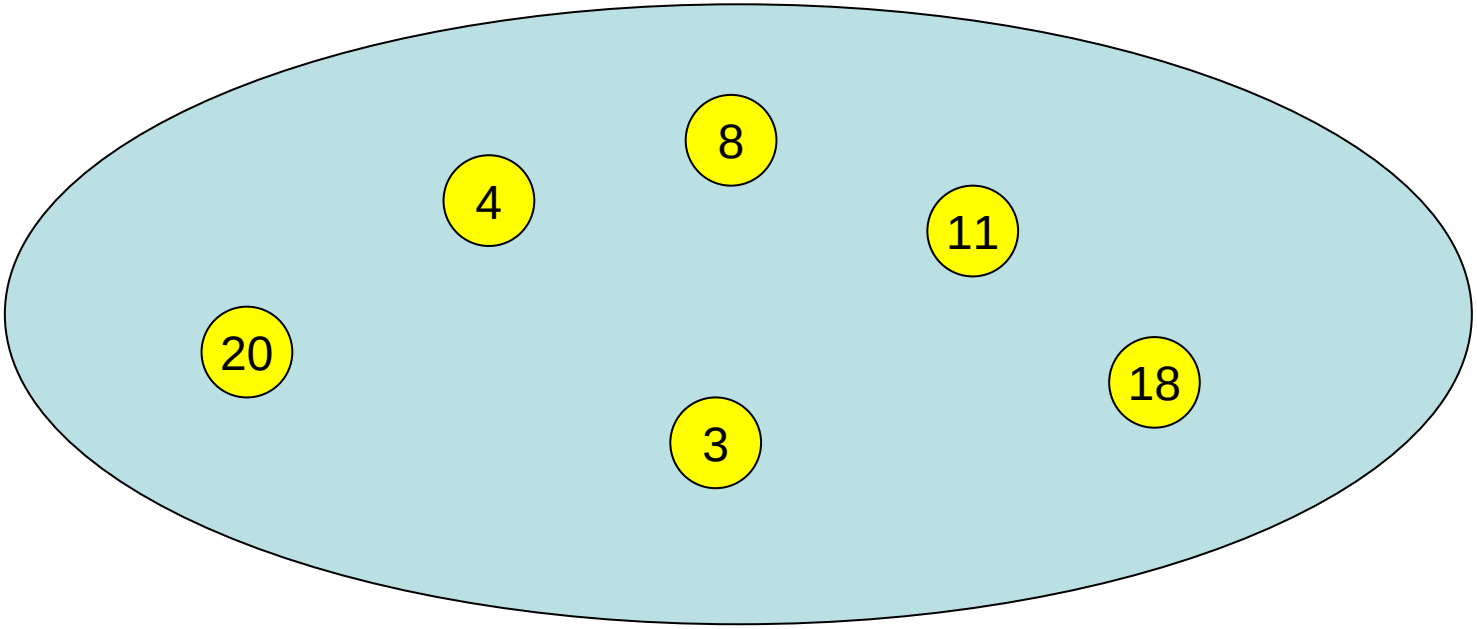
Effiziente Algorithmen und Datenstrukturen I
Kapitel 4: Hashing

Christian Scheideler

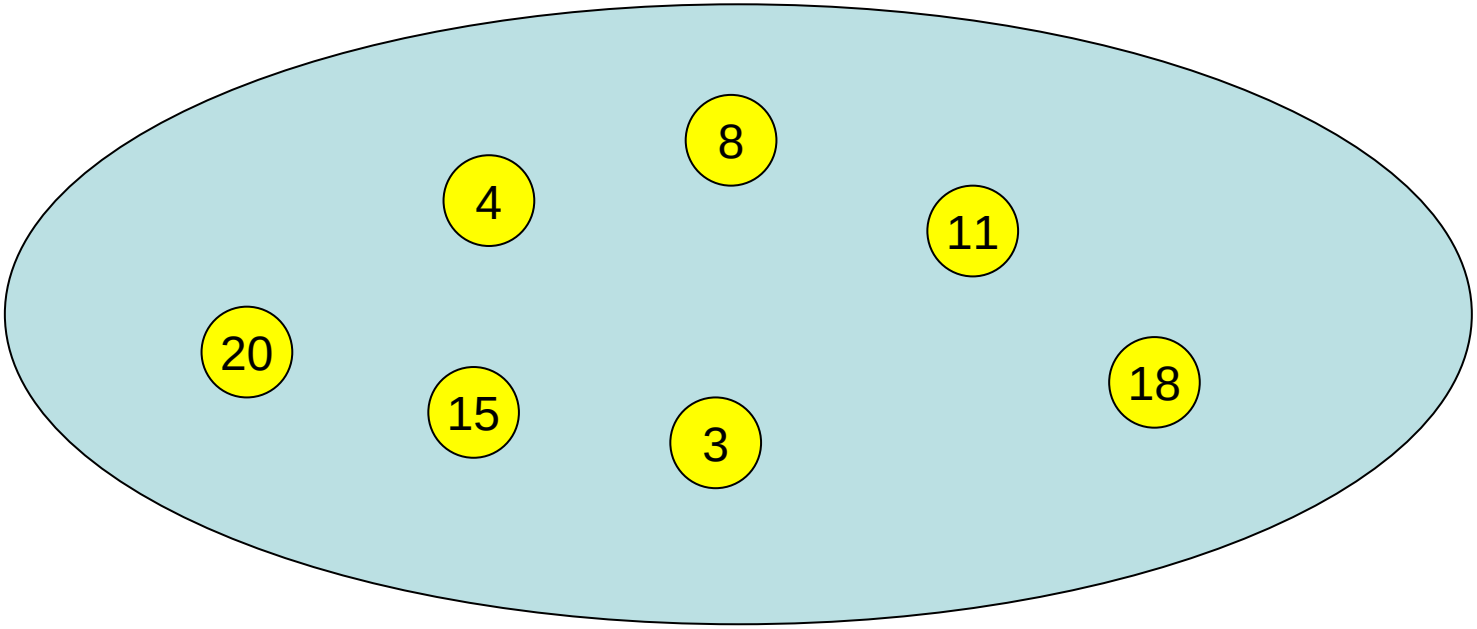
WS 2008

Wörterbuch



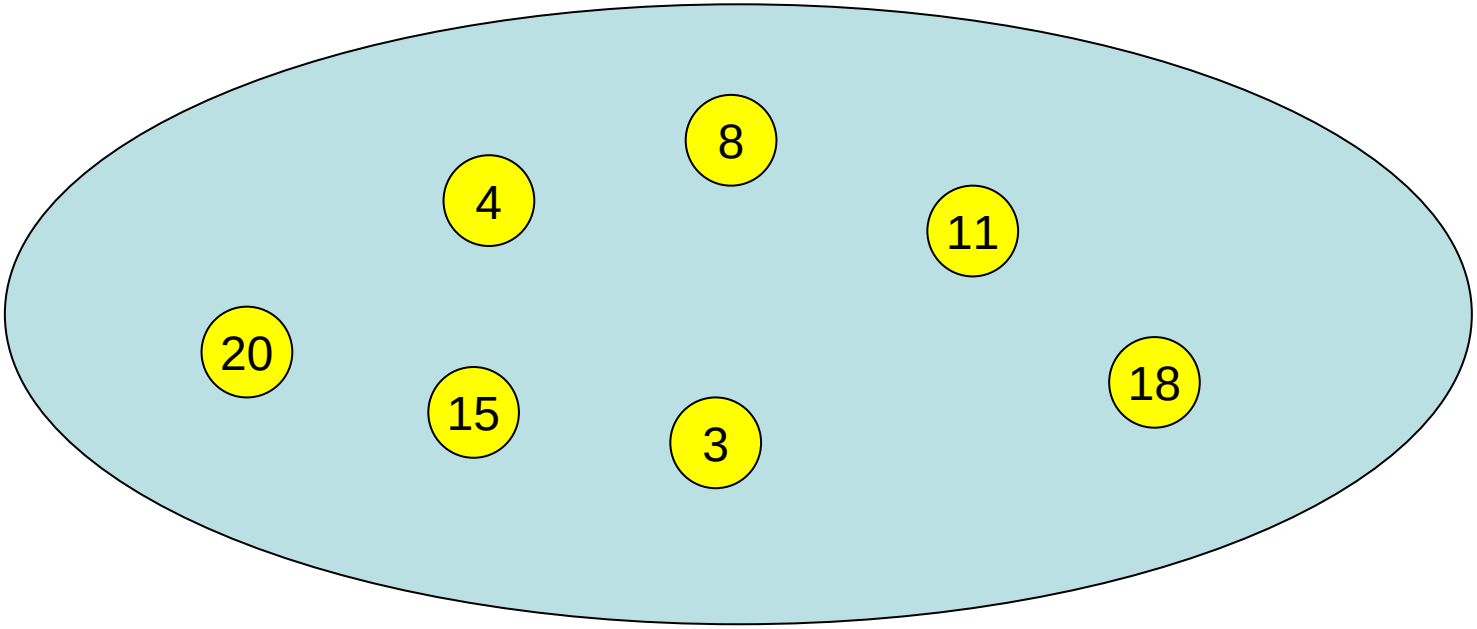
Wörterbuch

insert(15)



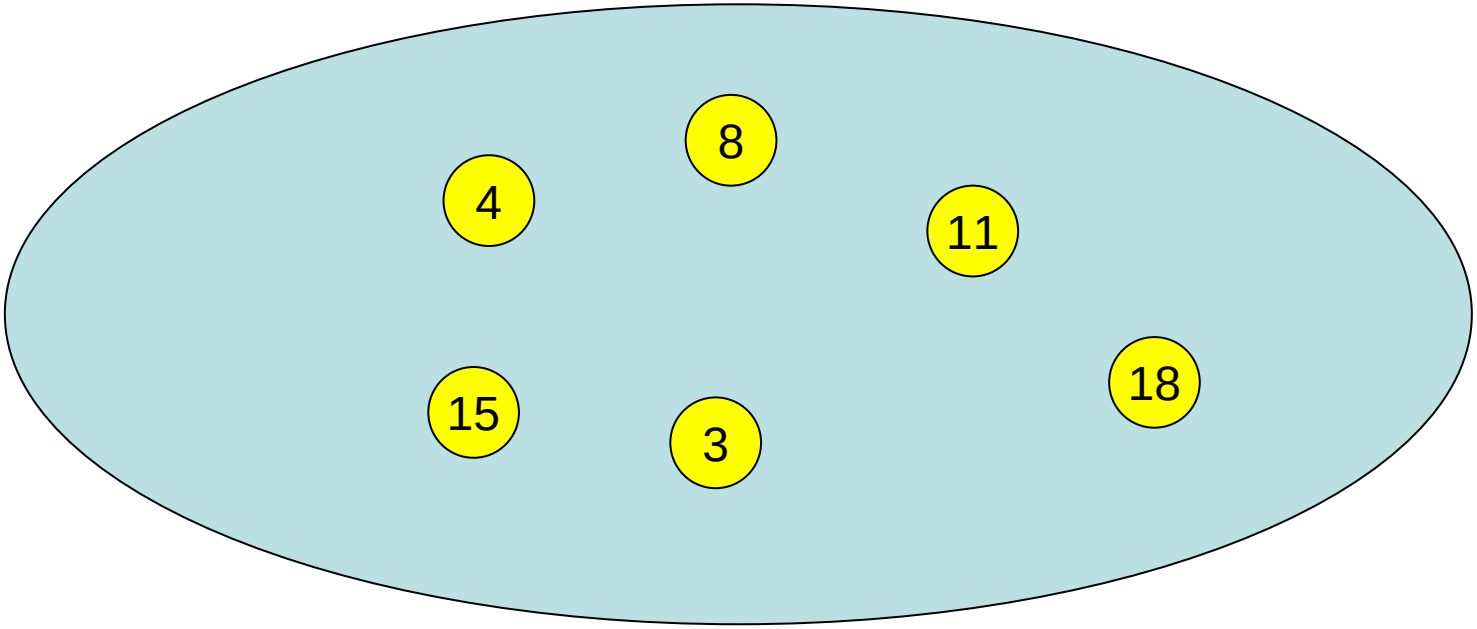
Wörterbuch

delete(20)



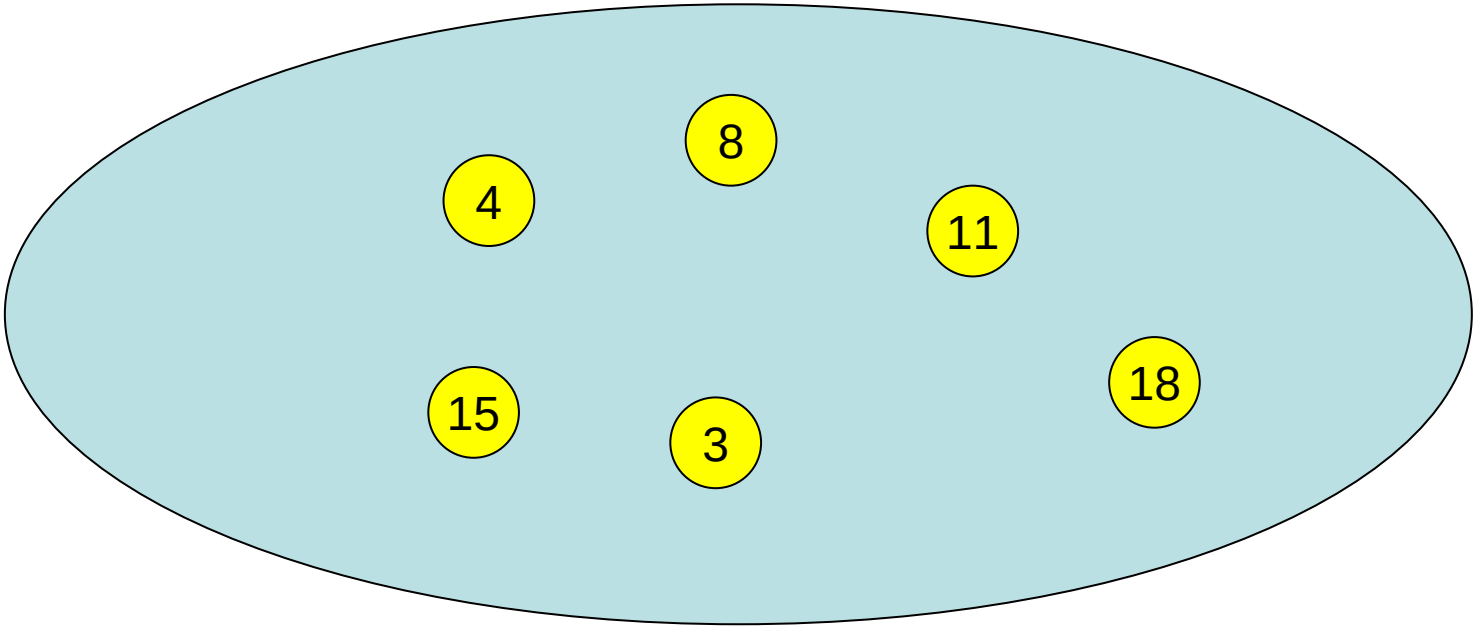
Wörterbuch

lookup(8) ergibt 8



Wörterbuch

lookup(7) ergibt $\perp$



Wörterbuch-Datenstruktur

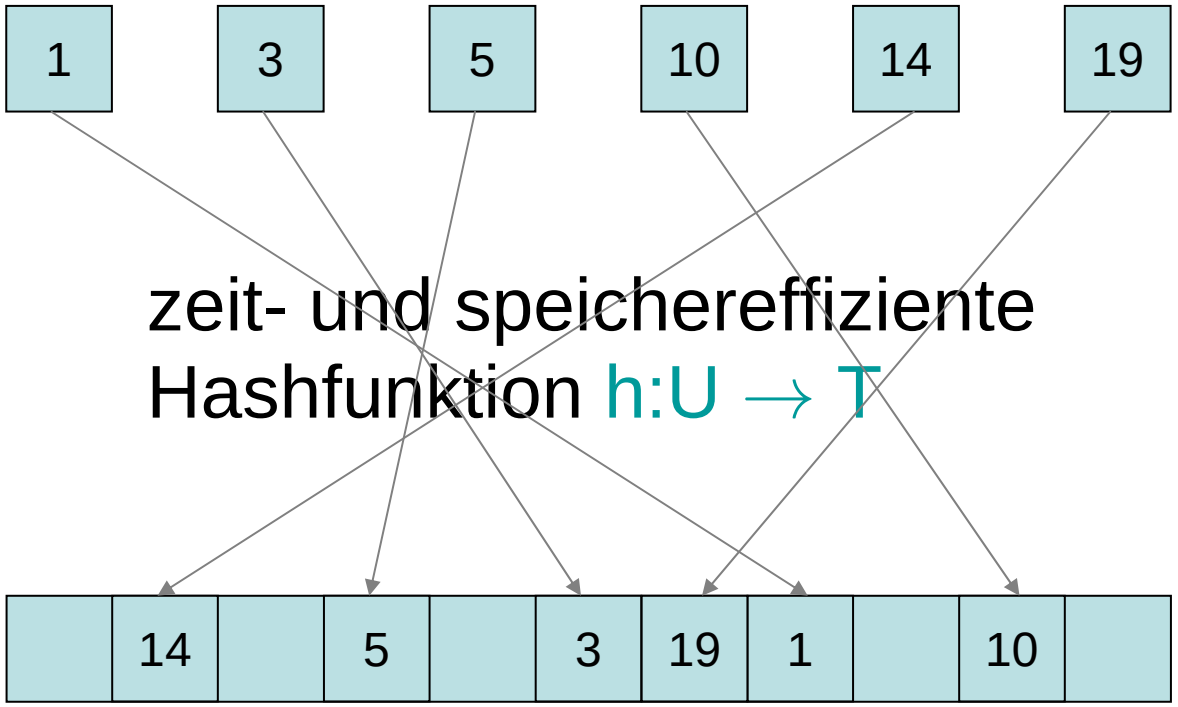
S: Menge von Elementen

Jedes Element e identifiziert über key(e).

Operationen:

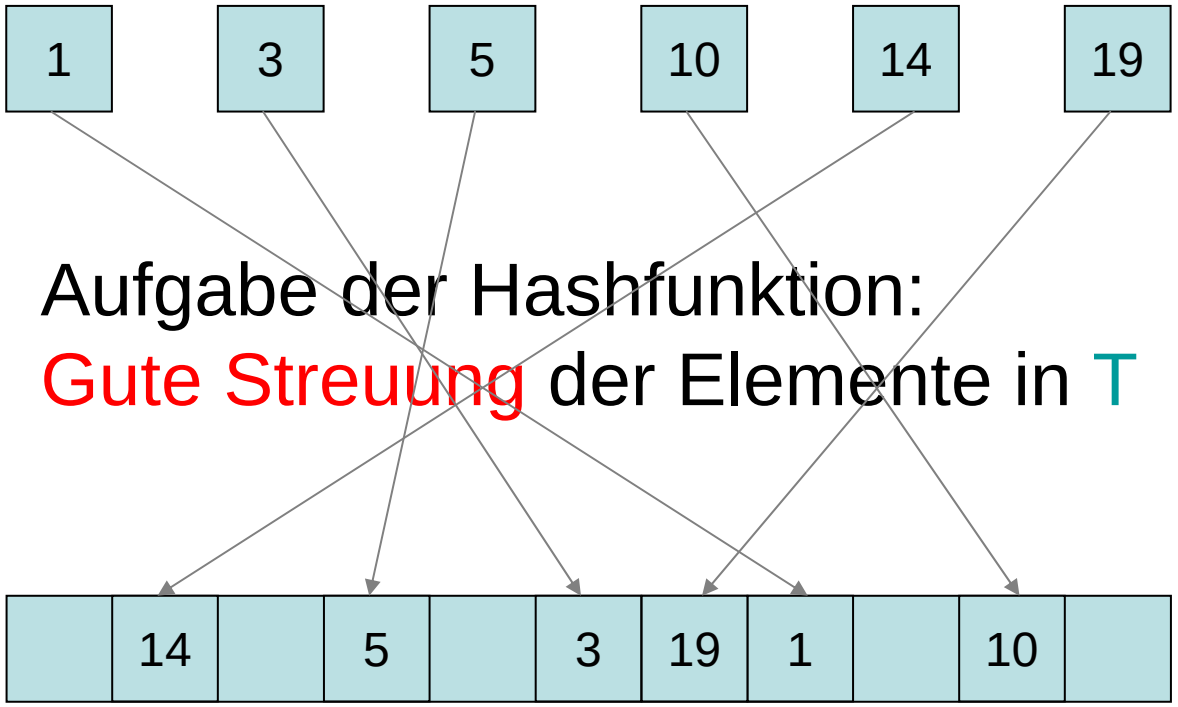
- S.insert(e: Element): $S:=S \cup \{e\}$
- S.delete(k: Key): $S:=S\backslash\{e\}$, wobei e das Element ist mit key(e)=k
- S.lookup(k: Key): Falls es ein $e \in S$ gibt mit key(e)=k, dann gib e aus, sonst gib $\perp$ aus

Hashing



Hashtabelle T

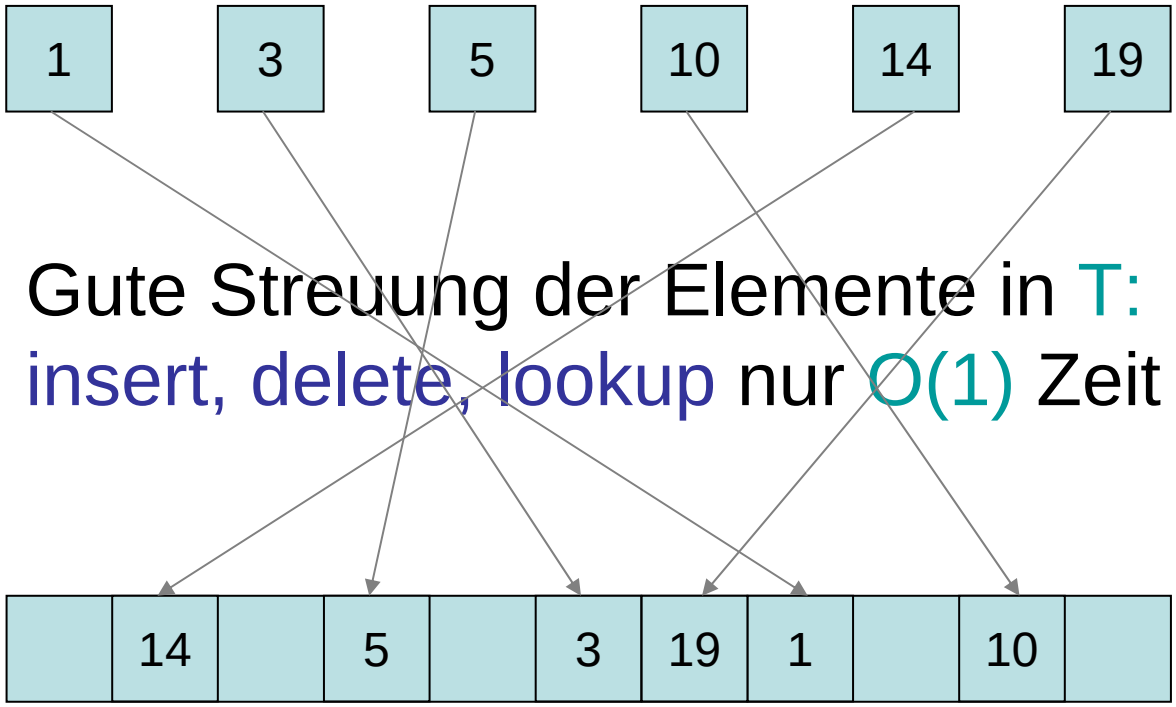
Hashing



Aufgabe der Hashfunktion:
Gute Streuung der Elemente in T

Hashtabelle T

Hashing



Gute Streuung der Elemente in T:
insert, delete, lookup nur O(1) Zeit

Hashtabelle T

Hashing (perfekte Streuung)

Procedure insert(e: Element)
  T[h(key(e))] := e

Procedure delete(k: Key)
  if key(T[h(k)])=k then T[h(k)] := $\perp$

Function lookup(k: Key): Element $\cup$ {$\perp$}
  return T[h(k)]

Hashing

Problem: gute Streuung

Fälle:

- Statisches Wörterbuch: nur lookup
- Dynamisches Wörterbuch: insert, delete und lookup

