Übersicht

Übersicht

Assoziative Arrays / Wörterbücher

- Assoziatives Array / Wörterbuch (dictionary) S:
  speichert eine Menge von Elementen
- Element e wird identifiziert über eindeutigen Schlüssel key(e)

Operationen:

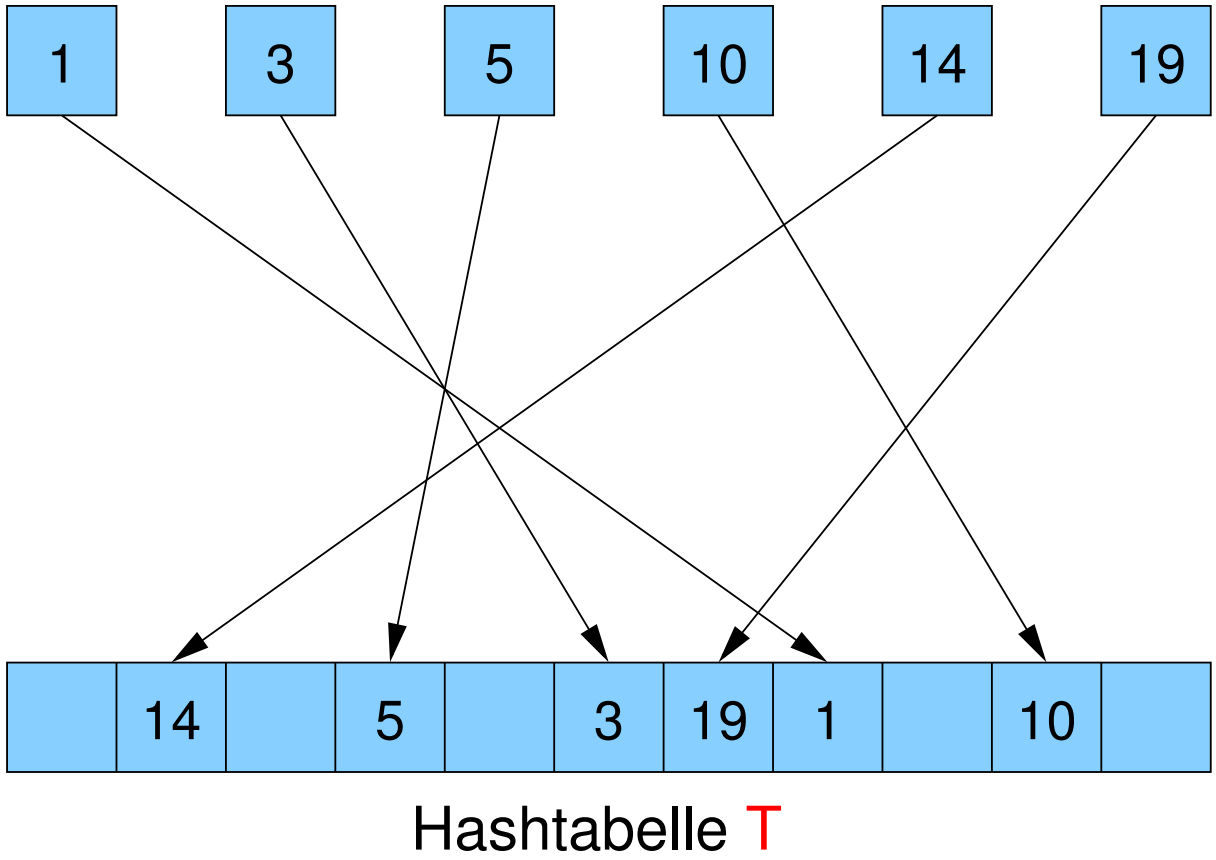
- S.insert(Elem e): $S := S \cup \{e\}$
- S.remove(Key k): $S := S \setminus \{e\}$,
  wobei e das Element mit key(e) $= k$ ist
- S.find(Key k):
  gibt das Element $e \in S$ mit key(e) $= k$ zurück, falls es existiert,
  sonst $\perp$ (entspricht Array-Indexoperator [ ], daher der Name)

Hashfunktion und Hashtabelle

Hashfunktion h :
Key $\mapsto \{0, \ldots, m-1\}$

$|\text{Key}| = N$

gespeicherte Elemente: n



Hashfunktion

Anforderungen:

- schneller Zugriff (Zeiteffizienz)
- platzsparend (Speichereffizienz)
  (z.B. surjektive Abbildung möglicher Schlüssel auf die Adressen)
- gute Streuung bzw. Verteilung der Elemente
  über die ganze Tabelle
- Idealfall: Element e direkt in Tabelleneintrag t[h(key(e))]

$\Rightarrow$ find, insert und remove in konstanter Zeit

(genauer: plus Zeit für Berechnung der Hashfunktion)

Hashing

Annahme: perfekte Streuung

```
insert(Elem e) {
   T[h(key(e))] = e;
}

remove(Key k) {
   T[h(k)] = null;
}

Elem find(Key k) {
   return T[h(k)];
}
```

statisches Wörterbuch: nur find
dynamisches Wörterbuch: insert, remove und find

Kollisionen

In der Praxis:

- perfekte Zuordnung zwischen den gespeicherten Schlüsseln und den Adressen der Tabelle nur bei statischem Array möglich
- leere Tabelleneinträge
- Schlüssel mit gleicher Adresse (Kollisionen)

Wie wahrscheinlich ist eine Kollision?

- Geburtstagsparadoxon: In einer Menge von 23 zufällig ausgewählten Personen gibt es mit Wahrscheinlichkeit $> 50\%$ zwei Leute, die am gleichen Tag Geburtstag feiern.
- Bei zufälliger Abbildung von 23 Schlüsseln auf die Adressen einer Hashtabelle der Größe 365 gibt es mit Wahrscheinlichkeit $> 50\%$ eine Kollision.

Wahrscheinlichkeit von Kollisionen

- Hilfsmittel: $\forall x \in \mathbb{R}: \quad 1 + x \leq \sum_{i=0}^{\infty} \frac{x^i}{i!} = e^x$

$\Rightarrow$ $\forall x \in \mathbb{R}: \quad \ln(1+x) \leq x$ (da $\ln(x)$ monoton wachsend ist)

- Pr[keine Kollision beim i-ten Schlüssel] $= \frac{m-(i-1)}{m}$ für $i \in [1 \ldots n]$

$$\begin{aligned} \Pr[\text{keine Kollision}] &= \prod_{i=1}^{n} \frac{m-(i-1)}{m} = \prod_{i=0}^{n-1} \left(1 - \frac{i}{m}\right) \\ &= e^{\left[\sum_{i=0}^{n-1} \ln(1-\frac{i}{m})\right]} \leq e^{\left[\sum_{i=0}^{n-1} (-\frac{i}{m})\right]} = e^{\left[-\frac{n(n-1)}{2m}\right]} \end{aligned}$$

