13. Hashing

AVL-Bäume:

- Sortierte Ausgabe aller Elemente in O(n) Zeit
- Suche, Minimum, Maximum, Nachfolger in O(log n) Zeit
- Einfügen, Löschen in O(log n) Zeit

Frage:

- Kann man Einfügen, Löschen und Suchen in O(1) Zeit?

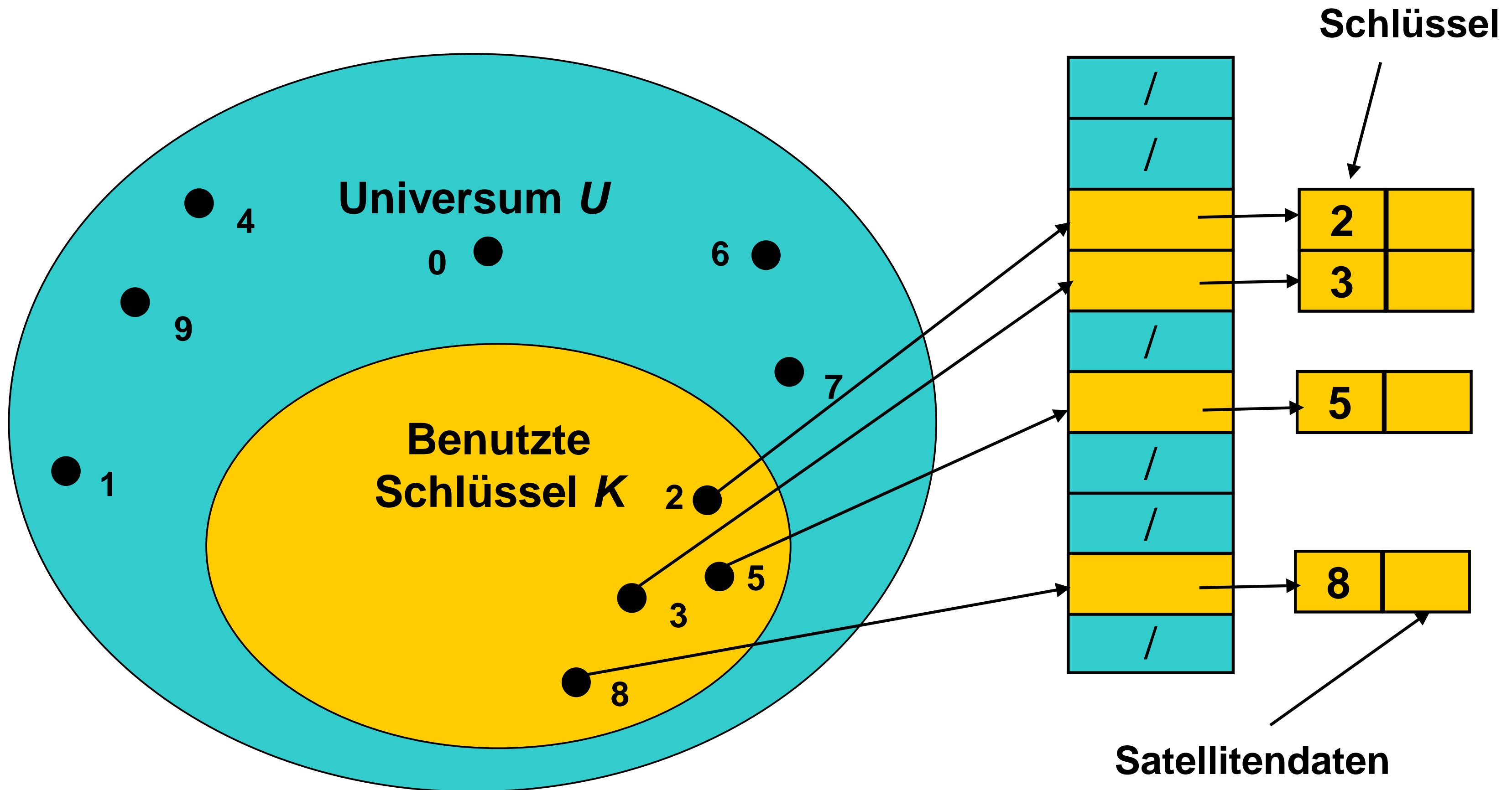
13. Hashing

- Hashing: einfache Methode, um Wörterbücher zu implementieren, d.h. Hashing unterstützt die Operationen Search, Insert, Remove.
- Worst-case Zeit für Search: $\Theta(n)$.
- In der Praxis jedoch sehr gut.
- Unter gewissen Annahmen, erwartete Suchzeit **O**(1).
- Hashing: Verallgemeinerung von direkter Adressierung durch Arrays.

Direkte Adressierung mit Arrays

- Schlüssel für Objekte der dynamischen Menge aus U:={0,…,u-1}. Menge U wird Universum genannt.
- Nehmen an, dass alle Objekte unterschiedliche Schlüssel haben.
- Legen Array T[0,…,u-1] an. Position k in T reserviert für Objekt mit Schlüssel k.
- $T[k]$ verweist auf Objekt mit Schlüssel k. Falls kein Objekt mit Schlüssel k in Struktur, so gilt $T[k]$=NIL.

Direkte Adressierung - Illustration

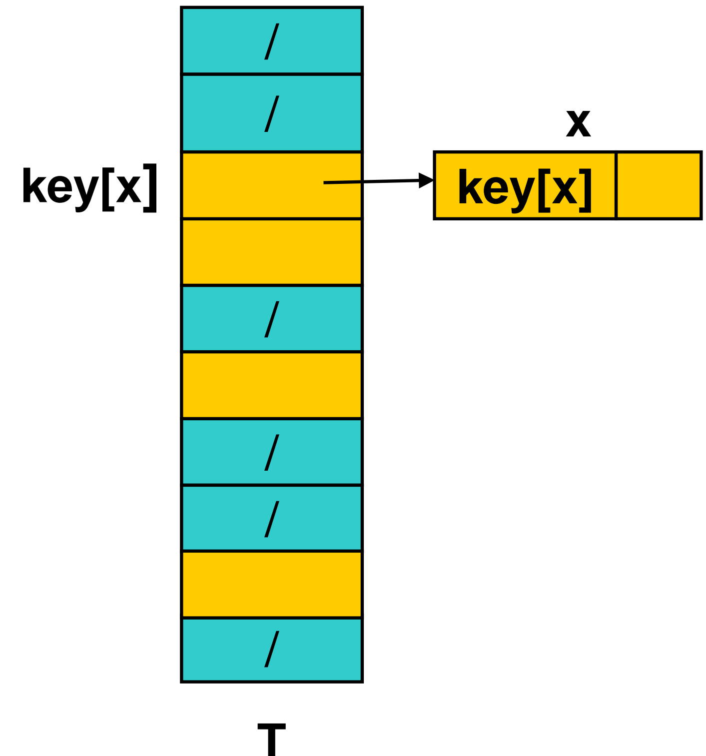


Operationen bei direkter Adressierung (1)

Insert(T,x):
  $T[key[x]] \leftarrow x$

Remove(T,k):
  $T[k] \leftarrow NIL$

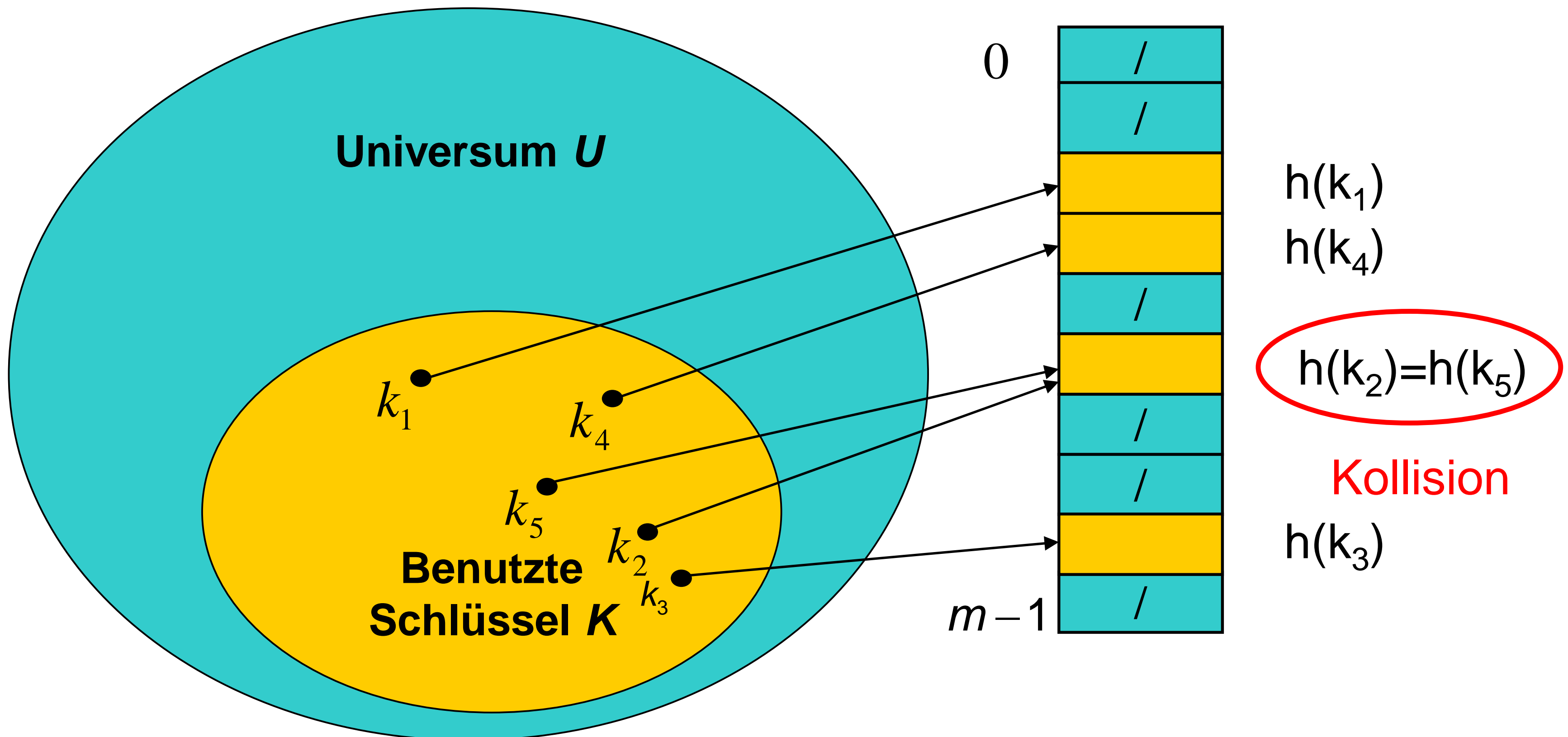
Search(k):
  return T[k]



