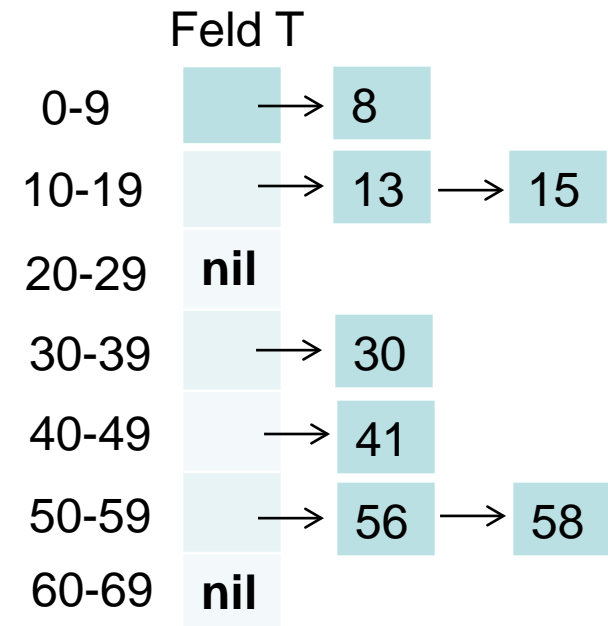
- $O(1)$



Datenstrukturen

Blocksuche(k)

1. $p \leftarrow \lfloor k/r \rfloor$
2. $\text{ListItem} \leftarrow \text{head}[T[p]]$
3. **while** $\text{ListItem} \neq \text{nil}$ and $\text{key}[\text{ListItem}] \neq k$ **do**
4. $\text{ListItem} \leftarrow \text{next}[\text{ListItem}]$
5. **return** ListItem



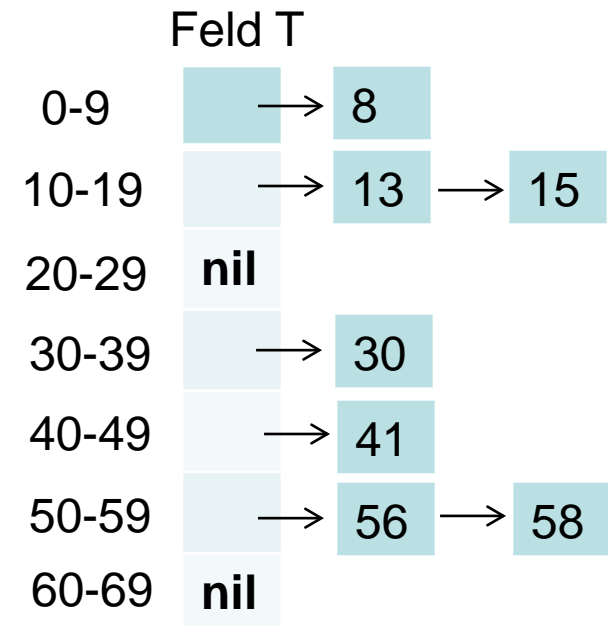
Datenstrukturen

Blocksuche(k)

1. $p \leftarrow \lfloor k/r \rfloor$
2. $\text{ListItem} \leftarrow \text{head}[T[p]]$
3. **while** $\text{ListItem} \neq \text{nil}$ and $\text{key}[\text{ListItem}] \neq k$ **do**
4. $\text{ListItem} \leftarrow \text{next}[\text{ListItem}]$
5. **return** ListItem

Beispiel

- Blocksuche(15)



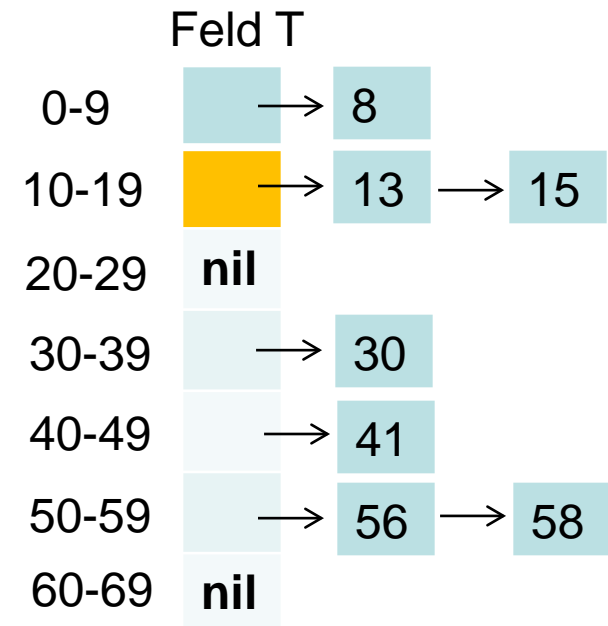
Datenstrukturen

Blocksuche(k)

1. $p \leftarrow \lfloor k/r \rfloor$
2. ListItem $\leftarrow \text{head}[T[p]]$
3. **while** ListItem $\neq \text{nil}$ and $\text{key}[\text{ListItem}] \neq k$ **do**
4. ListItem $\leftarrow \text{next}[\text{ListItem}]$
5. **return** ListItem

Beispiel

- Blocksuche(15)



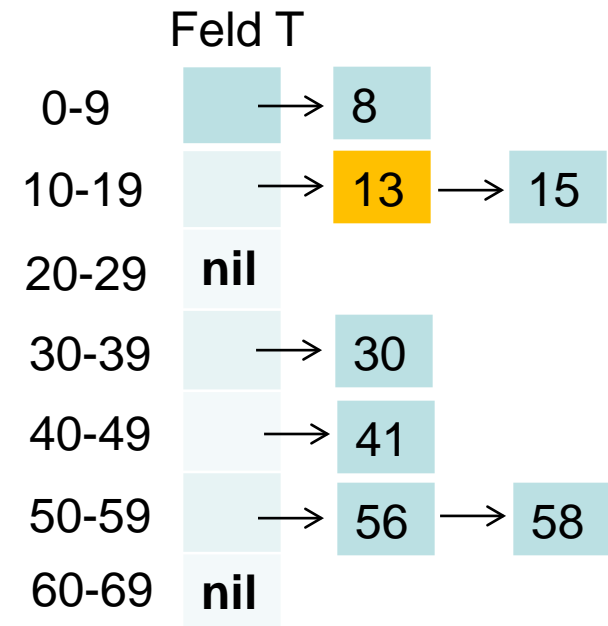
Datenstrukturen

Blocksuche(k)

1. $p \leftarrow \lfloor k/r \rfloor$
2. $\text{ListItem} \leftarrow \text{head}[T[p]]$
3. **while** ListItem \neq nil and key[ListItem] \neq k **do**
4. ListItem \leftarrow next[ListItem]
5. **return** ListItem

Beispiel

- Blocksuche(15)



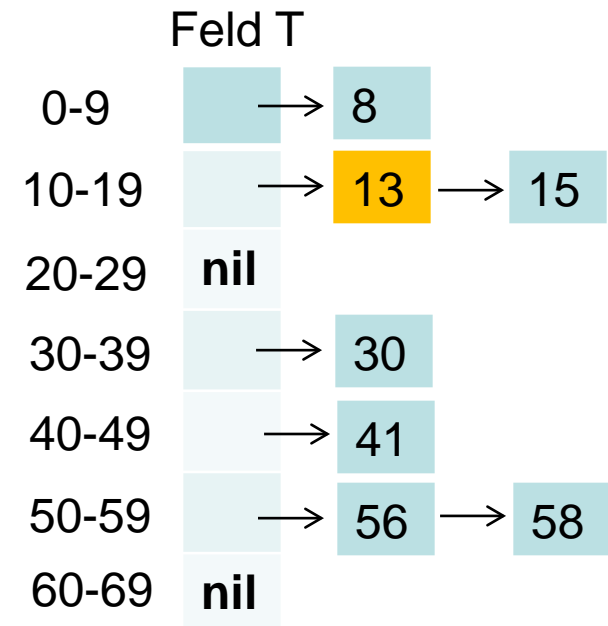
Datenstrukturen

Blocksuche(k)

1. $p \leftarrow \lfloor k/r \rfloor$
2. $\text{ListItem} \leftarrow \text{head}[T[p]]$
3. **while** $\text{ListItem} \neq \text{nil}$ and $\text{key}[\text{ListItem}] \neq k$ **do**
4. $\text{ListItem} \leftarrow \text{next}[\text{ListItem}]$
5. **return** ListItem

Beispiel

- Blocksuche(15)