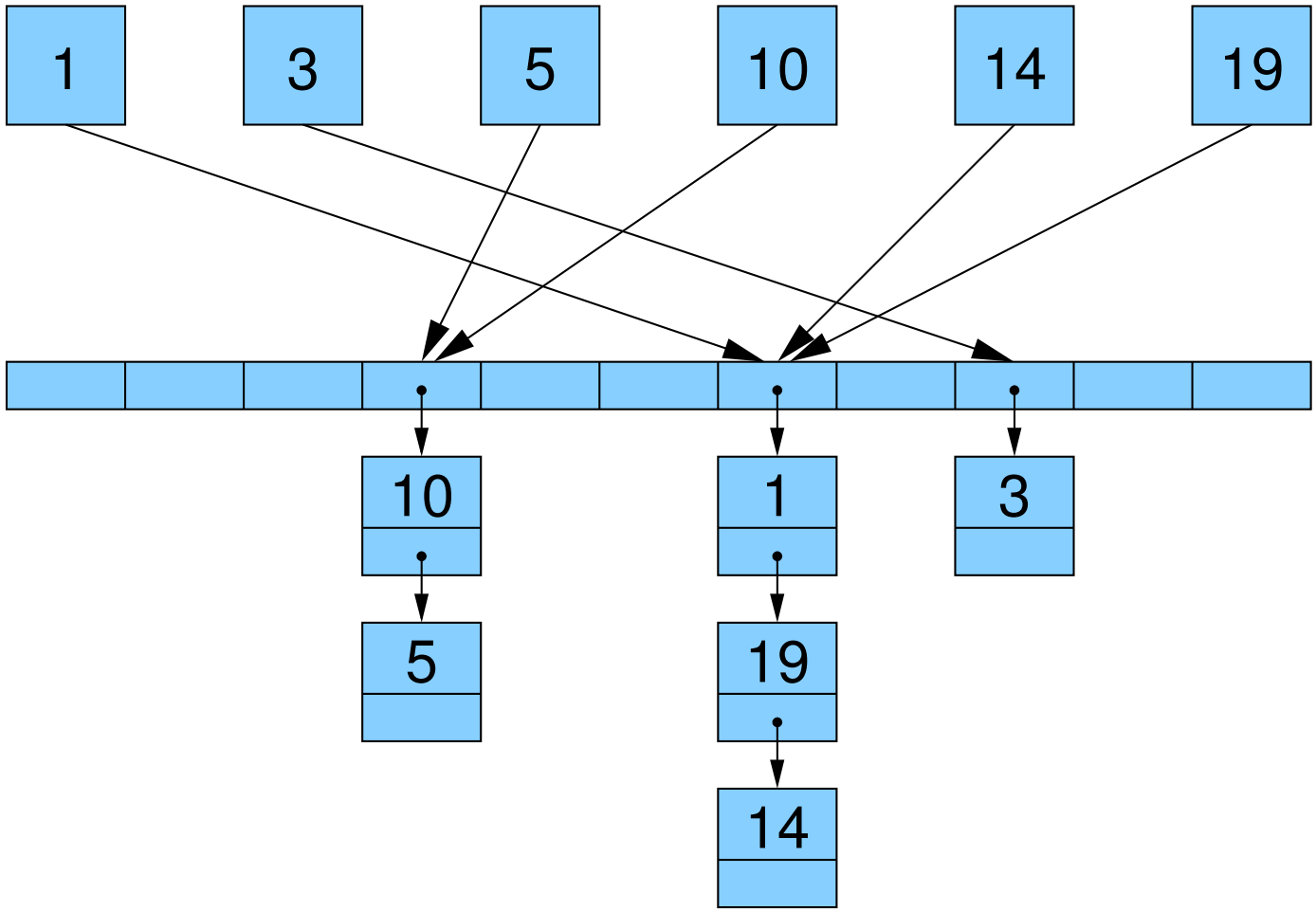
(da e^x monoton wachsend ist)

$\Rightarrow$ Bei gleichverteilt zufälliger Hashposition für jeden Schlüssel tritt für $n \in \omega(\sqrt{m})$ mit Wahrscheinlichkeit $1 - o(1)$ mindestens eine Kollision auf.

Übersicht

Dynamisches Wörterbuch

Hashing with Chaining:



Feld von Listen oder Zeigern

unsortierte verkettete Listen
(Ziel: Listen möglichst kurz)

Dynamisches Wörterbuch

Hashing with Chaining:

List<Elem>[m] T;

insert(Elem e) {
  T[h(key(e))].insert(e);
}

remove(Key k) {
  T[h(k)].remove(k);
}

find(Key k) {
  T[h(k)].find(k);
}

Hashing with Chaining

- Platzverbrauch: $\mathcal{O}(n+m)$
- insert benötigt konstante Zeit
- remove und find müssen u.U. eine ganze Liste scannen
- im worst case sind alle Elemente in dieser Liste

$\Rightarrow$ im worst case ist Hashing with chaining nicht besser als eine normale Liste

Hashing with Chaining

Gibt es Hashfunktionen, die garantieren, dass alle Listen kurz sind?

- nein, für jede Hashfunktion gibt es eine Adresse, der mindestens N/m mögliche Schlüssel zugeordnet sind
  (erweitertes Schubfachprinzip / pigeonhole principle)
- Meistens ist $n < N/m$ (weil N riesig ist).
- In diesem Fall kann die Suche zum Scan aller Elemente entarten.

$\Rightarrow$ Auswege

- Average-case-Analyse
- Randomisierung
- Änderung des Algorithmus
  (z.B. Hashfunktion abhängig von aktuellen Schlüsseln)

Hashing with Chaining

Betrachte als Hashfunktionsmenge die Menge aller Funktionen, die die Schlüsselmenge (mit Kardinalität N) auf die Zahlen $0, \ldots, m-1$ abbilden.

Satz

Falls n Elemente in einer Hashtabelle der Größe m mittels einer zufälligen Hashfunktion gespeichert werden, dann ist die erwartete Laufzeit von remove bzw. find in $\mathcal{O}(1 + n/m)$.

Unrealistisch: es gibt m^N solche Funktionen und man braucht $\log_2(m^N) = N \log_2 m$ Bits, um eine Funktion zu spezifizieren.

$\Rightarrow$ widerspricht dem Ziel, den Speicherverbrauch von N auf n zu senken!

Hashing with Chaining

Beweis.

- Betrachte feste Position $i = h(k)$ bei remove(k) oder find(k)
- Laufzeit ist Konstante plus Zeit für Scan der Liste
  also $O(1 + \mathbb{E}[X])$, wobei X Zufallsvariable für Länge von $T[i]$
- Zufallsvariable $X_e \in \{0, 1\}$ für jedes $e \in S$
- $X_e = 1 \Leftrightarrow h(\text{key}(e)) = i$
- Listenlänge $X = \sum_{e \in S} X_e$
- Erwartete Listenlänge $\mathbb{E}[X]$

$$
\begin{aligned}
&= \mathbb{E}\left[\sum_{e \in S} X_e\right] = \sum_{e \in S} \mathbb{E}[X_e] = \sum_{e \in S} 0 \cdot \Pr[X_e = 0] + 1 \cdot \Pr[X_e = 1] \\
&= \sum_{e \in S} \Pr[X_e = 1] = \sum_{e \in S} 1/m = n/m
\end{aligned}
$$

□

Übersicht

c-universelle Familien von Hashfunktionen

Wie konstruiert man zufällige Hashfunktionen?

Definition

Sei c eine positive Konstante.

Eine Familie H von Hashfunktionen auf $\{0, \ldots, m-1\}$ heißt c-universell, falls für jedes Paar $x \neq y$ von Schlüsseln gilt, dass

$$\left|\{h \in H : \; h(x) = h(y)\}\right| \leq \frac{c}{m}|H|.$$

D.h. bei zufälliger Auswahl der Hashfunktion $h \in H$ gilt

$$\forall \{x, y\}_{(x \neq y)} : \quad \Pr[h(x) = h(y)] \leq \frac{c}{m}$$

1-universelle Familien nennt man universell.

c-Universal Hashing with Chaining

Satz

Falls n Elemente in einer Hashtabelle der Größe m mittels einer zufälligen Hashfunktion h aus einer c-universellen Familie gespeichert werden, dann ist die erwartete Laufzeit von remove bzw. find in $O(1 + c \cdot n/m)$.