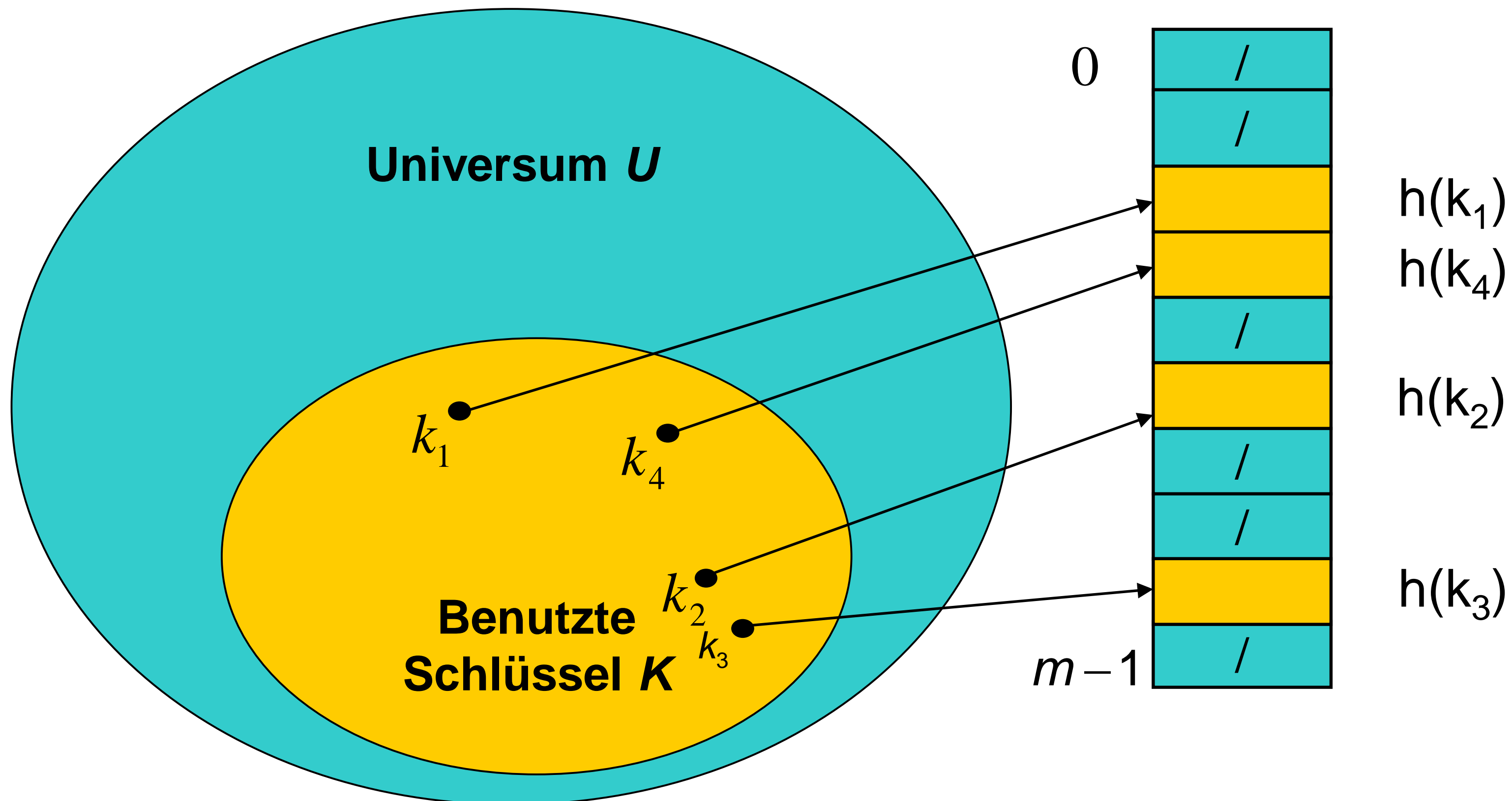
Operationen bei direkter Adressierung (2)

- Laufzeit jeweils **O**(1).
- Direkte Adressierung nicht möglich, wenn Universum U sehr groß ist.
- Speicherineffizient ($\Theta(|U|)$), wenn Menge der aktuell zu speichernden Schlüssel deutlich kleiner als U ist.
- Lösung: verwende Hashfunktion $h:U\rightarrow\{0,\ldots,m-1\}$, um die aktuelle Schlüsselmenge $K\subseteq U$ auf eine Hashtabelle T abzubilden mit $m=O(|K|)$.
- Problem: es kann zu Kollisionen kommen!

Kollisionen sind nicht auszuschließen



Hashing (ohne Kollisionen)

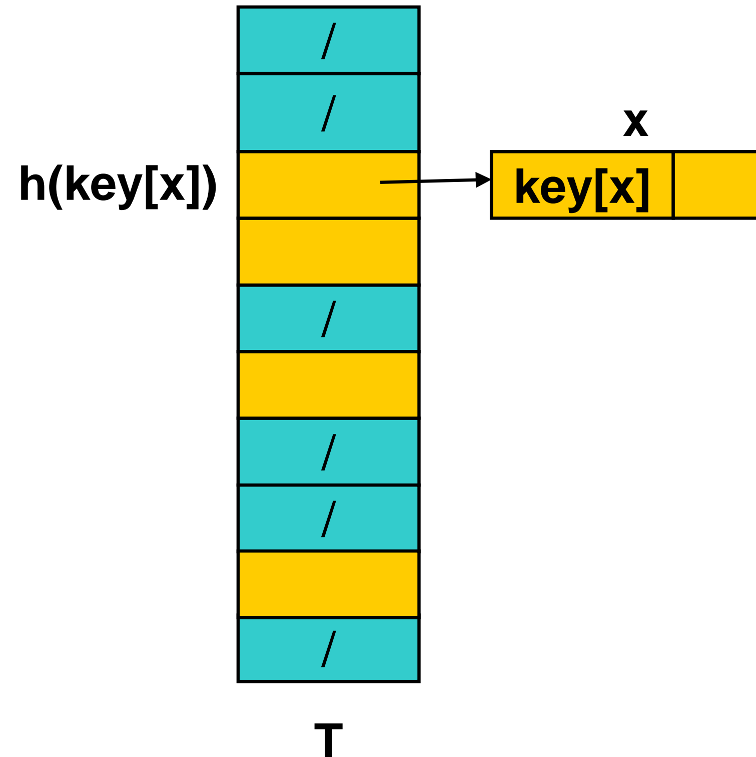


Hashing (ohne Kollisionen)

Insert(T,x):
    T[h(key[x])] ← x

Remove(T,k):
    if key[T[h(k)]]=k then
        T[h(k)] ← NIL

Search(T,k):
    if key[T[h(k)]]=k then
        return T[h(k)]
    else
        return NIL