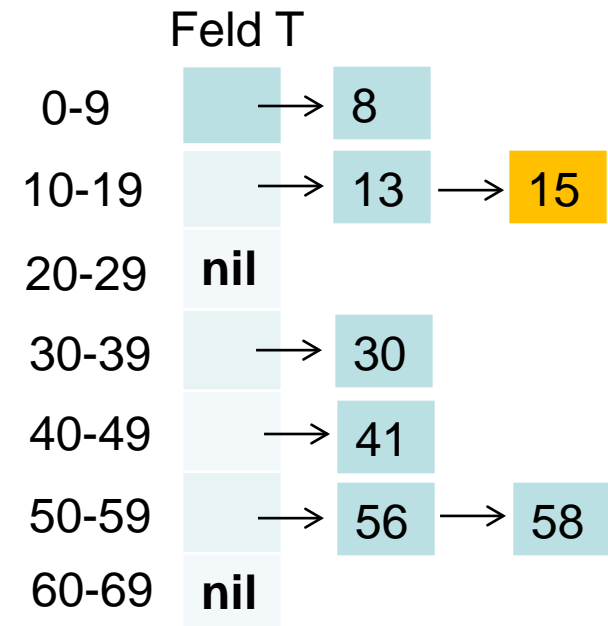
Datenstrukturen

Blocksuche(k)

1. $p \leftarrow \lfloor k/r \rfloor$
2. $\text{ListItem} \leftarrow \text{head}[T[p]]$
3. **while** $\text{ListItem} \neq \text{nil}$ and $\text{key}[\text{ListItem}] \neq k$ **do**
4. $\text{ListItem} \leftarrow \text{next}[\text{ListItem}]$
5. **return** ListItem

Beispiel

- Blocksuche(15)



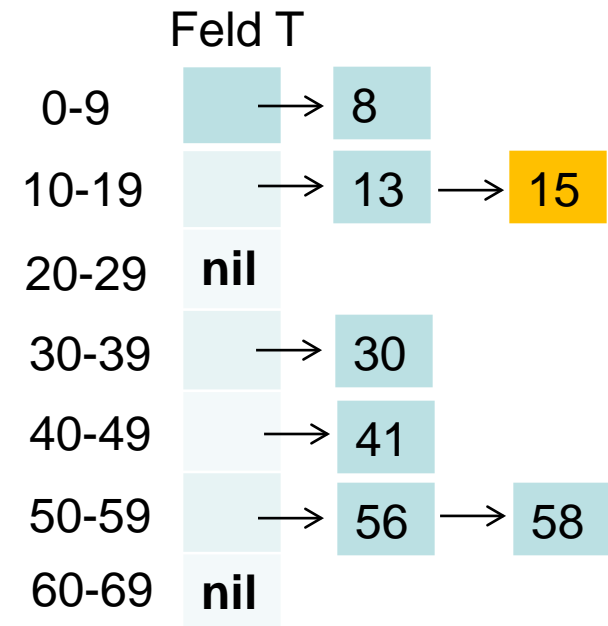
Datenstrukturen

Blocksuche(k)

1. $p \leftarrow \lfloor k/r \rfloor$
2. $\text{ListItem} \leftarrow \text{head}[T[p]]$
3. **while** $\text{ListItem} \neq \text{nil}$ and $\text{key}[\text{ListItem}] \neq k$ **do**
4. $\text{ListItem} \leftarrow \text{next}[\text{ListItem}]$
5. **return** ListItem

Beispiel

- Blocksuche(15)



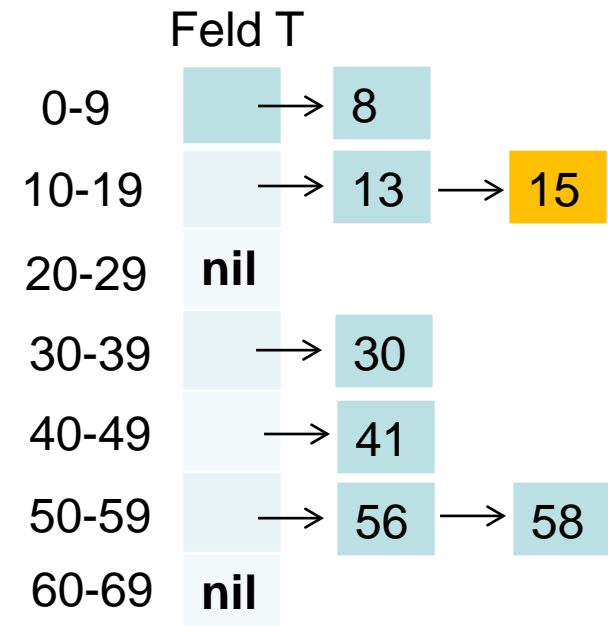
Datenstrukturen

Blocksuche(k)

1. $p \leftarrow \lfloor k/r \rfloor$
2. $\text{ListItem} \leftarrow \text{head}[T[p]]$
3. **while** $\text{ListItem} \neq \text{nil}$ and $\text{key}[\text{ListItem}] \neq k$ **do**
4. $\text{ListItem} \leftarrow \text{next}[\text{ListItem}]$
5. **return** ListItem

Beispiel

- Blocksuche(15)



Datenstrukturen

Analyse

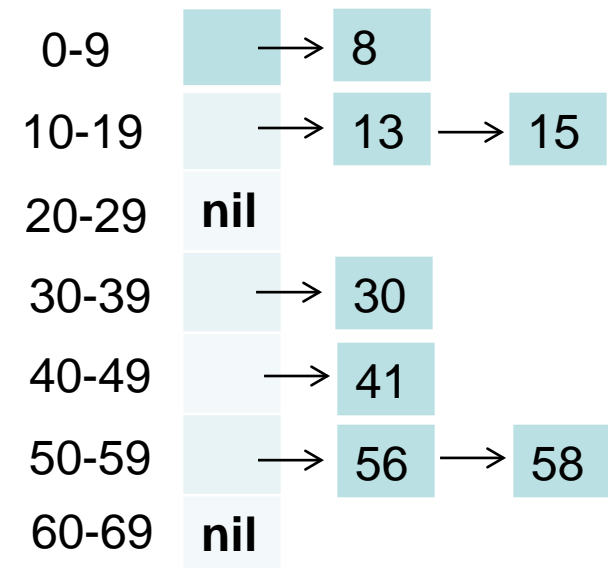
- Wollen Suchzeit für festen Schlüssel k analysieren

Worst-Case

- Alle Schlüssel aus demselben Block sind in Schlüsselmenge
- Suchzeit: $O(\min\{r, n\})$
- Ist $r > n$, so ist dies $O(n)$

Diskussion

- Ist das wirklich, was wir erwarten?
- Nein! Das ist eine sehr spezielle Eingabe
- Normalerweise sollte das besser funktionieren



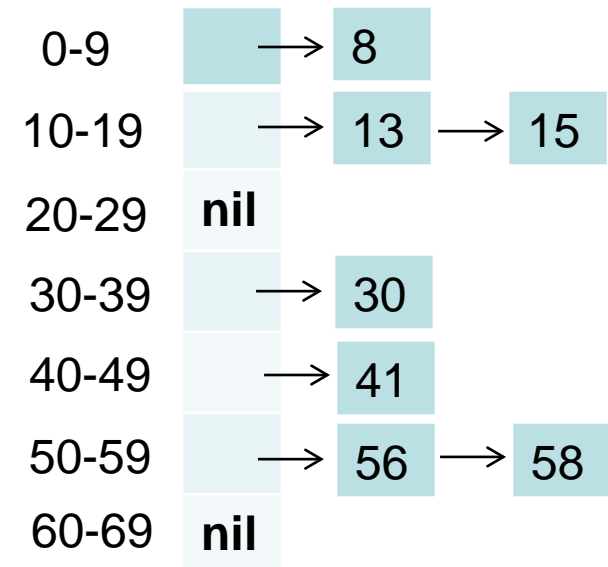
Datenstrukturen

Analyse

- Wollen Suchzeit für festen Schlüssel k analysieren

Average-Case

- Durchschnittliche Laufzeit über alle möglichen Schlüsselmengen



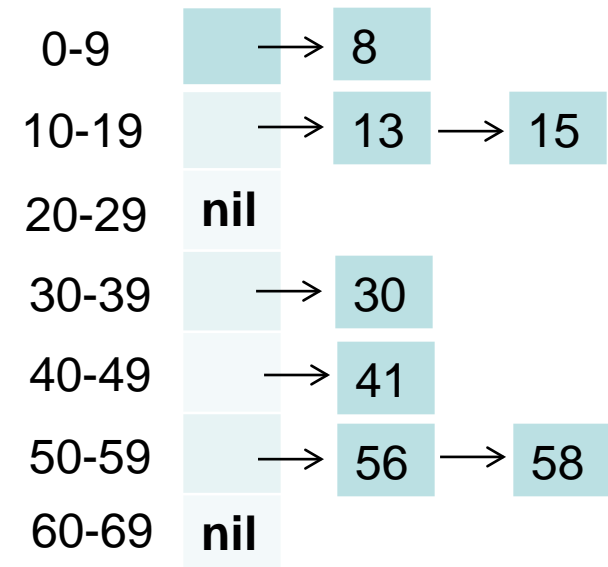
Datenstrukturen

Analyse

- Wollen Suchzeit für festen Schlüssel k analysieren

Average-Case

- Durchschnittliche Laufzeit über alle möglichen Schlüsselmengen
- Durchschnittliche Länge β jeder Liste ist $\beta = r \cdot n / m$