Beweis.

- Betrachte festen Schlüssel k
- Zugriffszeit X ist $O(1 +$ Länge der Liste $T[h(k)])$
- Zufallsvariable $X_e \in \{0, 1\}$ für jedes $e \in S$ zeigt an, ob e auf die gleiche Position wie k gehasht wird

c-Universal Hashing with Chaining

Beweis.

- $X_e = 1 \Leftrightarrow h(\text{key}(e)) = h(k)$
- Listenlänge $X = \sum_{e \in S} X_e$
- Erwartete Listenlänge

$$\begin{aligned}
\mathbb{E}[X] &= \mathbb{E}\left[\sum_{e \in S} X_e\right] \\
&= \sum_{e \in S} \mathbb{E}[X_e] = \sum_{e \in S} 0 \cdot \Pr[X_e = 0] + 1 \cdot \Pr[X_e = 1] \\
&= \sum_{e \in S} \Pr[X_e = 1] \leq \sum_{e \in S} c/m = n \cdot c/m
\end{aligned}$$

□

Beispiele für c-universelles Hashing

Einfache c-universelle Hashfunktionen?

Annahme: Schlüssel sind Bitstrings einer bestimmten Länge

Wähle als Tabellengröße m eine Primzahl

$\Rightarrow$ dann ist der Restklassenring modulo m (also $\mathbb{Z}_m$) ein Körper, d.h. es gibt zu jedem Element außer für die Null genau ein Inverses bzgl. Multiplikation

- Sei $w = \lfloor \log_2 m \rfloor$.
- unterteile die Bitstrings der Schlüssel in Teile zu je w Bits
- Anzahl der Teile sei k
- interpretiere jeden Teil als Zahl aus dem Intervall $[0, \ldots, 2^w - 1]$
- interpretiere Schlüssel x als k-Tupel solcher Zahlen:

$$\mathbf{x} = (x_1, \ldots, x_k)$$

Familie für universelles Hashing

Definiere für jeden Vektor

$$\mathbf{a} = (a_1, \ldots, a_k) \in \{0, \ldots, m-1\}^k$$

mittels Skalarprodukt

$$\mathbf{a} \cdot \mathbf{x} = \sum_{i=1}^{k} a_i x_i$$

eine Hashfunktion von der Schlüsselmenge
in die Menge der Zahlen $\{0, \ldots, m-1\}$

$$h_{\mathbf{a}}(\mathbf{x}) = \mathbf{a} \cdot \mathbf{x} \mod m$$

Familie für universelles Hashing

Satz

Wenn m eine Primzahl ist, dann ist

$$H = \left\{ h_{\mathbf{a}} : \mathbf{a} \in \{0, \ldots, m-1\}^k \right\}$$

eine [1-]universelle Familie von Hashfunktionen.

Oder anders:
das Skalarprodukt zwischen einer Tupeldarstellung des Schlüssels und einem Zufallsvektor modulo m definiert eine gute Hashfunktion.

Familie für universelles Hashing

Beispiel

- 32-Bit-Schlüssel, Hashtabellengröße $m = 269$

$\Rightarrow$ Schlüssel unterteilt in $k = 4$ Teile mit $w = \lfloor \log_2 m \rfloor = 8$ Bits

- Schlüssel sind also 4-Tupel von Integers aus dem Intervall $[0, 2^8 - 1] = \{0, \ldots, 255\}$, z.B. $\mathbf{x} = (11, 7, 4, 3)$
- Die Hashfunktion wird auch durch ein 4-Tupel von Integers, aber aus dem Intervall $[0, 269 - 1] = \{0, \ldots, 268\}$, spezifiziert z.B. $\mathbf{a} = (2, 4, 261, 16)$

$\Rightarrow$ Hashfunktion:

$$h_{\mathbf{a}}(\mathbf{x}) = (2x_1 + 4x_2 + 261x_3 + 16x_4) \bmod 269$$

$$h_{\mathbf{a}}(\mathbf{x}) = (2 \cdot 11 + 4 \cdot 7 + 261 \cdot 4 + 16 \cdot 3) \bmod 269 = 66$$

Eindeutiges a_j

Beweis

- Betrachte zwei beliebige verschiedene Schlüssel $\mathbf{x} = \{x_1, \ldots, x_k\}$ und $\mathbf{y} = \{y_1, \ldots, y_k\}$
- Wie groß ist $\Pr[h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y})]$?
- Sei j ein Index (von evt. mehreren möglichen) mit $x_j \neq y_j$ (muss es geben, sonst wäre $\mathbf{x} = \mathbf{y}$)

$\Rightarrow$ $(x_j - y_j) \not\equiv 0 \bmod m$
d.h., es gibt genau ein multiplikatives Inverses $(x_j - y_j)^{-1}$

$\Rightarrow$ gegeben Primzahl m und Zahlen $x_j, y_j, b \in \{0, \ldots m-1\}$ hat jede Gleichung der Form

$$a_j(x_j - y_j) \equiv b \bmod m$$

eine eindeutige Lösung: $a_j \equiv (x_j - y_j)^{-1} b \bmod m$

Wann wird $h(\mathbf{x}) = h(\mathbf{y})$?

Beweis.

Wenn man alle Variablen a_i außer a_j festlegt, gibt es exakt eine Wahl für a_j, so dass $h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y})$, denn

$$
\begin{aligned}
h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y}) \quad &\Leftrightarrow \quad \sum_{i=1}^{k} a_i x_i \equiv \sum_{i=1}^{k} a_i y_i && \mod m \\
&\Leftrightarrow \quad a_j(x_j - y_j) \equiv \sum_{i \neq j} a_i (y_i - x_i) && \mod m \\
&\Leftrightarrow \quad a_j \equiv (x_j - y_j)^{-1} \sum_{i \neq j} a_i (y_i - x_i) && \mod m
\end{aligned}
$$

Wie oft wird $h(\mathbf{x}) = h(\mathbf{y})$?

Beweis.

- Es gibt m^{k-1} Möglichkeiten, Werte für die Variablen a_i mit $i \neq j$ zu wählen.
- Für jede solche Wahl gibt es genau eine Wahl für a_j, so dass $h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y})$.
- Für $\mathbf{a}$ gibt es insgesamt m^k Auswahlmöglichkeiten.
- Also

$$\Pr[h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y})] = \frac{m^{k-1}}{m^k} = \frac{1}{m}$$

□

Familie für k-universelles Hashing

Definiere für $a \in \{1, \ldots, m-1\}$ die Hashfunktion

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

(mit $x_i \in \{0, \ldots, m-1\}$)

Satz

Für jede Primzahl m ist

$$H' = \{h'_a : \ a \in \{1, \ldots, m-1\}\}$$

eine k-universelle Familie von Hashfunktionen.

Familie für k-universelles Hashing

Beweisidee:

Für Schlüssel $\mathbf{x} \neq \mathbf{y}$ ergibt sich folgende Gleichung:

$$\begin{aligned} h'_a(\mathbf{x}) &= h'_a(\mathbf{y}) \\ h'_a(\mathbf{x}) - h'_a(\mathbf{y}) &\equiv 0 \bmod m \\ \sum_{i=1}^{k} a^{i-1}(x_i - y_i) &\equiv 0 \bmod m \end{aligned}$$

Anzahl der Nullstellen des Polynoms in a ist durch den Grad des Polynoms beschränkt (Fundamentalsatz der Algebra), also durch $k-1$.