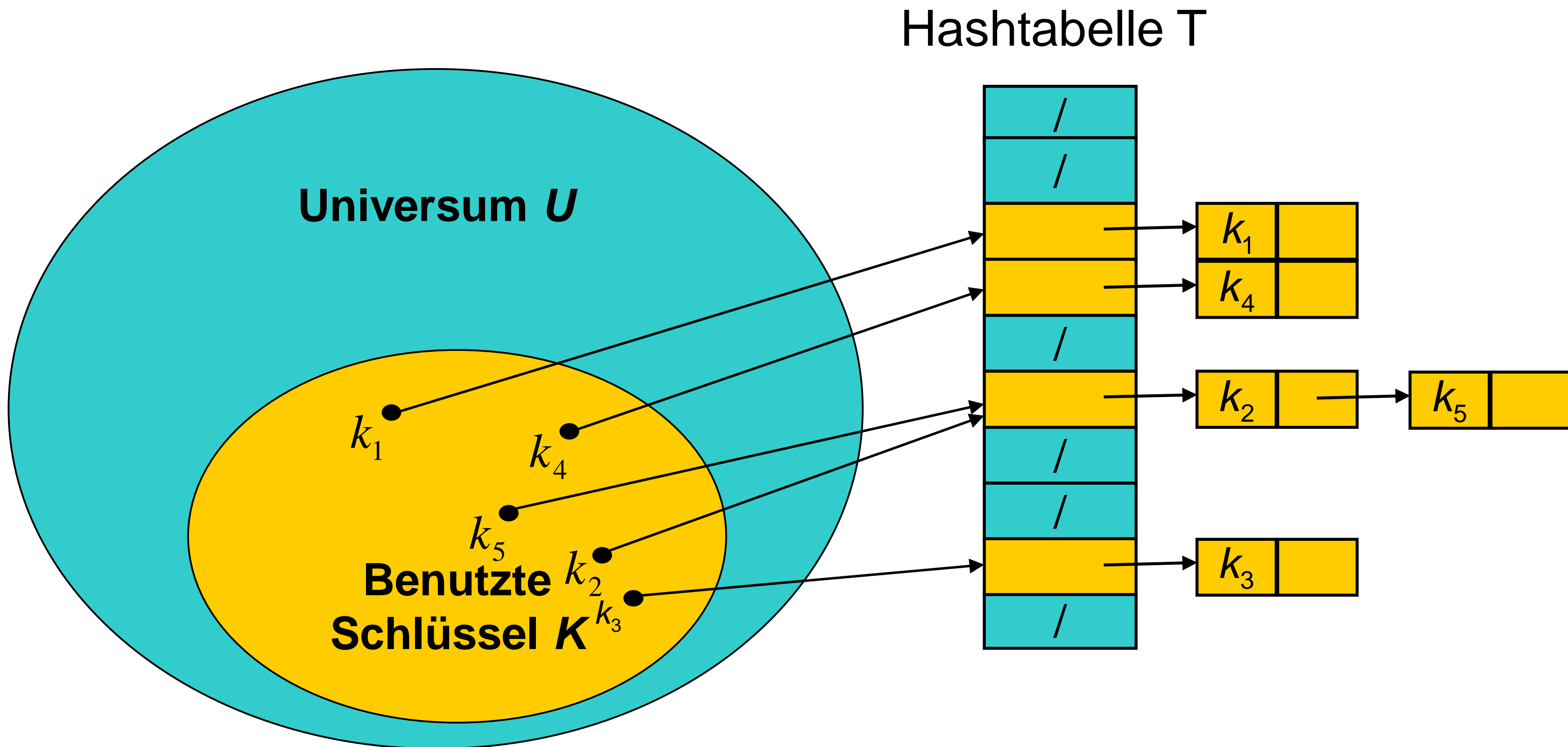


Maßnahmen zur Kollisionsauflösung

Mögliche Strategien:

- Geschlossene Adressierung
  - Kollisionsauflösung durch Listen
- Offene Adressierung
  - Lineares/Quadratisches Hashing: es wird nach der nächsten freien Stelle in T gesucht
- Kuckuckshashing: geschickte Verwendung von zwei Hashfunktionen

Hashing mit Listen - Illustration



Hashing mit Listen

Chained-Hash-Insert(T,x):

Füge x vorne in Liste T[h(key[x])] ein

Chained-Hash-Remove(T,k):

Entferne alle x aus Liste T[h(k)] mit key[x]=k

Chained-Hash-Search(T,k):

Suche nach Element mit Schlüssel k in Liste T[h(k)]

- Laufzeit für Insert **O**(1).
- Laufzeit für Remove, Search proportional zur Länge von $T[h(k)]$.

Hashing mit Listen

- m Größe der Hashtabelle T, n Anzahl der gespeicherten Objekte. Dann heisst $\alpha := n/m$ der Lastfaktor von T.
- α ist die durchschnittliche Anzahl von Elementen in einer verketteten Liste.
- Werden alle Objekte auf denselben Wert gehasht, so benötigt Suche bei n Elementen Zeit $\Theta(n)$. Dasselbe Verhalten wie verkettete Listen.
- Im Durchschnitt aber ist Suche deutlich besser, falls eine *gute* Hashfunktion h benutzt wird.
- Gute Hashfunktion sollte Werte wie zufällige Funktion streuen, aber was genau heißt das?

Universelles Hashing

Problem: Wie konstruiert man genügend zufällige Hashfunktionen?

Definition 13.1 [universelles Hashing]:
Sei c eine positive Konstante. Eine Familie H von Hashfunktionen auf $\{0,\ldots,m-1\}$ 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

Satz 13.2: 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 $T[i]$ mit mindestens einem Schlüssel die erwartete Anzahl Schlüssel in $T[i]$ in $O(1+c\cdot n/m)$.

Beweis:

- Betrachte festen Schlüssel k_0 mit Position $T[i]$
- Zufallsvariable $X_k \in \{0,1\}$ für jeden Schlüssel $k \in K \setminus \{k_0\}$
- $X_k = 1 \Leftrightarrow h(k)=i$
- $X = \sum_{k \neq k_0} X_k$: Anzahl Elemente in Position $T[i]$ von k_0
- $E[X] = \sum_{k \neq k_0} E[X_k] = \sum_{k \neq k_0} \Pr[X_k=1] \leq \sum_{k \neq k_0} (c/m)$
  $= (n-1)c/m$

H ist c-universell

Universelles Hashing

Aus Satz 13.2: Tabellengröße $\Theta(|K|)$ ergibt erwartet konstante Zeit für Insert, Remove und Search.

Aber wie konstruieren wir eine c-universelle Hash-funktion?

Betrachte die Familie der Hashfunktionen

$$h_a(x) = a \cdot x \pmod{m}$$

mit $a,x \in [m]^d$ (wobei $[m] = \{0,\ldots,m-1\}$ und $a \cdot x = \sum_i a_i x_i$).

Satz 13.3: $H=\{ h_a: a \in [m]^d \}$ ist eine 1-universelle Familie von Hashfunktionen, falls m prim ist.

Universelles Hashing

Beweis:

- Betrachte zwei feste Schlüssel $x \neq y$ mit $x=(x_1,\ldots,x_d)$ und $y=(y_1,\ldots,y_d)$
- Anzahl Möglichkeiten für a, dass $h_a(x)=h_a(y)$:

$$h_a(x)=h_a(y) \Leftrightarrow \sum_i a_i x_i = \sum_i a_i y_i \pmod{m}$$
$$\Leftrightarrow a_j(y_j-x_j) = \sum_{i\neq j} a_i (x_i-y_i) \pmod{m}$$

  für ein j mit $x_j \neq y_j$
- Falls m prim, dann gibt es für jede Wahl von $a_1,\ldots,a_{j-1}, a_{j+1},\ldots,a_d$ genau ein a_j, das diese Gleichung erfüllt.
- Also ist die Anzahl Möglichkeiten für a gleich m^{d-1}.
- Daher ist $|\{h \in H: h(x)=h(y)\}| \leq (1/m)\,|H|$.

Universelles Hashing

Praktische Hashfunktionsklasse: Tabulationshashing

- Sei $U=[d]^c$ und die Hashfunktion $h:U\rightarrow[m]$ definiert als

$$h(x_1,\ldots,x_c)=\bigoplus_{i=1}^{c} T_i[x_i]$$

für Felder $T_i[0..d-1]$ mit zufällig gewählten Einträgen aus $[m]$, wobei $\oplus$ das bitweise XOR ist.

Beispiel:

- $c=d=4$, $m=8$
- $T_1[0..3]=\{4,7,3,4\}$, $T_2[0..3]=\{2,5,6,1\}$, $T_3[0..3]=\{7,1,0,3\}$, $T_4[0..3]=\{3,6,5,1\}$
- $h(2,3,1,1)$ $= T_1[2] \oplus T_2[3] \oplus T_3[1] \oplus T_4[1]$
  $= 3 \oplus 1 \oplus 1 \oplus 6$ (zur Basis 10)
  $= 011 \oplus 001 \oplus 001 \oplus 110$ (zur Basis 2)
  $= 101$ (zur Basis 2)
  $= 5$ (zur Basis 10)

Universelles Hashing

Praktische Hashfunktionsklasse: Tabulationshashing

- Sei $U=[d]^c$ und die Hashfunktion $h:U\to[m]$ definiert als

$$h(x_1,\ldots,x_c)=\bigoplus_{i=1}^{c} T_i[x_i]$$

  für Felder $T_i[0..d-1]$ mit zufällig gewählten Einträgen aus $[m]$, wobei $\oplus$ das bitweise XOR ist.

Diese Hashfunktion verhält sich fast wie eine vollständig zufällige Hashfunktion (siehe Mihai Patrascu und Mikkel Thorup. The Power of Simple Tabulation Hashing. Journal of the ACM, 59(3), 2011).

Anwendungsmöglichkeiten:

- Der Einfachheit halber: m ist 2-Potenz ($\to$ dyn. Felder).
- 32-bit Schlüssel: für $c=4$ haben wir 4 Tabellen mit jeweils 256 Einträgen aus $[m]$
- 64-bit Schlüssel: für $c=16$ haben wir 16 Tabellen mit jeweils 16 Einträgen aus $[m]$

Maßnahmen zur Kollisionsauflösung

Mögliche Strategien:

- Geschlossene Adressierung
  - Kollisionsauflösung durch Listen
- Offene Adressierung
  - Lineares/Quadratisches Hashing: es wird nach der nächsten freien Stelle in T gesucht
- Kuckuckshashing: geschickte Verwendung von zwei Hashfunktionen

Offene Adressierung (1)

- Geschlossene Adressierung weist Objekt mit gegebenem Schlüssel feste Position in Hashtabelle zu.
- Bei Hashing durch offene Adressierung wird Objekt mit Schlüssel keine feste Position zugewiesen.
- Position abhängig vom Schlüssel und bereits belegten Positionen in Hashtabelle.
- Konkret: Für neues Objekt wird erste freie Position gesucht. Dazu wird Hashtabelle nach freier Position durchsucht.