Übersicht

Klassische Hashverfahren

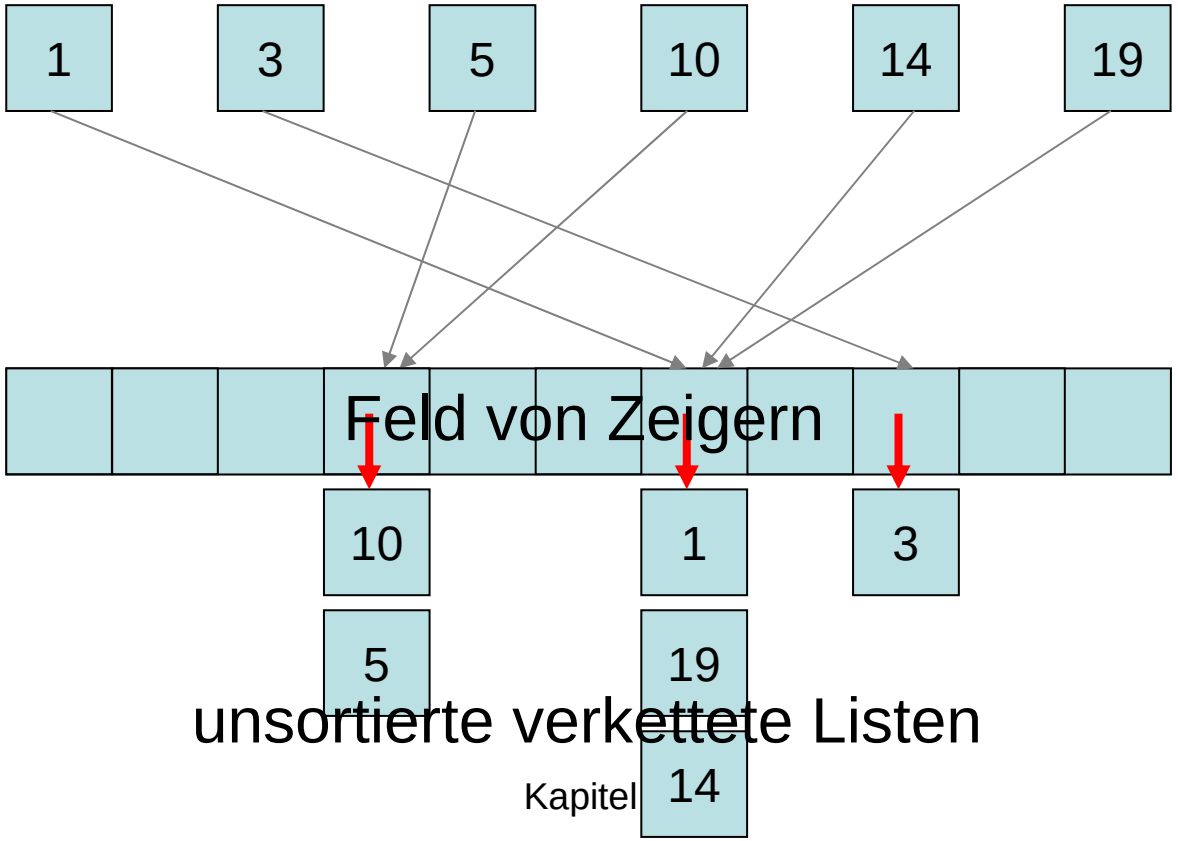
- Hashing mit Chaining
- Hashing mit Linear Probing
- FKS-Hashing

Neuere Hashverfahren

- Cuckoo Hashing
- Trie Hashing
- Konsistentes Hashing und SHARE

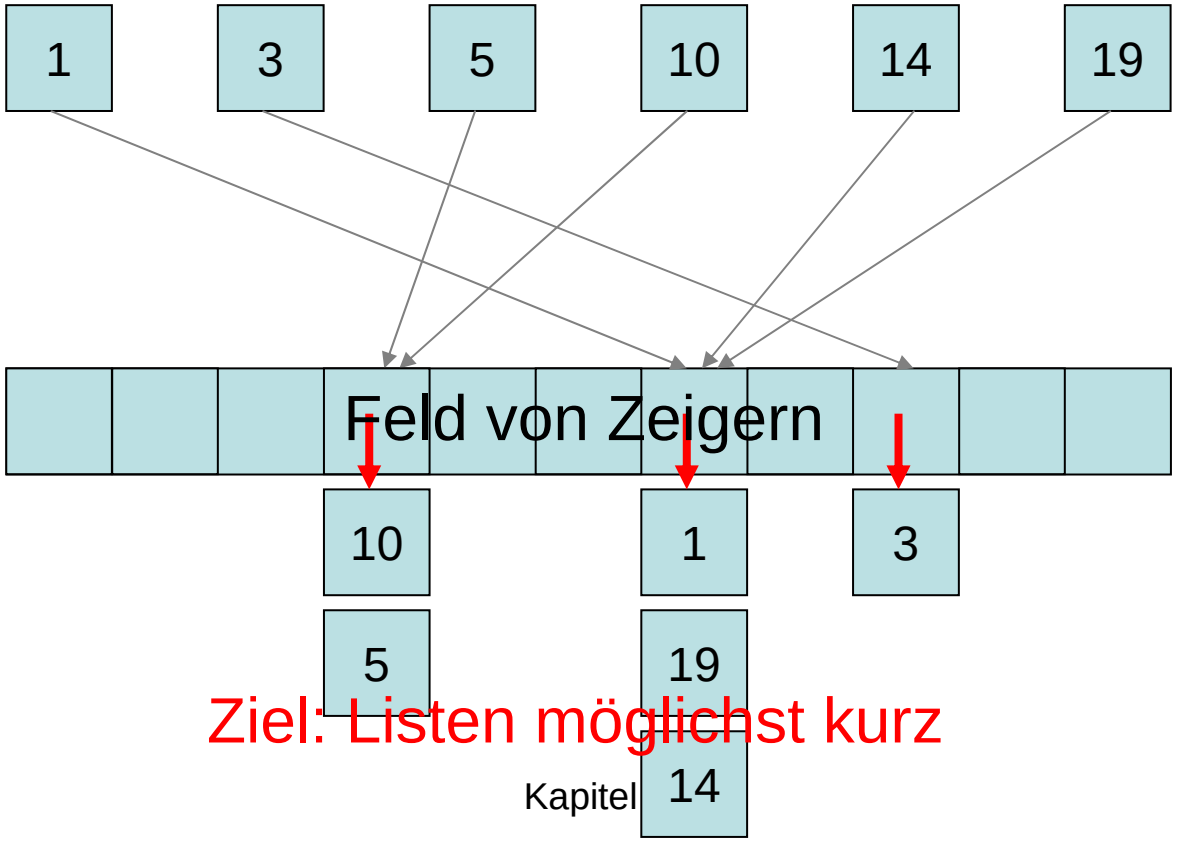
Dynamisches Wörterbuch

Hashing with Chaining:



Dynamisches Wörterbuch

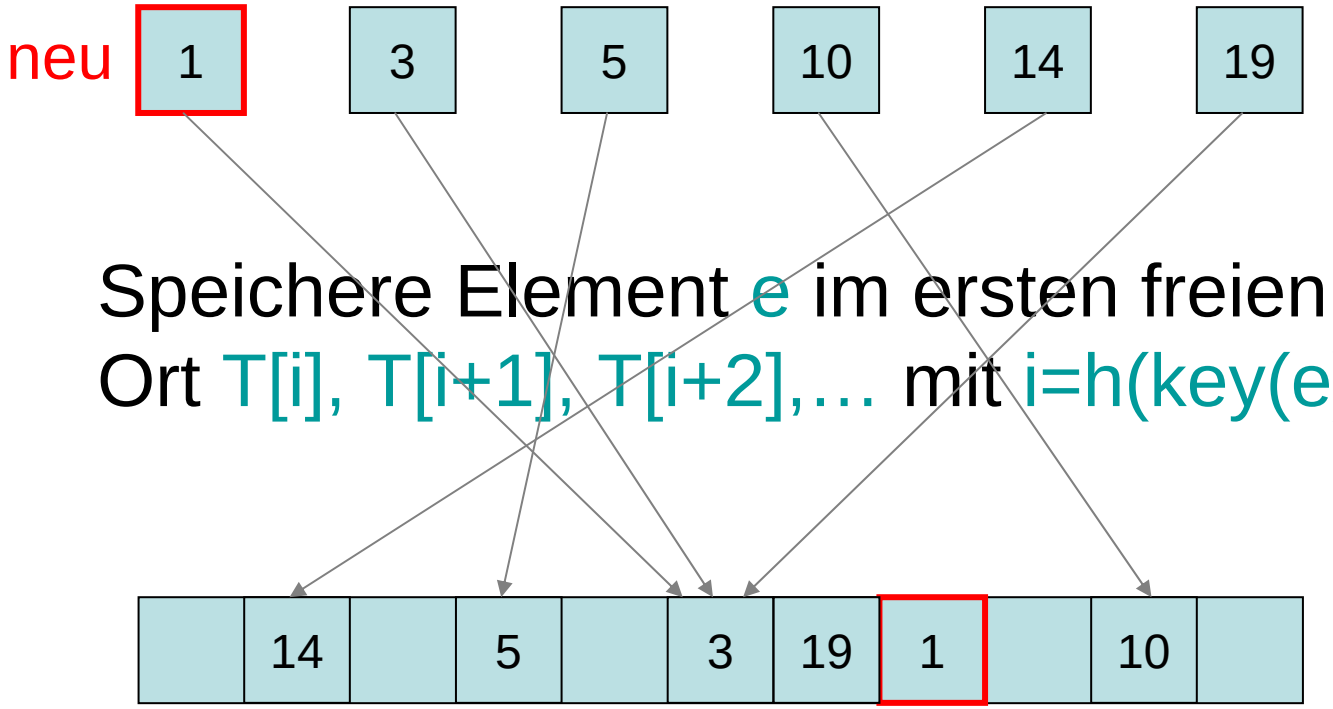
Hashing with Chaining:



Ziel: Listen möglichst kurz

Dynamisches Wörterbuch

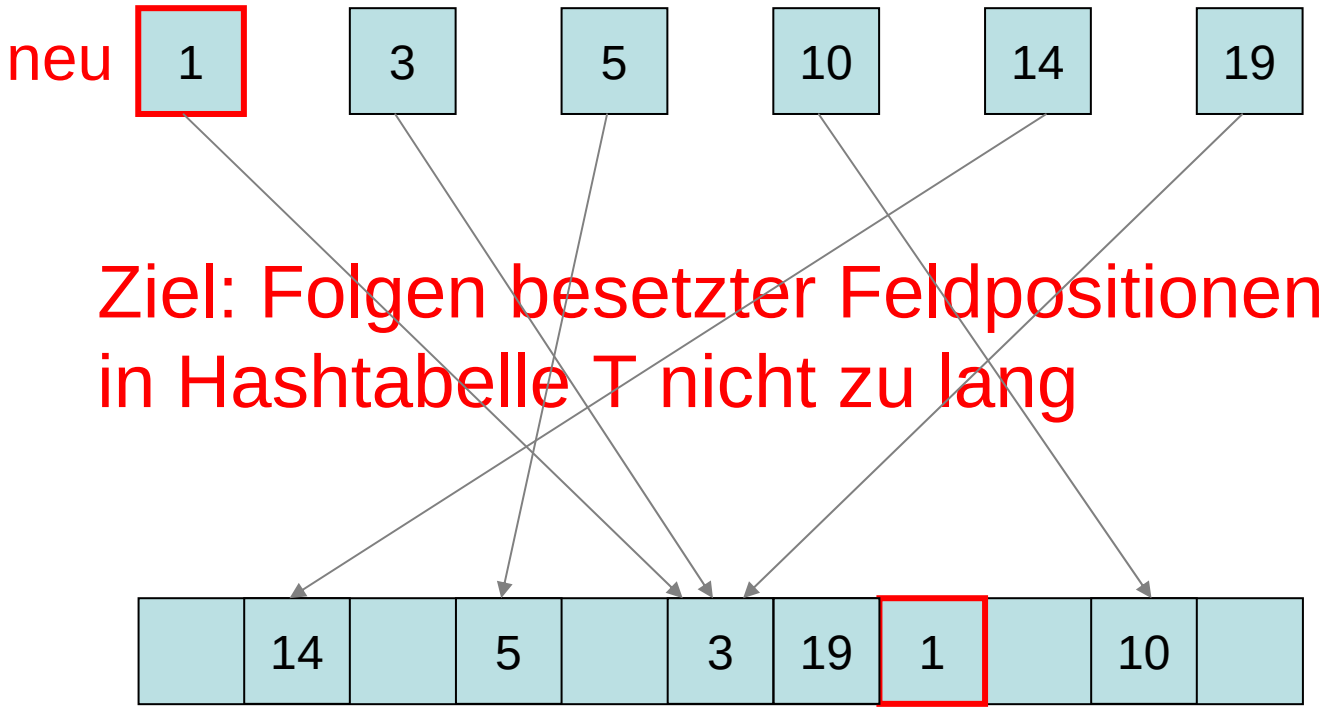
Hashing with Linear Probing:



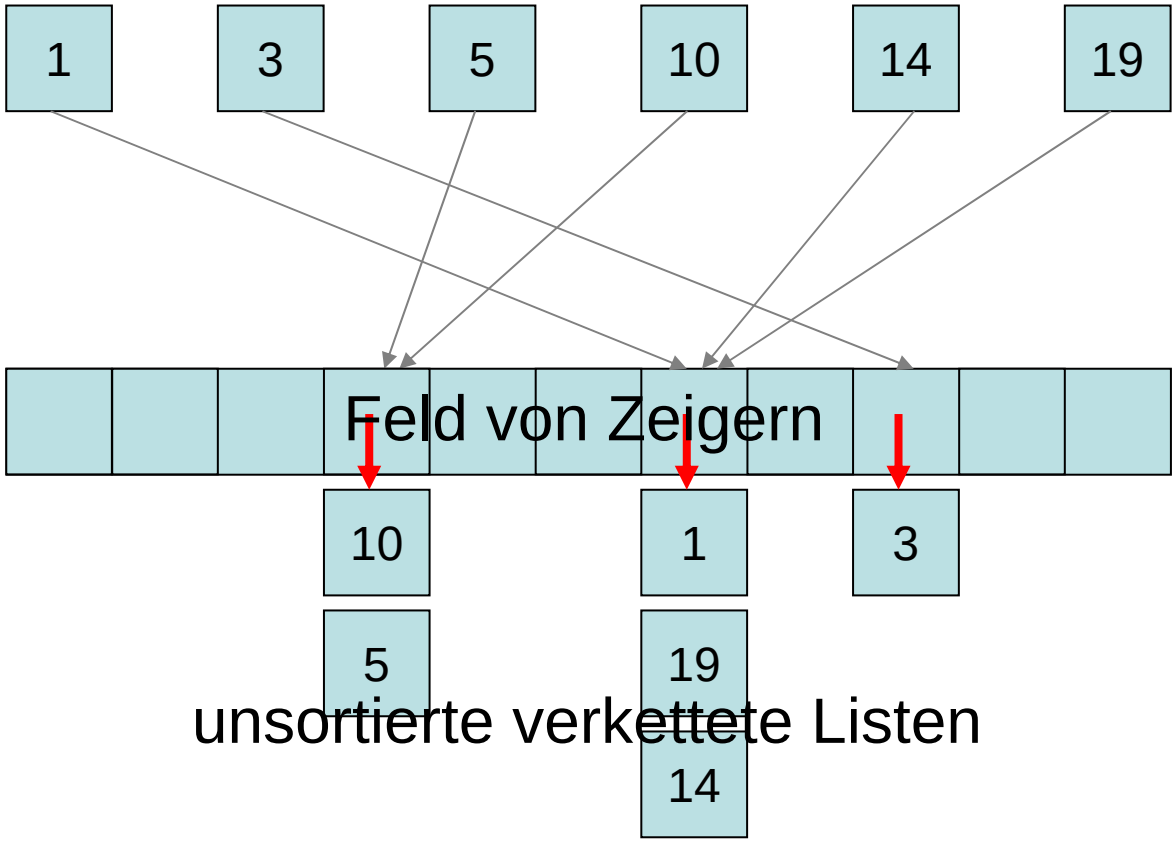
Speichere Element e im ersten freien Ort T[i], T[i+1], T[i+2],… mit i=h(key(e))

Dynamisches Wörterbuch

Hashing with Linear Probing:



Hashing with Chaining



Hashing with Chaining

- T: Array [0..m-1] of List
- List: DS mit Operationen insert, delete, lookup

Procedure insert(e: Element)
  T[h(key(e))].insert(e)

Procedure delete(k: Key)
  T[h(k)].delete(k)

Function lookup(k: Key): Element $\cup$ {$\perp$}
  return T[h(k)].lookup(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 $T[i]$ gleich 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

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

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 $O(c \cdot n/m)$.

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
- $E[X] = \sum_e E[X_e] = \sum_e Pr[X_e=1] \leqslant \sum_e (c/m) = (n-1)c/m$

Universelles Hashing

Betrachte die Familie der Hashfunktionen

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

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

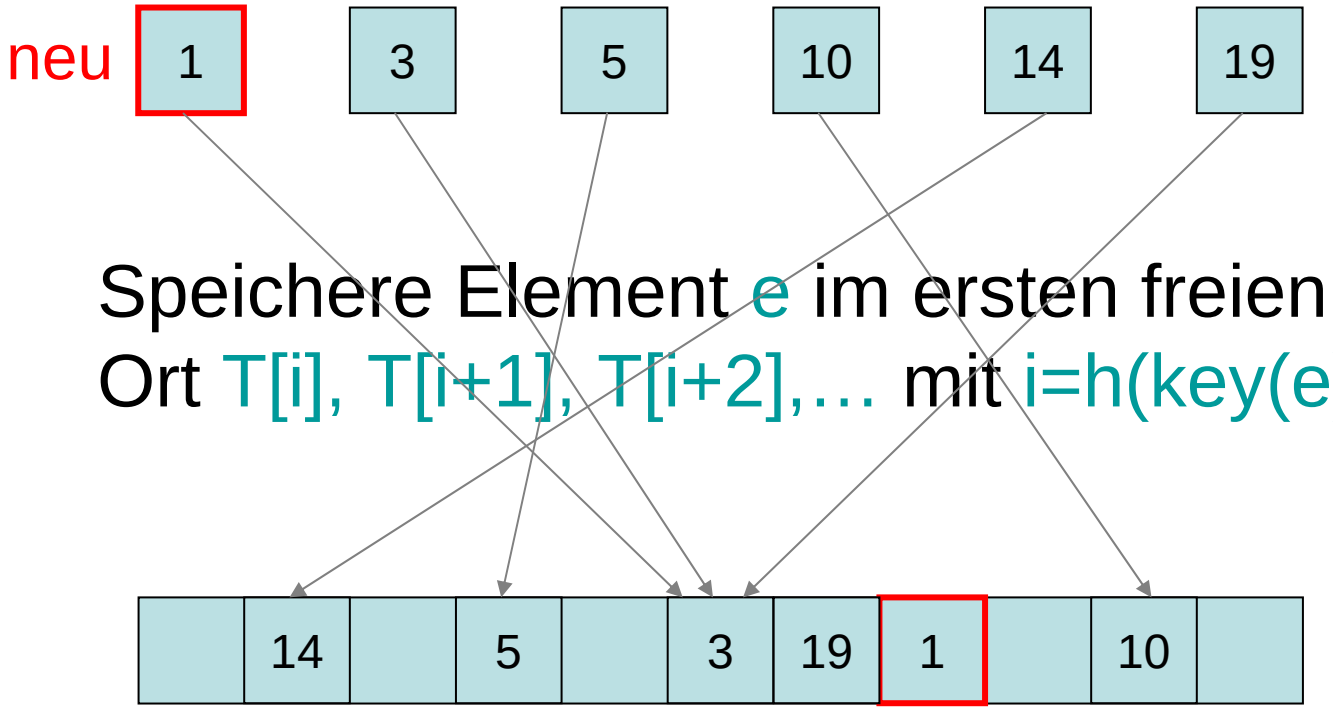
Beweis:

- Betrachte zwei feste Schlüssel $x \neq y$ mit $x=(x_1,\ldots,x_k)$ und $y=(y_1,\ldots,y_k)$
- 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 $a_1,\ldots,a_{j-1}$, $a_{j+1},\ldots,a_k$ genau ein a_j, das diese Gleichung erfüllt.
- Also ist die Anzahl Möglichkeiten für a gleich m^{k-1}
- Daher ist $Pr[h_a(x)=h_a(y)] = m^{k-1}/m^k = 1/m$

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

```
T: Array [0..m-1] of Element     // m>n

Procedure insert(e: Element)
    i := h(key(e))
    while T[i]<>⊥ & key(T[i])<>key(e) do i := i+1 (mod m)
    T[i] := e

Function lookup(k: Key): Element ∪ {⊥}
    i:=h(k)
    while T[i]<>⊥ & key(T[i])<>k do i:=i+1 (mod 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 $\perp$)
3. Stelle die folgende Invariante sicher:
   Für jedes $e \in S$ mit idealer Position $i=h(key(e))$ und aktueller Position j gilt

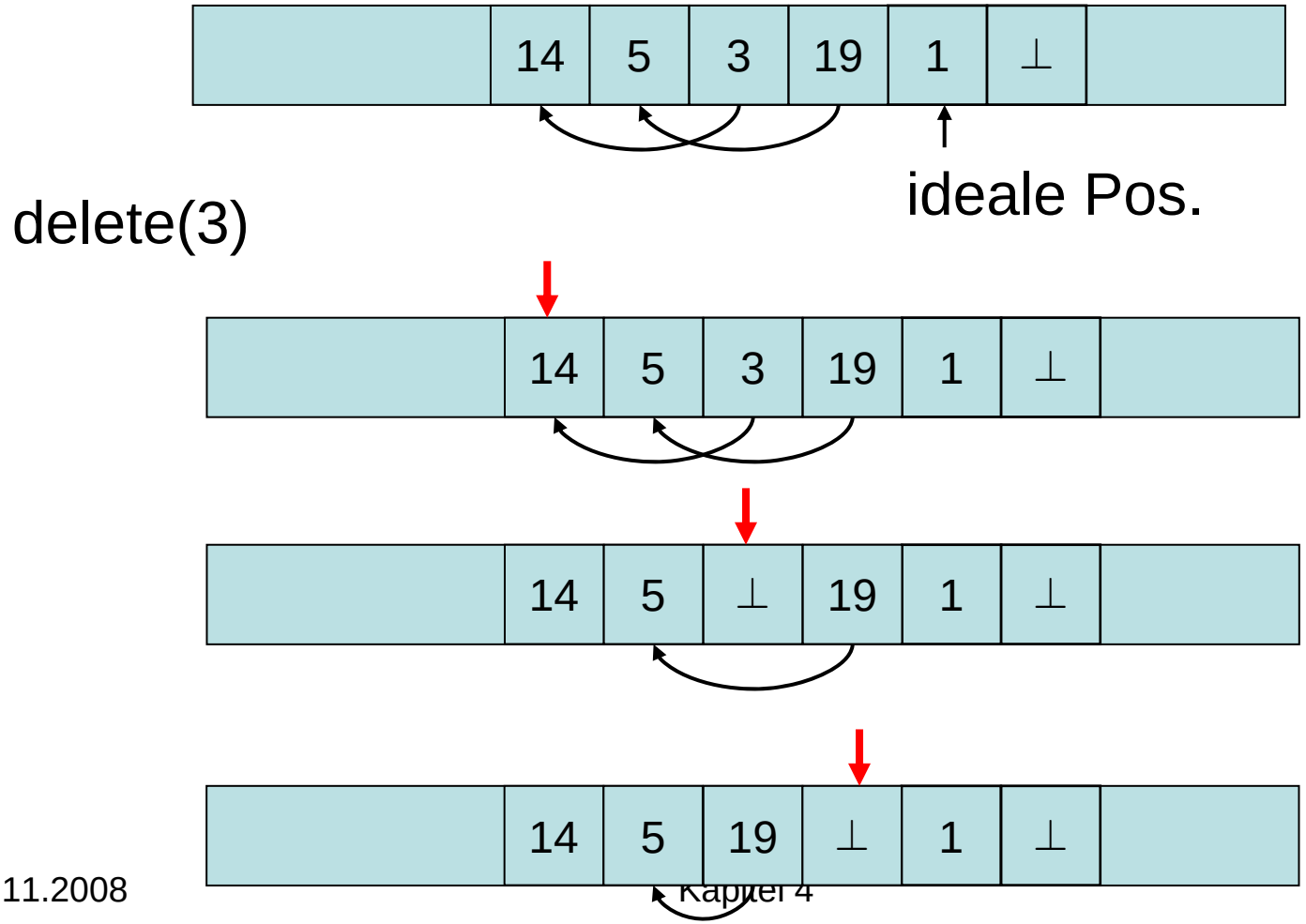
   $T[i], T[i+1] \ldots, T[j]$ sind besetzt

Hashing with Linear Probing