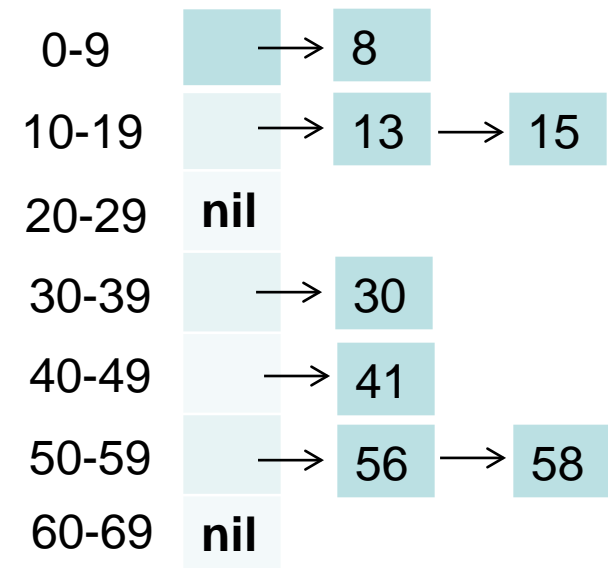
Datenstrukturen

Analyse

- Wollen Suchzeit für festen Schlüssel k analysieren

Average-Case

- Durchschnittliche Laufzeit über alle möglichen Schlüsselmengen
- Durchschnittliche Länge β jeder Liste ist $\beta = r \cdot n / m$
- Durchschnittliche Suchzeit $O(1 + \beta)$



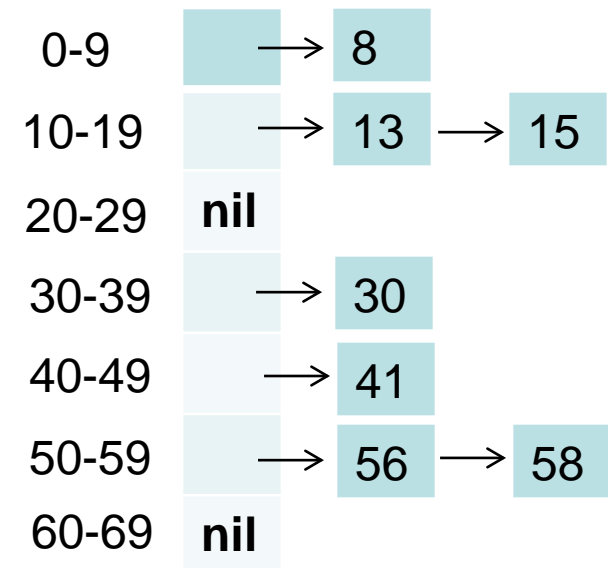
Datenstrukturen

Analyse

- Wollen Suchzeit für festen Schlüssel k analysieren

Average-Case

- Durchschnittliche Laufzeit über alle möglichen Schlüsselmengen
- Durchschnittliche Länge β jeder Liste ist $\beta = r \cdot n / m$
- Durchschnittliche Suchzeit $O(1 + \beta)$
- Speicherplatz $O(m/r + n)$



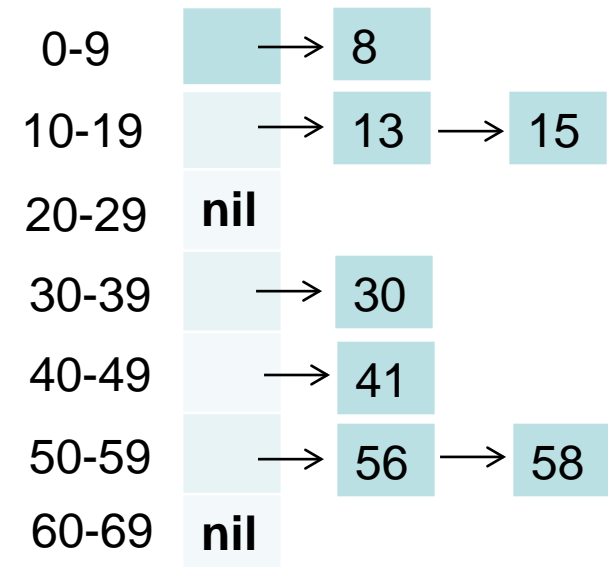
Datenstrukturen

Analyse

- Wollen Suchzeit für festen Schlüssel k analysieren

Average-Case

- Durchschnittliche Laufzeit über alle möglichen Schlüssel Mengen
- Durchschnittliche Länge β jeder Liste ist $\beta = r \cdot n / m$
- Durchschnittliche Suchzeit $O(1 + \beta)$
- Speicherplatz $O(m/r + n)$
- Setze Blockgröße $r = m/n$



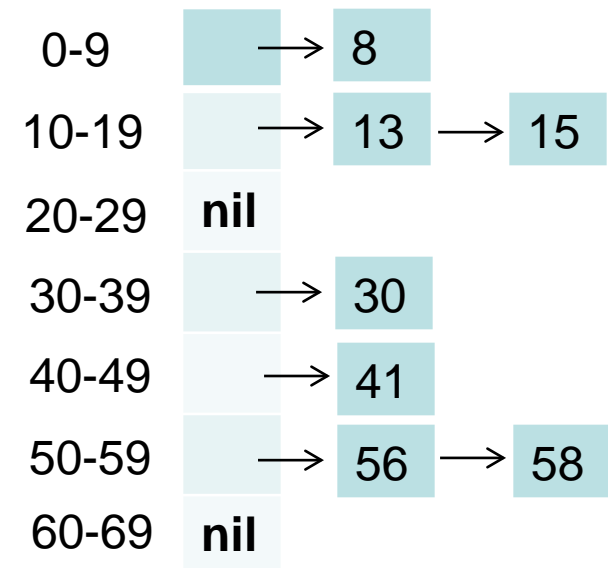
Datenstrukturen

Analyse

- Wollen Suchzeit für festen Schlüssel k analysieren

Average-Case

- Durchschnittliche Laufzeit über alle möglichen Schlüsselmenngen
- Durchschnittliche Länge β jeder Liste ist $\beta = r \cdot n / m$
- Durchschnittliche Suchzeit $O(1 + \beta)$
- Speicherplatz $O(m/r + n)$
- Setze Blockgröße $r = m/n$
- $\Rightarrow O(1)$ durch. Suchzeit und $O(n)$ Speicher



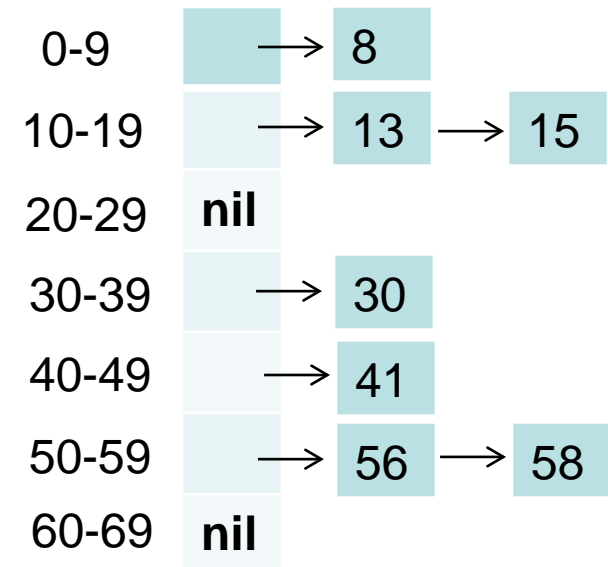
Datenstrukturen

Analyse

- Wollen Suchzeit für festen Schlüssel k analysieren

Average-Case

- Durchschnittliche Laufzeit über alle möglichen Schlüssel Mengen
- Durchschnittliche Länge β jeder Liste ist $\beta = r \cdot n / m$
- Durchschnittliche Suchzeit $O(1 + \beta)$
- Speicherplatz $O(m/r + n)$
- Setze Blockgröße $r = m/n$
- $\Rightarrow O(1)$ durch. Suchzeit und $O(n)$ Speicher



Datenstrukturen

Satz

- Sei $U = \{0, \dots, m-1\}$ eine Grundmenge von Schlüsseln (Universum). Sei T ein Feld mit m/r Einträgen und jeder Eintrag von T entspreche einem Block von r Werten aus U . Dann gilt, dass die durchschnittliche Suchzeit nach einem beliebigen, aber festen Schlüssel k durch $O(1+\beta)$ beschränkt ist, wobei $\beta = r \cdot n/m$ und der Durchschnitt über alle n -elementigen Teilmengen von U gebildet wird.