Offene Adressierung (2)

- Keine Listen zur Kollisionsvermeidung. Wenn Anzahl eingefügter Objekte m ist, dann sind keine weiteren Einfügungen mehr möglich.
- Listen zur Kollisionsvermeidung möglich, aber Ziel von offener Adressierung ist es, Verfolgen von Verweisen zu vermeiden.
- Suche von Objekten in der Regel schneller, da keine Listen linear durchsucht werden müssen. Aber Laufzeit für Einfügen jetzt nicht mehr im worst cae $\Theta(1)$.

Hashfunktionen bei offener Adressierung

- Hashfunktion der Form $h:U\times\{0,1,2,\ldots\} \rightarrow [m]$ legt für jeden Schlüssel fest, in welcher Reihenfolge für Objekte mit diesem Schlüssel nach freier Position in Hashtabelle gesucht wird (zunächst $h(k,0)$, dann $h(k,1)$ usw.).

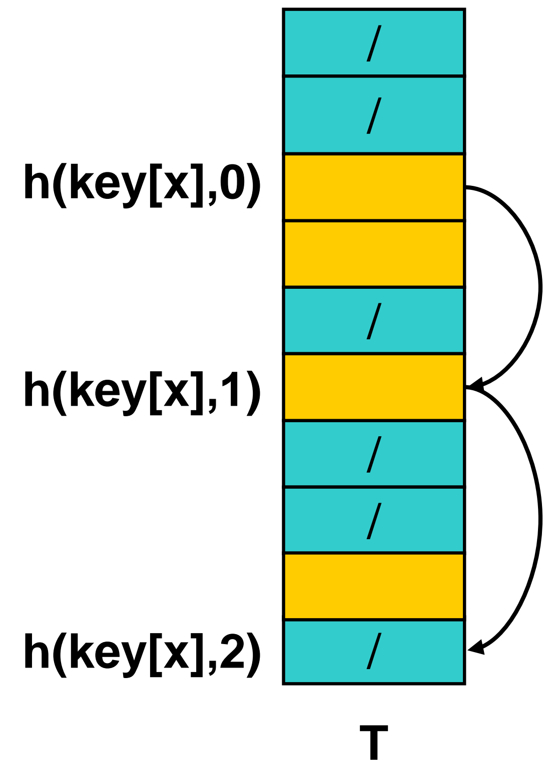
- Idealerweise formt die Folge $(h(k,0),\ldots,h(k,m-1))$ eine Permutation von $[m]$, so dass keine Position der Hashtabelle ausgelassen wird.

- $(h(k,0), h(k,1), \ldots.)$ heißt Testfolge bei Schlüssel k.

Einfügen bei offener Adressierung

```
Hash-Insert(T,x)
  for i←0 to m-1 do
    j←h(key[x],i)
    if T[j] = NIL then T[j]←x; return
  error „Hashtabelle vollständig gefüllt“

Hash-Search(T,k)
  i←0
  repeat
    j←h(k,i)
    if key[T[j]] = k then return T[j]
                else i←i+1
  until T[j]=NIL or i=m
  return NIL
```



Probleme beim Entfernen

- Wir können Felder *i* mit gelöschten Schlüsseln nicht wieder mit NIL belegen, denn dann wird Suche nach Schlüsseln, bei deren Einfügung Position *i* getestet wird, fehlerhaft sein!

- Mögliche Lösung ist, Felder gelöschter Schlüssel mit DELETED zu markieren. Aber dann werden Laufzeiten für Hash-Insert und Hash-Delete nicht mehr nur vom Lastfaktor $\alpha = n/m$ abhängen.

- Daher Anwendung von offener Adressierung oft nur, wenn keine Objekte entfernt werden müssen.

Probleme beim Entfernen

- Ein möglicher Trick, um Remove Operationen dennoch zu erlauben, funktioniert wie folgt:
  - Markiere Felder gelöschter Schlüssel mit DELETED.
  - Ein Zähler zählt die Anzahl gelöschter Einträge.
  - Übersteigt die Anzahl gelöschter Einträge die Anzahl der verbleibenden Elemente in der Hashtabelle, wird die Hashtabelle nochmal komplett neu aufgebaut, indem alle noch vorhandenen Elemente nochmal neu in die anfangs leere Hashtabelle mittels Hash-Insert eingefügt werden.

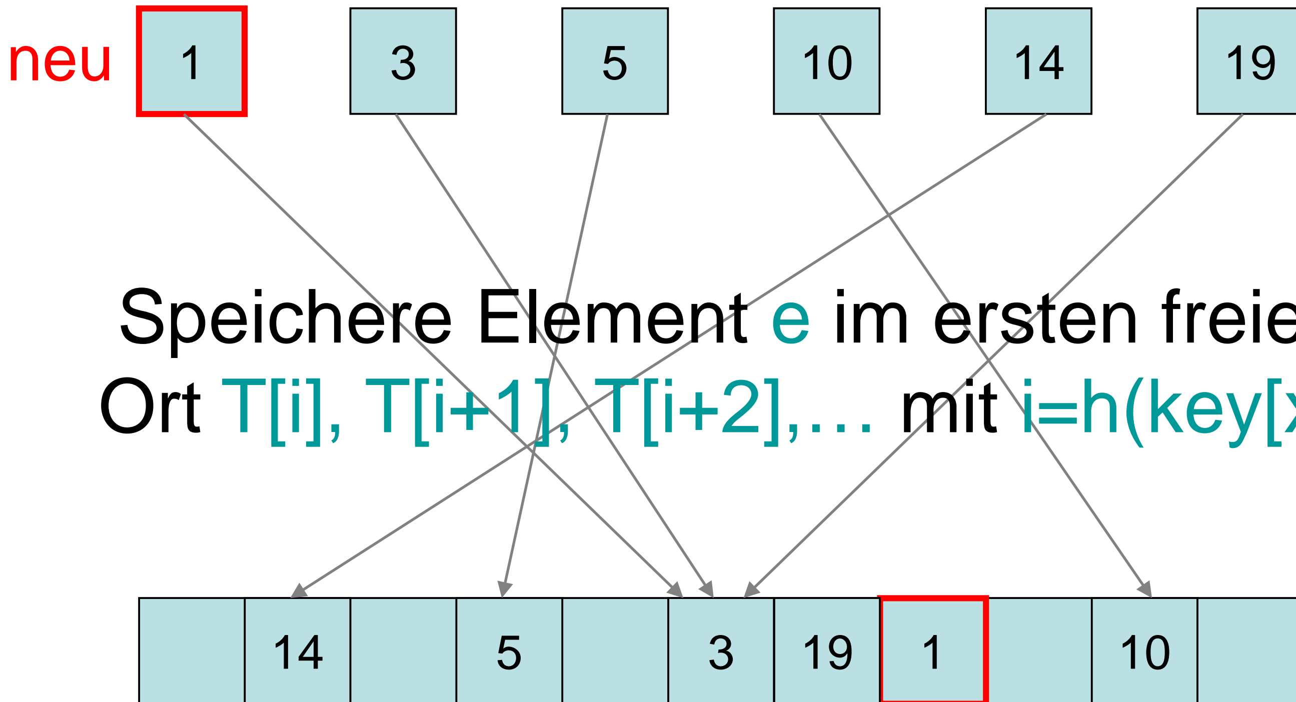
- Potenzialanalyse: Erwartete amortisierte Laufzeit von Insert, Remove und Search konstant (für $m=\Omega(|K|)$).

Mögliche Hashfunktionen

1. $h': U \to \{0,1,\dots,m-1\}$ Funktion.
   Lineares Hashen:
   $$h(k,i) = \left(h'(k) + i\right) \bmod m.$$

2. $h': U \to \{0,1,\dots,m-1\}$ Funktion, $c_1, c_2 \neq 0$.
   Quadratisches Hashen:
   $$h(k,i) = \left(h'(k) + c_1 i + c_2 i^2\right) \bmod m.$$

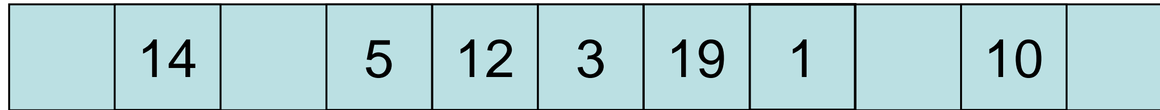
Lineares Hashing



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

Lineares Hashing

Lauf:



Lauf der Länge 5

Satz 13.4: Falls n Elemente in einer Hashtabelle T der Größe $m \geq 2en$ 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)

Lineares Hashing

Beweis:

n: Anzahl Elemente, $m \geq 2en$: Größe der Tabelle

Anzahl Möglichkeiten für Anfänge des Laufs: k

Anzahl Möglichkeiten zur Wahl von k Objekten: $\binom{n}{k} \leq \left(\frac{en}{k}\right)^k$