```
Procedure delete(k: Key)
    i := h(k)
    j := i
    empty := 0          // Indicator für T[i]=⊥
    while T[j]<>⊥ do
        if h(key(T[j]))<=i & empty then
            T[i] := T[j]
            T[j] := ⊥
            i := j          // repariere Invariante ab T[i]
        if key(T[j])=k then
            T[j] := ⊥       // entferne Element mit Schlüssel k
              i := j
              empty := 1
        j := j+1 (mod m)
```

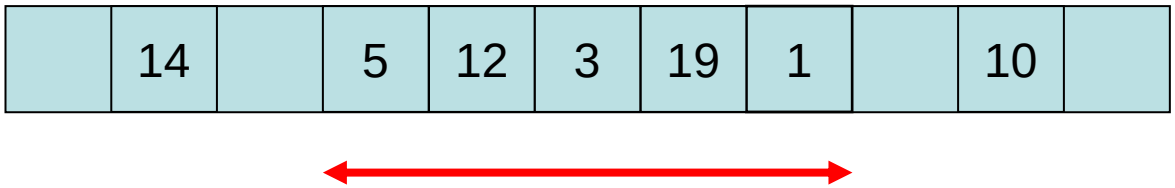
Klappt nur, solange i>h(k). Wie sieht allg. Fall aus?

Beispiel für delete Operation



Hashing with Linear Probing

Lauf:



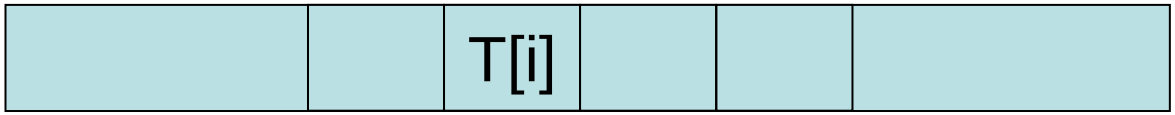
Lauf der Länge 5

Theorem 4.5: 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



Besetzter Lauf der Länge k

Anzahl Möglichkeiten zur Wahl von k Elementen:

$$\binom{n}{k} < (en/k)^k$$

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

Hashing with Linear Probing

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

- p_k := Pr[T[i] in Lauf der Länge k] $< k\,(1/2)^k$
- E[Länge des Laufs über T[i]]
  $= \sum_{k \geqslant 0} k \cdot p_k < \sum_{k \geqslant 0} k^2 (1/2)^k = O(1)$

Also erwartet konstanter Aufwand für Operationen insert, delete und lookup.

Dynamisches Wörterbuch

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

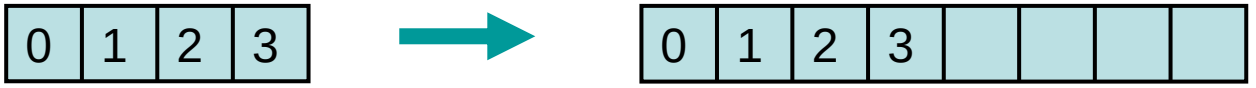
Lösung: Reallokation

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

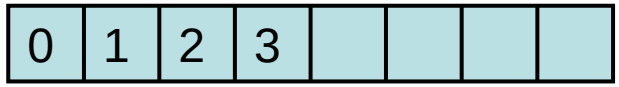
Dynamische Has

Vereinfachung!
(evtl. für Hashtabelle schon ab n>m/2 nötig)

- Tabellenverdopplung (n>m):

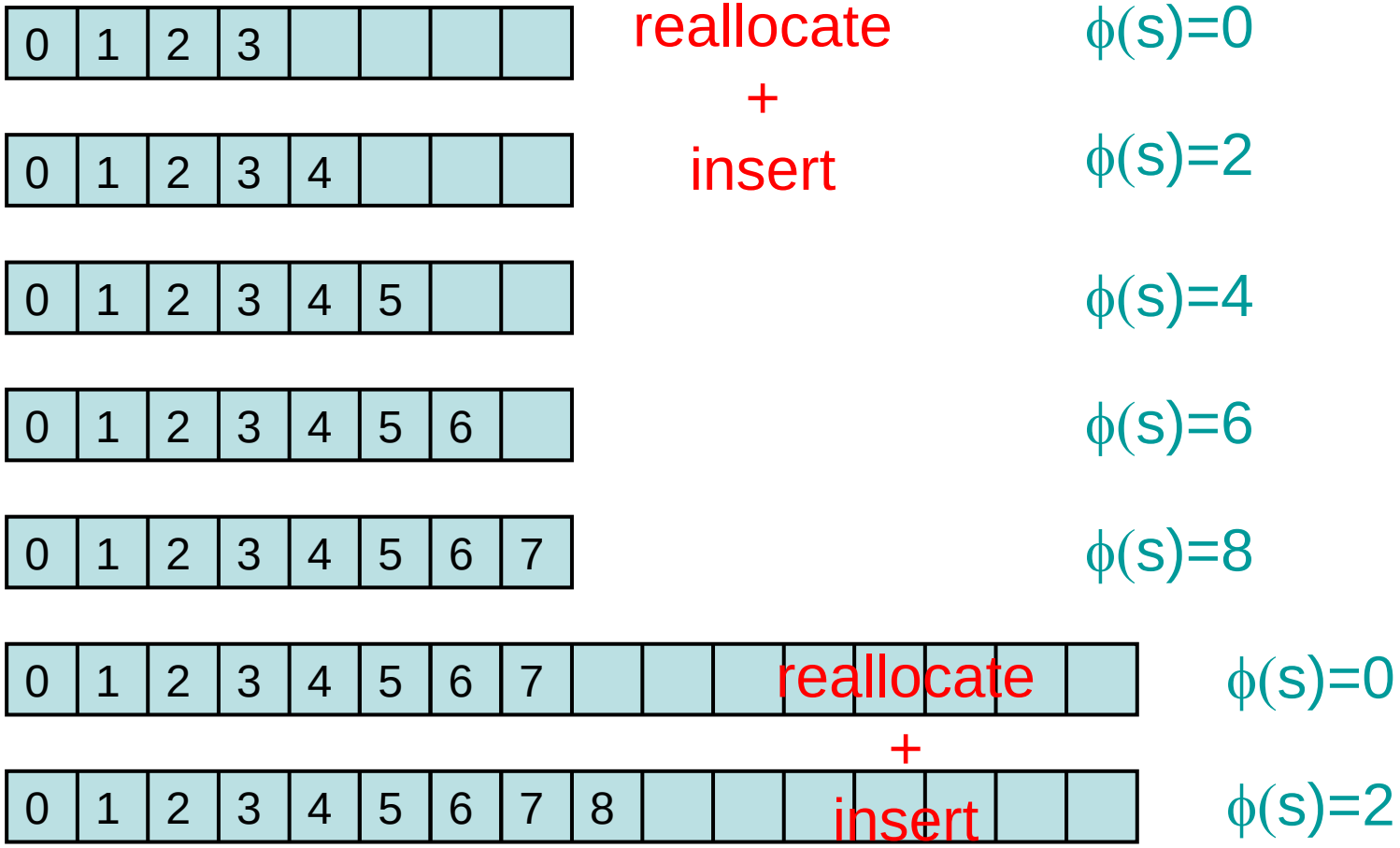


- Tabellenhalbierung (n⩽m/4):

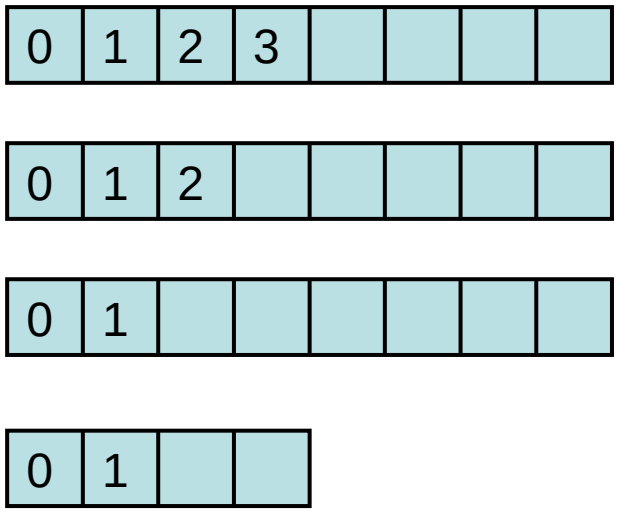
- Von 

  - Nächste Verdopplung: ⩾ n insert Ops
  - Nächste Halbierung: ⩾ n/2 delete Ops

Dynamische Hashtabelle



Dynamische Hashtabelle



delete
+
reallocate

$\phi(s)=0$

$\phi(s)=2$

$\phi(s)=4$

$\phi(s)=0$

Generelle Formel für $\phi(s)$:
(w_s: Feldgröße von s, n_s: Anzahl Einträge)

$$\phi(s) = 2|w_s/2 - n_s|$$

Dynamische Hashtabelle

Generelle Formel für $\phi(s)$:
(w_s: Feldgröße von s, n_s: Anzahl Einträge)

$$\phi(s) = 2|w_s/2 - n_s|$$

Theorem 4.6:
Sei $\Delta\phi = \phi(s') - \phi(s)$ für $s \rightarrow s'$. Für die erwarteten amortisierten Laufzeiten gilt:

- insert: $E[t_{ins}] + \Delta\phi = O(1)$
- delete: $E[t_{del}] + \Delta\phi = O(1)$

Beweis: Übung

Dynamische Hashtabelle

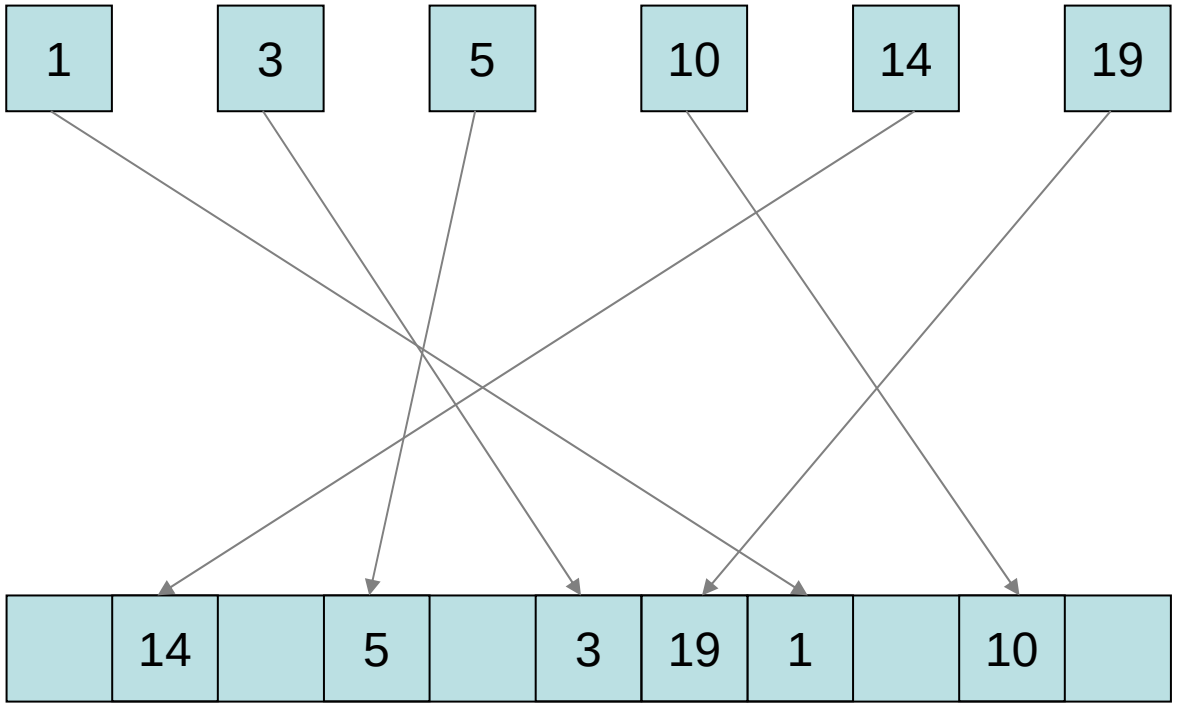
Problem: Tabellengröße m sollte prim sein
(für gute Verteilung der Schlüssel)

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

Statisches Wörterbuch

Ziel: perfekte Hashtabelle

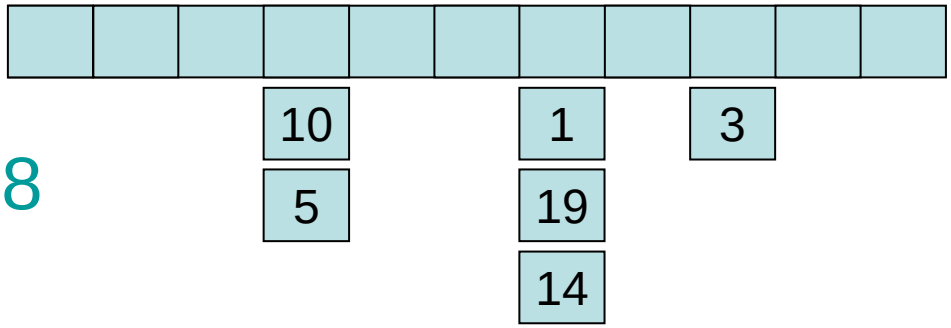


Statisches Wörterbuch

- 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.

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

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



Statisches Wörterbuch

Lemma 4.7: $E[C(h)] \leqslant c \cdot n(n-1)/m$, und für mindestens die Hälfte der $h \in H$ ist $C(h) \leqslant 2c \cdot n(n-1)/m$.

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: $C(h) = \sum_{e \neq e'} X_{e,e'}$
- $E[C(h)] = \sum_{e \neq e'} E[X_{e,e'}] \leqslant \sum_{e \neq e'} (c/m) = c \cdot n(n-1)/m$

Statisches Wörterbuch

…für $\geqslant 1/2$ der $h \in H$ ist $C(h) \leqslant 2c \cdot n(n-1)/m$:

- Nach Markov:

$$\Pr[X \geqslant k \cdot E[X]] \leqslant 1/k$$

- Also gilt

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

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

Statisches Wörterbuch

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

Lemma 4.8: $C(h) = \sum_i b_i^h(b_i^h-1)$.

Beweis:

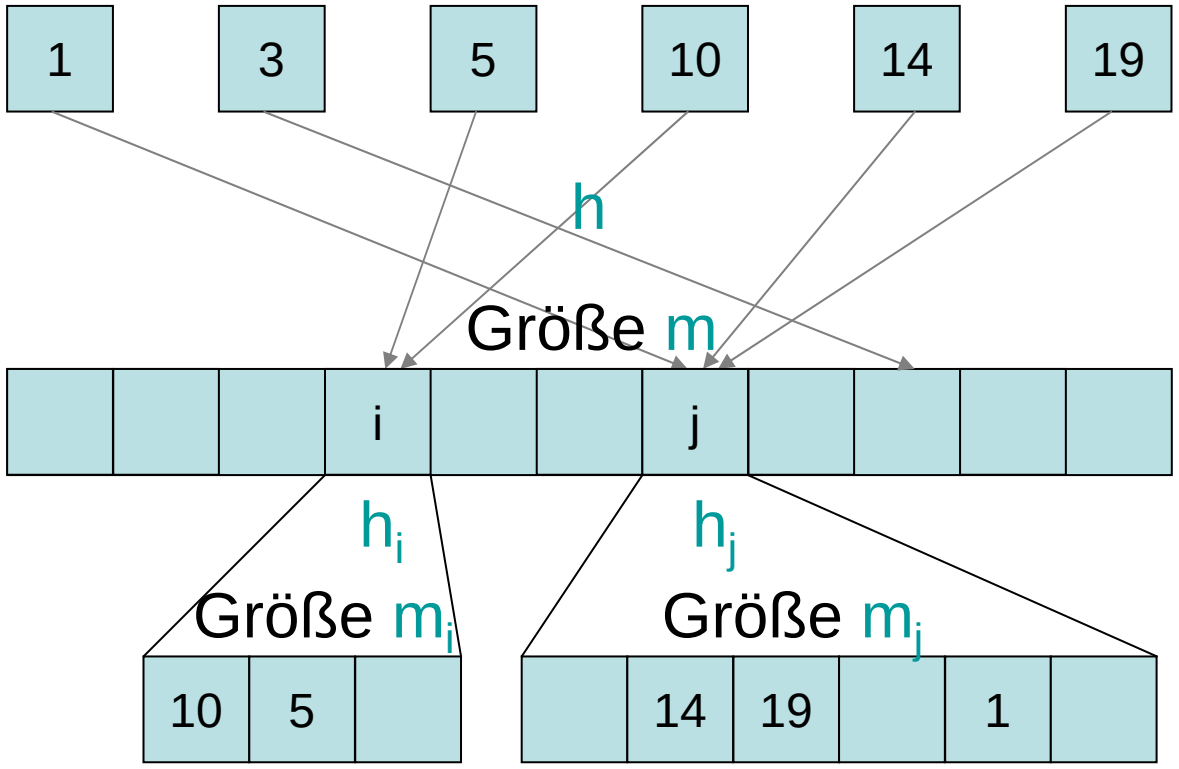
- Zufallsvariable $C_i(h)$: Anzahl Kollisionen in Bucket i
- Es gilt: $C_i(h) = b_i^h(b_i^h-1)$.
- Also ist $C(h) = \sum_i C_i(h) = \sum_i b_i^h(b_i^h-1)$.

Statisches Wörterbuch

Konstruktion der Hashtabelle:

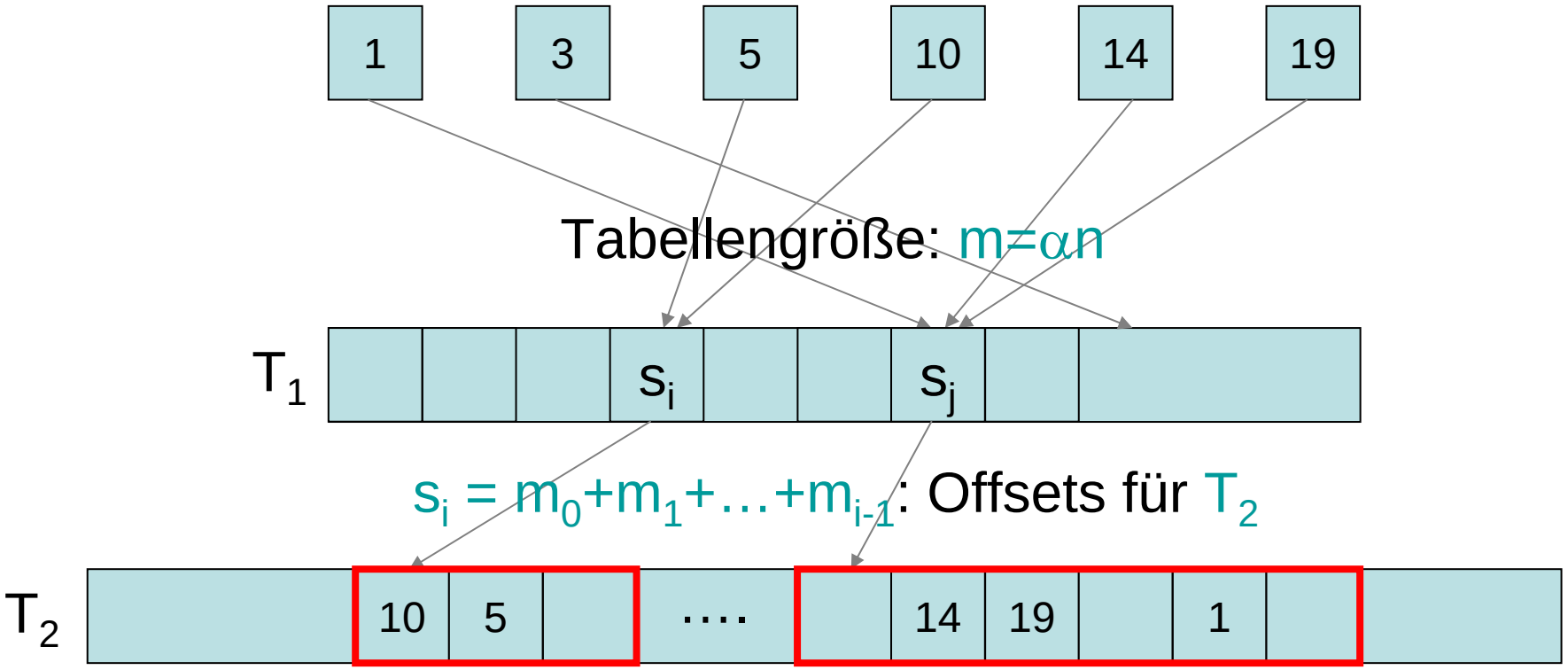
- Wähle eine Funktion $h \in H_{\alpha\eta}$ mit $C(h) \leqslant 2cn(n-1)/(\alpha\, n) = 2cn/\alpha$, α konstant
  (Lemma 4.7: zufällige Wahl mit Wahrscheinlichkeit $\geqslant 1/2$ erfolgreich)
- Für jede Position i sei $m_i = 2c\, b_i^h(b_i^h-1)+1$.
- Wähle eine Funktion $h_i \in H_{m_i}$ mit $C(h_i)<1$
  (Lemma 4.7: zufällige Wahl mit Wahrscheinlichkeit $\geqslant 1/2$ erfolgreich)