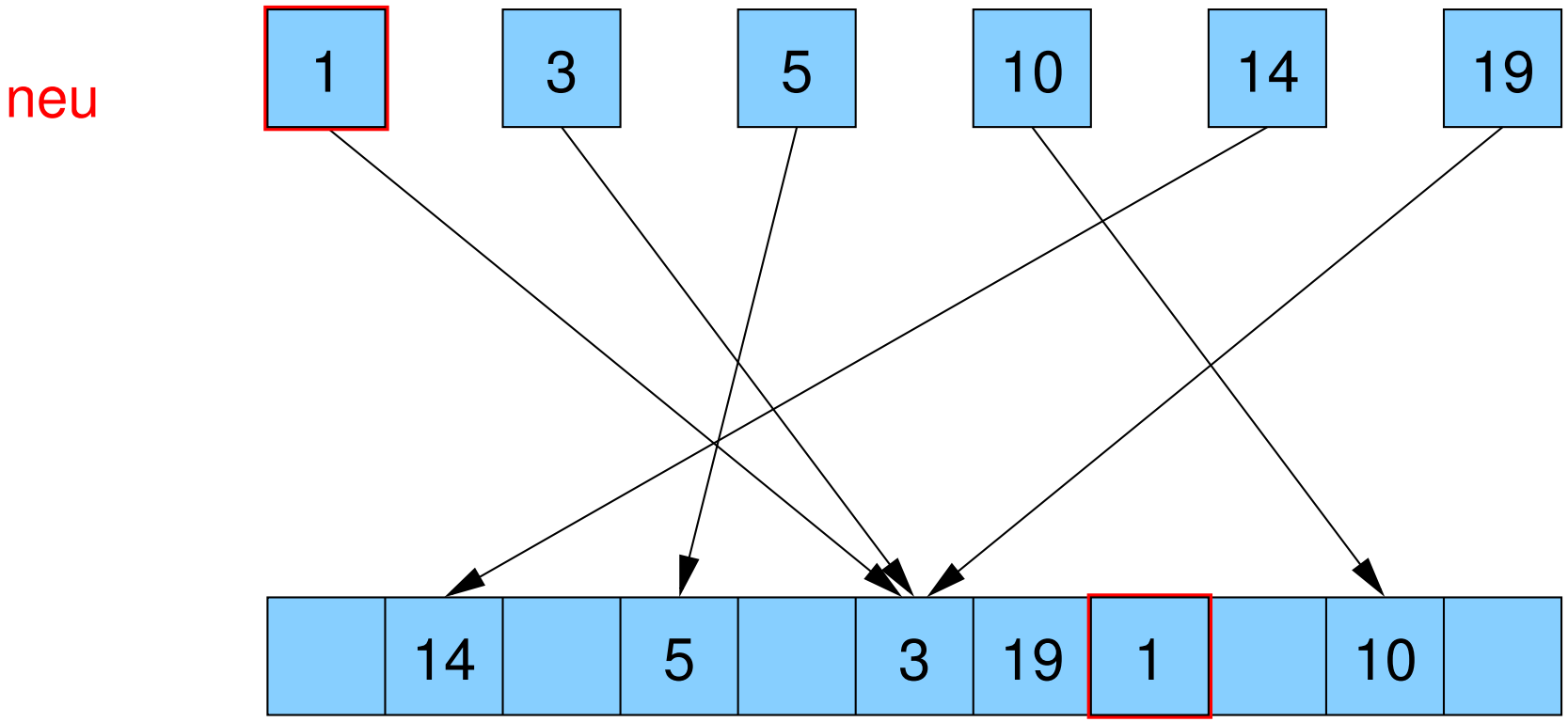
Falls $k \leq m$ können also höchstens $k-1$ von $m-1$ möglichen Werten für a zum gleichen Hashwert für $\mathbf{x}$ und $\mathbf{y}$ führen.

Aus $\Pr[h(\mathbf{x}) = h(\mathbf{y})] \leq \frac{\min\{k-1, m-1\}}{m-1} \leq \frac{k}{m}$ folgt, dass H' k-universell ist.

Übersicht

Dynamisches Wörterbuch

Hashing with Linear Probing:



Speichere Element e im ersten freien
Ort T[i], T[$i+1$], T[$i+2$], ... mit $i == h(\text{key}(e))$
(Ziel: Folgen besetzter Positionen möglichst kurz)

Hashing with Linear Probing

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

insert(Elem e) {
    i = h(key(e));
    while (T[i] ≠ null ∧ T[i] ≠ e)
        i = (i+1) % m;
    T[i] = e;
}

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

Vorteil:

Es werden im Gegensatz zu Hashing with Chaining (oder auch im Gegensatz zu anderen Probing-Varianten) nur zusammenhängende Speicherzellen betrachtet.

$\Rightarrow$ Cache-Effizienz!

Hashing with Linear Probing

Problem: Löschen von Elementen

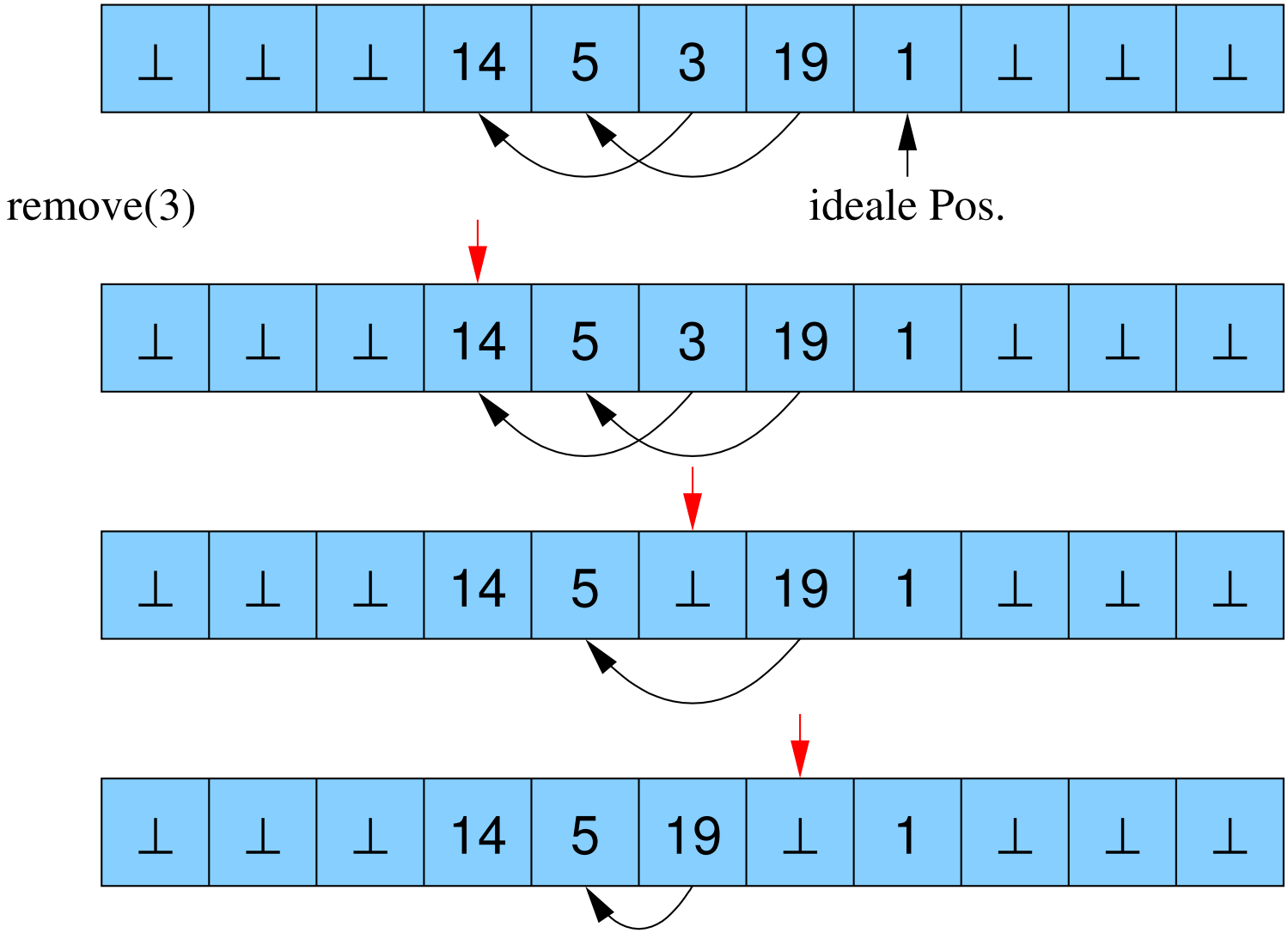
1. Löschen verbieten
2. Markiere Positionen als gelöscht
   (mit speziellem Zeichen $\neq \perp$)
   Suche endet bei $\perp$, aber nicht bei markierten Zellen