Datenstrukturen

Diskussion

- Ist Durchschnitt das richtige Maß für eine Laufzeitanalyse?
- Durchschnitt \neq Durchschnitt
(unsere Durchschnittsbildung nimmt an, dass jede Teilmenge gleichwahrscheinlich auftritt; dies ist vermutlich nicht realistisch)

Beispiel

- Universum ist die Menge der long ints
- Schlüssel sind Kundennummern
- Häufig starten Kundennummern bei einem bestimmten Wert und steigen von dort an (z.B. 1 bis 5323)
- „Durchschnittsannahme“ nicht richtig

Datenstrukturen

Problem

- Wir kennen die „typische“ Datenverteilung nicht
- Diese kann insbesondere von der Anwendung abhängen
- Um eine gute Vorhersage der Laufzeit zu machen, müssten wir bei der Durchschnittsbildung aber die typische Datenverteilung berücksichtigen

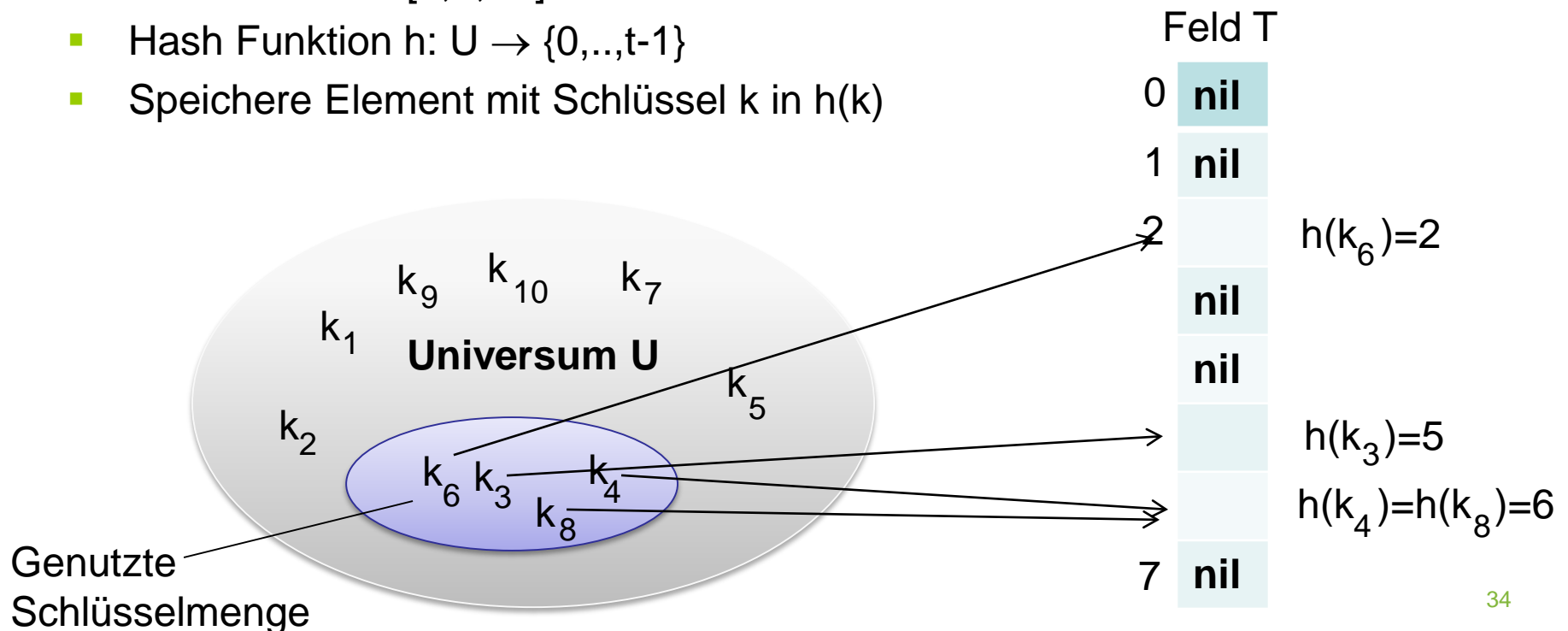
Abhilfe

- Wir werden die Aufteilung zufällig machen

Datenstrukturen

Hashing

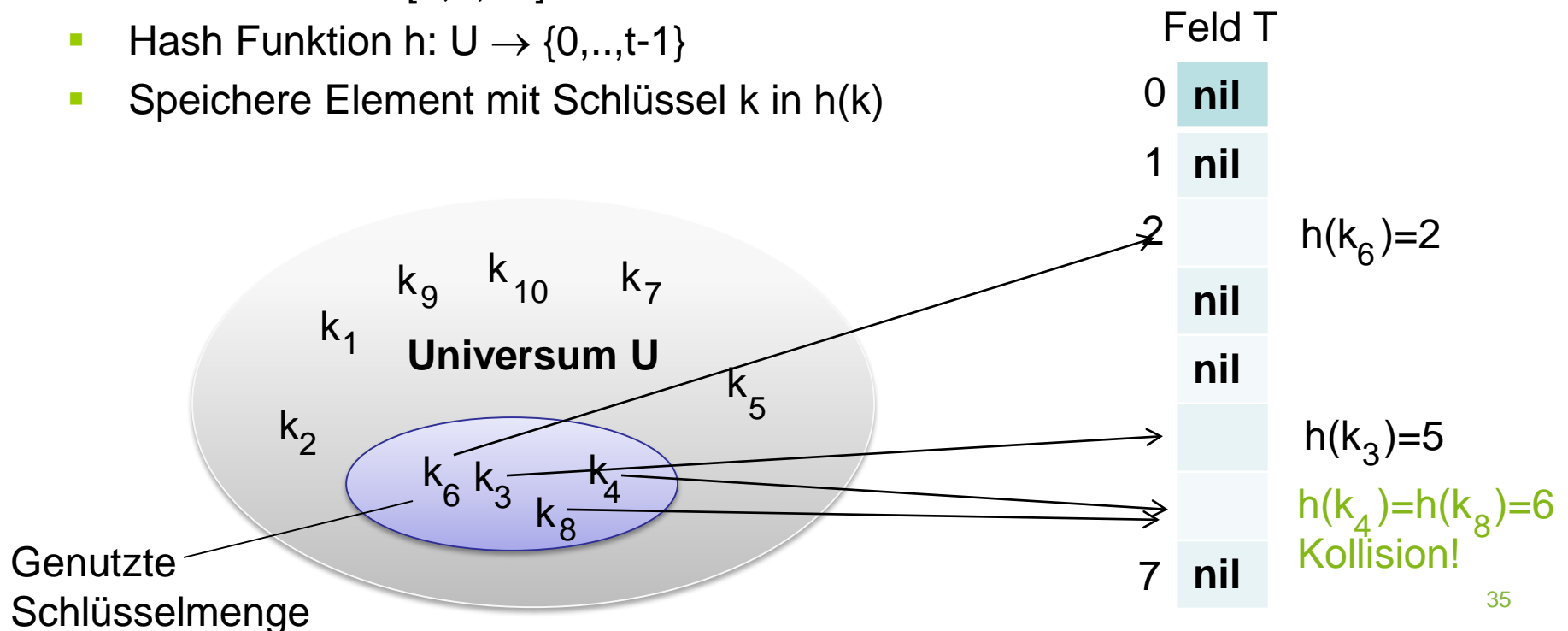
- Universum $U = \{0, \dots, m-1\}$
- Hash Tabelle $T[0, \dots, t-1]$
- Hash Funktion $h: U \rightarrow \{0, \dots, t-1\}$
- Speichere Element mit Schlüssel k in $h(k)$