Statisches Wörterbuch



Keine Kollisionen in Subtabellen

Statisches Wörterbuch



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

Statisches Wörterbuch

Theorem 4.9: Für jede Menge von n Schlüsseln gibt es eine perfekte Hashfunktion der Größe $\Theta(n)$, die in erwarteter Zeit $\Theta(n)$ konstruiert werden kann.

Sind perfekte Hashfunktionen auch dynamisch konstruierbar??

Übersicht

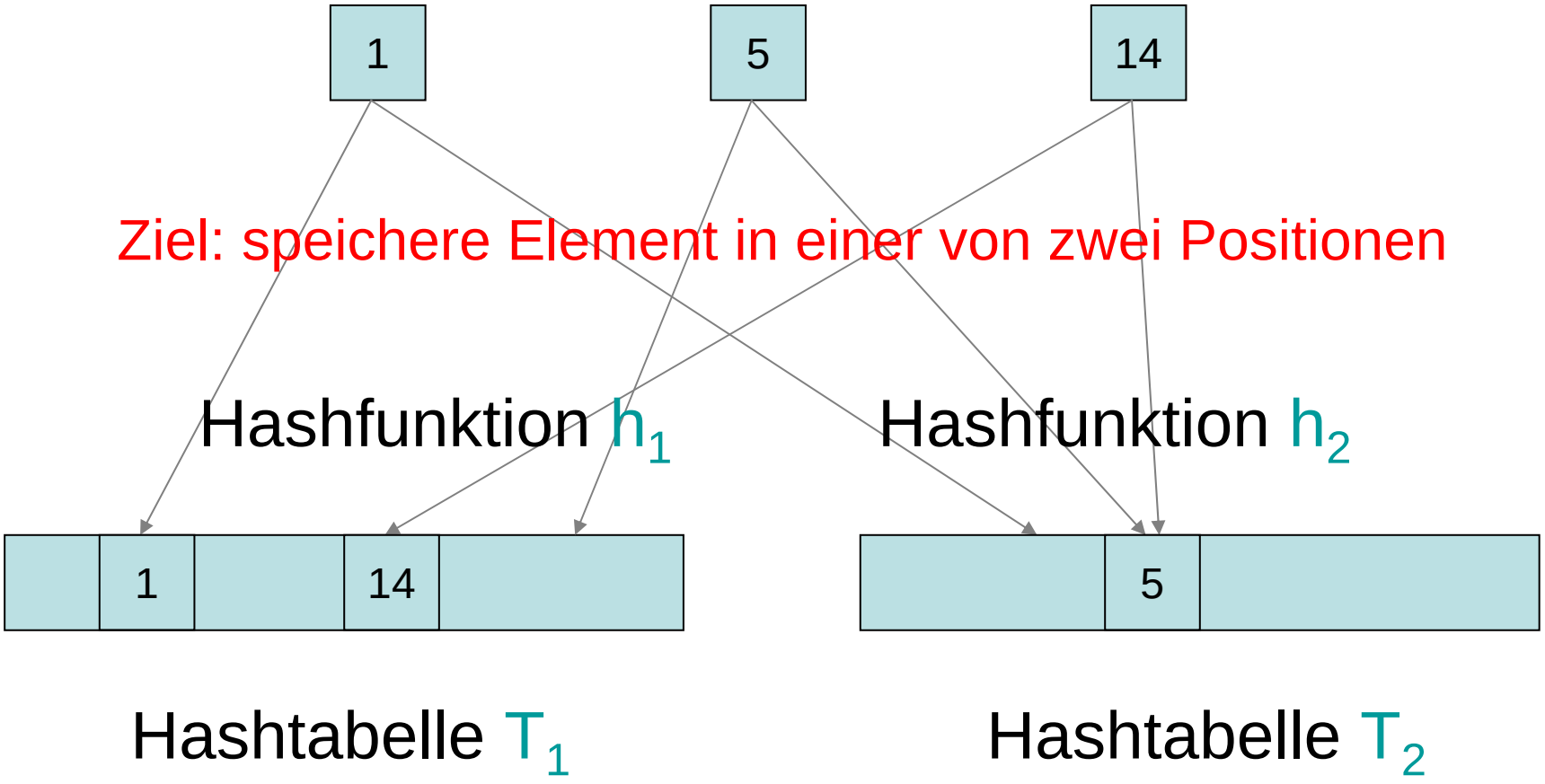
Klassische Hashverfahren

- Hashing mit Chaining
- Hashing mit Linear Probing
- FKS-Hashing

Neuere Hashverfahren

- Cuckoo Hashing
- Trie Hashing
- Konsistentes Hashing und SHARE

Cuckoo Hashing



Cuckoo Hashing

T_1, T_2: Array [0..m-1] of Element

Function lookup(k: Key): Element $\cup \{\perp\}$
  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 $\perp$

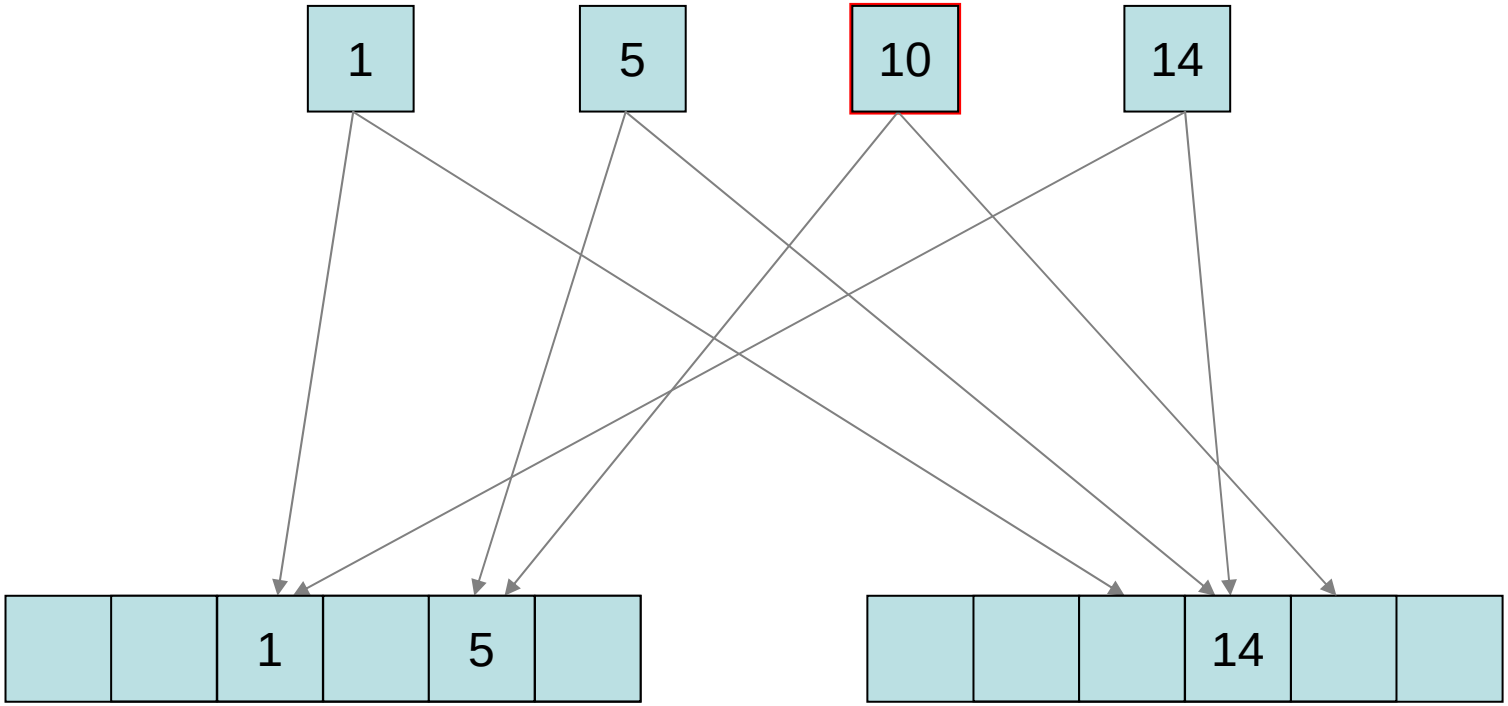
Procedure delete(k: Key)
  if $key(T_1[h_1(k)])=k$ then $T_1[h_1(k)]:=\perp$
  if $key(T_2[h_2(k)])=k$ then $T_2[h_2(k)]:=\perp$

Cuckoo Hashing

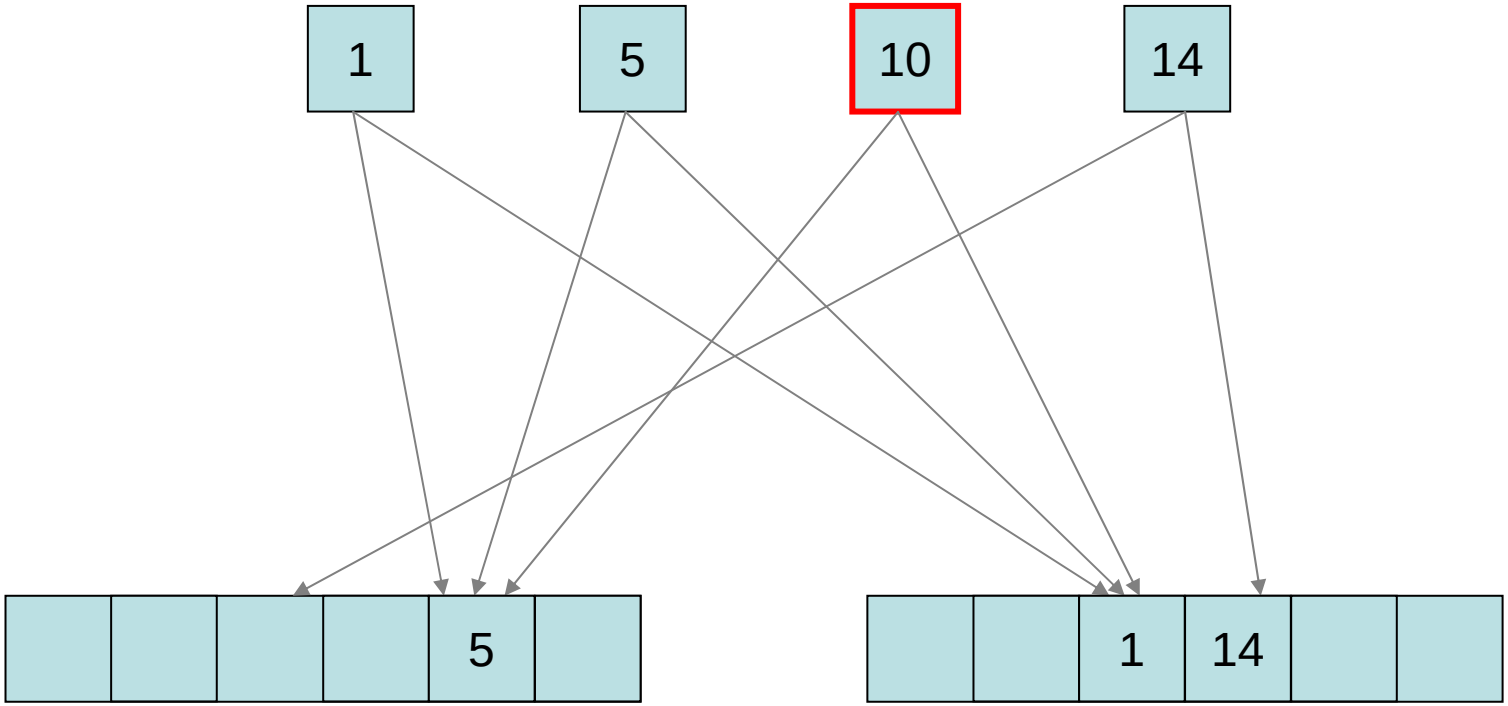
```
Procedure insert(e: Element)
    // key(e) schon in Hashtabelle?
    if key(T1[h1(key(e))])=key(e) then
        T1[h1(key(e))] := e; return
    if key(T2[h2(key(e))])=key(e) then
        T2[h2(key(e))] := e; return
    // nein, dann einfügen
    while true do
        e ↔ T1[h1(key(e))]  // tausche e mit Pos. in T1
        if e=⊥ then return
        e ↔ T2[h2(key(e))]  // tausche e mit Pos. in T2
        if e=⊥ then return
```

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

Cuckoo Hashing



Cuckoo Hashing



Endlosschleife!

Cuckoo Hashing

Laufzeiten:

- lookup, delete: O(1) (worst case!)
- insert: O(Länge der Umlegefolge + evtl. Aufwand für Rehash bei Endlosschleife)

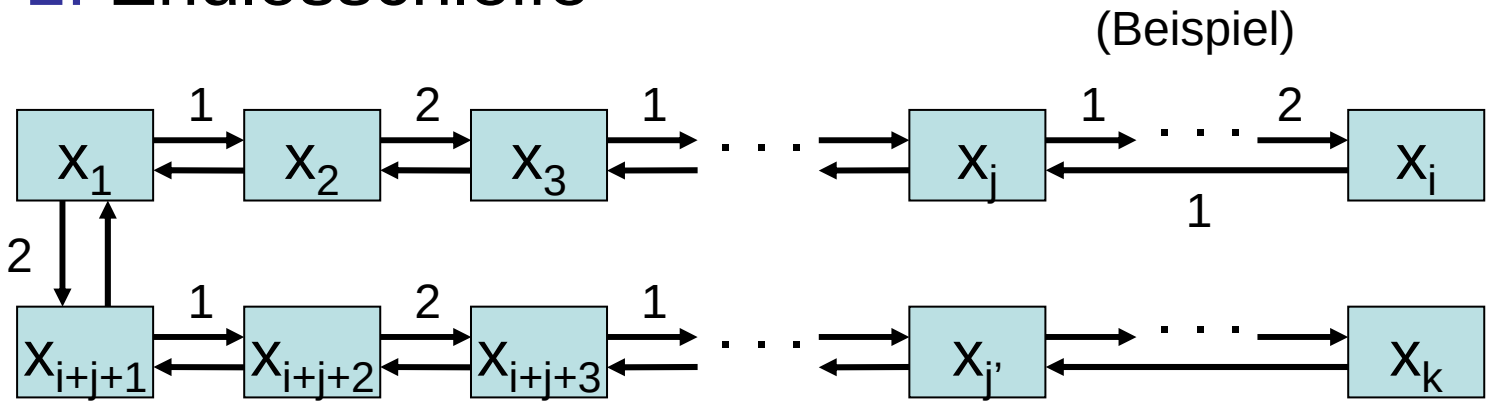
Laufzeit der insert-Operation: 2 Fälle

1. insert läuft nach t Runden in eine Endlosschleife
2. insert terminiert nach t Runden

Im folgenden: n: #Schlüssel, m: Tabellengröße

Cuckoo Hashing

Fall 1: Endlosschleife



- $v \leqslant k$: Anzahl verschiedener Schlüssel
- Anzahl Möglichkeiten, um Endlosschleife zu formen (x_1 fest) ist max.

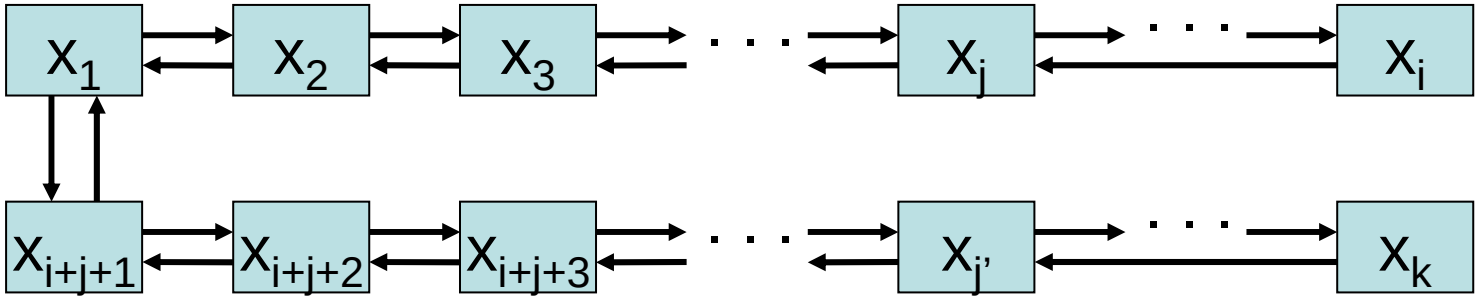
#Komb. (wer,wo)

$v^4\ n^{v-1}\ m^{v-1}$

#Mögl. i, j, k, j'

Cuckoo Hashing

Fall 1: Endlosschleife

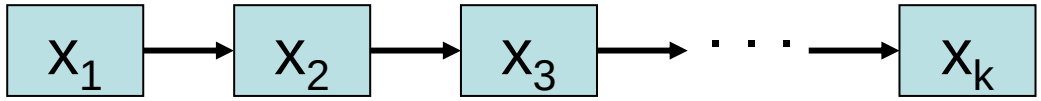


- Anzahl Möglichkeiten: $v^4\, n^{v-1}\, m^{v-1}$
- Wahrscheinlichkeit: $(1/m)^{2v}$ (jeder mit 2 Pos.)
- Falls $m=(1+\varepsilon)n$, $\varepsilon>0$ konstant, dann

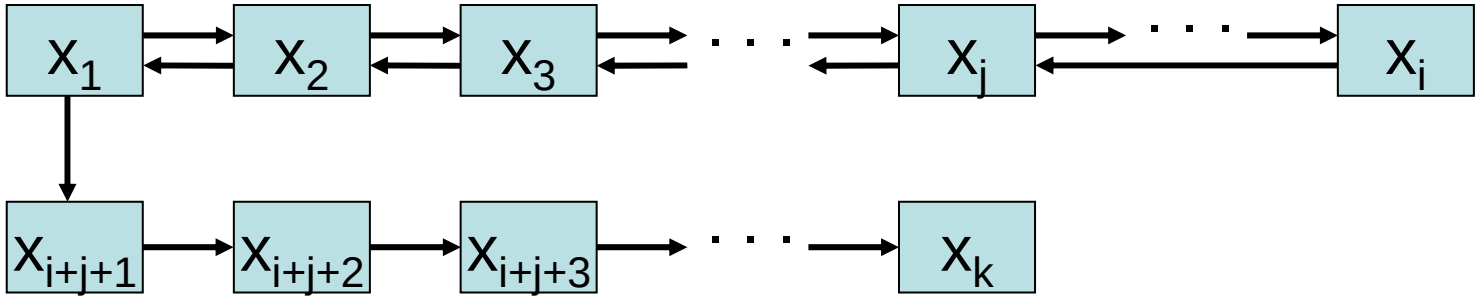
$$\Pr[\text{Fall 1}] = \sum_{v\geqslant 3} v^4\, n^{v-1}\, m^{v-1}\, (1/m)^{2v}$$
$$\leqslant (1/n)^2 \sum_{v\geqslant 3} v^4/(1+\varepsilon)^{2v} = O(1/(\varepsilon^8 n^2))$$

Cuckoo Hashing

Fall 2: Terminierung nach t Runden

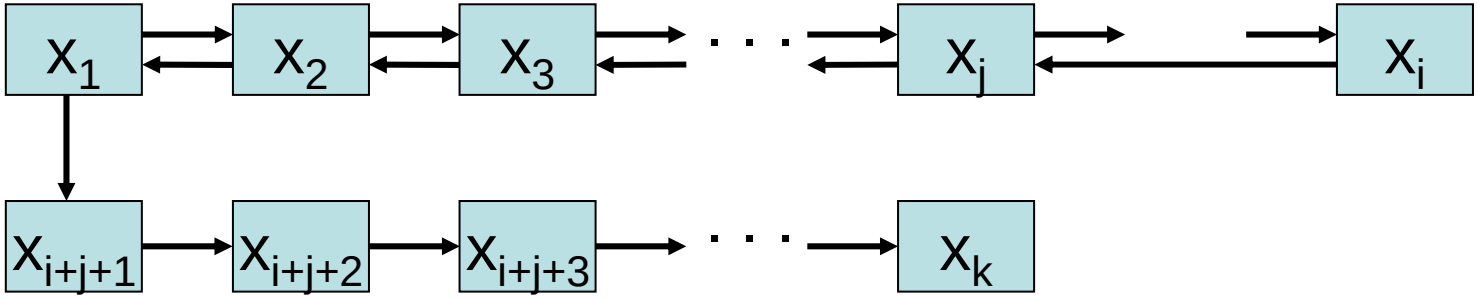


oder



Cuckoo Hashing

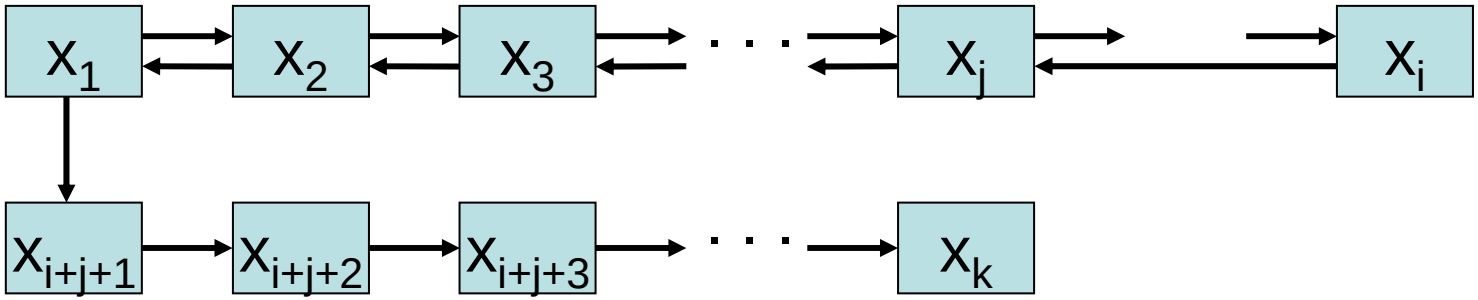
Fall 2: Terminierung nach t Runden



Lemma 4.10: Angenommen, insert besuche v verschiedene Schlüssel. Dann enthält endweder $x_1,\ldots,x_i$ oder $x_{i+j},\ldots,x_k$ mindestens $v/3$ verschiedene Schlüssel

Cuckoo Hashing

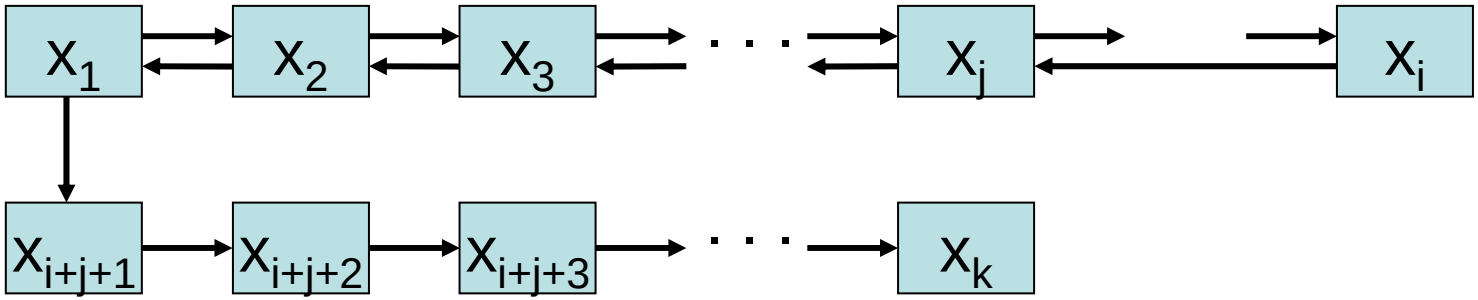
Fall 2: Terminierung



- Sei $x'_1,\ldots,x'_i$ Folge $v/3$ verschiedener Schlüssel ($x'_1=x_1$)
- Geg. x_1, so $n^{v/3-1}$ Möglichkeiten für $x'_2,\ldots,x'_i$
- Pr[Kette $x'_1,\ldots,x'_i$ gültig] $\leqslant 2\,(1/m)^{v/3-1}$
- Pr[Kette mit v versch. Schlüsseln] $\leqslant 2(n/m)^{v/3-1}$

Cuckoo Hashing

Fall 2: Terminierung



E[#Verschiebungen von insert]

$= \sum_{t \geqslant 1} t \cdot \Pr[\ \#\text{Verschiebungen}=t]$

$= O(\ \sum_{v \geqslant 1} v \cdot \Pr[\ \#\text{versch. Schlüssel}=v])$

$= O(\ \sum_{v \geqslant 1} v \cdot 2(1/(1+\varepsilon))^{v/3-1}\ ) = O(1+1/\varepsilon)$

Cuckoo Hashing

Rehash benötigt $O(n)$ erwartete Zeit.

Konsequenz: Die erwartete Laufzeit von insert ist $O(1+1/\varepsilon + n \cdot 1/(\varepsilon^8 n^2)) = O(1+1/\varepsilon)$

Problem: Wahrscheinlichkeit für Endlos-schleife recht hoch!

Lösung: verwende Notfallcache

Cuckoo Hashing mit Cache

T_1, T_2: Array [0..m-1] of Element

C: Array [0..c-1] of Element (c konstant)

Function lookup(k: Key): Element $\cup$ {$\perp$}
  if $\exists$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 $\perp$

Cuckoo Hashing mit Cache