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

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

Lineares Hashing

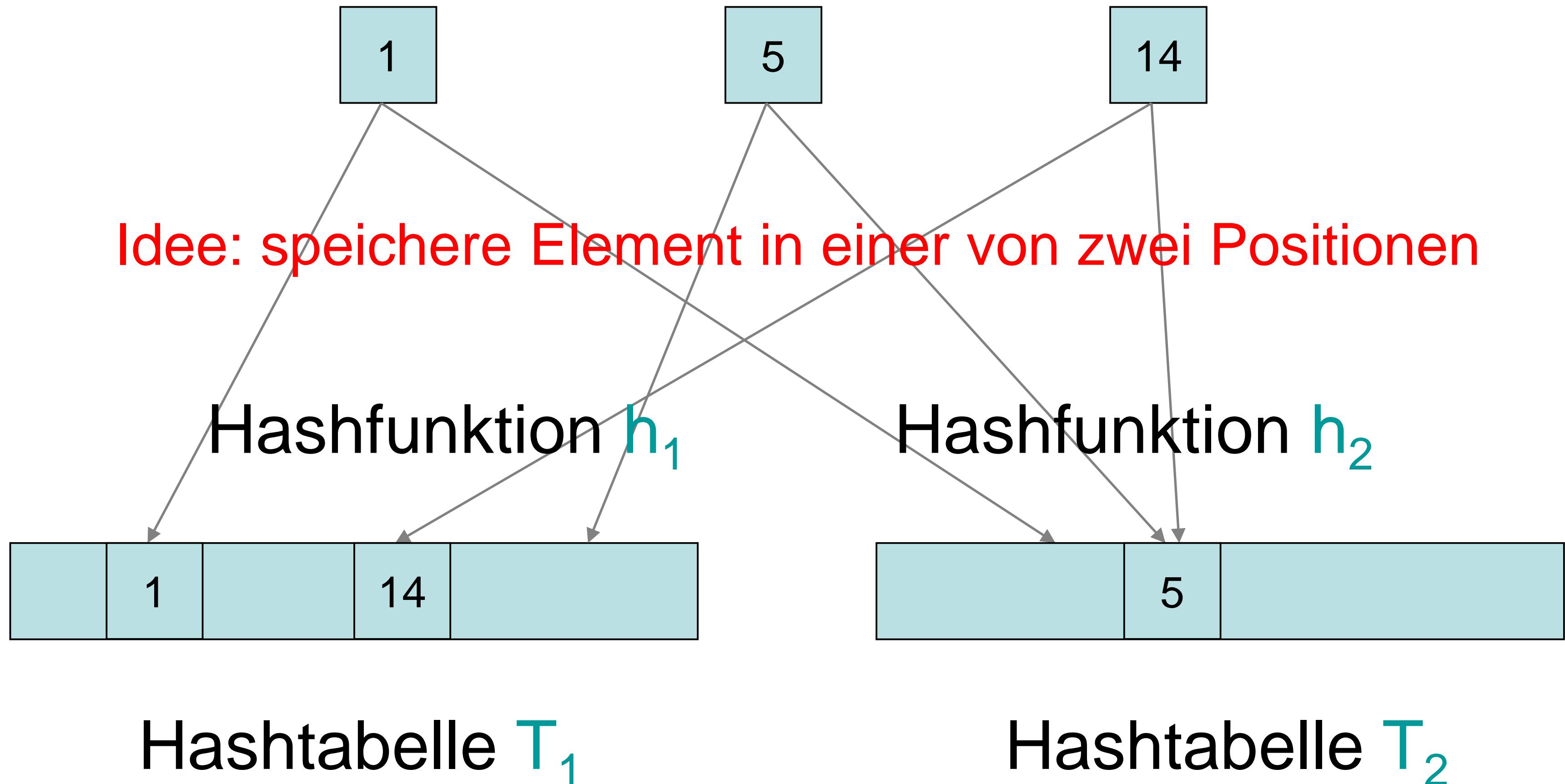
- p_k := Pr[T[i] in Lauf der Länge k] $\leq k \cdot (1/2)^k$
- E[Länge des Laufs über T[i]]
  $= \sum_{k \geq 0} k \cdot p_k \leq \sum_{k \geq 0} k^2 \cdot (1/2)^k = O(1)$
- Also erwartet konstanter Aufwand für Operationen Insert, Remove und Search.
- Vorteil des linearen Hashings: eng zusammenliegender Bereich des Adressraums wird durchsucht (gut für Caching).

Maßnahmen zur Kollisionsauflösung

Mögliche Strategien:

- Geschlossene Adressierung
  - Kollisionsauflösung durch Listen
- Offene Adressierung
  - Lineares/Quadratisches Hashing: es wird nach der nächsten freien Stelle in T gesucht
- Kuckuckshashing: geschickte Verwendung von zwei Hashfunktionen

Kuckuckshashing



Kuckuckshashing

T_1, T_2: Hashtabellen der Größe m

Search(T_1,T_2,k):

if $key[T_1[h_1(k)]]=k$ then return $T_1[h_1(k)]$
if $key[T_2[h_2(k)]]=k$ then return $T_2[h_2(k)]$
return NIL

Remove(T_1,T_2,k):

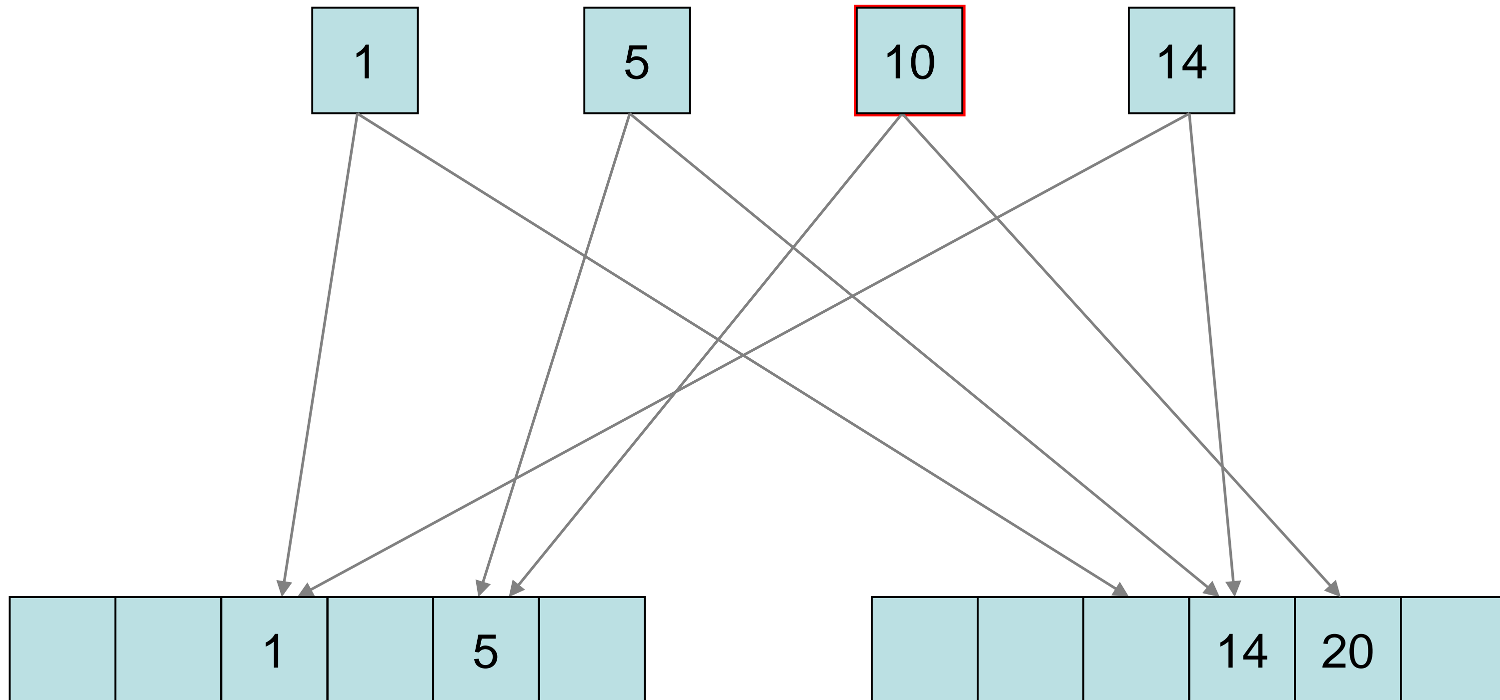
if $key[T_1[h_1(k)]]=k$ then $T_1[h_1(k)]\leftarrow NIL$
if $key[T_2[h_2(k)]]=k$ then $T_2[h_2(k)]\leftarrow NIL$