   Problem: Anzahl echt freier Zellen sinkt monoton
   $\Rightarrow$ Suche wird evt. langsam oder periodische Reorganisation
3. Invariante sicherstellen:

   Für jedes $e \in S$ mit idealer Position $i = h(key(e))$ und aktueller Position j gilt:
   T[i], T[i+1],. . . ,T[j] sind besetzt

Hashing with Linear Probing

Löschen / Aufrechterhaltung der Invariante



Übersicht

Dynamisches Wörterbuch

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

Lösung: Reallokation

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

Dynamisches Wörterbuch

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

Lösung:

- Für jedes k gibt es eine Primzahl in $\left[k^3, (k+1)^3\right]$
- Jede Zahl $z \leq (k+1)^3$, die nicht prim ist, muss einen Teiler $t \leq \sqrt{(k+1)^3} = (k+1)^{3/2}$ haben.
- Für eine gewünschte ungefähre Tabellengröße m' (evt. nicht prim) bestimme k so, dass $k^3 \leq m' \leq (k+1)^3$
- Größe des Intervalls:
  $(k+1)^3 - k^3 + 1 = (k^3 + 3k^2 + 3k + 1) - k^3 + 1 = 3k^2 + 3k + 2$
- Für jede Zahl $j = 2, \ldots, (k+1)^{3/2}$:
  streiche die Vielfachen von j in $[k^3, (k+1)^3]$
- Für jedes j kostet das Zeit $\left((k+1)^3 - k^3 + 1\right)/j \in O(k^2/j)$

Dynamisches Wörterbuch

- Hilfsmittel: Wachstum der harmonischen Reihe

$$\ln n \quad \leq \quad H_n = \sum_{i=1}^{n} \frac{1}{i} \quad \leq \quad 1 + \ln n$$

- insgesamt:

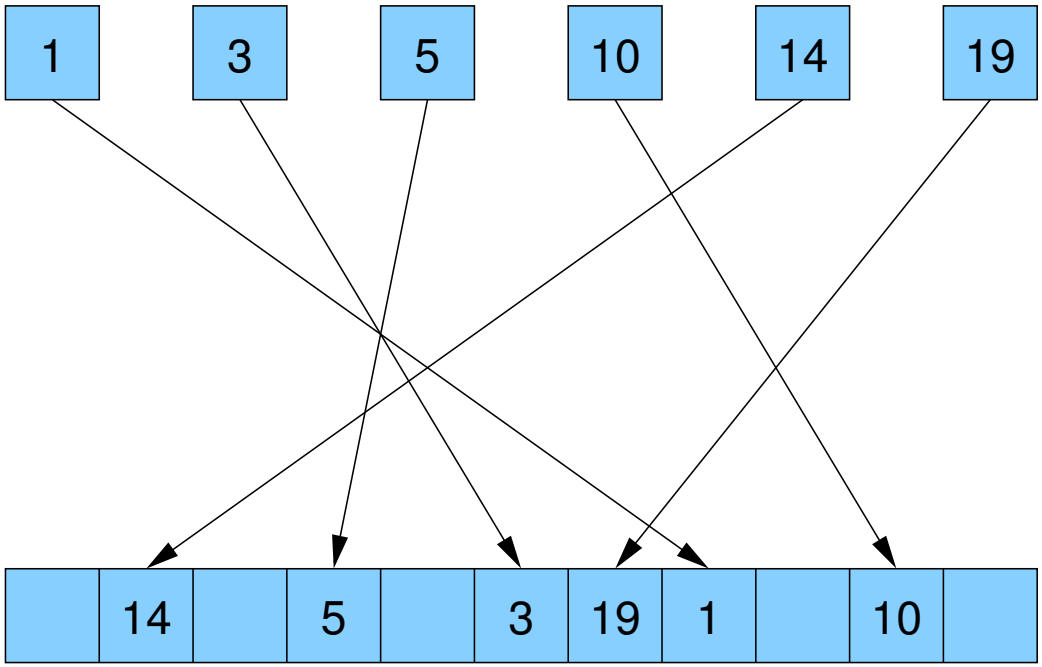
$$\begin{aligned}
\sum_{2 \leq j \leq (k+1)^{3/2}} O\left(\frac{k^2}{j}\right) \quad &\leq \quad k^2 \sum_{2 \leq j \leq (k+1)^{3/2}} O\left(\frac{1}{j}\right) \\
&\in \quad k^2 \cdot O\left(\ln\left((k+1)^{3/2}\right)\right) \\
&\in \quad O(k^2 \ln k) \\
&\in \quad o(m)
\end{aligned}$$

$\Rightarrow$ Kosten zu vernachlässigen im Vergleich zur Initialisierung der Tabelle der Größe m (denn m ist kubisch in k)

Übersicht

Perfektes Hashing für statisches Wörterbuch

- bisher: konstante *erwartete* Laufzeit
  falls m im Vergleich zu n genügend groß gewählt wird
  (nicht ausreichend für Real Time Scenario)
- Ziel: konstante Laufzeit im worst case für find()
  durch perfekte Hashtabelle ohne Kollisionen
- Annahme: statische Menge S von n Elementen