```
Procedure delete(k: Key)
   if ∃i: key(C[i])=k then C[i]:=⊥; return
   if key(T1[h1(k)])=k then
      // lösche k aus T1 und ersetze es evtl. aus C
      p:=h1(k); T1[p]:=⊥
      if ∃i: h1(key(C[i]))=p then T1[p]:=C[i]; C[i]:=⊥
   if key(T2[h2(k)])=k then
      // lösche k aus T2 und ersetze es evtl. aus C
      p:=h2(k); T2[p]:=⊥
       if ∃i: h2(key(C[i]))=p then T2[p]:=C[i]; C[i]:=⊥
```

Cuckoo Hashing mit Cache

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

Cuckoo Hashing mit Cache

Theorem 4.11: Die Wahrscheinlichkeit, dass mehr als c Elemente zu einem Zeitpunkt in C sind, ist $O(1/n^c)$.

Beweis: aufwendig, hier nicht behandelt

Problem: Mit Hashing nur exakte Suche machbar.

Ist Hashing erweiterbar auf komplexere Suchoperationen??

Übersicht

Klassische Hashverfahren

- Hashing mit Chaining
- Hashing mit Linear Probing
- FKS-Hashing

Neuere Hashverfahren

- Cuckoo Hashing
- Trie Hashing
- Konsistentes Hashing und SHARE

Präfix Suche

- Alle Schlüssel kodiert als binäre Folgen $\{0,1\}^W$
- Präfix eines Schlüssels $x \in \{0,1\}^W$: eine beliebige Teilfolge von x, die mit dem ersten Bit von x startet (z.B. ist 101 ein Präfix von 10110100)

Problem: finde für einen Schlüssel $x \in \{0,1\}^W$ einen Schlüssel $y \in S$ mit größtem gemeinsamen Präfix

Lösung: Trie Hashing

Trie

Ein Trie ist ein Suchbaum über einem Alphabet Σ, der die folgenden Eigenschaften erfüllt:

- Jede Baumkante hat einen Label $c \in \Sigma$
- Jeder Schlüssel $x \in \Sigma^k$ ist von der Wurzel des Tries über den eindeutigen Pfad der Länge k zu erreichen, dessen Kantenlabel zusammen x ergeben.

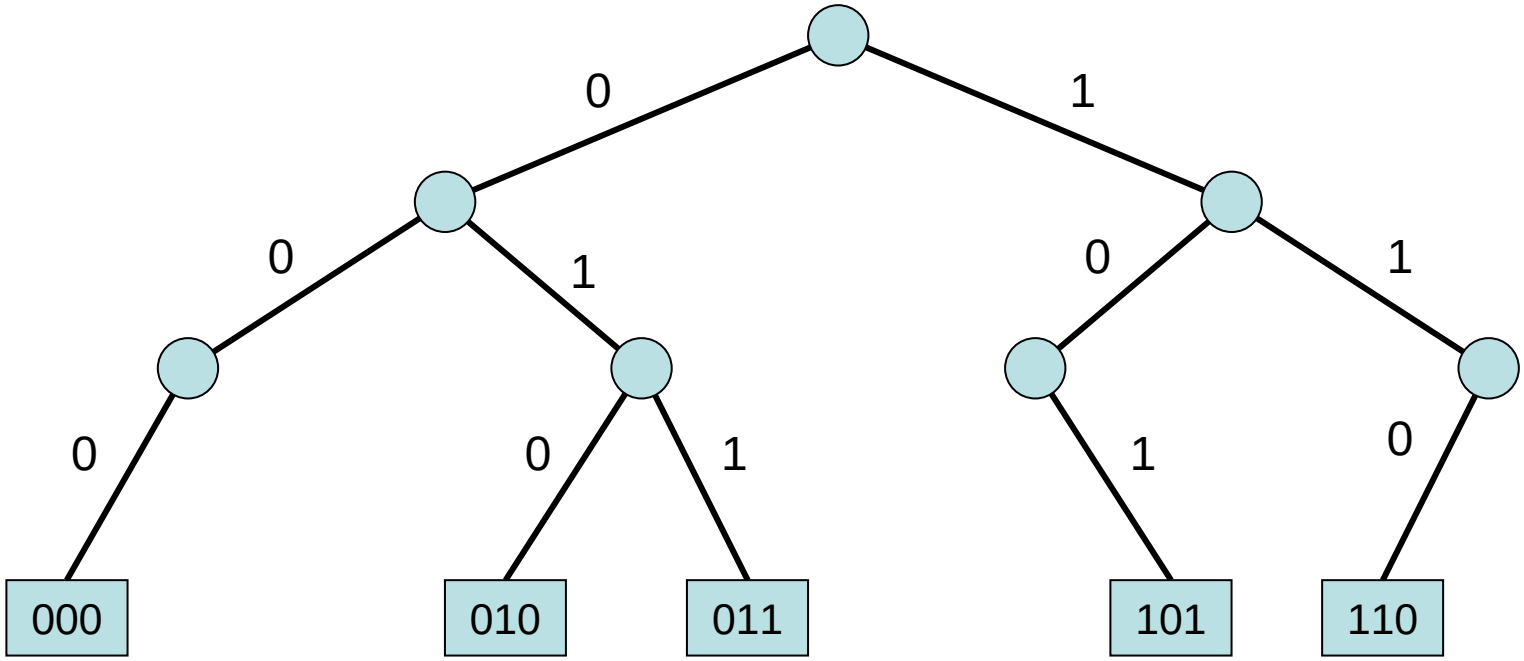
Hier: alle Schlüssel aus $\{0,1\}^W$

Beispiel:

(0,2,3,5,6) mit W=3 ergibt (000,010,011,101,110)

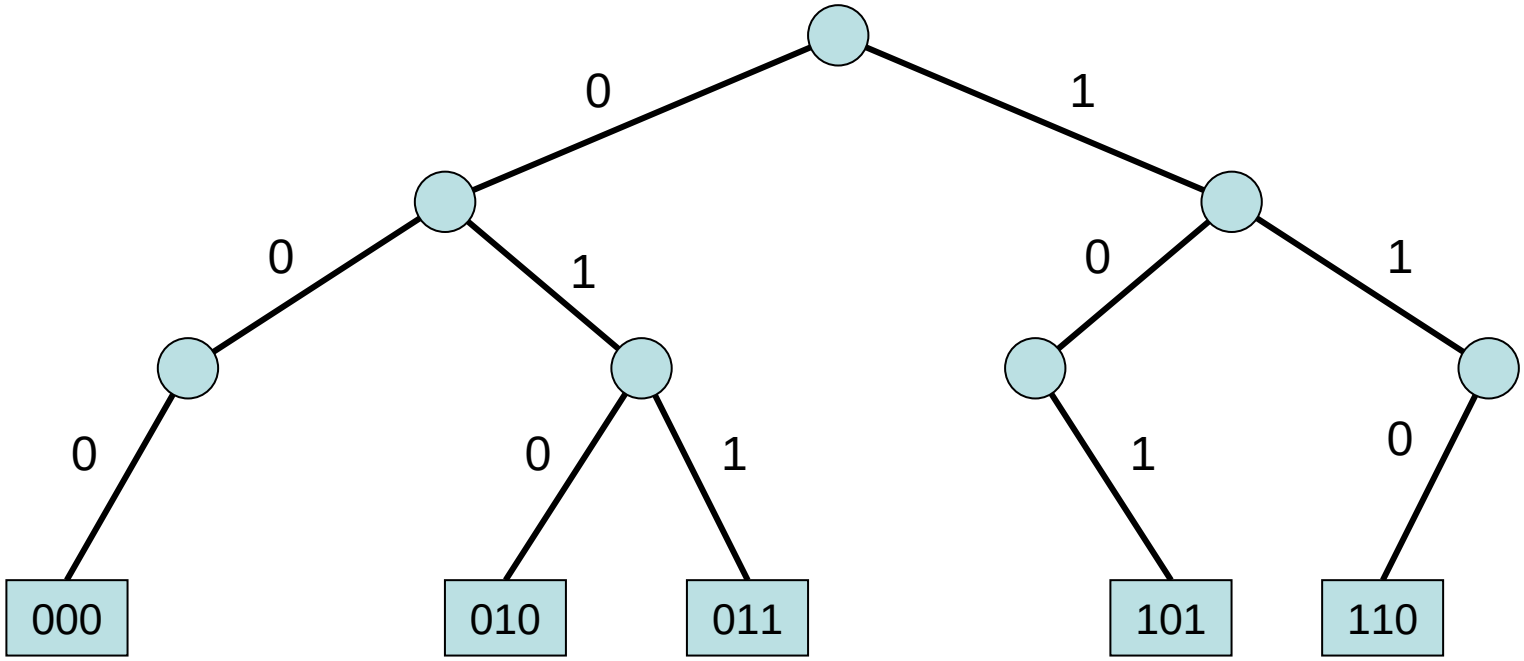
Trie

Trie aus Beispielzahlen:



Trie

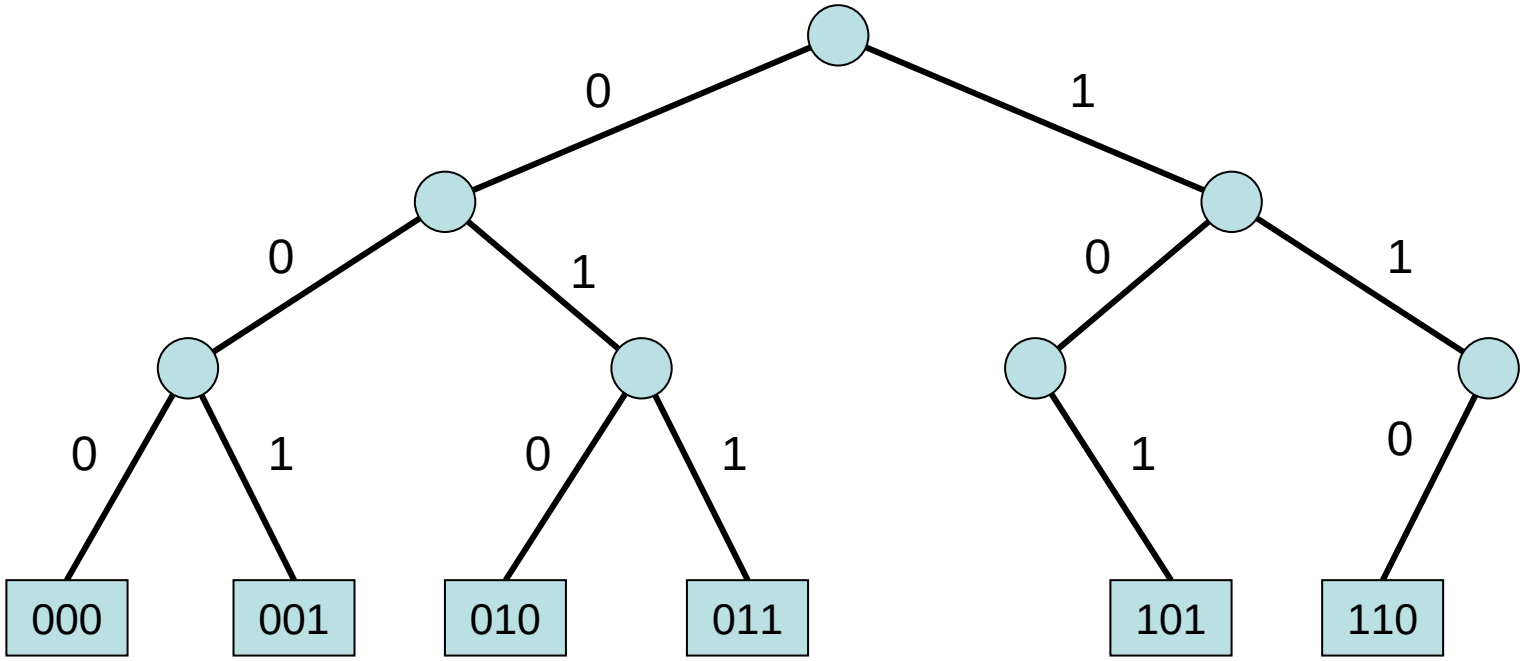
Präfixsuche für Zahl 4 (4 entspricht 100):



Ausgabe: 5 (größter gem. Präfix)

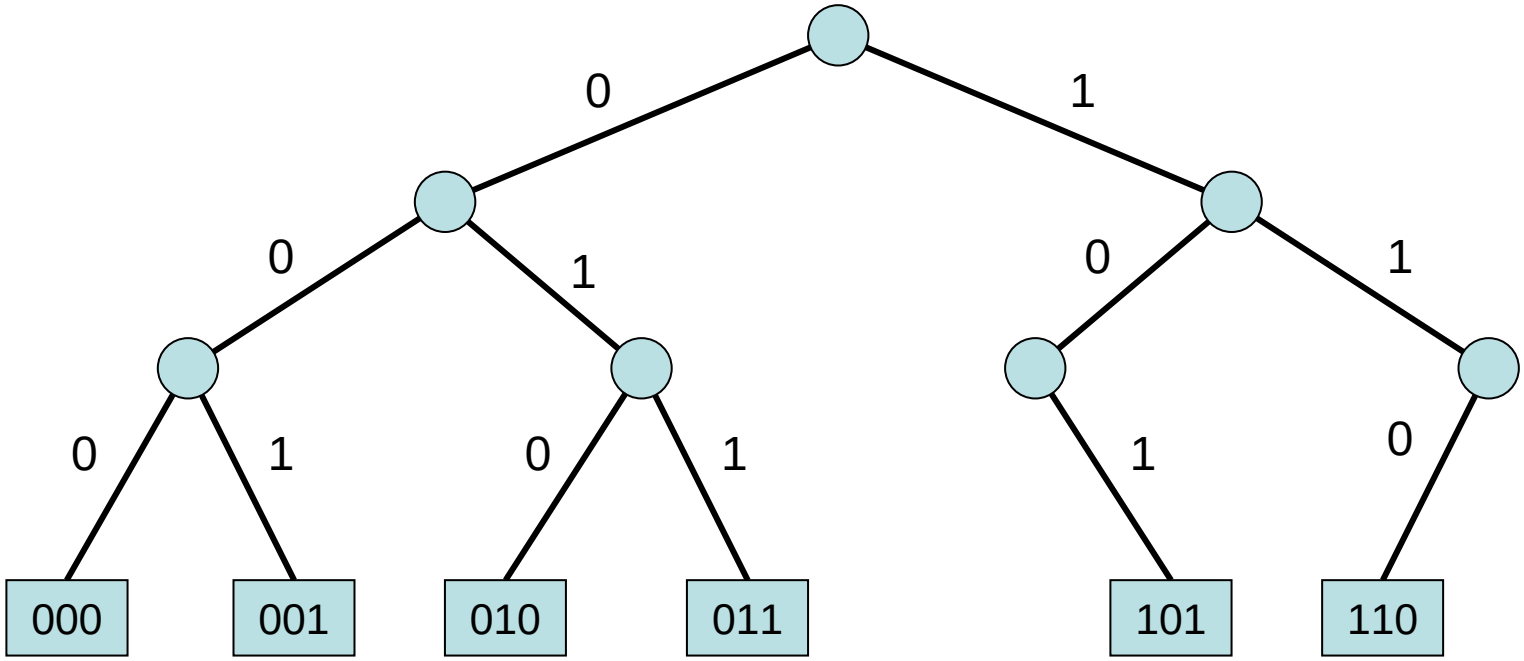
Trie

Insert(1) (1 entspricht 001):



Trie

Delete(5):



Patricia Trie

Probleme:

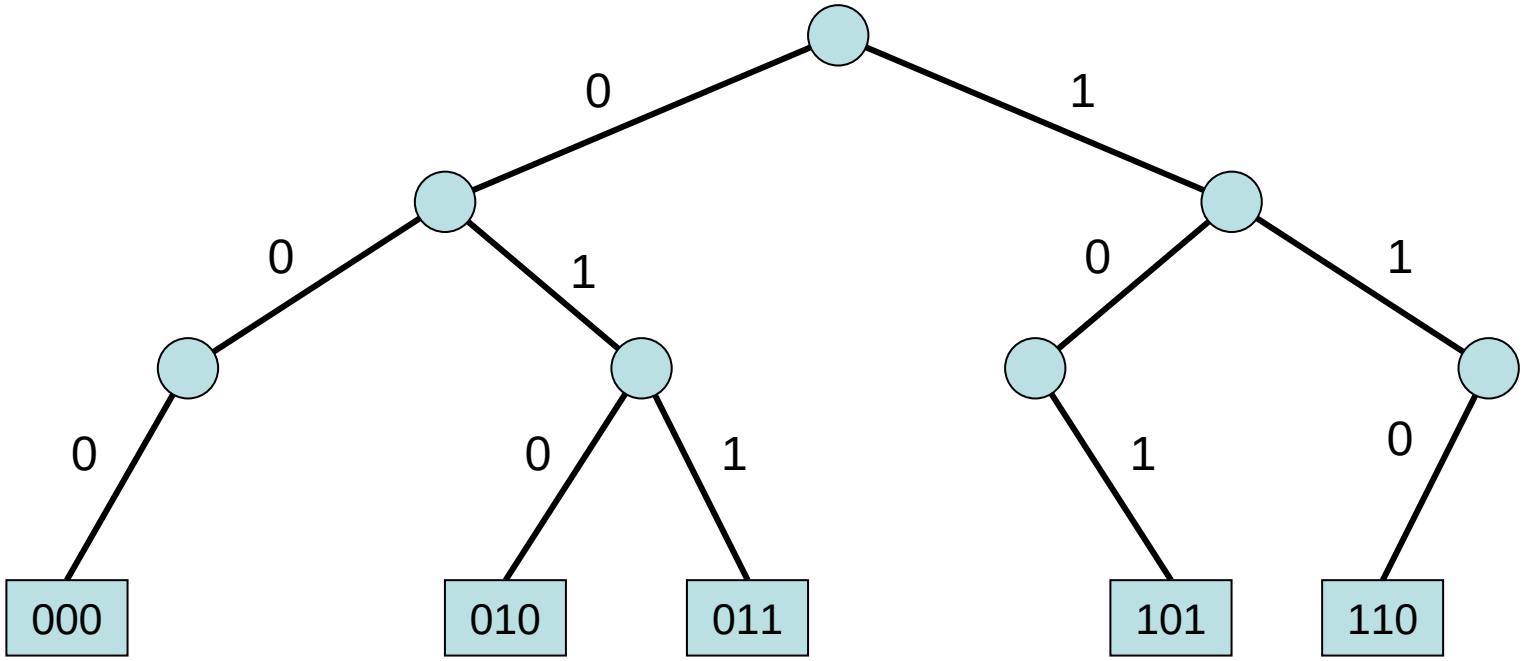
- Präfixsuche im Trie mit Schlüsseln aus $x \in \{0,1\}^W$ kann $\Theta(W)$ Zeit dauern.
- Insert und delete benötigen bis zu $\Theta(W)$ Restrukturierungen

Verbesserung: verwende Patricia Tries

Ein Patricia Trie ist ein Trie, in dem alle Knotenketten (d.h. Folgen von Knoten mit Grad 1) zu einer Kante verschmolzen werden.

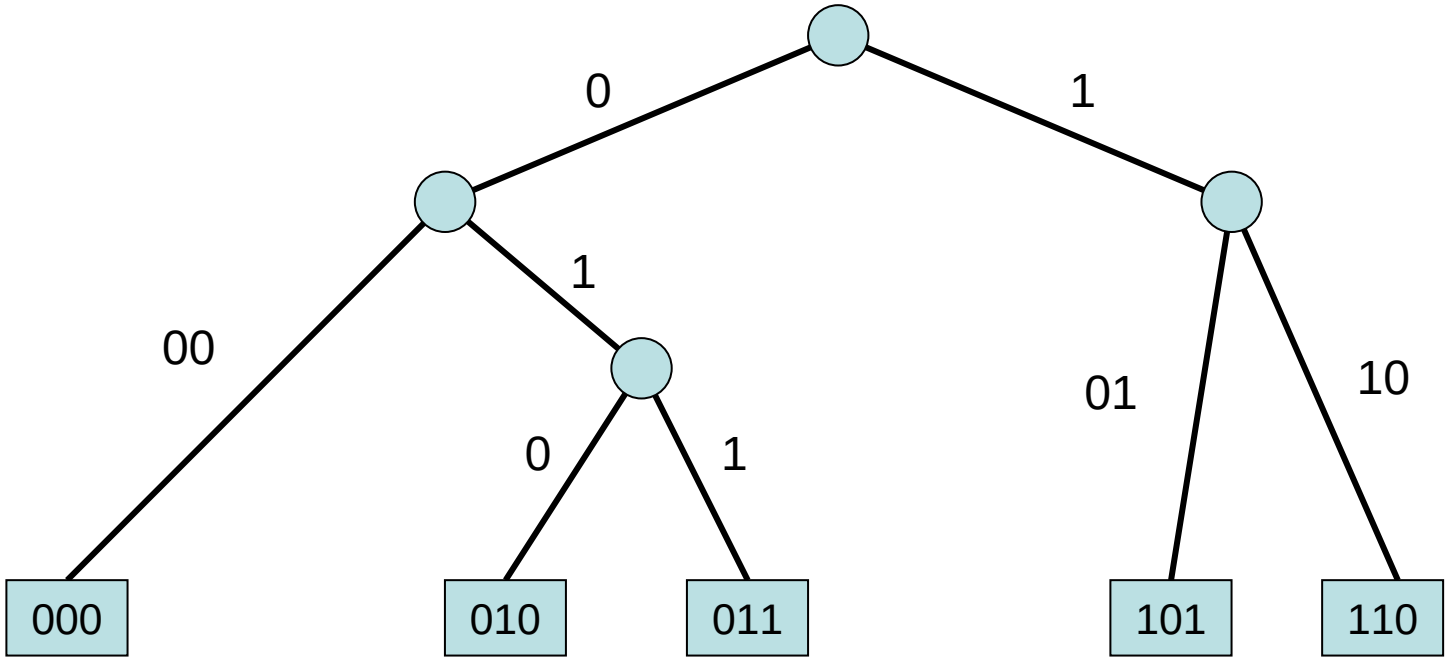
Trie

Trie aus Beispielzahlen:



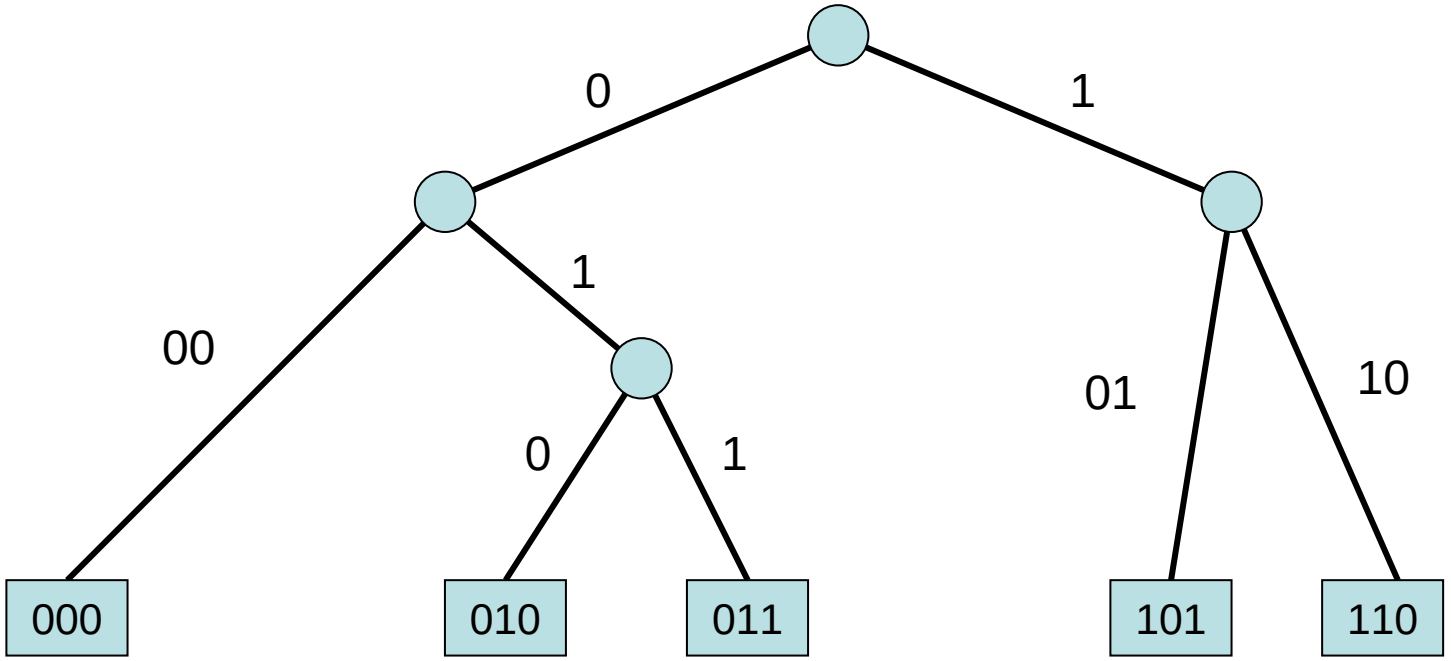
Patricia Trie

Jeder Baumknoten hat Grad 2.



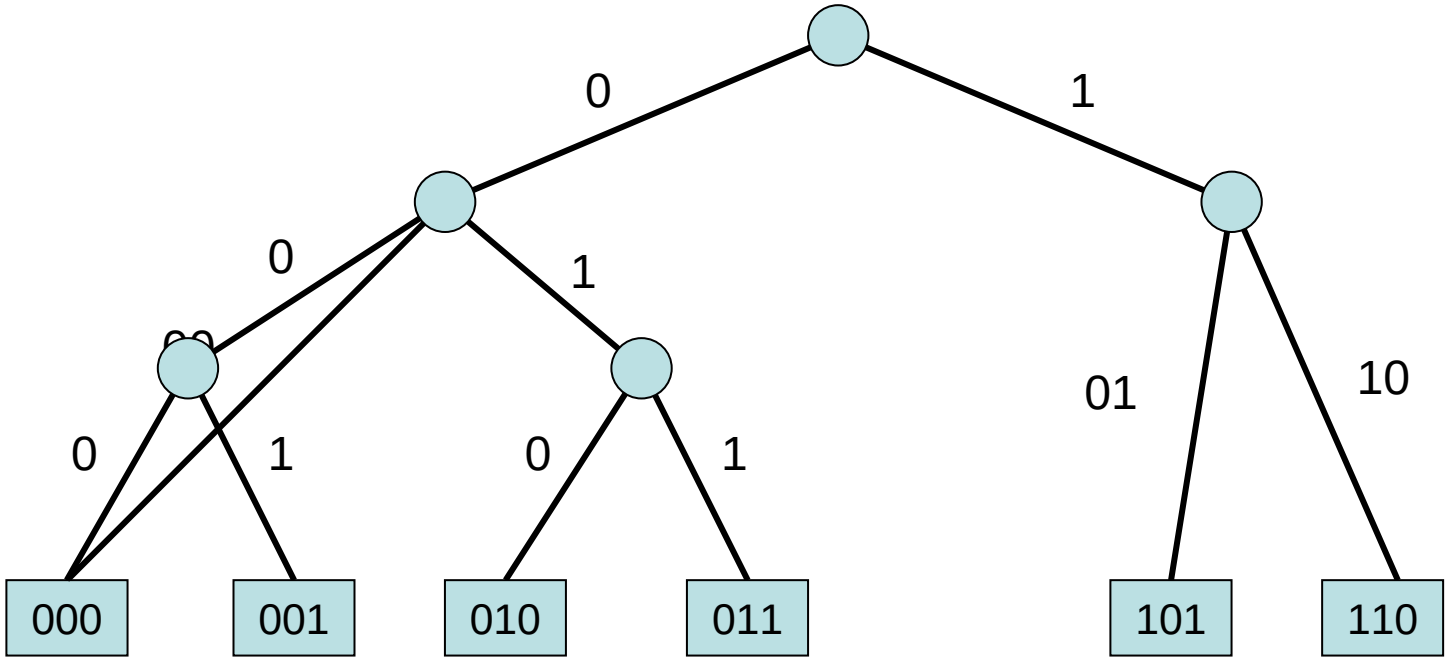
Patricia Trie

Search(4):



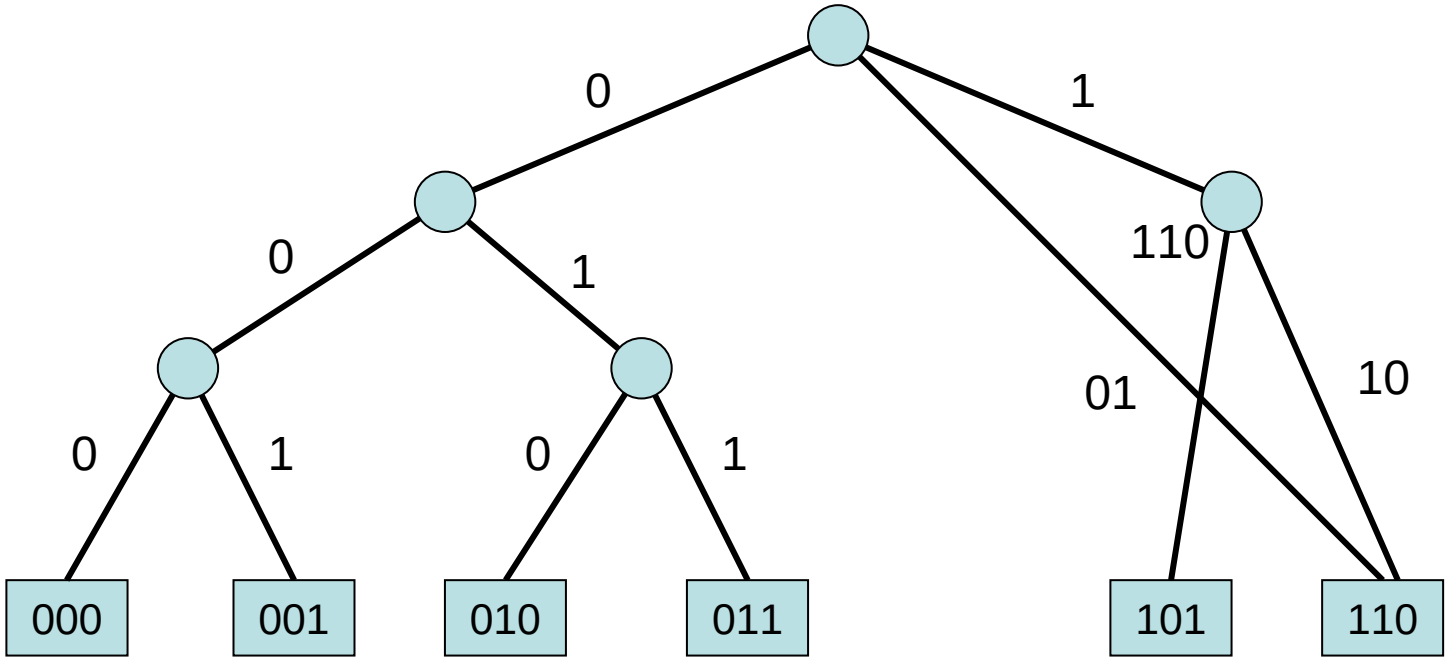
Patricia Trie