Datenstrukturen

Hashing

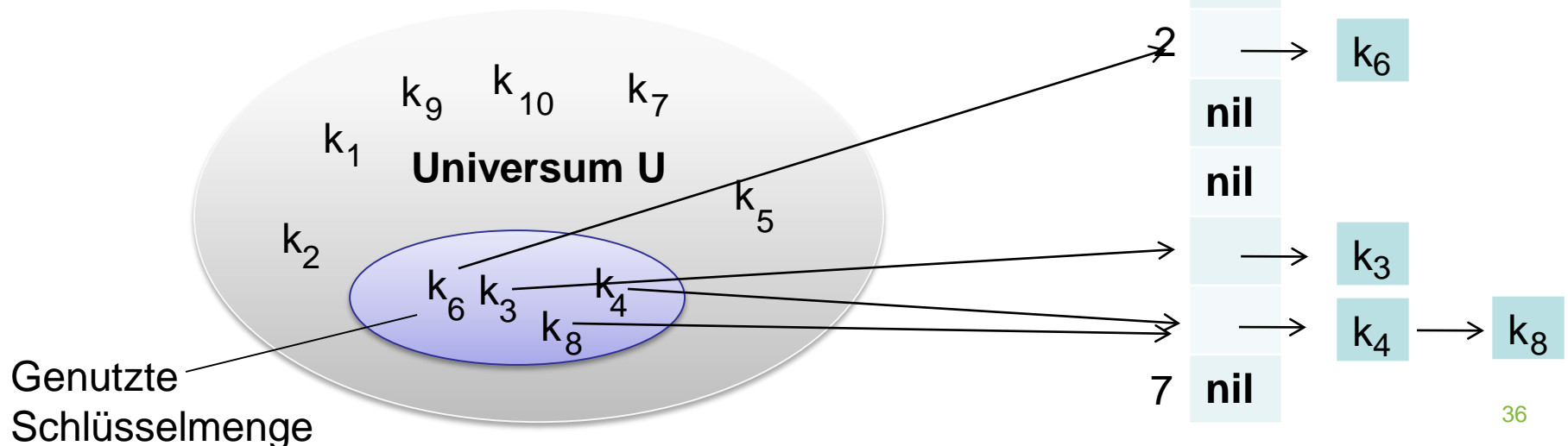
- Universum $U = \{0, \dots, m-1\}$
- Hash Tabelle $T[0, \dots, t-1]$
- Hash Funktion $h: U \rightarrow \{0, \dots, t-1\}$
- Speichere Element mit Schlüssel k in $h(k)$



Datenstrukturen

Hashing mit Verkettung

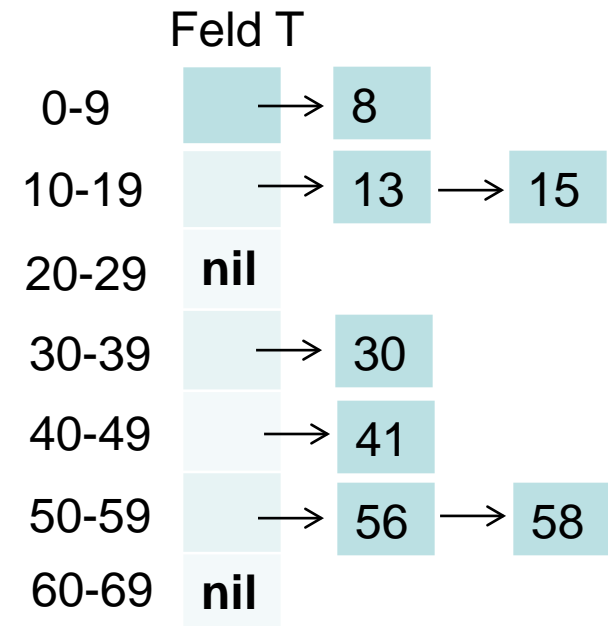
- Universum $U = \{0, \dots, m-1\}$
- Hash Tabelle $T[0, \dots, t-1]$
- Hash Funktion $h: U \rightarrow \{0, \dots, t-1\}$
- Speichere Element mit Schlüssel k in $h(k)$
- Löse Kollisionen durch Listen auf (wie vorhin)



Datenstrukturen

Beispiel

- Wenn wir $h: U \rightarrow \{0, \dots, t-1\}$ durch $h(x) = \lfloor x \cdot t/m \rfloor$ definieren, so haben wir die auf Blockbildung basierende Datenstruktur (mit Blockgröße $r=m/t$)
- Dieses ist also ein Spezialfall des Hashing-Szenarios
- Die Hauptschwierigkeit beim Hashing ist die Frage, wie man h geschickt wählt



Datenstrukturen

Operationen

Einfügen(x)

- Füge neuen Schlüssel k am Ende der Liste $T[h(\text{key}[x])]$ ein

Löschen(x)

- Lösche Element x aus Liste $T[h(\text{key}[x])]$

Suche(k)

- Suche nach k in Liste $T[h(k)]$

Datenstrukturen

Wie sieht eine gute Hashfunktion aus?

- Benutzte Schlüssel sollten möglichst gleichmäßig auf Tabelle verteilt werden
- Guter Kandidat wäre eine zufällige Funktion
(die natürlich nur einmal zu Beginn zufällig gewählt wird und dann fest ist)
- Sobald h festliegt, gibt es immer eine schlechte Eingabe für h mit Worst-Case Suchzeit $O(n)$ bei n Elementen in der Datenstruktur
- Wir suchen aber für gegebene Schlüsselmenge eine gute Funktion h

Last Faktor α

- Durchschnittliche Länge einer Kollisionsliste, d.h. $\alpha = n/t$

Datenstrukturen

Idee

- Wähle h zufällig (aus einer Menge von geeigneten Kandidaten H)

Annahme (einfaches gleichverteiltes Hashing)

- Jedes k aus U wird mit Wahrscheinlichkeit $1/t$ auf $i \in \{0, \dots, t-1\}$ abgebildet
- Diese Wahrscheinlichkeit ist komplett unabhängig vom Bild aller anderen Elemente

Auswahlprozess für h

- Für jede $k \in U$ würfele einen Wert w zwischen 0 und $t-1$ und setze $h[k]=w$
- H : Menge aller Funktionen von U nach $\{0, \dots, t-1\}$

Datenstrukturen

Idee

- Wähle h zufällig (aus einer Menge von geeigneten Kandidaten H)

Annahme (einfaches gleichverteiltes Hashing)

- Jedes k aus U wird mit Wahrscheinlichkeit $1/t$ auf $i \in \{0, \dots, t-1\}$ abgebildet
- Diese Wahrscheinlichkeit ist komplett unabhängig vom Bild aller anderen Elemente

Weitere Annahme

- $h(k)$ kann in $O(1)$ Zeit berechnet werden

Datenstrukturen

Satz

- Sei $M \subseteq U$ eine beliebige Teilmenge von n Schlüsseln und sei h eine Hashfunktion, die zufällig unter der Annahme des einfachen gleichverteilten Hashings ausgewählt wurde. Werden die Kollisionen die unter h auftreten durch Verkettung aufgelöst, so benötigt eine Suche nach Schlüssel $k \notin M$ eine durchschnittliche Laufzeit von $O(1+\alpha)$.

Datenstrukturen

Satz

- Sei $M \subseteq U$ eine beliebige Teilmenge von n Schlüsseln und sei h eine Hashfunktion, die zufällig unter der Annahme des einfachen gleichverteilten Hashings ausgewählt wurde. Werden die Kollisionen die unter h auftreten durch Verkettung aufgelöst, so benötigt eine Suche nach Schlüssel $k \notin M$ eine durchschnittliche Laufzeit von $O(1+\alpha)$.

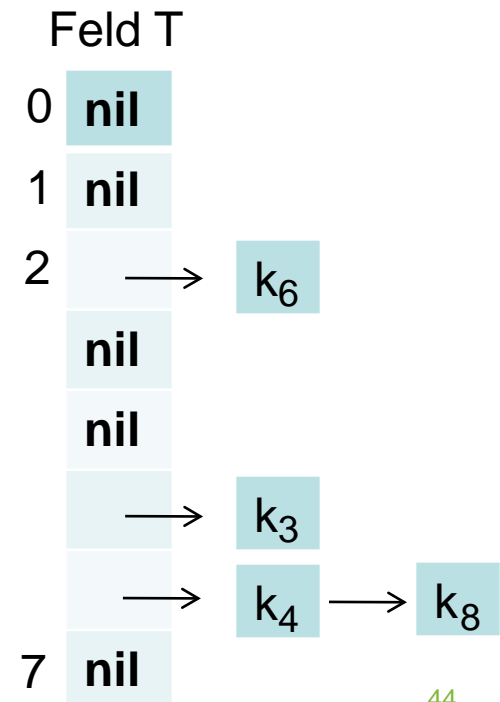
Beweis

- Jeder Schlüssel k wird unter der Annahme des einfachen gleichverteilten Hashings auf jede Position in T mit derselben Wahrscheinlichkeit abgebildet.
- Also ist die durchschnittliche Suchzeit nach k gerade die durchschnittliche Suchzeit bis zum Ende der t Listen. Diese ist $O(1+\alpha)$, denn α ist durchschn. Listenlänge.
- Damit ergibt sich inklusive Berechnung von $h(k)$ eine Suchzeit von $O(1+\alpha)$.