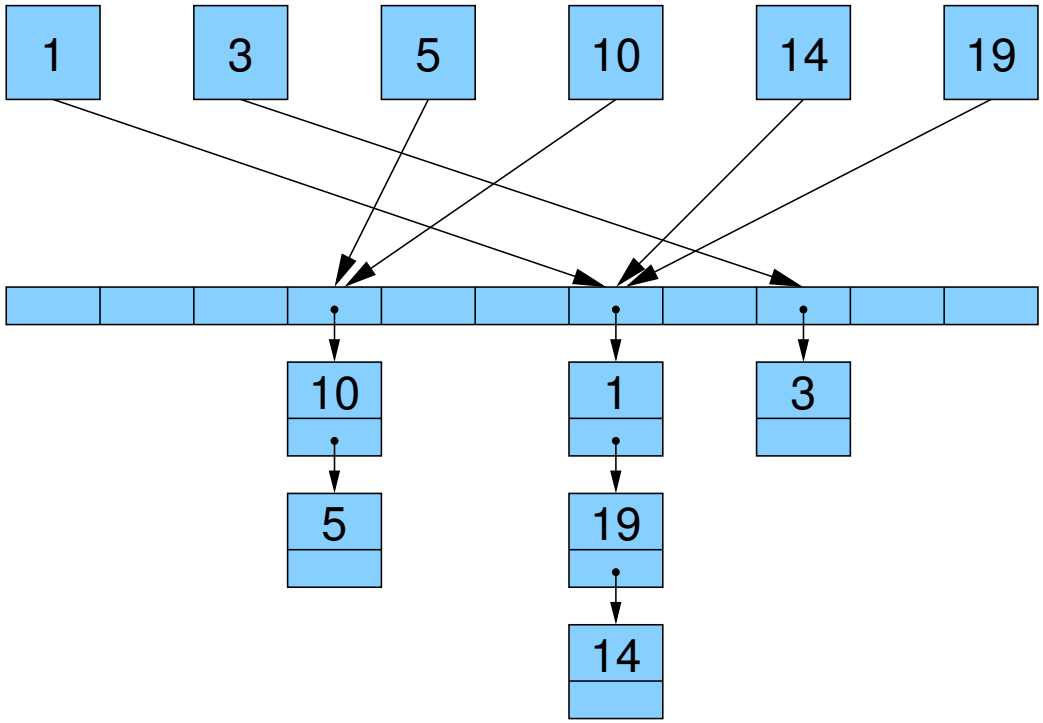
Statisches Wörterbuch

- S: feste Menge von n Elementen mit Schlüsseln k_1 bis k_n
- H_m: c-universelle Familie von Hashfunktionen auf $\{0, \ldots, m-1\}$
  (Hinweis: 2-universelle Familien existieren für alle m)
- $C(h)$ für $h \in H_m$: Anzahl Kollisionen in S für h, d.h.

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

Beispiel:

$C(h) = 2 + 6 = 8$



Erwartete Anzahl von Kollisionen

Lemma

Für die Anzahl der Kollisionen gilt: $\mathbb{E}[C(h)] \leq cn(n-1)/m.$

Beweis.

- Definiere $n(n-1)$ Indikator-Zufallsvariablen $X_{ij}(h)$:
  Für $i \neq j$ sei $X_{ij}(h) = 1 \Leftrightarrow h(k_i) = h(k_j)$.
- Dann ist $C(h) = \sum_{i \neq j} X_{ij}(h)$

$$\mathbb{E}[C] = \mathbb{E}\left[\sum_{i \neq j} X_{ij}\right] = \sum_{i \neq j} \mathbb{E}\left[X_{ij}\right] = \sum_{i \neq j} \Pr\left[X_{ij} = 1\right] \leq n(n-1) \cdot c/m$$

$\Rightarrow$ Für quadratische Tabellengröße ist die erwartete Anzahl von Kollisionen (und damit die erwartete worst-case-Laufzeit für find) eine Konstante. □

Markov-Ungleichung

Satz (Markov-Ungleichung)

Für jede nichtnegative Zufallsvariable X und Konstante $k > 0$ gilt:

$$\Pr[X \geq k] \leq \frac{\mathbb{E}[X]}{k} \qquad \textit{und für } \mathbb{E}[X] > 0\textit{:} \quad \Pr[X \geq k \cdot \mathbb{E}[X]] \leq \frac{1}{k}$$

Beweis.

$$\begin{aligned}
\mathbb{E}[X] &= \sum_{z \in \mathbb{R}} z \cdot \Pr[X = z] \\
&\geq \sum_{z \geq k \cdot \mathbb{E}[X]} z \cdot \Pr[X = z] \geq \sum_{z \geq k \cdot \mathbb{E}[X]} k \cdot \mathbb{E}[X] \cdot \Pr[X = z] \\
&\geq k \cdot \mathbb{E}[X] \cdot \Pr[X \geq k \cdot \mathbb{E}[X]]
\end{aligned}$$

□

Hashfunktionen mit wenig Kollisionen

Lemma

Für mindestens die Hälfte der Funktionen $h \in H_m$ gilt:

$$C(h) \leq 2cn(n-1)/m$$

Beweis.

- Aus Lemma $\mathbb{E}[C(h)] \leq cn(n-1)/m$ und Markov-Ungleichung $\Pr[X \geq k \cdot \mathbb{E}[X]] \leq \frac{1}{k}$ folgt für $n \geq 2$:

$$\Pr[C(h) \geq 2cn(n-1)/m] \leq \Pr\left[C(h) \geq 2\mathbb{E}[C(h)]\right] \leq \frac{1}{2}$$

$\Rightarrow$ Für höchstens die Hälfte der Funktionen ist $C(h) \geq \frac{2cn(n-1)}{m}$

$\Rightarrow$ Für mindestens die Hälfte der Funktionen ist $C(h) \leq \frac{2cn(n-1)}{m}$ $(n \geq 1)$

□

Hashfunktionen ohne Kollisionen

Lemma

Wenn $m \geq cn(n-1)+1$, dann bildet mindestens die Hälfte der Funktionen $h \in H_m$ die Schlüssel injektiv in die Indexmenge der Hashtabelle ab.

Beweis.

- Für mindestens die Hälfte der Funktionen $h \in H_m$ gilt $C(h) < 2$.
- Da $C(h)$ immer eine gerade Zahl sein muss, folgt aus $C(h) < 2$ direkt $C(h) = 0$.

$\Rightarrow$ keine Kollisionen (bzw. injektive Abbildung)

□

- Wähle zufällig $h \in H_m$ mit $m \geq cn(n-1)+1$
- Prüfe auf Kollision $\Rightarrow$ behalten oder erneut wählen

$\Rightarrow$ Nach durchschnittlich 2 Versuchen erfolgreich

Statisches Wörterbuch

Ziel: lineare Tabellengröße

Idee: zweistufige Abbildung der Schlüssel

- Wähle Hashfunktion h mit wenig Kollisionen (≈ 2 Versuche)

$\Rightarrow$ 1. Stufe bildet Schlüssel auf Buckets von konstanter durchschnittlicher Größe ab

- Wähle Hashfunktionen h_ℓ ohne Kollisionen (≈ 2 Versuche pro h_ℓ)

$\Rightarrow$ 2. Stufe benutzt quadratisch viel Platz für jedes Bucket, um alle Kollisionen aus der 1. Stufe aufzulösen