Insert(1):



Patricia Trie

Delete(5):



Patricia Trie

- Search, insert und delete wie im einfachen binären Suchbaum, nur dass wir Labels an den Kanten haben.
- Suchzeit immer noch O(W), aber Restrukturierungsaufwand nur noch O(1)

Idee: Um Suchzeit zu verringern, betten wir Patricia Trie in Hashtabelle ein.

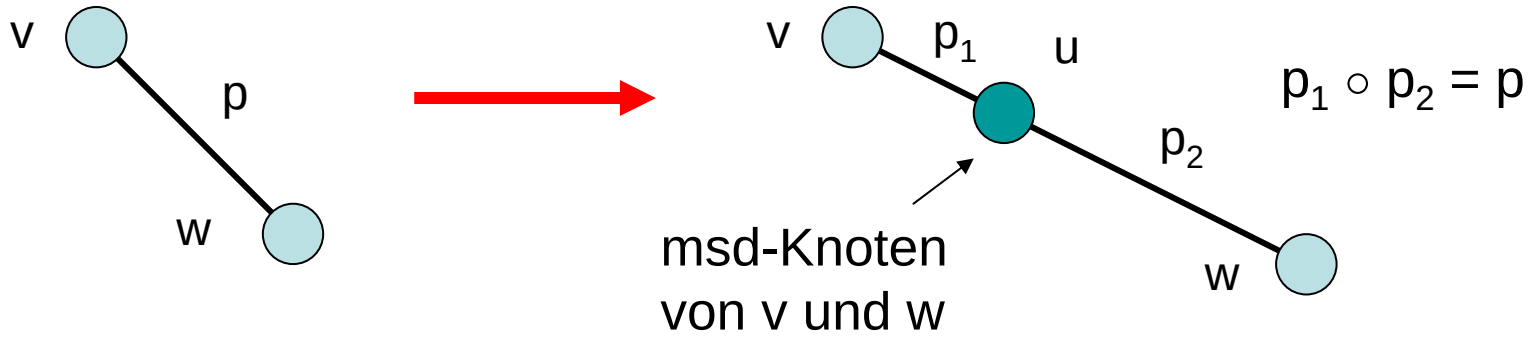
Trie Hashing

- Für eine Bitfolge x sei |x| die Länge von x.
- Für einen Baumknoten v sei b(v) die konkatenierte Bitfolge der Kanten des eindeutigen Weges von der Wurzel zu v (d.h. der gem. Präfix aller Elemente unter v).
- Betrachte eine Bitfolge b mit Binärdarstellung von |b| gleich $(x_k,\ldots,x_0)$. Sei b' ein Präfix von b.
  Die msd-Folge m(b',b) von b' und b ist das Präfix von b der Länge $\sum_{i=j}^{k} x_i\, 2^i$ mit j=msd(|b|,|b'|).
  (Zur Definition von msd siehe die Radix-Heaps.)

Beispiel: Betrachte b=01101001010 und b'=011010. Dann ist |b|=11 oder binär 1011 und |b'|=6 oder binär 110, d.h. msd(|b|,|b'|)=3. Also ist von m(b',b)= 01101001.

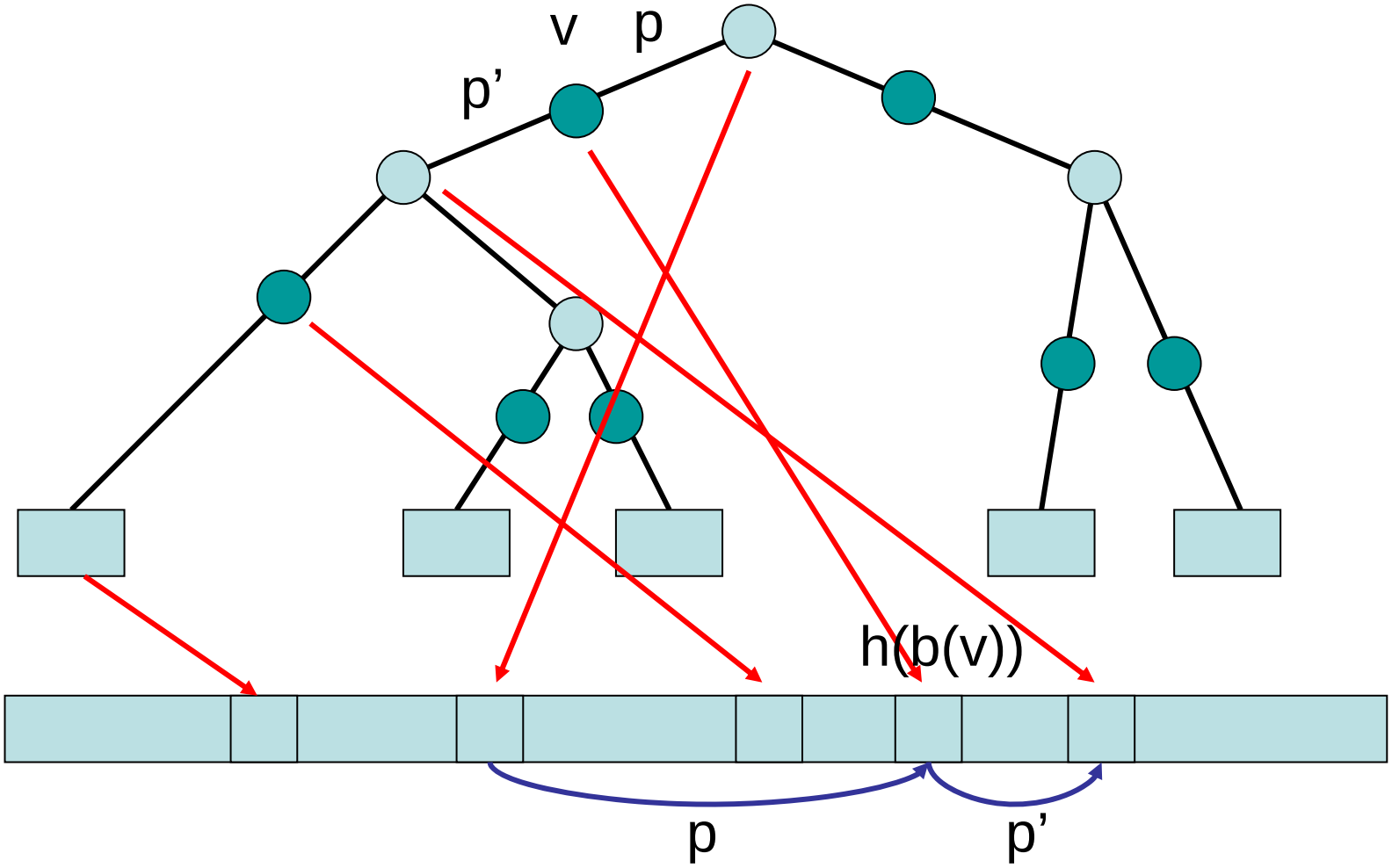
Trie Hashing

Zentrale Idee: Wir ersetzen jede Kante e={v,w} im Patricia Trie durch zwei Kanten {v,u} und {u,w} mit b(u)=m(b(v),b(w)) und speichern resultierenden Trie in der gegebenen Hashtabelle ab.



Zur Vereinfachung: Hashing mit Chaining

Trie Hashing



Trie Hashing

Datenstruktur:

Jeder Hasheintrag zu einem Baumknoten v speichert:

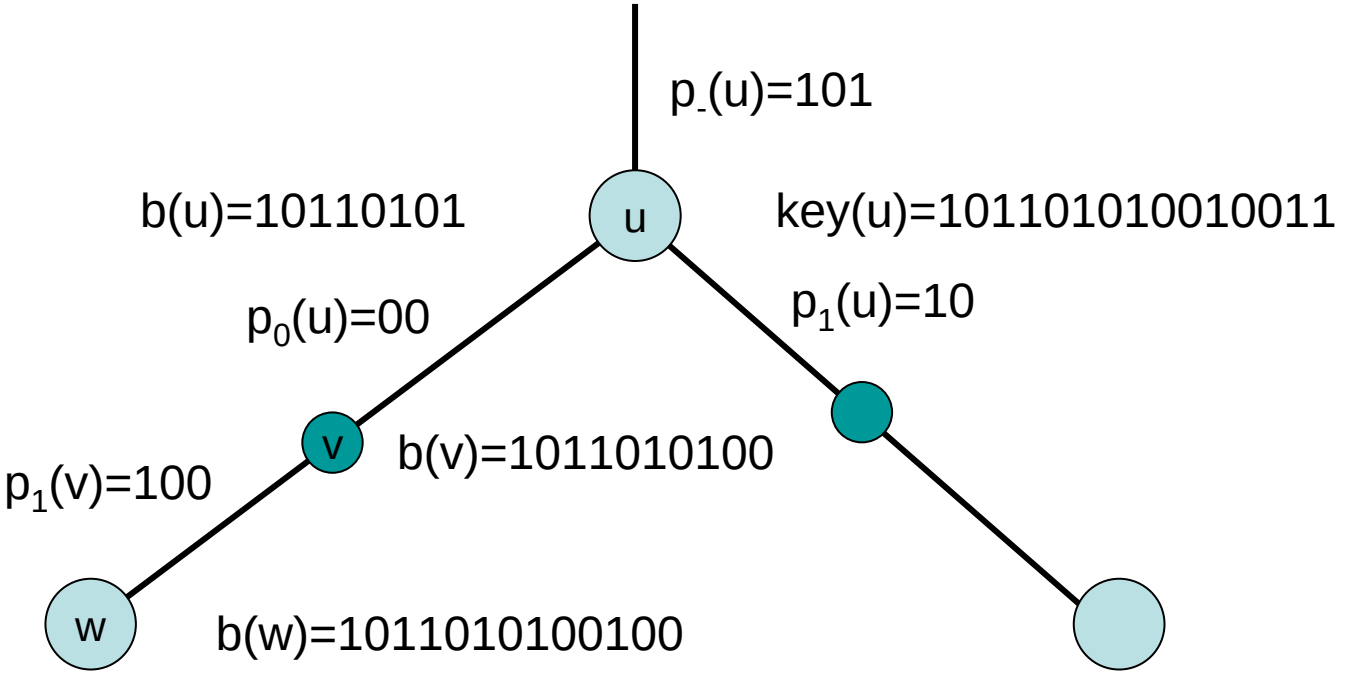
1. Bitfolge $b(v)$ zu v
2. Schlüssel $key(v)$ eines Elements e, falls v Knoten im Original Patricia Trie ist (für Suchergebnis)
3. Bitfolgen $p_x(v)$ der Kanten bzgl. nächstem Bit $x \in \{0,1\}$ zu Kindern von v (für Suche)
4. Bitfolge $p_{-}(v)$ zum Vaterknoten (für Restrukturierung)

Jeder Hasheintrag zu einem Element e speichert:

1. Schlüssel von e
2. Ort des Schlüssels von e im Baum (siehe 2. oben)

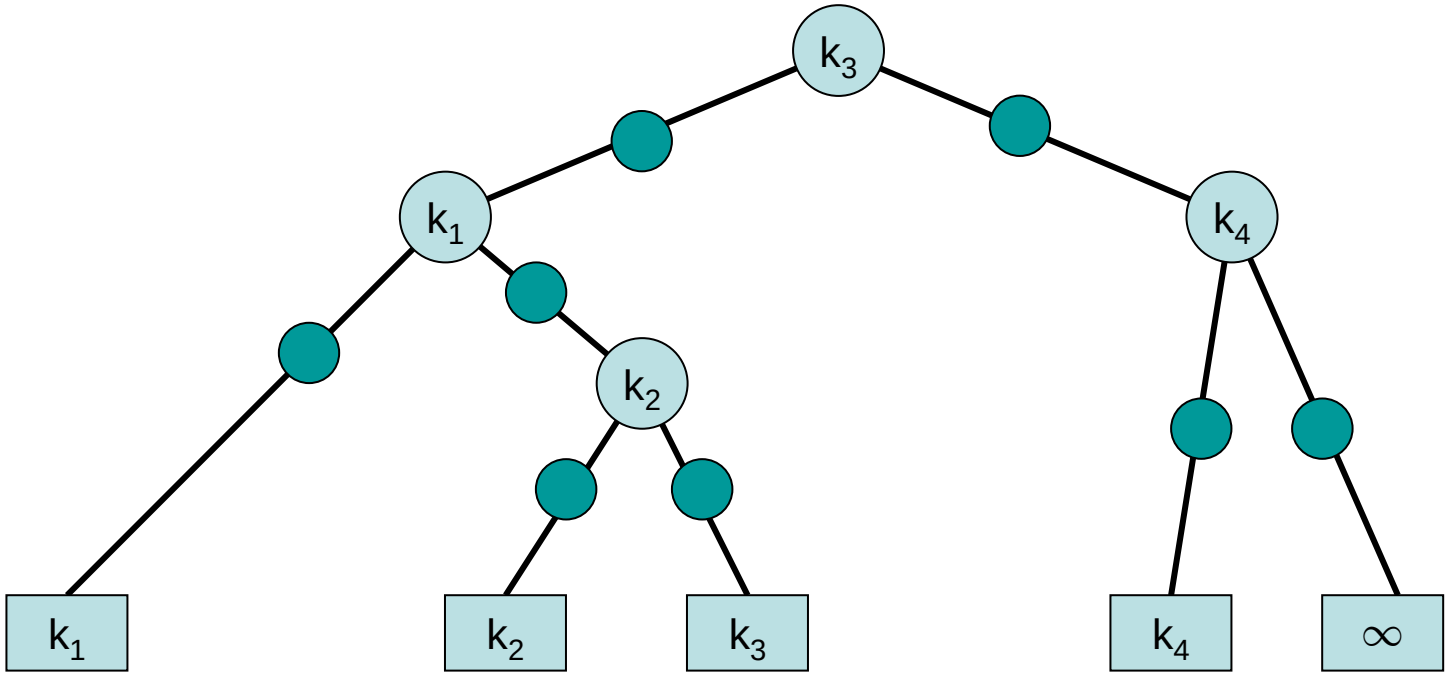
Trie Hashing

Beispiel (ohne dass msd-Knoten stimmen):



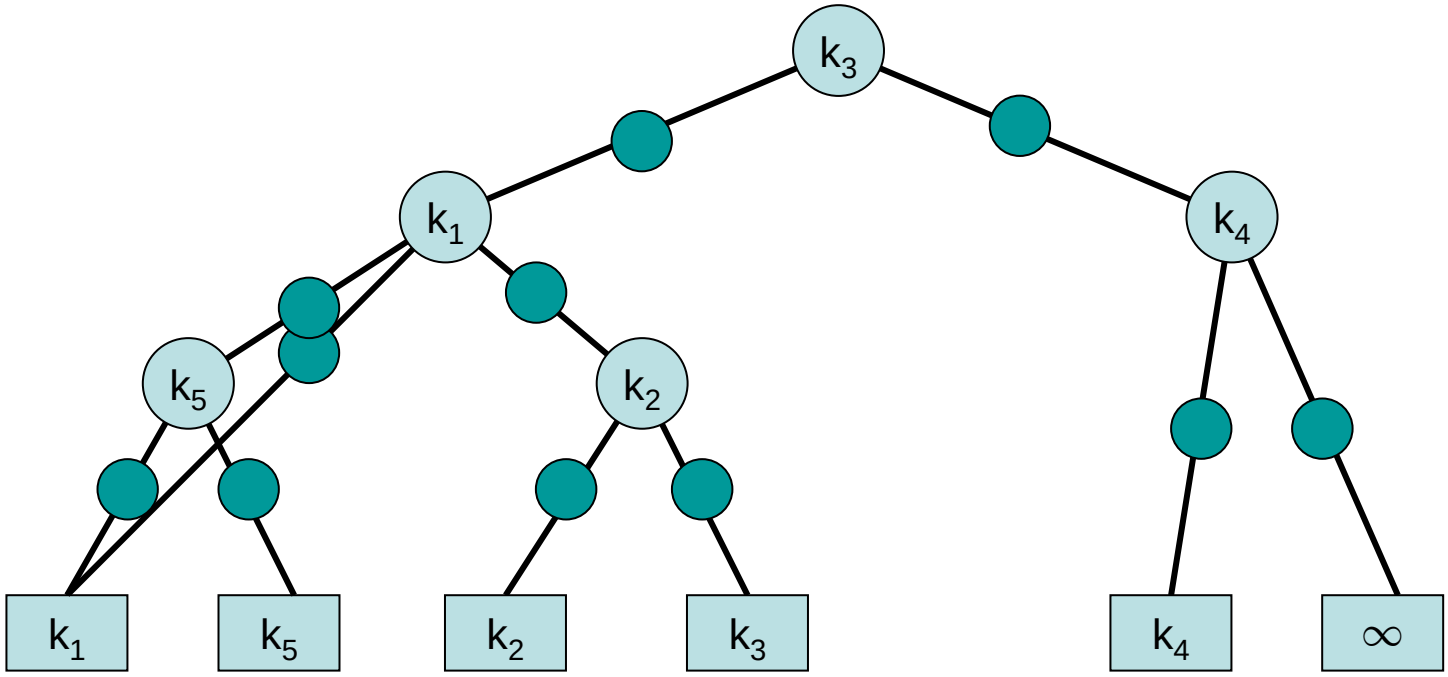
Trie Hashing

Wir vergessen zunächst die Suchproblematik und illustrieren insert und delete.



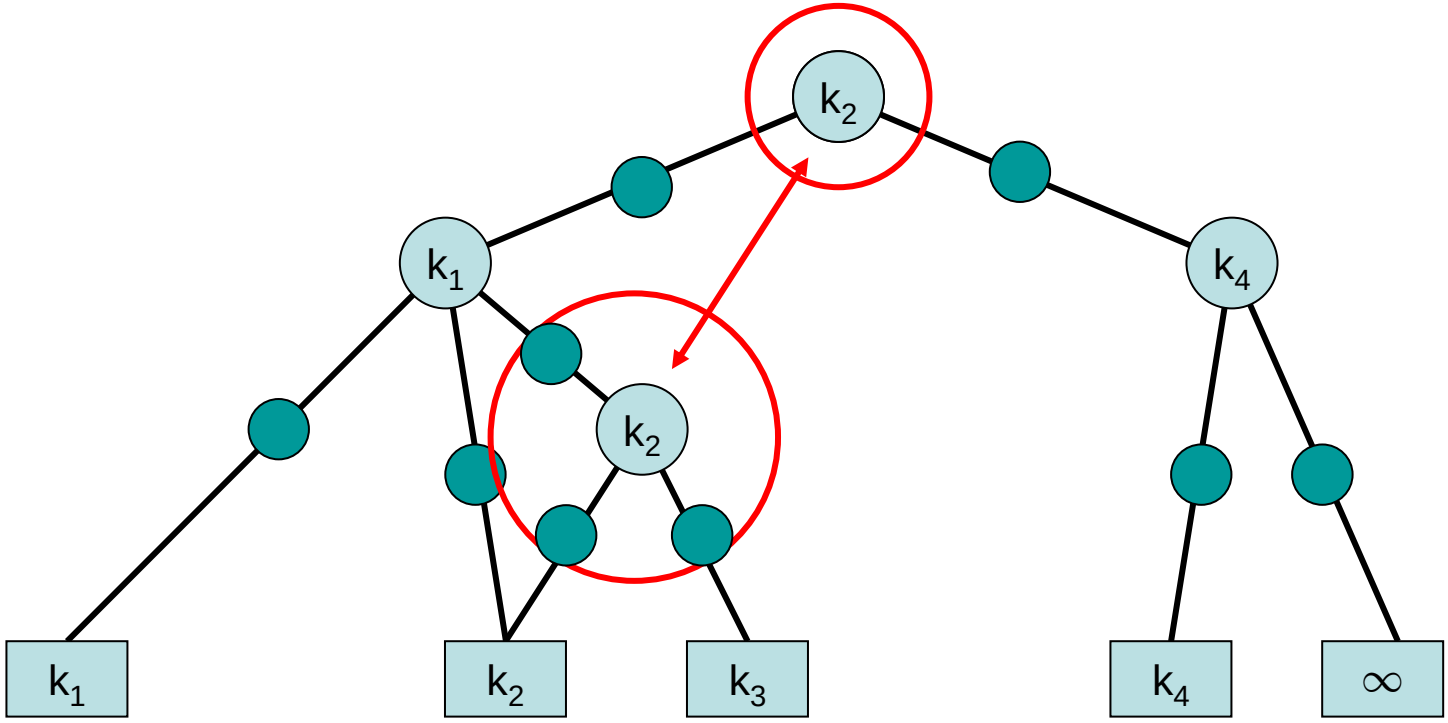
Trie Hashing

Insert(e) mit key(e)=k_5: wie im bin. Suchbaum



Trie Hashing

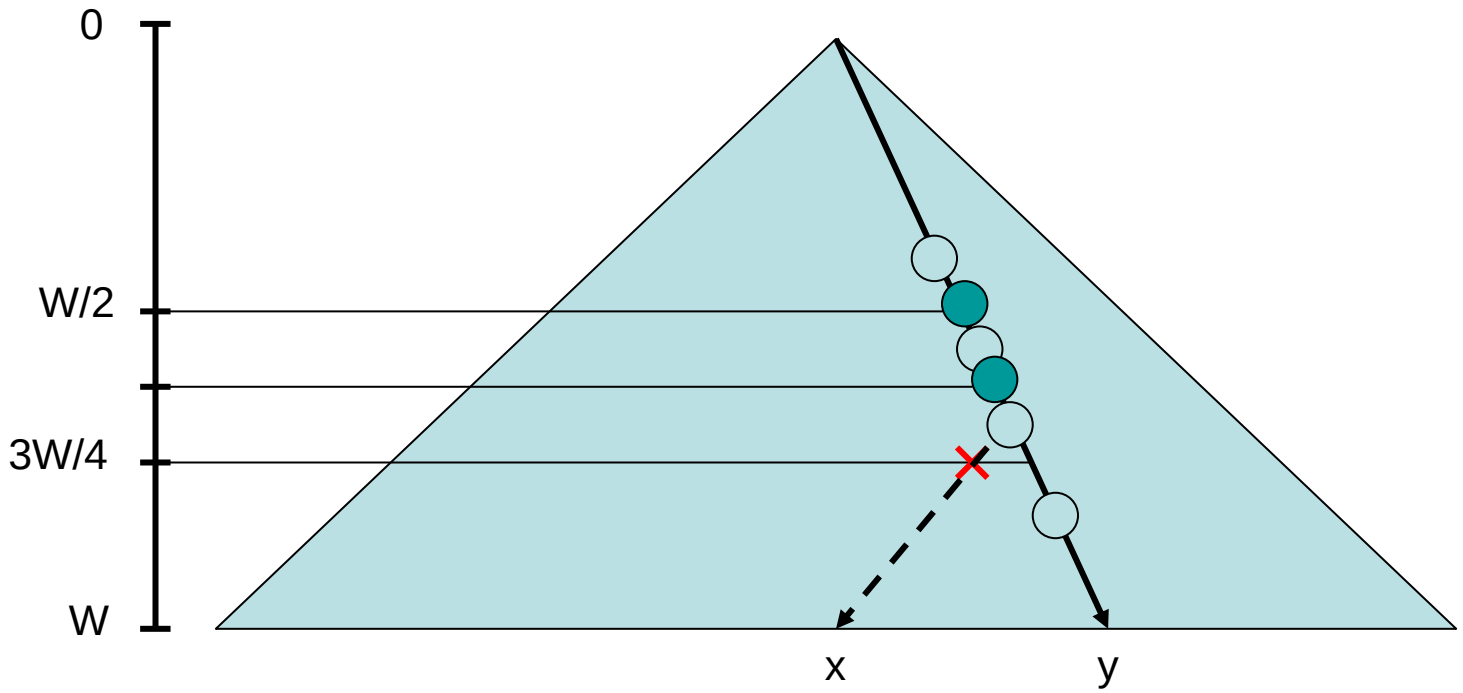
Delete(k_3): wie im binären Suchbaum



Trie Hashing

Search(x): (W Zweierpotenz)

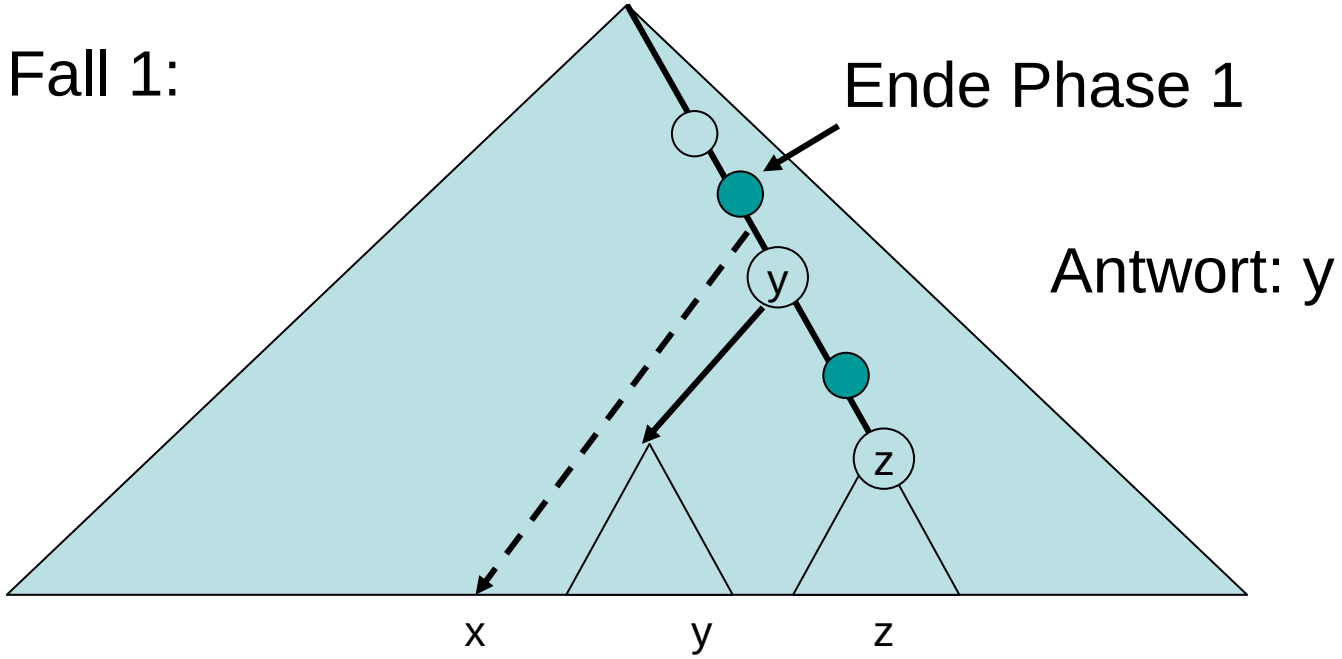
Phase 1: binäre Suche über msd-Knoten



Trie Hashing

Search(x): (W Zweierpotenz)

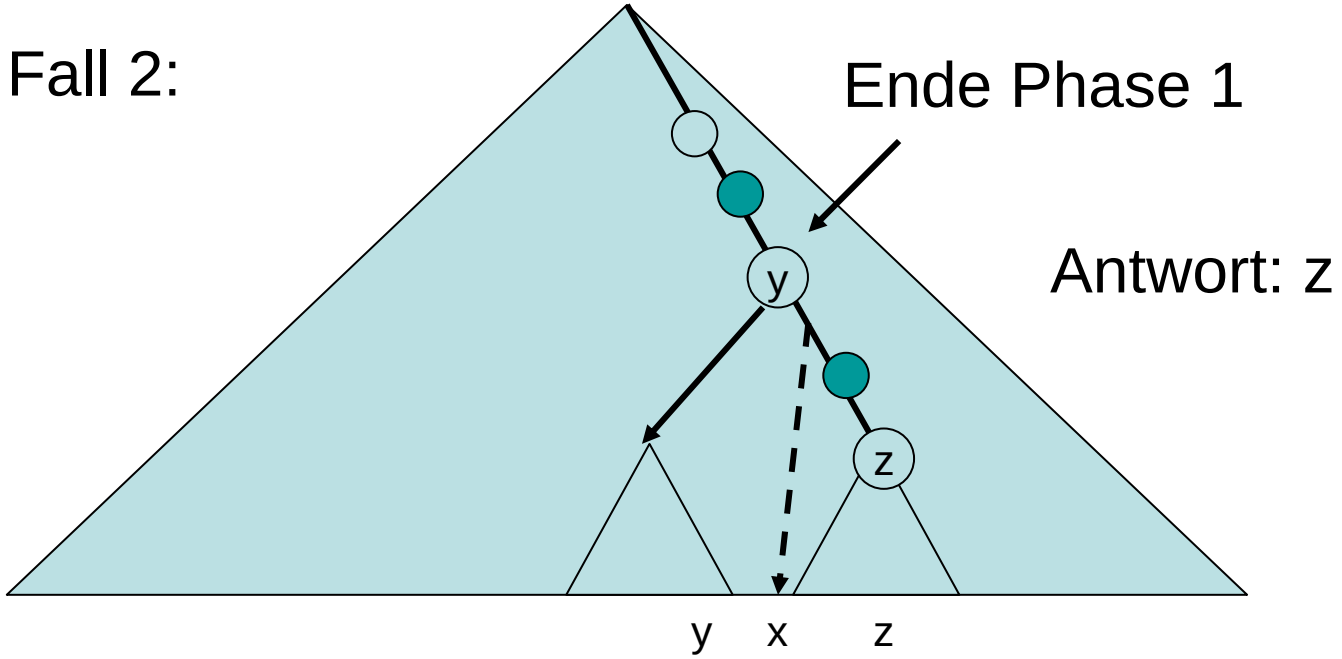
Phase 2: lies Schlüssel aus Baumknoten



Trie Hashing

Search(x): (W Zweierpotenz)

Phase 2: lies Schlüssel aus Baumknoten



Trie Hashing

- $x \in \{0,1\}^W$ habe Darstellung $(x_1,\ldots,x_W)$
- Annahme: keine Kollisionen, eine Hashf. h