Datenstrukturen

Satz

- Sei $M \subseteq U$ eine beliebige Teilmenge von n Schlüsseln und sei h eine Hashfunktion, die zufällig unter der Annahme des einfachen gleichverteilten Hashings ausgewählt wurde. Werden die Kollisionen die unter h auftreten durch Verkettung aufgelöst, so benötigt eine **Suche nach Schlüssel $k \in M$** eine durchschnittliche Laufzeit von $O(1 + \alpha)$. **Dabei wird der Durchschnitt über die Auswahl von h und den Schlüssel $k \in M$ gebildet.**



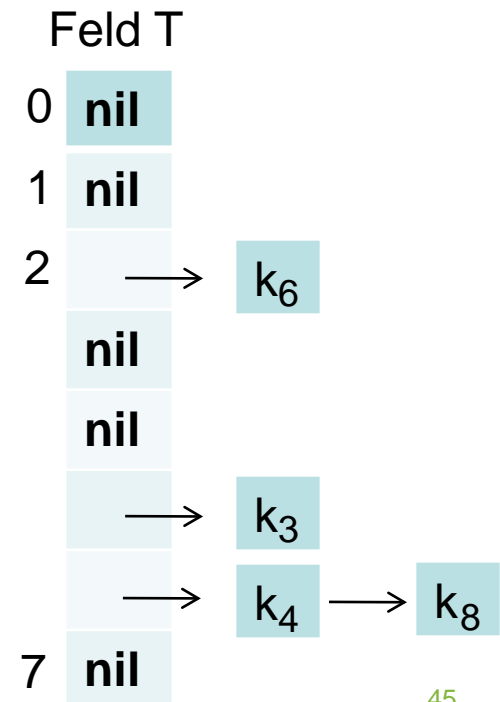
Datenstrukturen

Satz

- Sei $M \subseteq U$ eine beliebige Teilmenge von n Schlüsseln und sei h eine Hashfunktion, die zufällig unter der Annahme des einfachen gleichverteilten Hashings ausgewählt wurde. Werden die Kollisionen die unter h auftreten durch Verkettung aufgelöst, so benötigt eine Suche nach Schlüssel $k \in M$ eine durchschnittliche Laufzeit von $O(1 + \alpha)$. Dabei wird der Durchschnitt über die Auswahl von h und den Schlüssel $k \in M$ gebildet.

Schwierigkeit

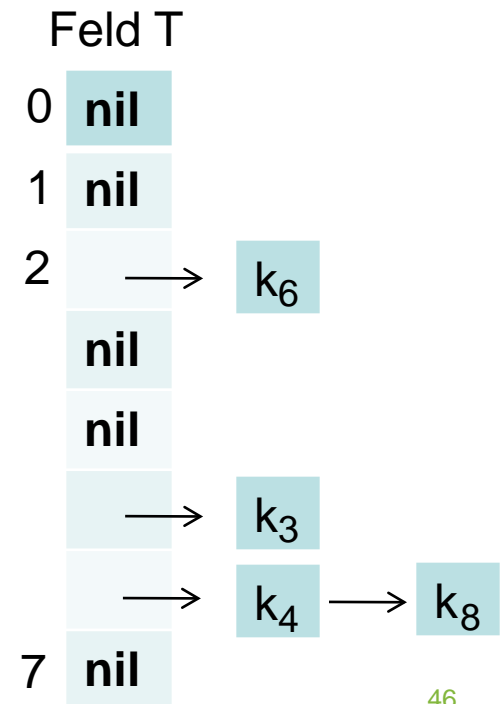
- Die Suchzeit hängt von Position des gesuchten Elements in Kollisionsliste ab
- Suchzeit hängt von Einfügereihenfolge und Implementierung ab



Datenstrukturen

Beweis

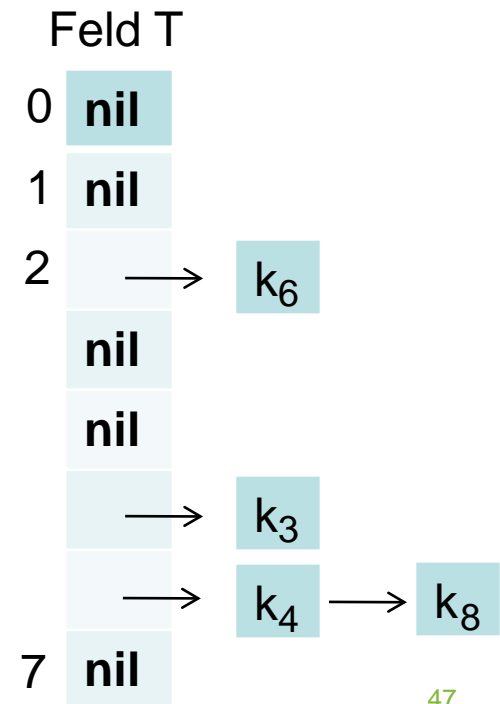
- Annahme: Einfügen am Ende der Listen



Datenstrukturen

Beweis

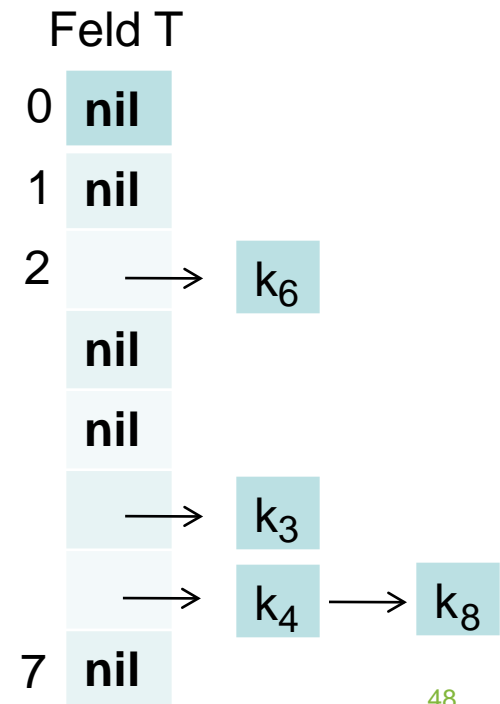
- Annahme: Einfügen am Ende der Listen
- Betrachte Elemente in Einfügereihenfolge



Datenstrukturen

Beweis

- Annahme: Einfügen am Ende der Listen
- Betrachte Elemente in Einfügereihenfolge
- Zunächst: Durchschn. Suchzeit für i -tes Element

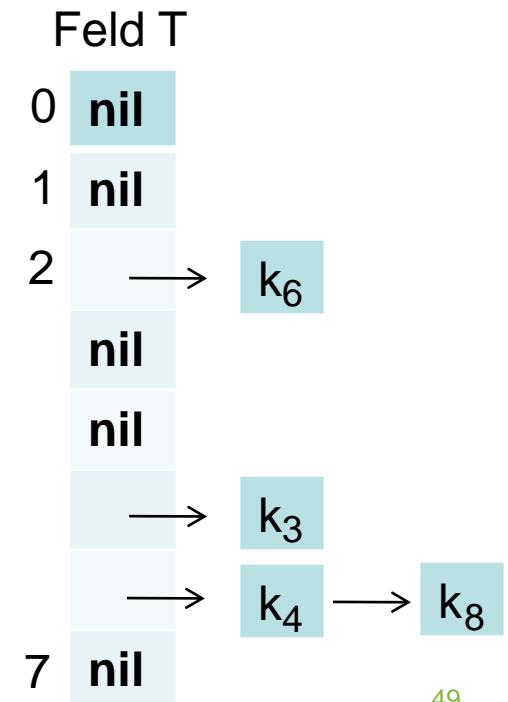


Datenstrukturen

Beweis

- Annahme: Einfügen am Ende der Listen
- Betrachte Elemente in Einfügereihenfolge
- Zunächst: Durchschn. Suchzeit für i-tes Element

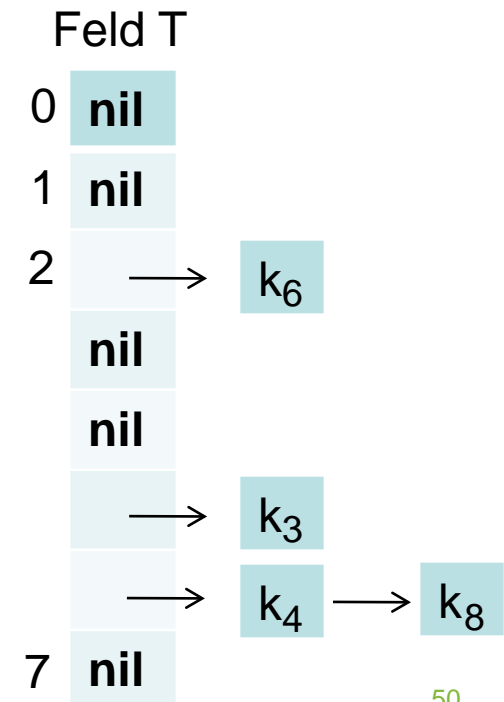
- **Situation vor Einfügen von i:**
- i-1 Elemente eingefügt