Statisches Wörterbuch

- B_ℓ^h: Menge der Elemente in S, die h auf ℓ abbildet, $\ell \in \{0, \ldots, m-1\}$ und $h \in H_m$
- b_ℓ^h: Kardinalität von B_ℓ^h, also $b_\ell^h := \left|B_\ell^h\right|$
- Für jedes ℓ führen die Schlüssel in B_ℓ^h zu $b_\ell^h(b_\ell^h - 1)$ Kollisionen
- Also ist die Gesamtzahl der Kollisionen

$$C(h) = \sum_{\ell=0}^{m-1} b_\ell^h(b_\ell^h - 1)$$

Perfektes statisches Hashing: 1. Stufe

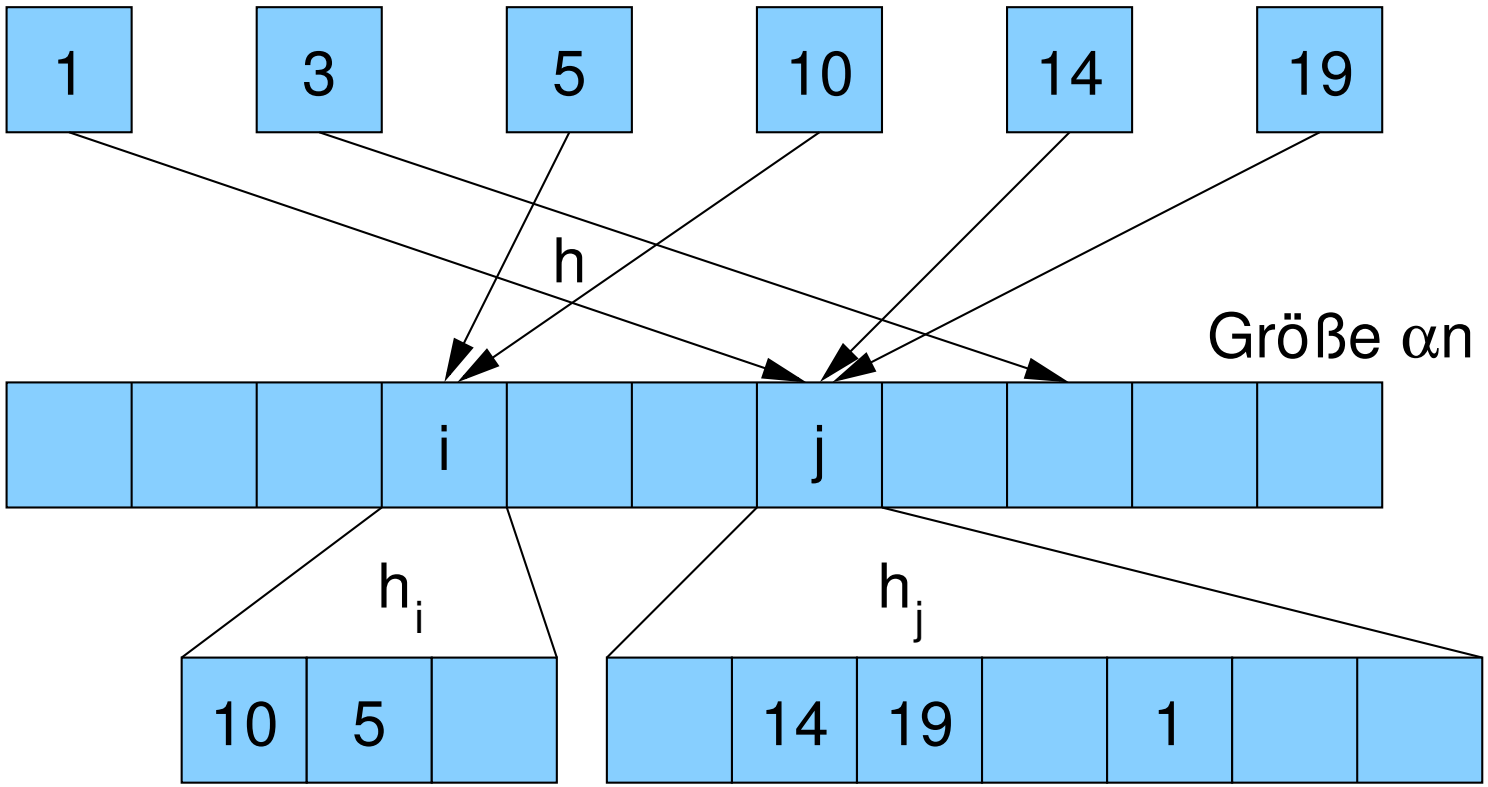
- 1. Stufe der Hashtabelle soll linear viel Speicher verwenden, also $\lceil \alpha n \rceil$ Adressen, wobei wir Konstante α später festlegen.
- Wähle Funktion $h \in H_{\lceil \alpha n \rceil}$, um S in Teilmengen B_ℓ aufzuspalten.

  Wähle h dabei so lange zufällig aus $H_{\lceil \alpha n \rceil}$ aus, bis gilt:

$$C(h) \leq \frac{2cn(n-1)}{\lceil \alpha n \rceil} \leq \frac{2cn(n-1)}{\alpha n} \leq \frac{2cn}{\alpha}$$

  Da das für mindestens die Hälfte der Hashfunktionen in $H_{\lceil \alpha n \rceil}$ gilt (vorletztes Lemma), erwarten wir dafür ≤ 2 Versuche.
- Für jedes $\ell \in \{0, \ldots, \lceil \alpha n \rceil - 1\}$ seien B_ℓ die Elemente, die durch h auf Adresse ℓ abgebildet werden und $b_\ell = |B_\ell|$ deren Anzahl.

Perfektes statisches Hashing



Perfektes statisches Hashing: 2. Stufe

- Für jedes B_ℓ:

  Berechne $m_\ell = cb_\ell(b_\ell - 1) + 1$.

  Wähle zufällig Funktion $h_\ell \in H_{m_\ell}$, bis h_ℓ die Menge B_ℓ injektiv in $\{0, \ldots, m_\ell - 1\}$ abbildet (also ohne Kollisionen).
  Mindestens die Hälfte der Funktionen in H_{m_ℓ} tut das.

- Hintereinanderreihung der einzelnen Tabellen ergibt eine Gesamtgröße der Tabelle von $\sum_\ell m_\ell$