```
search(x):
    // gleich gefunden, dann fertig
    if key(T[h(x)])=x then return T[h(x)]
    // Phase 1: binäre Suche auf x
    s:=⌊log W⌋; k:=0; v:=T[h(ε)]; p:=p_{x_1}(v)  // v: Wurzel des Patricia Tries
    while s >= 0 do
        // gibt es Element mit Präfix (x_1,…,x_{k+2^s}) ?
        if (x_1,…,x_{k+2^s}) = b(T[h(x_1,…,x_{k+2^s})])     // (msd-)Knoten existiert
            then k:=k+2^s; v:=T[h(x_1,…,x_k)]; p:= (x_1,…,x_k) ∘ p_{x_{k+1}}(v)
            else if (x_1,…,x_{k+2^s}) ist Präfix von p
                    // Kante aus v deckt (x_1,…,x_{k+2^s}) ab
                    then k:=k+2^s
        s:=s-1
    // weiter mit Phase 2…
```

Trie Hashing

search(x): (Fortsetzung)

// Phase 2: Laufe zu Knoten mit größtem Präfix

while $b(v)$ ist Präfix von $(x_1,\ldots,x_k)$ do

if $p_{x_{k+1}}(v)$ existiert then

$k:=k+|p_{x_{k+1}}(v)|$

$v:=T[h(b(v) \circ p_{x_{k+1}}(v))]$

else

$k:=k+|p_{\overline{x}_{k+1}}(v)|$

$v:=T[h(b(v) \circ p_{\overline{x}_{k+1}}(v))]$

if v ist msd-Knoten then

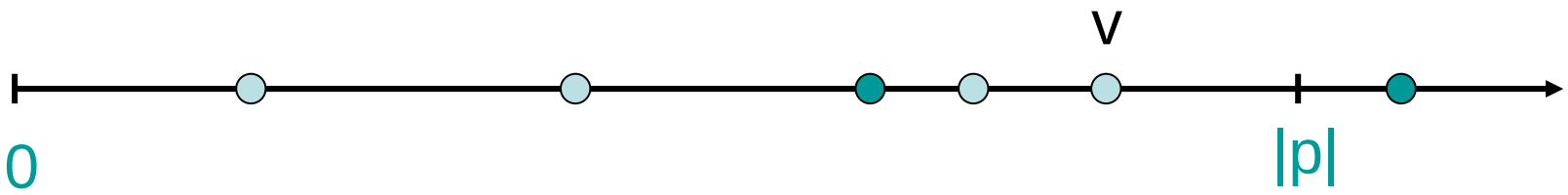
$v:=T[h(b(v) \circ p)]$ für Bitfolge p aus v

return key(v)

Trie Hashing

Korrektheit von Phase 1:

- Sei p der größte gemeinsame Präfix von x und einem Element $y \in S$ und sei $|p|=(z_k,\ldots,z_0)$.
- Patricia Trie enthält Route für Präfix p
- Sei v letzter Knoten auf Route bis p
- Fall 1: v ist Patricia Knoten

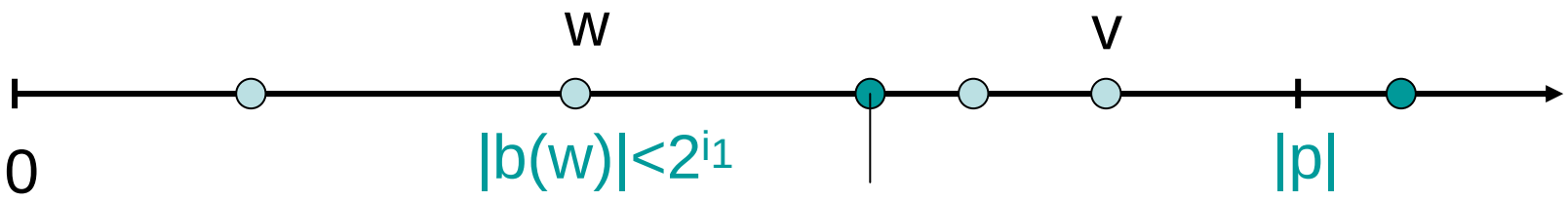


Binärdarstellung von $|b(v)|$ habe Einsen an den Positionen $i_1,i_2,\ldots$ (i_1 maximal)

Trie Hashing

Korrektheit von Phase 1:

- Sei p der größte gemeinsame Präfix von x und einem Element $y \in S$ und sei $|p|=(z_k,\ldots,z_0)$.
- Patricia Trie enthält Route für Präfix p
- Sei v letzter Knoten auf Route bis p
- Fall 1: v ist Patricia Knoten

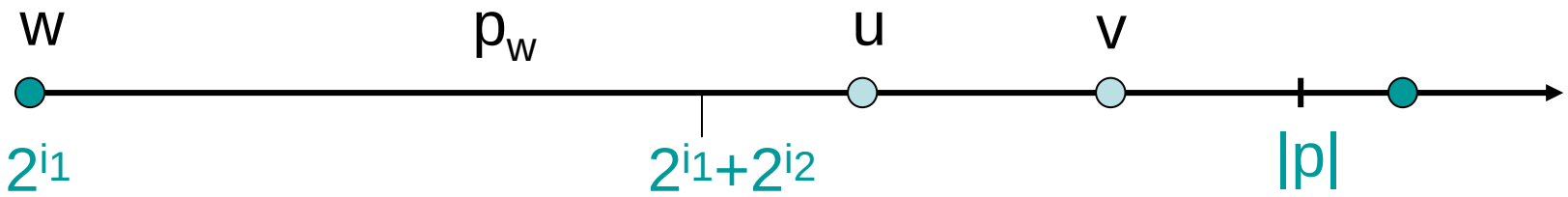


msd-Knoten muss bei 2^{i_1} existieren,
wird bei binärer Suche gefunden

Trie Hashing

Korrektheit von Phase 1:

- Sei p der größte gemeinsame Präfix von x und einem Element $y \in S$ und sei $|p|=(z_k,\ldots,z_0)$.
- Patricia Trie enthält Route für Präfix p
- Sei v letzter Knoten auf Route bis p
- Fall 1: v ist Patricia Knoten



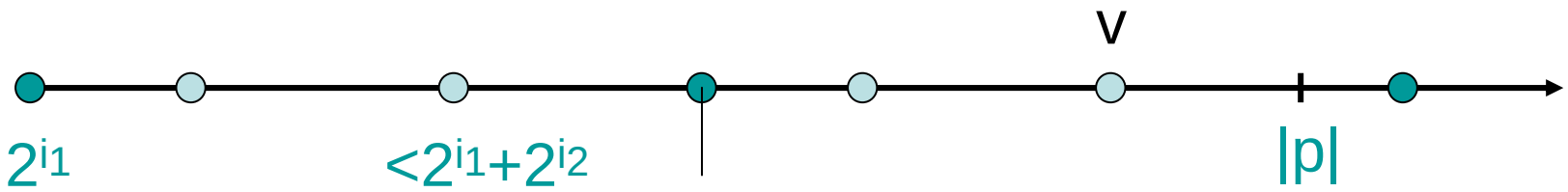
a) kein msd-Knoten bei $2^{i1}+2^{i2}$: nur dann, wenn kein Patricia-Knoten u mit $2^{i1} \leqslant |b(u)| < 2^{i1}+2^{i2}$,

aber das wird durch p_w erkannt und abgedeckt

Trie Hashing

Korrektheit von Phase 1:

- Sei p der größte gemeinsame Präfix von x und einem Element $y \in S$ und sei $|p|=(z_k,\ldots,z_0)$.
- Patricia Trie enthält Route für Präfix p
- Sei v letzter Knoten auf Route bis p
- Fall 1: v ist Patricia Knoten

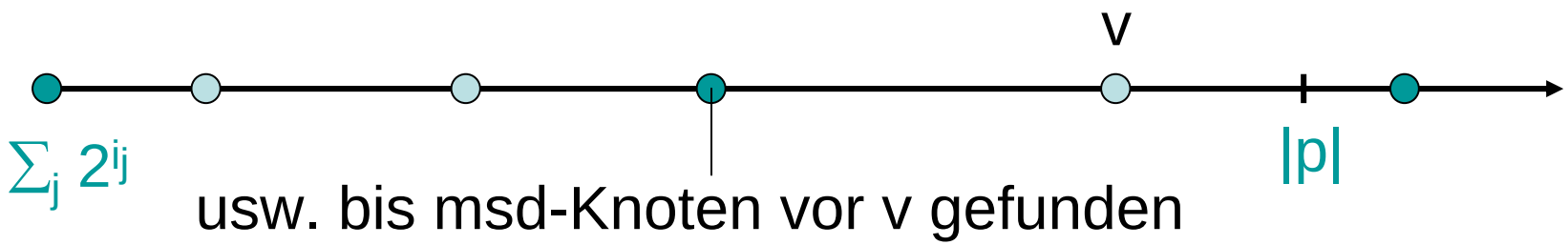


b) msd-Knoten bei $2^{i1}+2^{i2}$: wird durch binäre Suche gefunden

Trie Hashing

Korrektheit von Phase 1:

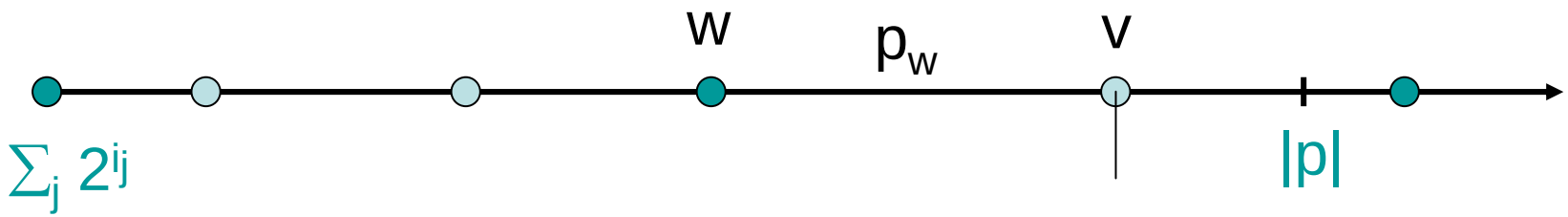
- Sei p der größte gemeinsame Präfix von x und einem Element $y \in S$ und sei $|p|=(z_k,\ldots,z_0)$.
- Patricia Trie enthält Route für Präfix p
- Sei v letzter Knoten auf Route bis p
- Fall 1: v ist Patricia Knoten



Trie Hashing

Korrektheit von Phase 1:

- Sei p der größte gemeinsame Präfix von x und einem Element $y \in S$ und sei $|p|=(z_k,\ldots,z_0)$.
- Patricia Trie enthält Route für Präfix p
- Sei v letzter Knoten auf Route bis p
- Fall 1: v ist Patricia Knoten

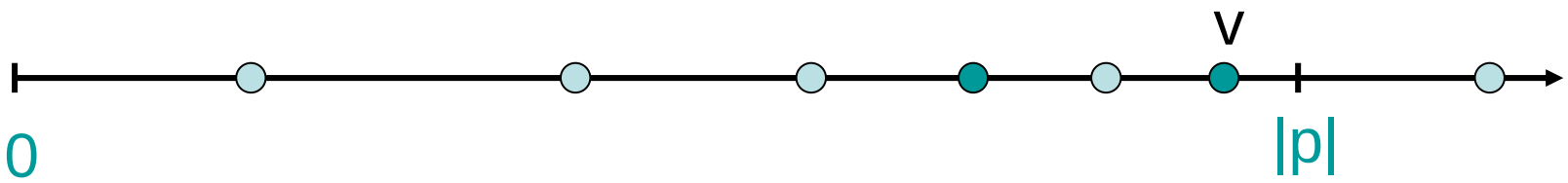


Durch Beachtung von p_w im Algorithmus wird dann auch v gefunden.

Trie Hashing

Korrektheit von Phase 1:

- Sei p der größte gemeinsame Präfix von x und einem Element $y \in S$ und sei $|p|=(z_k,\ldots,z_0)$.
- Patricia Trie enthält Route für Präfix p
- Sei v letzter Knoten auf Route bis p
- Fall 2: v ist msd-Knoten



In diesem Fall wird v gefunden (Argumente wie für Fall 1)

Trie Hashing

Laufzeit von search:

- Phase 1 (binäre Suche): O(log W) Zeit
- Phase 2 (entlang Patricia Trie): O(1) Zeit

Also Laufzeit O(log W).

Laufzeit von insert:

- O(log W) für search-Operation
- O(1) Restrukturierungen im Patricia Trie

Also Laufzeit O(log W).

Laufzeit von delete: O(1) (nur Restrukturierung)

Trie Hashing

Zwei Modelle für Laufzeit:

1. $W=(\log n)^{O(1)}$ und Operationen auf Bitfolgen der Länge W kosten eine Zeiteinheit: Laufzeit $O(\log \log n)$ für insert, delete und search.
2. $W=\Omega(\log n)$ beliebig groß und Operationen auf Bitfolgen der Länge $O(\log n)$ kosten eine Zeiteinheit: Laufzeit $O(W/\log n + \log \log n)$ für insert, delete und search, was optimal für $W=\Omega(\log n \log \log n)$ ist.

Übersicht

Klassische Hashverfahren

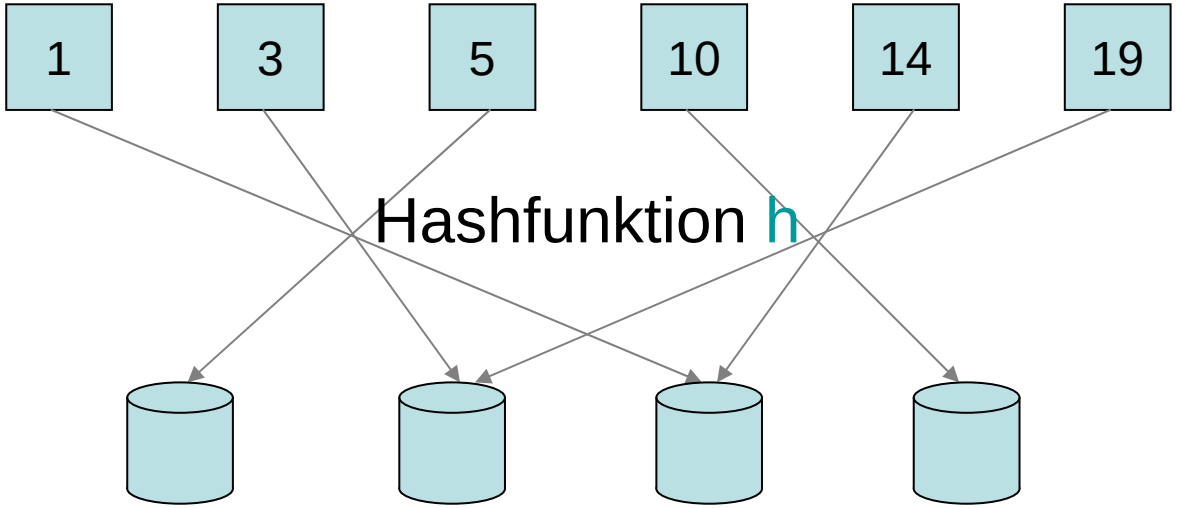
- Hashing mit Chaining
- Hashing mit Linear Probing
- FKS-Hashing

Neuere Hashverfahren

- Cuckoo Hashing
- Trie Hashing
- Konsistentes Hashing und SHARE

Verteiltes Wörterbuch

Hashing auch für verteiltes Speichern anwendbar:



Problem: Menge der Speichermedien verändert sich (Erweiterungen, Ausfälle,…)

Verteiltes Wörterbuch

Grundlegende Operationen:

- insert(d): fügt neues Datum d ein
- delete(k): löscht Datum d mit key(d)=k
- lookup(k): gibt Datum d zurück mit key(d)=k
- join(v): Knoten (Speicher) v kommt hinzu
- leave(v): Knoten v wird rausgenommen

Verteiltes Wörterbuch

Anforderungen:

1. Fairness: Jedes Speichermedium mit c% der Kapazität speichert (erwartet) c% der Daten.