Kuckuckshashing

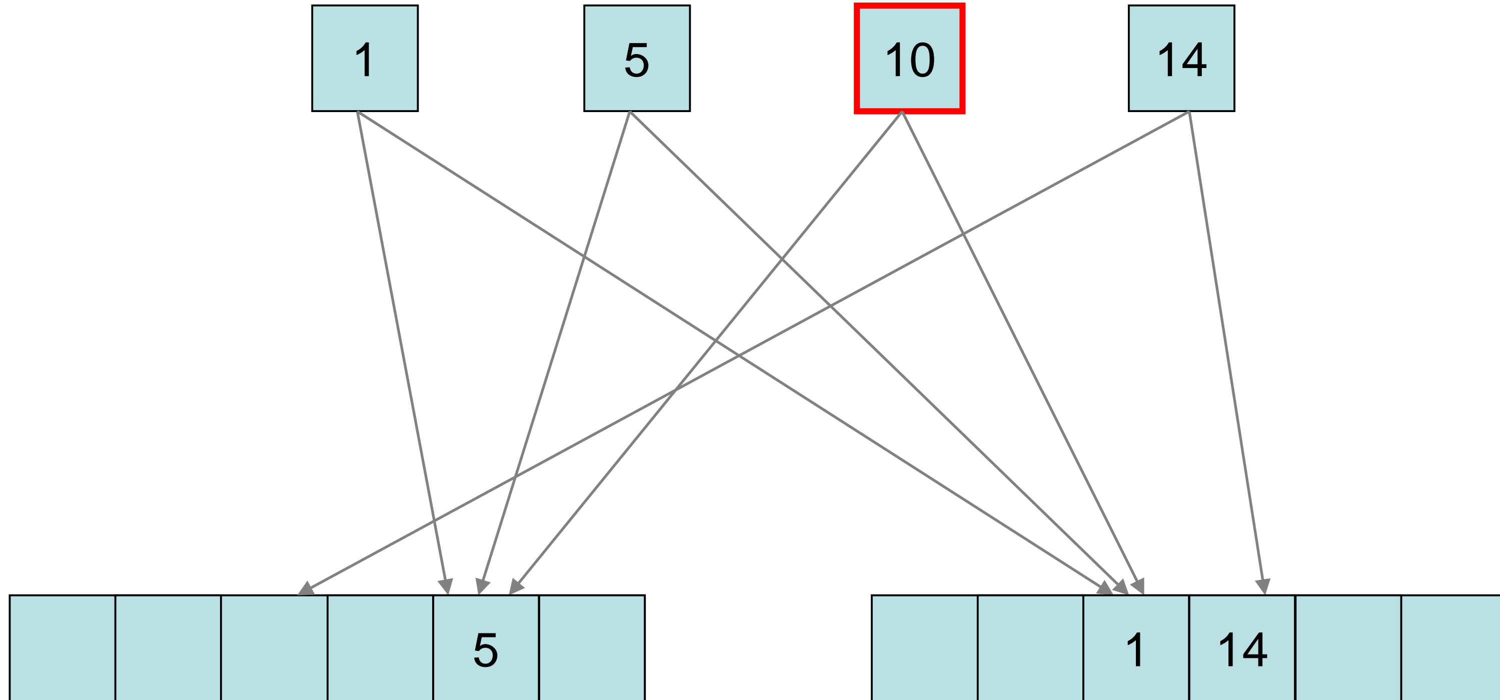
```
Insert(T_1,T_2,x):
  // key[x] schon in Hashtabelle?
  if T_1[h_1(key[x])]=NIL or key[T_1[h_1(key[x])]]=key[x] then
      T_1[h_1(key[x])]←x; return
  if T_2[h_2(key[x])]=NIL or key[T_2[h_2(key[x])]]=key[x] then
      T_2[h_2(key[x])] ←x; return
  // nein, dann einfügen
  while true do
      x↔T_1[h_1(key[x])]  // tausche x mit Pos. in T_1
      if x=NIL then return
      x↔T_2[h_2(key[x])]  // tausche x mit Pos. in T_2
      if x=NIL then return
```

Oder maximal d·log n oft, wobei Konstante d so gewählt ist, dass die Wahrscheinlichkeit eines Erfolgs unter der einer Endlosschleife liegt. Bei Misserfolg kompletter Rehash mit neuen, zufälligen h_1, h_2

Kuckuckshashing



Kuckuckshashing



Endlosschleife!

Kuckuckshashing

Laufzeiten:
- Search, Remove: O(1) (worst case!)
- Insert: O(Länge der Umlegefolge + evtl. Aufwand für Rehash bei Endlosschleife)

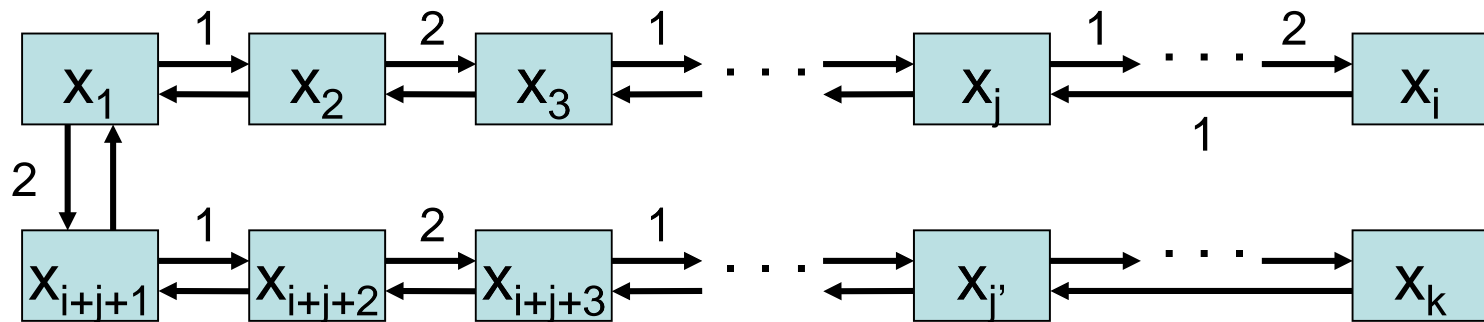
Laufzeit der insert-Operation: 2 Fälle
1. Insert läuft nach t Runden in eine End-losschleife
2. Insert terminiert nach t Runden

Im folgenden: n: #Schlüssel, m: Tabellengröße

Kuckuckshashing

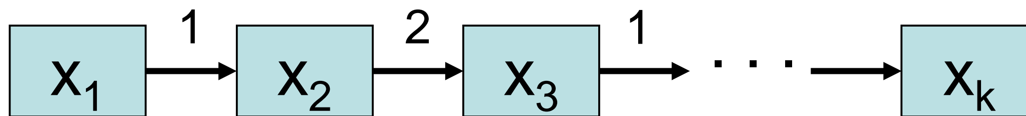
Fall 1: Endlosschleife

Folgende Elementabhängigkeiten existieren in diesem Fall ($x \xrightarrow{i} y$: x verdrängt y für HF i):

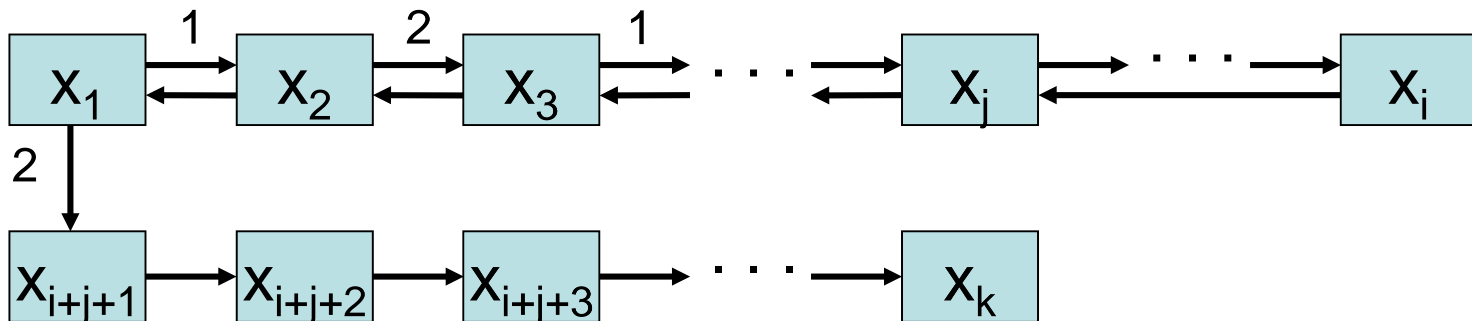


Kuckuckshashing

Fall 2: Terminierung nach t Runden



oder



Kuckuckshashing

Man kann zeigen:

- Die erwartete Laufzeit von Insert ist $O(1+1/\varepsilon)$ bei Tabellengrößen $m=(1+\varepsilon)n$.
- Rehash benötigt $O(n)$ erwartete Zeit, passiert aber nur mit Wahrscheinlichkeit $O(1/n)$.

Problem: Wahrscheinlichkeit für Endlosschleife noch recht hoch!

Lösung: verwende Notfallcache
(siehe: A. Kirsch, M. Mitzenmacher, U. Wieder. More robust hashing: Cuckoo hashing with a stash. In European Symposium on Algorithms, pp. 611-622, 2008.)