Datenstrukturen

Beweis

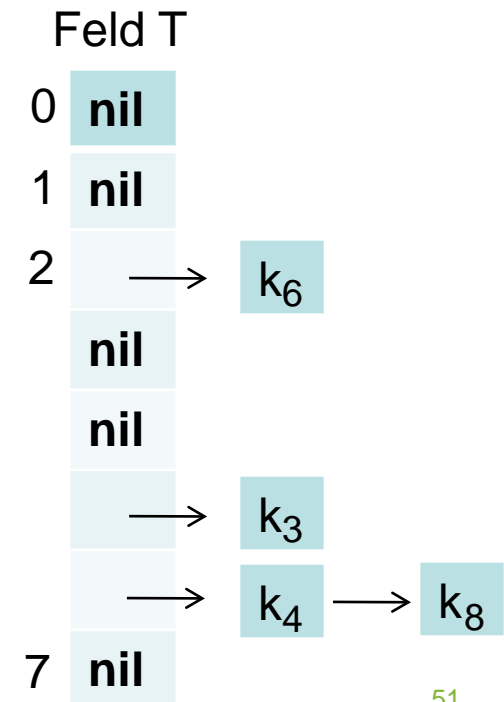
- Annahme: Einfügen am Ende der Listen
- Betrachte Elemente in Einfügereihenfolge
- Zunächst: Durchschn. Suchzeit für i -tes Element
- **Situation vor Einfügen von i :**
- $i-1$ Elemente eingefügt
- Durchschn. Listenlänge $(i-1)/t$



Datenstrukturen

Beweis

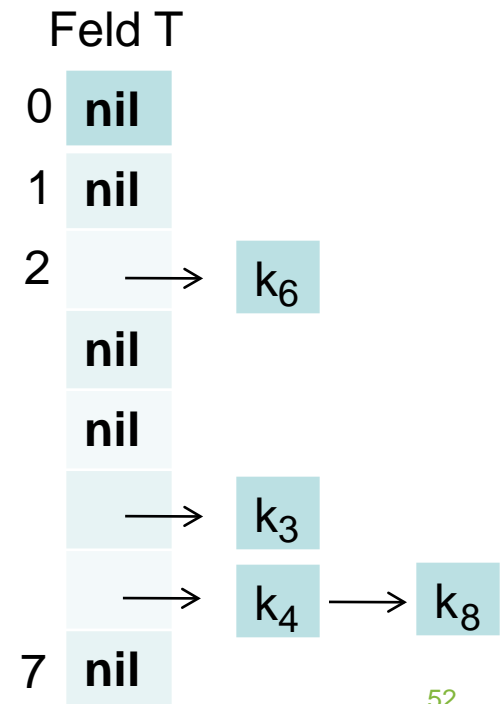
- Annahme: Einfügen am Ende der Listen
- Betrachte Elemente in Einfügereihenfolge
- Zunächst: Durchschn. Suchzeit für i -tes Element
- **Situation vor Einfügen von i :**
- $i-1$ Elemente eingefügt
- Durchschn. Listenlänge $(i-1)/t$
- $h(i)$ ist zufällig aus $\{0, \dots, t-1\}$



Datenstrukturen

Beweis

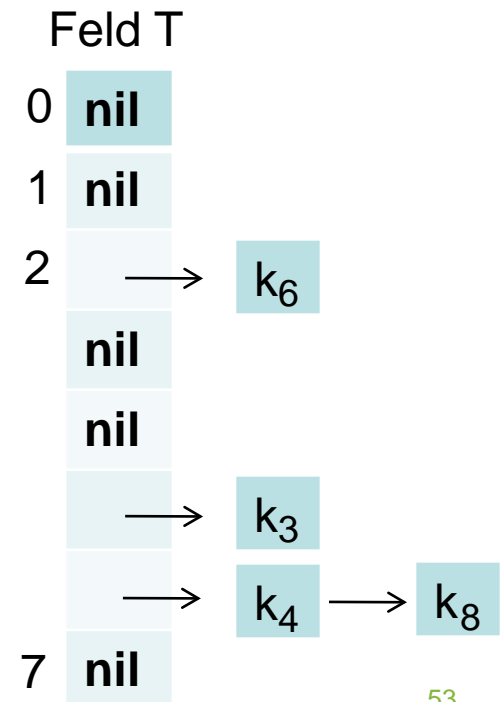
- Annahme: Einfügen am Ende der Listen
- Betrachte Elemente in Einfügereihenfolge
- Zunächst: Durchschn. Suchzeit für i -tes Element
- **Situation vor Einfügen von i :**
- $i-1$ Elemente eingefügt
- Durchschn. Listenlänge $(i-1)/t$
- $h(i)$ ist zufällig aus $\{0, \dots, t-1\}$
- Damit durchschn. Länge der Liste, in der i ist:
- $1 + (i-1)/t$



Datenstrukturen

Beweis

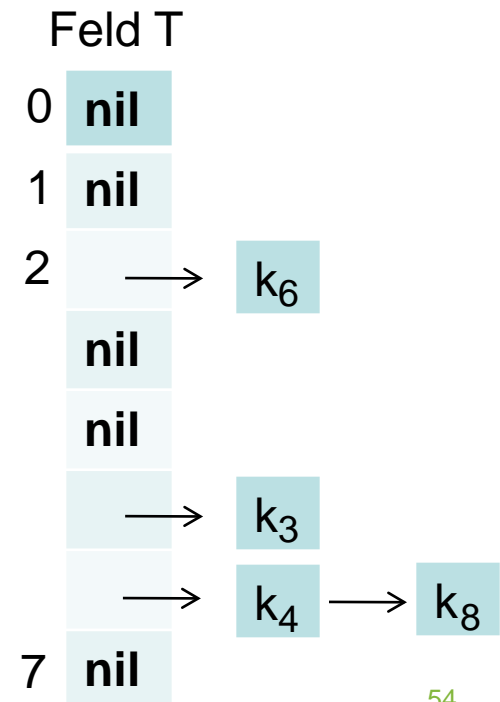
- Annahme: Einfügen am Ende der Listen
- Betrachte Elemente in Einfügereihenfolge
- Zunächst: Durchschn. Suchzeit für i -tes Element
- **Situation vor Einfügen von i :**
 - $i-1$ Elemente eingefügt
 - Durchschn. Listenlänge $(i-1)/t$
 - $h(i)$ ist zufällig aus $\{0, \dots, t-1\}$
 - Damit durchschn. Länge der Liste, in der i ist:
 $1 + (i-1)/t$
- **Durchschn. Suchzeit für i -tes Element:**
 - $O(1 + (i-1)/t)$



Datenstrukturen

Beweis

- Annahme: Einfügen am Ende der Listen
- Betrachte Elemente in Einfügereihenfolge
- Zunächst: Durchschn. Suchzeit für i-tes Element
- **Situation vor Einfügen von i:**
 - i-1 Elemente eingefügt
 - Durchschn. Listenlänge $(i-1)/t$
 - $h(i)$ ist zufällig aus $\{0, \dots, t-1\}$
 - Damit durchschn. Länge der Liste, in der i ist:
 $1 + (i-1)/t$
- **Durchschn. Suchzeit für i-tes Element:**
 - $O(1 + (i-1)/t)$

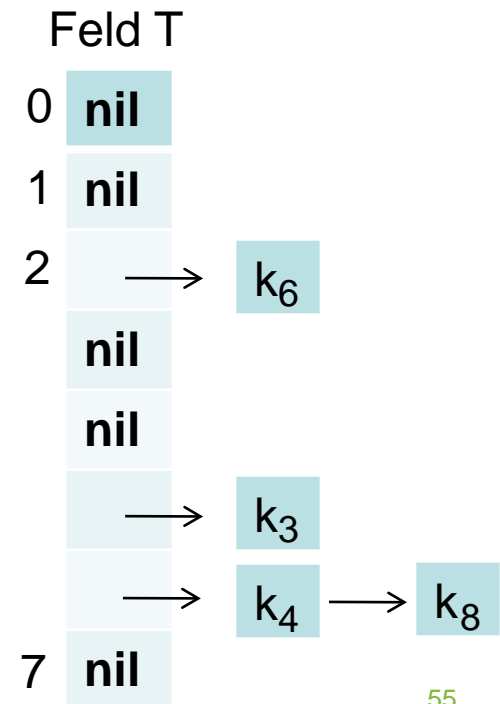


Datenstrukturen

Beweis

- Durchschnitt über alle n Elemente aus M:

$$\frac{1}{n} \sum_{i=1}^n \left(1 + \frac{i-1}{t}\right) = 1 + \frac{1}{nt} \sum_{i=1}^n (i-1)$$