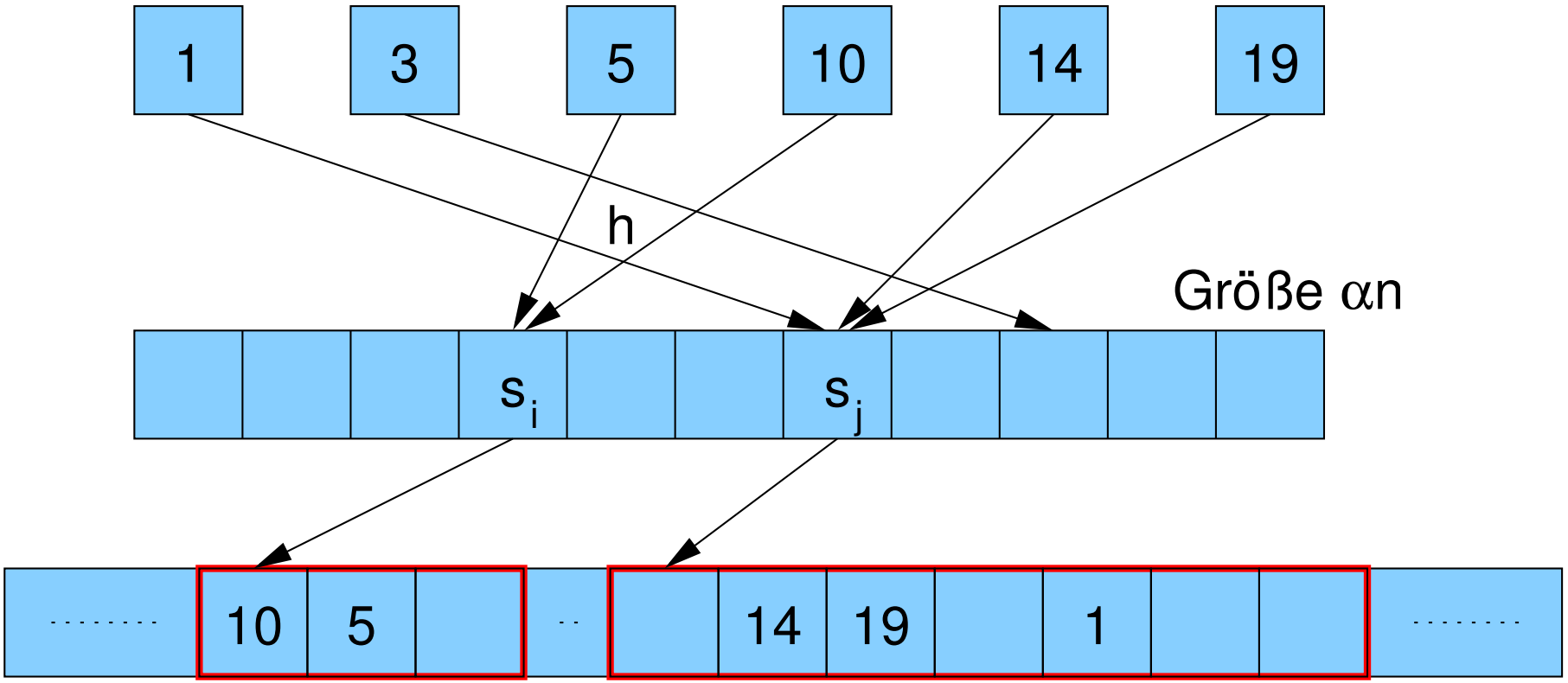
- Teiltabelle für B_ℓ beginnt an Position $s_\ell = m_0 + m_1 + \ldots + m_{\ell-1}$ und endet an Position $s_\ell + m_\ell - 1$

- Für gegebenen Schlüssel x, berechnen die Anweisungen

$$\ell = h(x); \quad \text{return } s_\ell + h_\ell(x);$$

  dann eine injektive Funktion auf der Menge S.

Perfektes statisches Hashing



Perfektes statisches Hashing

- Die Funktion ist beschränkt durch:

$$\begin{aligned}
\sum_{\ell=0}^{\lceil \alpha n \rceil - 1} m_\ell &= \sum_{\ell=0}^{\lceil \alpha n \rceil - 1} (c \cdot b_\ell (b_\ell - 1) + 1) \qquad \text{(siehe Def. der } m_\ell\text{'s)} \\
&\leq c \cdot C(h) + \lceil \alpha n \rceil \\
&\leq c \cdot 2cn/\alpha + \alpha n + 1 \\
&\leq (2c^2/\alpha + \alpha)n + 1
\end{aligned}$$

- Zur Minimierung der Schranke betrachte die Ableitung

$$\begin{aligned}
f(\alpha) &= (2c^2/\alpha + \alpha)n + 1 \\
f'(\alpha) &= (-2c^2/\alpha^2 + 1)n
\end{aligned}$$

$\Rightarrow$ $f'(\alpha) = 0$ liefert $\alpha = \sqrt{2}c$

$\Rightarrow$ Adressbereich: $0 \ldots 2\sqrt{2}cn$

Perfektes statisches Hashing

Satz

Für eine beliebige Menge von n Schlüsseln kann eine perfekte Hashfunktion mit Zielmenge $\{0,\ldots,2\sqrt{2}cn\}$ *in linearer erwarteter Laufzeit konstruiert werden.*

- Da wir wissen, dass wir für beliebige m eine 2-universelle Familie von Hashfunktionen finden können, kann man also z.B. eine Hashfunktion mit Adressmenge $\{0,\ldots,4\sqrt{2}n\}$ in linearer erwarteter Laufzeit konstruieren. (Las Vegas-Algorithmus)

- Unsere Minimierung hat nicht den Platz berücksichtigt, der benötigt wird, um die Werte s_ℓ, sowie die ausgewählten Hashfunktionen h_ℓ zu speichern.

$\Rightarrow$ Berechnung von α sollte eigentlich angepasst werden (entsprechend Speicherplatz pro Element, pro m_ℓ und pro h_ℓ)

Perfektes dynamisches Hashing

Kann man perfekte Hashfunktionen auch dynamisch konstruieren?

ja, z.B. mit Cuckoo Hashing

- 2 Hashfunktionen h_1 und h_2
- 2 Hashtabellen T_1 und T_2
- bei find und remove jeweils in beiden Tabellen nachschauen
- bei insert abwechselnd beide Tabellen betrachten, das zu speichernde Element an die Zielposition der aktuellen Tabelle schreiben und wenn dort schon ein anderes Element stand, dieses genauso in die andere Tabelle verschieben usw.
- evt. Anzahl Verschiebungen durch $2 \log n$ beschränken, um Endlosschleife zu verhindern
  (ggf. kompletter Rehash mit neuen Funktionen h_1, h_2)

Übersicht

Probleme beim linearen Sondieren

- Offene Hashverfahren allgemein:

  Erweiterte Hashfunktion $h(k, i)$ gibt an, auf welche Adresse ein Schlüssel k abgebildet werden soll, wenn bereits i Versuche zu einer Kollision geführt haben

- Lineares Sondieren (Linear Probing):

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

- Primäre Häufung (primary clustering):
  tritt auf, wenn für Schlüssel k_1, k_2 mit unterschiedlichen Hashwerten $h(k_1) \neq h(k_2)$ ab einem bestimmten Punkt i_1 bzw. i_2 die gleiche Sondierfolge auftritt:

$$\exists i_1, i_2 \quad \forall j : \quad h(k_1, i_1 + j) = h(k_2, i_2 + j)$$

Probleme beim quadratischen Sondieren

- Quadratisches Sondieren (Quadratic Probing):

$$h(k,i) = \left(h(k) + c_1 i + c_2 i^2\right) \bmod m \qquad (c_2 \neq 0)$$

$$\text{oder:} \quad h(k,i) = \left(h(k) - (-1)^i \lceil i/2 \rceil^2\right) \bmod m$$

- $h(k,i)$ soll möglichst surjektiv auf die Adressmenge $\{0,\ldots,m-1\}$ abbilden, um freie Positionen auch immer zu finden. Bei

$$h(k,i) = \left(h(k) - (-1)^i \lceil i/2 \rceil^2\right) \bmod m$$

z.B. durch Wahl von m prim $\wedge$ $m \equiv 3 \bmod 4$

- Sekundäre Häufung (secondary clustering):
tritt auf, wenn für Schlüssel k_1, k_2 mit gleichem Hashwert $h(k_1) = h(k_2)$ auch die nachfolgende Sondierfolge gleich ist:

$$\forall i: \quad h(k_1,i) = h(k_2,i)$$

Double Hashing

- Auflösung der Kollisionen der Hashfunktion h
  durch eine zweite Hashfunktion h':

$$h(k,i) = [h(k) + i \cdot h'(k)] \bmod m$$

  wobei für alle k gelten soll, dass $h'(k)$ teilerfremd zu m ist,

$$\begin{array}{lrcl} \text{z.B.} & h'(k) & = & 1 + k \bmod m - 1 \\ \text{oder} & h'(k) & = & 1 + k \bmod m - 2 \end{array}$$

  für Primzahl m

- primäre und sekundäre Häufung werden weitgehend vermieden,
  aber nicht komplett ausgeschlossen