Kuckuckshashing mit Cache

T_1, T_2: Hashtabellen der Größe m

C: Notfallcache der Größe c (c konstant)

Search(T_1, T_2,C,k):

```
if ∃i: key[C[i]]=k then return C[i]
if key[T_1[h_1(k)]]=k then return T_1[h_1(k)]
if key[T_2[h_2(k)]]=k then return T_2[h_2(k)]
return NIL
```

Kuckuckshashing mit Cache

```
Remove(T1, T2,C,k):
  if ∃i: key[C[i]]=k then C[i]←NIL; return
  if key[T1[h1(k)]]=k then
    // lösche k aus T1 und ersetze es evtl. aus C
    p←h1(k); T1[p] ←NIL
    if ∃i: h1(key[C[i]])=p then T1[p]←C[i]; C[i]←NIL
  if key[T2[h2(k)]]=k then
    // lösche k aus T2 und ersetze es evtl. aus C
    p←h2(k); T2[p] ←NIL
    if ∃i: h2(key[C[i]])=p then T2[p]←C[i]; C[i]←NIL
```

Kuckuckshashing mit Cache

```
Insert(T₁, T₂,C,x):
  // key[x] schon in Hashtabelle?
  if ∃i: key[C[i]]=key[x] then C[i]←x; return
  if T₁[h₁(key[x])]=NIL or key[T₁[h₁(key[x])]]=key[x] then
      T₁[h₁(key[x])]←x; return
  if T₂[h₂(key[x])]=NIL or key[T₂[h₂(key[x])]]=key[x] then
      T₂[h₂(key[x])]←x; return
  // nein, dann füge x ein
  r:=1
  while r<d·log n do        // d: Konstante >2
      x↔T₁[h₁(key[x])]     // tausche e mit Pos. in T₁
      if x=NIL then return
      x↔T₂[h₂(key[x])]     // tausche e mit Pos. in T₂
      if x=NIL then return
      r←r+1
  if ∃i: C[i]=NIL then C[i]←x
                  else rehash
```

Kuckuckshashing mit Cache

Für zufällige Hashfunktionen kann man (für ein etwas aufwändigeres Verfahren) zeigen (aber das angegebene hat simultiv dieselbe Eigenschaft):

Satz 13.5: Die Wahrscheinlichkeit, dass mehr als c Elemente zu einem Zeitpunkt in C sind, ist $O(1/n^c)$.

Beweis: aufwändig, hier nicht behandelt

Der Satz gilt auch für Tabulationshashing.

Dynamische Hashtabelle

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

Lösung: Reallokation

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

Für Tabulationshashing kein Problem, da die Tabellen-größe eine 2-Potenz sein kann, aber für die andere präsentierte Hashklasse muss die Tabellengröße prim sein.

Dynamische Hashtabelle

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

Lösung:

- Für jedes k gibt es Primzahl in $[k^3,(k+1)^3]$
- Wähle primes m, so dass $m \in [k^3, (k+1)^3]$
- Jede nichtprime Zahl in $[k^3,(k+1)^3]$ muss Teiler $< \sqrt{(k+1)^3}$ haben
  $\rightarrow$ erlaubt effiziente Primzahlfindung über Sieb des Eratosthenes

Dynamische Hashtabelle

Primzahlbestimmung in $\{2,3,...,n\}$:

Sieb des Eratosthenes:

```
// initialisiere gestrichen-Feld
for i←2 to n do
    gestrichen[i]←FALSE
// siebe nichtprime Zahlen aus
for i←2 to √n do
  if not gestrichen[i] then
        for j←i to n/i do
            gestrichen[j·i]←TRUE
// gib Primzahlen aus
for i←2 to n do
    if not gestrichen[i] then print i
```

Dynamische Hashtabelle

Primzahlbestimmung in $[k^3,(k+1)^3]$:

- Wende Sieb des Eratosthenes mit Werten von 1 bis $\sqrt{(k+1)^3}$ auf $[k^3,(k+1)^3]$ an.

Laufzeit:

- Es gibt $\Theta(k^2)$ Elemente in $[k^3, (k+1)^3]$
- Sieb des Eratosthenes mit Wert i auf $[k^3,(k+1)^3]$: Laufzeit $\Theta(k^2/i)$.
- Gesamtlaufzeit: $\Sigma_i\ \Theta(k^2/i) = \Theta(k^2 \ln k)=o(m)$

Perfektes Hashing

- Soll nur Suchen unterstützt werden (d.h. alle Objekte werden nur zu Beginn eingefügt), so kann das Hashing verbessert werden.

- *n* Objekte werden zu Beginn in Zeit $\Theta(n)$ eingefügt.

- Jede Suche benötigt danach im *worst-case* Zeit $\Theta(1)$.

- Idee wie bei Hashing mit Kollisionsverwaltung. Allerdings werden nicht Listen zur Kollisionsverwaltung benutzt, sondern wieder Hashtabellen.

- Verfahren wird perfektes Hashing genannt.

Perfektes Hashing

Exkurs in Wahrscheinlichkeitstheorie:

- Zufallsvariable: $X:U\to\mathbb{R}$
- Erwartungswert von X:

$$E[X] = \Sigma_{u\in U} \Pr[u]\cdot X(u)$$
$$= \Sigma_{x\in\mathbb{R}} x\cdot\Pr[X=x]$$

Markov Ungleichung: Für jede nichtnegative Zufallsvariable X gilt:

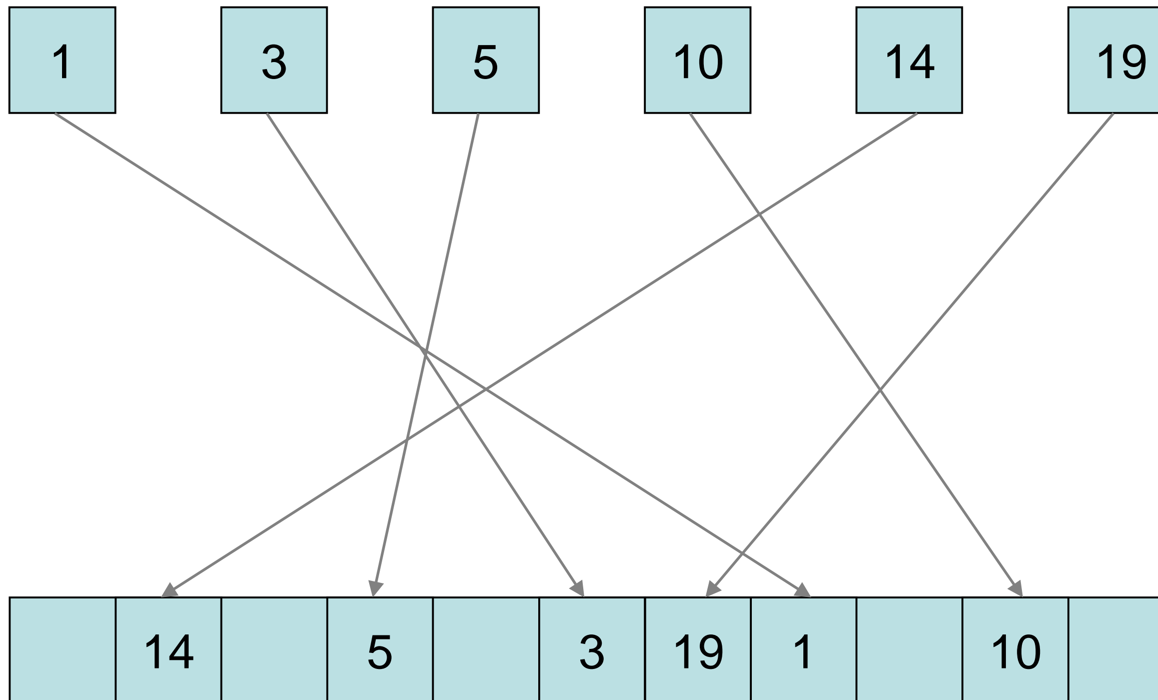
$$\Pr[X\geq k\cdot E[X]]\leq 1/k$$

Beweis:

$$E[X]\geq\Sigma_{x\geq k} x\cdot\Pr[X=x]\geq\Sigma_{x\geq k} k\cdot\Pr[X=x]=k\cdot\Pr[X\geq k]$$

Perfektes Hashing

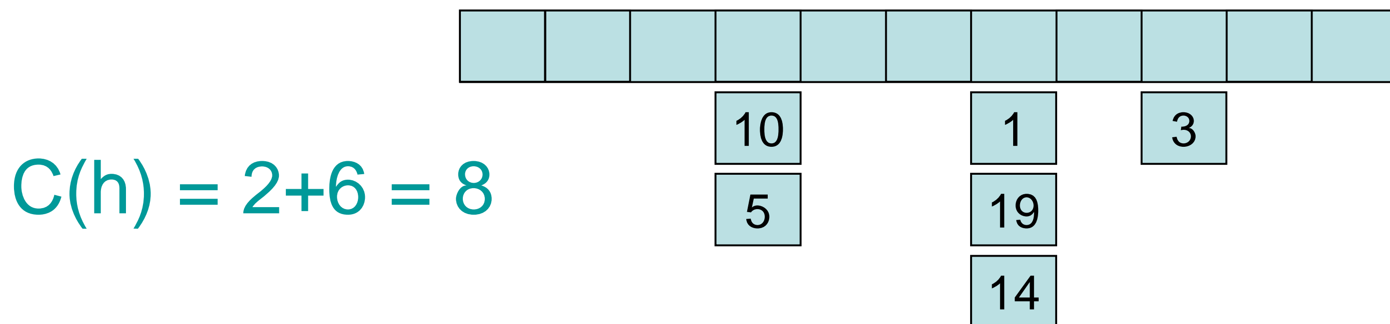
Ziel: perfekte Hashtabelle



Perfektes Hashing

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

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



Perfektes Hashing

Lemma 13.6: $E[C(h)] \leq c \cdot n(n-1)/m$, und für mindestens die Hälfte der $h \in H_m$ ist

$$C(h) \leq 2c \cdot n(n-1)/m.$$

Beweis:

- Zufallsvariable $X_{e,e'} \in \{0,1\}$ für jedes Paar $e,e' \in S$
- $X_{e,e'} = 1 \Leftrightarrow h(key[e])=h(key[e'])$
- Es gilt: $C(h) = \sum_{e \neq e'} X_{e,e'}$
- $E[C(h)] = \sum_{e \neq e'} E[X_{e,e'}] \leq \sum_{e \neq e'} (c/m) = c \cdot n(n-1)/m$

Perfektes Hashing

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

- Nach Markov:

$$Pr[X \geq k \cdot E[X]] \leq 1/k$$

- Also gilt

$$Pr[C(h) \geq 2c \cdot n(n-1)/m] \leq ½$$

- Da die Hashfunktion uniform zufällig gewählt wird, folgt Behauptung.