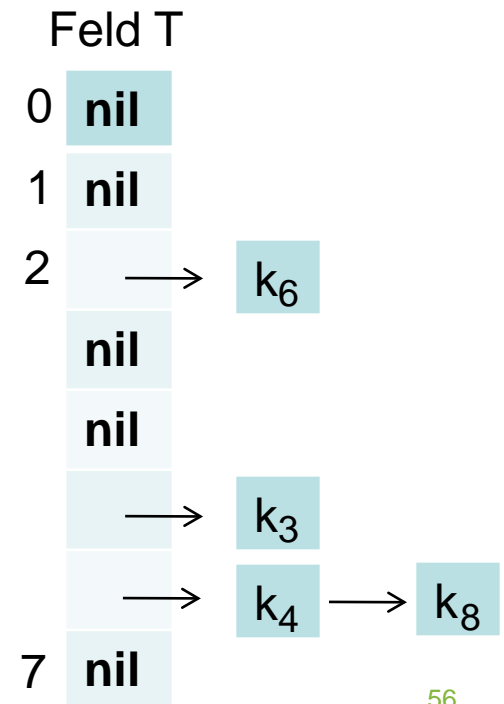


Datenstrukturen

Beweis

- Durchschnitt über alle n Elemente aus M:

$$\begin{aligned} \frac{1}{n} \sum_{i=1}^n \left(1 + \frac{i-1}{t}\right) &= 1 + \frac{1}{nt} \sum_{i=1}^n (i-1) \\ &= 1 + \left(\frac{1}{nt}\right) \left(\frac{(n-1)n}{2}\right) = 1 + \frac{\alpha}{2} - \frac{1}{2t} \end{aligned}$$

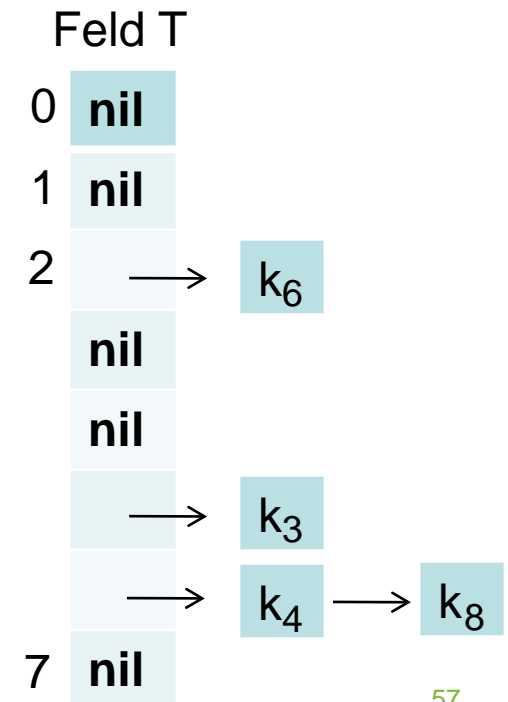


Datenstrukturen

Beweis

- Durchschnitt über alle n Elemente aus M:

$$\begin{aligned} \frac{1}{n} \sum_{i=1}^n \left(1 + \frac{i-1}{t}\right) &= 1 + \frac{1}{nt} \sum_{i=1}^n (i-1) \\ &= 1 + \left(\frac{1}{nt}\right) \left(\frac{(n-1)n}{2}\right) = 1 + \frac{\alpha}{2} + \frac{1}{2t} \\ &= O(1 + \alpha) \end{aligned}$$



Datenstrukturen

Interpretation

- Ist die Größe der Hash-Tabelle proportional zur Anzahl gespeicherter Elemente, dann ist die durchschn. Suchzeit $O(1)$

Frage

- Wie realistisch ist Annahme des einfachen gleichverteilten Hashing?

Datenstrukturen

Interpretation

- Ist die Größe der Hash-Tabelle proportional zur Anzahl gespeicherter Elemente, dann ist die durchschn. Suchzeit $O(1)$

Frage

- Wie realistisch ist Annahme des einfachen gleichverteilten Hashing?
- Die Menge H aller Funktionen von U nach $\{0, \dots, t-1\}$ erfüllt Anforderung

Datenstrukturen

Interpretation

- Ist die Größe der Hash-Tabelle proportional zur Anzahl gespeicherter Elemente, dann ist die durchschn. Suchzeit $O(1)$

Frage

- Wie realistisch ist Annahme des einfachen gleichverteilten Hashing?
- Die Menge H aller Funktionen von U nach $\{0, \dots, t-1\}$ erfüllt Anforderung
- Kann man eine Funktion aus H effizient abspeichern?

Datenstrukturen

Kann man eine Funktion aus H effizient abspeichern?

- Wenn es $|H|$ unterschiedliche Funktionen gibt, dann benötigen wir mindestens $\lceil \log |H| \rceil$ viele Bits, um jede Funktion aus H beschreiben zu können

Datenstrukturen

Kann man eine Funktion aus H effizient abspeichern?

- Wenn es $|H|$ unterschiedliche Funktionen gibt, dann benötigen wir mindestens $\lceil \log |H| \rceil$ viele Bits, um jede Funktion aus H beschreiben zu können
- **Argument:**
 - Man muss mindestens so viele unterschiedliche Bitstrings haben wie Funktionen in H
 - Die Anzahl unterschiedlicher Bitstrings der Länge k ist 2^k

Datenstrukturen

Kann man eine Funktion aus H effizient abspeichern?

- Wenn es $|H|$ unterschiedliche Funktionen gibt, dann benötigen wir mindestens $\lceil \log |H| \rceil$ viele Bits, um jede Funktion aus H beschreiben zu können
- Argument:
 - Man muss mindestens so viele unterschiedliche Bitstrings haben wie Funktionen in H
 - Die Anzahl unterschiedlicher Bitstrings der Länge k ist 2^k
- Jedes Element von U kann auf t unterschiedliche Werte abgebildet werden

Datenstrukturen

Kann man eine Funktion aus H effizient abspeichern?

- Wenn es $|H|$ unterschiedliche Funktionen gibt, dann benötigen wir mindestens $\lceil \log |H| \rceil$ viele Bits, um jede Funktion aus H beschreiben zu können
- Argument:
 - Man muss mindestens so viele unterschiedliche Bitstrings haben wie Funktionen in H
 - Die Anzahl unterschiedlicher Bitstrings der Länge k ist 2^k
- Jedes Element von U kann auf t unterschiedliche Werte abgebildet werden
- Es gibt als $t^{|U|}$ unterschiedliche Funktionen in H

Datenstrukturen

Kann man eine Funktion aus H effizient abspeichern?

- Wenn es $|H|$ unterschiedliche Funktionen gibt, dann benötigen wir mindestens $\lceil \log |H| \rceil$ viele Bits, um jede Funktion aus H beschreiben zu können
- Argument:
 - Man muss mindestens so viele unterschiedliche Bitstrings haben wie Funktionen in H
 - Die Anzahl unterschiedlicher Bitstrings der Länge k ist 2^k
- Jedes Element von U kann auf t unterschiedliche Werte abgebildet werden
- Es gibt als $t^{|U|}$ unterschiedliche Funktionen in H
- Wir benötigen also mindestens $|U| \log t$ Bits, um Funktionen aus H abspeichern zu können

Datenstrukturen

Kann man eine Funktion aus H effizient abspeichern?

- Wenn es $|H|$ unterschiedliche Funktionen gibt, dann benötigen wir mindestens $\lceil \log |H| \rceil$ viele Bits, um jede Funktion aus H beschreiben zu können
- Argument:
 - Man muss mindestens so viele unterschiedliche Bitstrings haben wie Funktionen in H
 - Die Anzahl unterschiedlicher Bitstrings der Länge k ist 2^k
- Jedes Element von U kann auf t unterschiedliche Werte abgebildet werden
- Es gibt als $t^{|U|}$ unterschiedliche Funktionen in H
- Wir benötigen also mindestens $|U| \log t$ Bits, um Funktionen aus H abspeichern zu können

Datenstrukturen

Kurzes Fazit: Was ist eine gute Hashfunktion?

- Eine Funktion die eine gute Verteilung der Daten auf die Tabelle gewährleistet
- Problem: Wir kennen die Daten a priori nicht
- Oft kennen wir auch die Verteilung der Daten nicht
- Eine zufällige Funktion wäre gut, aber die können wir nicht speichern

Datenstrukturen

Die Divisionsmethode

- k wird abgebildet auf den Rest von k durch t
- Es gilt also $h(k) = k \bmod t$

Beispiel

- $t=12$ und $k=100$
- Dann gilt $8t+4 = 100$ und somit $h(100) = 4$

Datenstrukturen

Die Divisionsmethode

- k wird abgebildet auf den Rest von k durch t
- Es gilt also $h(k) = k \bmod t$

Was sind gute Werte für m (ohne Beweis bzw. empirisch)?

- Ist t Zweierpotenz, dann „zählen“ nur die niedrigwertigen Bits (meistens schlecht)
- Gute Werte sind normalerweise Primzahlen, die nicht zu nah an Zweierpotenzen liegen

Datenstrukturen

Zusammenfassung und Ausblick

- Hashing mit Verkettung
- Divisionsmethode als Hashfunktion
- Viele weitere Methoden des Hashing