Perfektes Hashing

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

Lemma 13.7: $C(h) = \sum_i b_i^h(b_i^h-1)$.

Beweis:

- Zufallsvariable $C_i(h)$: Anzahl Kollisionen in $T[i]$
- Es gilt: $C_i(h) = b_i^h(b_i^h-1)$.
- Also ist $C(h) = \sum_i C_i(h) = \sum_i b_i^h(b_i^h-1)$.

Perfektes Hashing

Konstruktion der Hashtabelle:

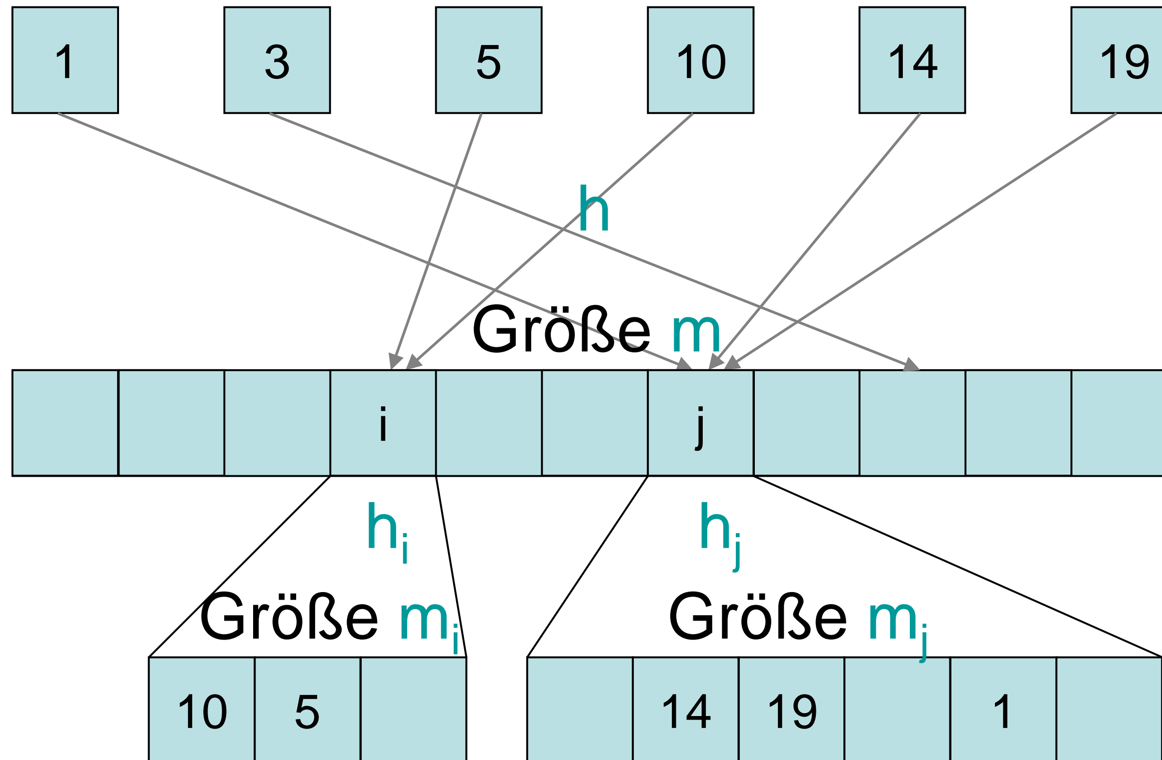
1) Wahl einer Hashfunktion h für S:

- Wähle eine Funktion $h \in H_{\alpha n}$ mit $C(h) \leq 2cn(n-1)/(\alpha\, n) \leq 2cn/\alpha$, α konstant (Lemma 13.6: zufällige Wahl mit Wahrscheinlichkeit $\geq 1/2$ erfolgreich)
- Für jede Position T[i] seien $m_i = 2c\, b_i^h(b_i^h-1)+1$ und $S_i = \{ x \in S \mid h(key[x])=i \}$

2) Wahl einer Hashfunktion h_i für jedes S_i:

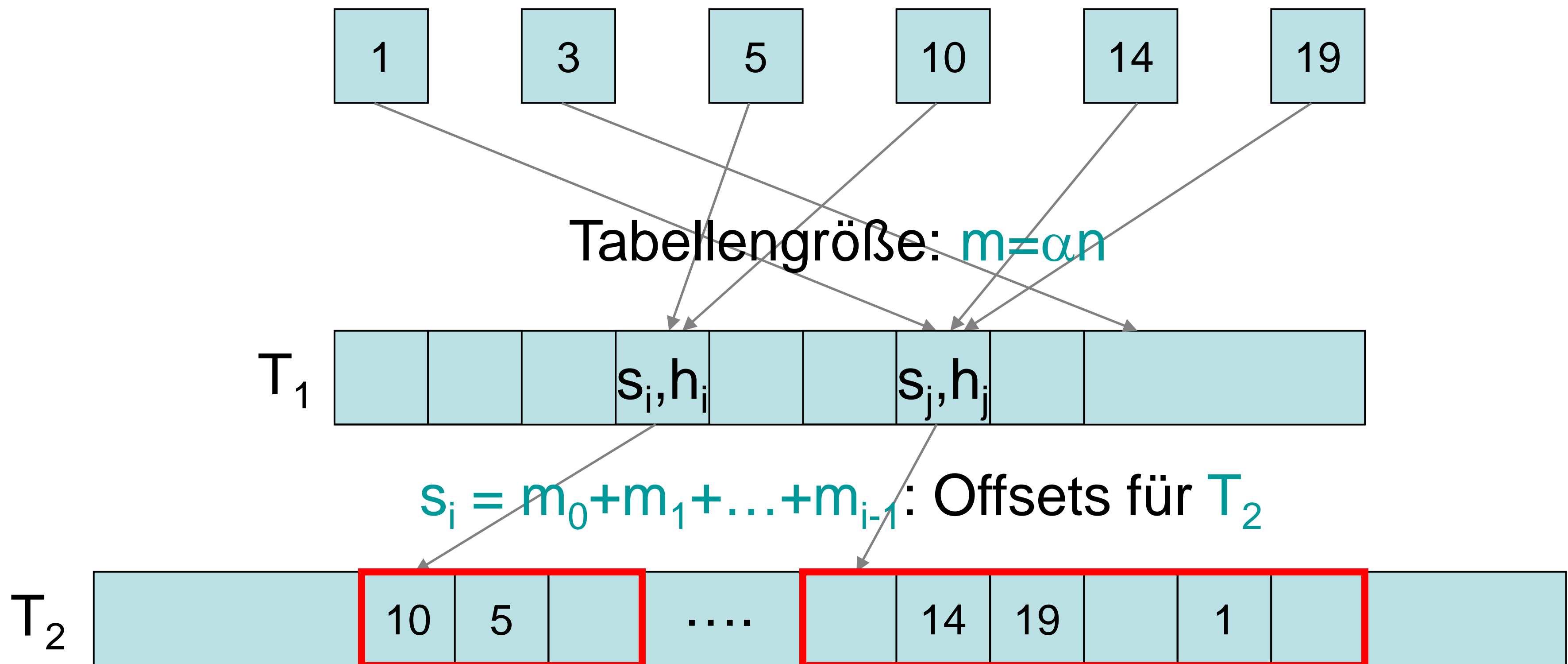
- Wähle eine Funktion $h_i \in H_{m_i}$ mit $C(h_i)<1$ (Lemma 13.6: zufällige Wahl mit Wahrscheinlichkeit $\geq 1/2$ erfolgreich)

Perfektes Hashing



Keine Kollisionen in Subtabellen

Perfektes Hashing



Tabellengröße: $\sum_i m_i \le \sum_i (2cb_i^h(b_i^h-1)+1) \le 4c^2n/\alpha + \alpha n$

Perfektes Hashing

Satz 13.8: Für jede Menge von n Schlüsseln gibt es eine perfekte Hashfunktion der Größe $\Theta(n)$, die in erwarteter Zeit $\Theta(n)$ konstruiert werden kann.

Beweis: Betrachte den folgenden Algorithmus.

```
FKS-Hashing(S)
  repeat                          erwartet O(1) Läufe (siehe Folie 57)
    h ← random-hash(H_αn)
  until C(h)≤2cn/α                O(n) Zeit
  for i←0 to m-1 do
    m_i ← 2c b_i^h(b_i^h-1)+1
    repeat                        erwartet O(1) Läufe (siehe Folie 57)
      h_i ← random-hash(H_m_i)
    until C(h_i)<1                O(n) Zeit
```

Insgesamt also erwartete Laufzeit $O(n)$.

Speicherplatz $O(n)$: siehe Folie 59

Changelog

30.05.16: Folien 35, 41, 44, 50

03.06.16: Folien 54, 55, 57, 60