2. Effizienz: Die Speicherstrategie sollte zeit- und speichereffizient sein.
3. Redundanz: Die Kopien eines Datums sollten unterschiedlichen Speichern zugeordnet sein.
4. Adaptivität: Für jede Kapazitätsveränderung von c% im System sollten nur O(c%) der Daten umverteilt werden, um 1.-3. zu bewahren.

Verteiltes Wörterbuch

Uniforme Speichersysteme: jeder Knoten (Speicher) hat dieselbe Kapazität.

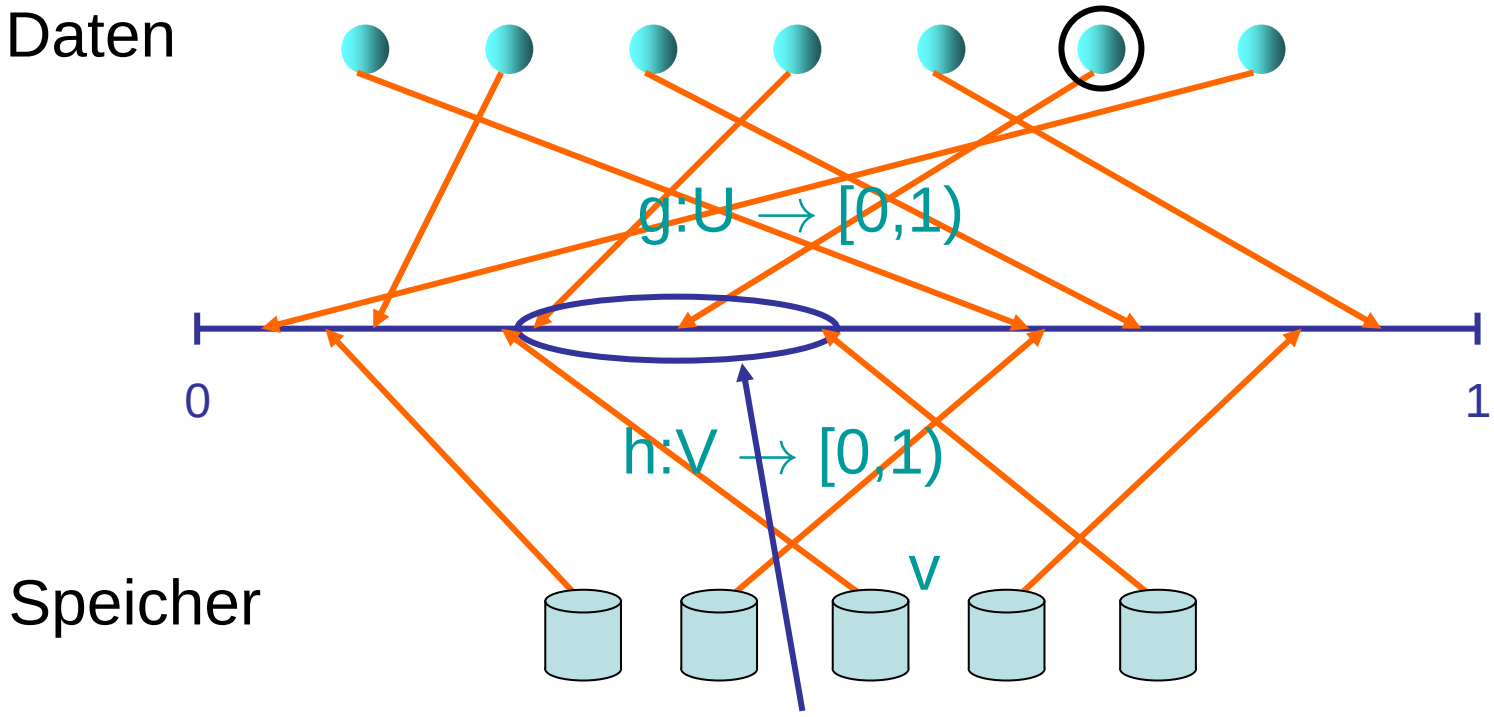
Nichtuniforme Speichersysteme: Kapazitäten können beliebig unterschiedlich sein

Vorgestellte Strategien:

- Uniforme Speichersysteme: konsistentes Hashing
- Nichtuniforme Speichersysteme: SHARE

Konsistentes Hashing

Wähle zwei zufällige Hashfunktionen h, g



Region, für die Speicher v zuständig ist

Konsistentes Hashing

- V: aktuelle Knotenmenge
- $succ(v)$: nächster Nachfolger von v in V bzgl. Hashfunktion h (wobei $[0,1)$ als Kreis gesehen wird)
- $pred(v)$: nächster Vorgänger von v in V bzgl. Hashfunktion h

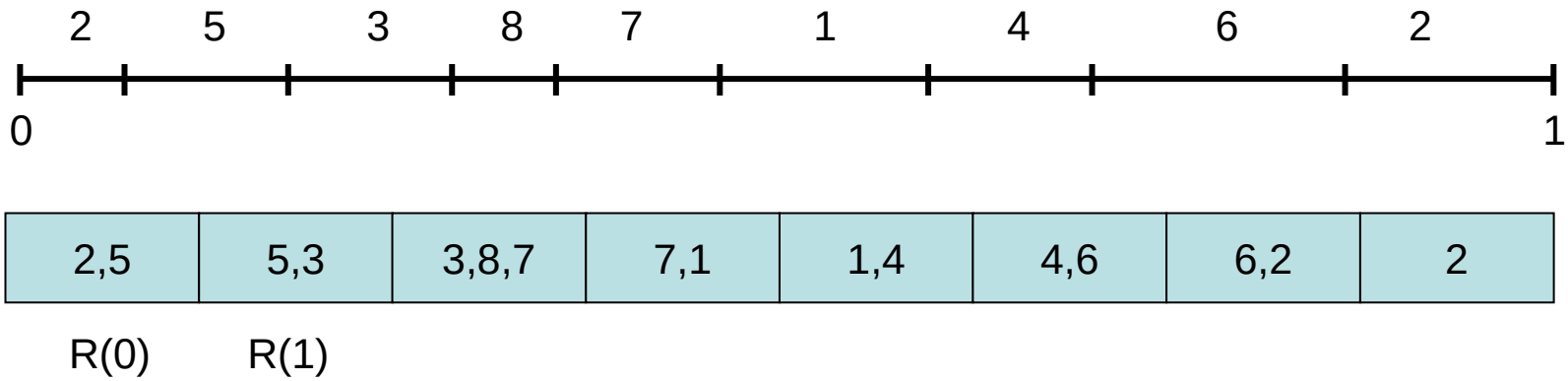
Zuordnungsregeln:

- Eine Kopie pro Datum: Jeder Knoten v speichert alle Daten d mit $g(d) \in I(v)$ mit $I(v)=[h(v), h(succ(v)))$.
- $k>1$ Kopien pro Datum: speichere jedes Datum d im Knoten v oben und seinen $k-1$ nächsten Nachfolgern bzgl. h

Konsistentes Hashing

Effiziente Datenstruktur:

- Verwende Hashtabelle T mit $m=\Theta(n)$ Positionen.
- Jede Position $T[i]$ mit $i\in\{0,\ldots,m-1\}$ ist für die Region $R(i)=[i/m, (i+1)/m)$ in $[0,1)$ zuständig und speichert alle Knoten v mit $I(v) \cap R(i) \neq \emptyset$.



Konsistentes Hashing

Effiziente Datenstruktur:

- Verwende Hashtabelle T mit $m=\Theta(n)$ Positionen.
- Jede Position $T[i]$ mit $i\in\{0,\ldots,m-1\}$ ist für die Region $R(i)=[i/m, (i+1)/m)$ in $[0,1)$ zuständig und speichert alle Knoten v mit $I(v) \cap R(i) \neq \emptyset$.

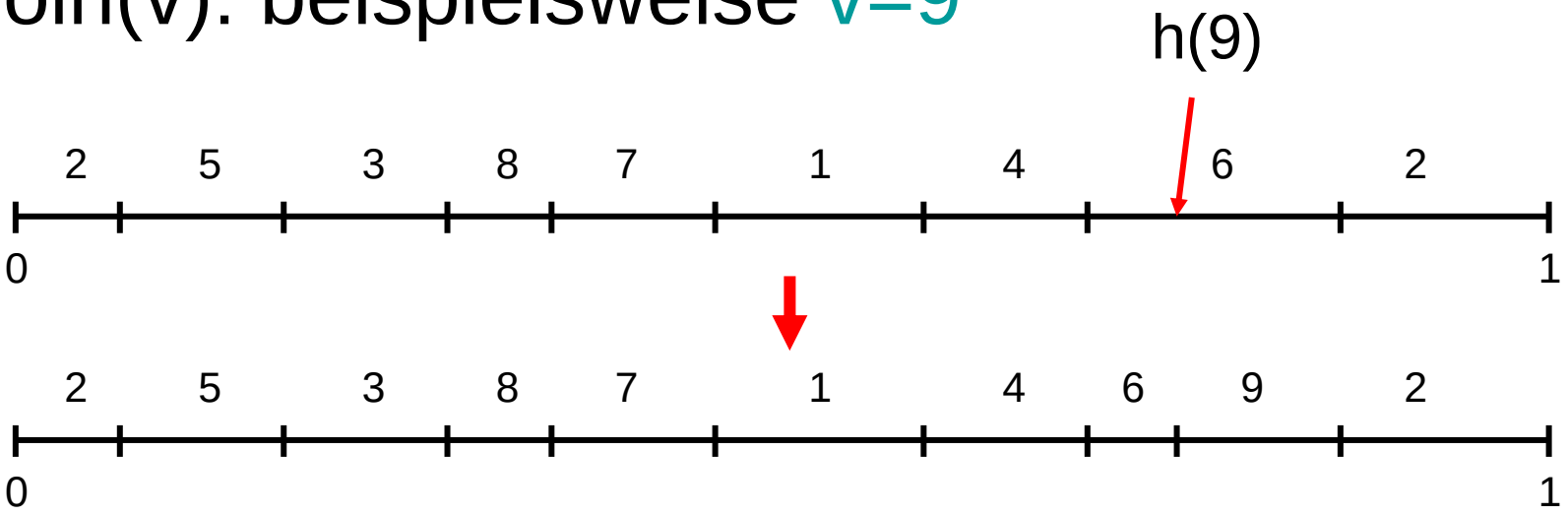
Einfach zu zeigen:

- $T[i]$ enthält erwartet konstant viele Elemente und höchstens $O(\log n / \log \log n)$ mit hoher W.keit.
- D.h. lookup(d) (die Ermittlung des zuständigen Knotens) für ein Datum d benötigt erwartet konstante Zeit

Konsistentes Hashing

Operationen:

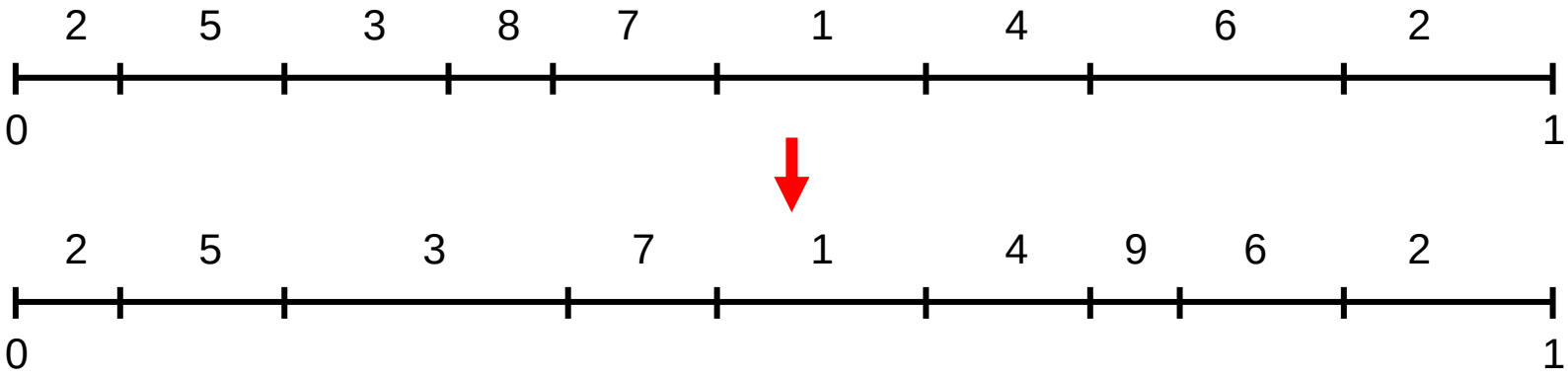
- insert und delete: siehe Zuordnungsregel
- lookup: mittels Datenstruktur
- join(v): beispielsweise v=9



Konsistentes Hashing

Operationen:

- insert und delete: siehe Zuordnungsregel
- lookup: mittels Datenstruktur
- leave(v): beispielsweise v=8



Konsistentes Hashing

Theorem 4.12:

- Konsistentes Hashing ist effizient und redundant.
- Jeder Knoten speichert im Erwartungswert 1/n der Daten, d.h. konsistentes Hashing ist fair.
- Bei Entfernung/Hinzufügung eines Speichers nur Umplatzierung von erwartungsgemäß 1/n der Daten

Problem: Schwankung um 1/n hoch!

Lösung: verwende $\Theta(\log n)$ viele Hashfunktionen für die Knoten (d.h. jeder Knoten hat $\Theta(\log n)$ Punkte in $[0,1)$).

Aber stelle sicher, dass Redundanzforderung noch gilt!

SHARE

Situation hier: wir haben Knoten mit beliebigen relativen Kapazitäten $c_1,\ldots,c_n$, d.h. $\sum_i c_i = 1$.

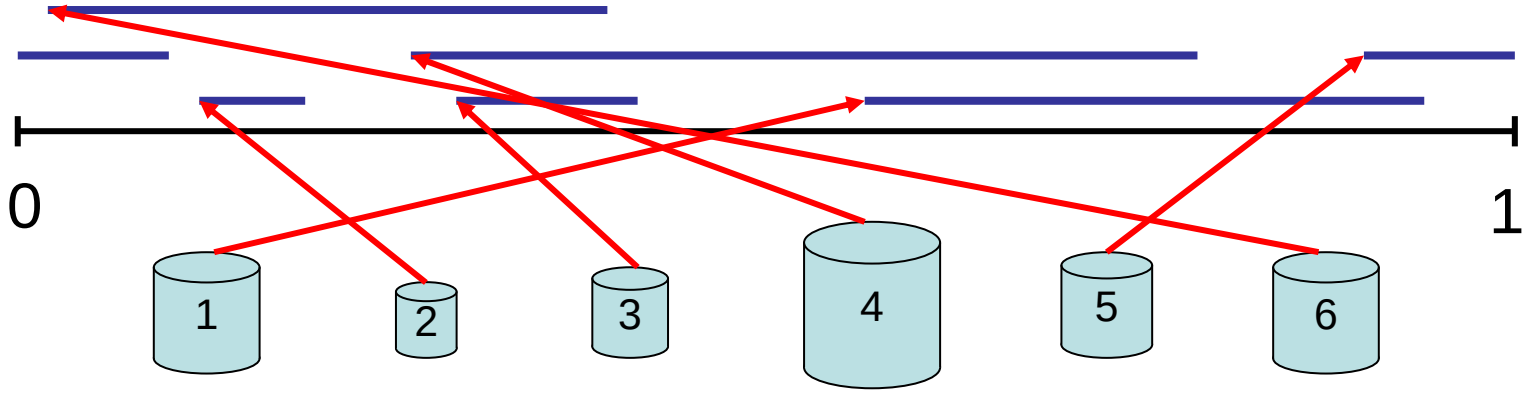
Problem: konsistentes Hashing funktioniert nicht gut, da Knoten nicht einfach in virtuelle Knoten gleicher Kapazität aufgeteilt werden können.

Lösung: SHARE

SHARE

SHARE ist ein zweistufiges Verfahren.

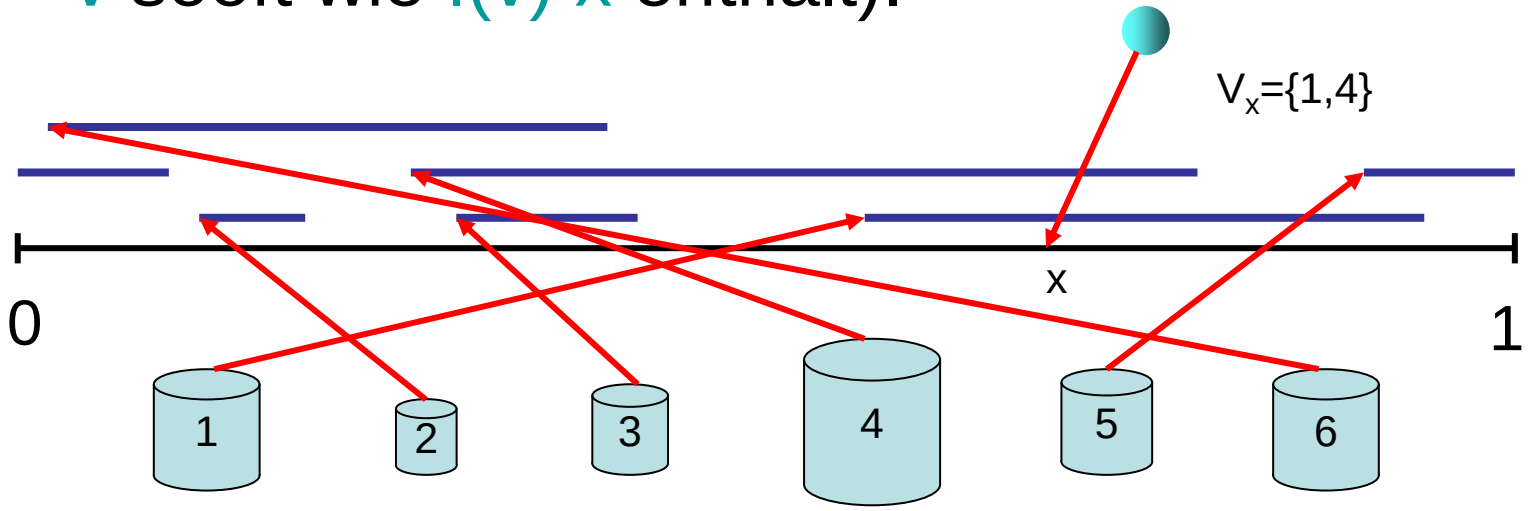
1. Stufe: Jedem Knoten v wird ein Intervall $I(v)$ in $[0,1)$ der Länge $s \cdot c_v$ zugeordnet, wobei $s=\Theta(\log n)$ ein fester Stretch-Faktor ist. Die Startpunkte der Intervalle sind durch eine Hashfunktion $h:V \rightarrow [0,1)$ gegeben.



SHARE

SHARE ist ein zweistufiges Verfahren.

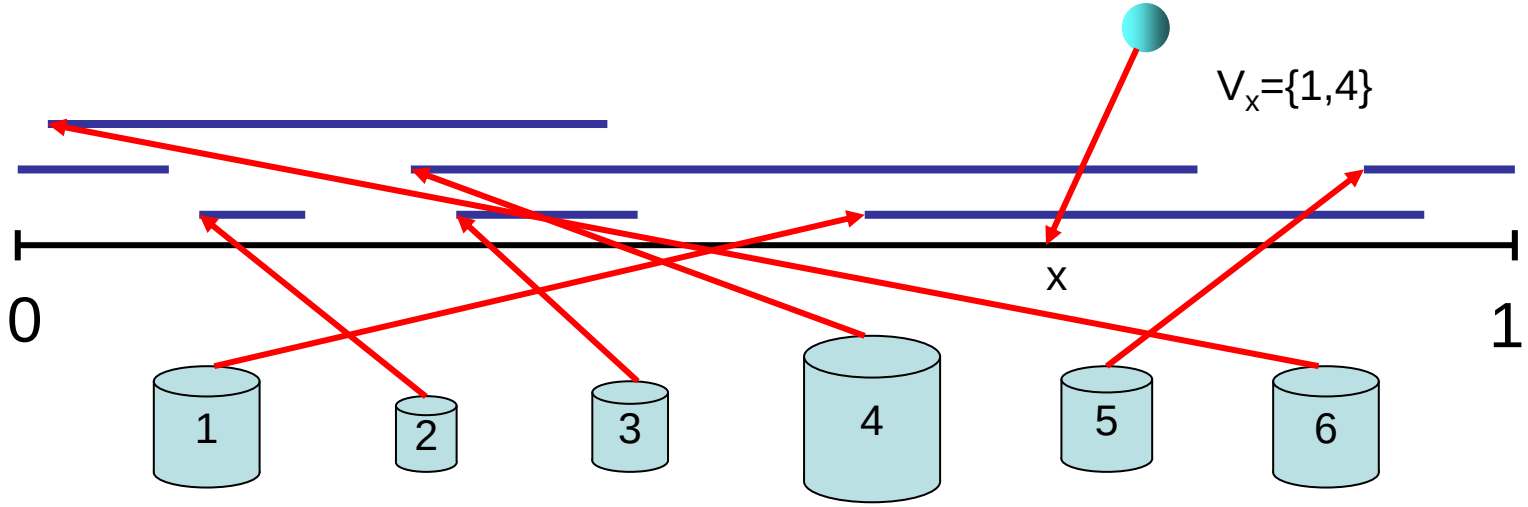
1. Stufe: Jedem Datum d wird mittels einer Hashfunktion $g:U \rightarrow [0,1)$ ein Punkt $x \in [0,1)$ zugewiesen und die Multimenge V_x aller Knoten v bestimmt mit $x \in I(v)$ (für $|I(v)|>1$ zählt v sooft wie I(v) x enthält).



SHARE

SHARE ist ein zweistufiges Verfahren.

2. Stufe: Für Datum d wird mittels konsistentem Hashing mit Hashfunktionen h' und g' (die für alle Punkte $x \in [0,1)$ gleich sind) ermittelt, welcher Knoten in V_x Datum d speichert.



SHARE

Effiziente Datenstruktur:

- 1. Stufe: verwende Hashtabelle wie für konsistentes Hashing, um alle möglichen Multimengen für alle Bereiche $[i/m,(i+1)/m)$ zu speichern.
- 2. Stufe: verwende separate Hashtabelle der Größe $\Theta(k)$ für jede mögliche Multimenge aus der 1. Stufe mit k Elementen (es gibt maximal $2n$ Multimengen, da es nur n Intervalle mit jeweils 2 Endpunkten gibt)

Laufzeit:

- 1. Stufe: $O(1)$ erw. Zeit zur Bestimmung der Multimenge
- 2. Stufe: $O(1)$ erw. Zeit zur Bestimmung des Knotens

SHARE

Theorem 4.12:

1. SHARE ist effizient.
2. Jeder Knoten i speichert im Erwartungswert c_i-Anteil der Daten, d.h. SHARE ist fair.
3. Bei jeder relativen Kapazitätsveränderung um $c \in [0,1)$ nur Umplatzierung eines erwarteten c-Anteils der Daten notwendig

Problem: Redundanz nicht einfach zu garantieren!

Lösung: SPREAD (erst dieses Jahr veröffentlicht)

SHARE

Beweis:

Punkt 2:

- $s=\Theta(\log n)$: Erwartete Anzahl Intervalle über jeden Punkt in $[0,1)$ ist s und Abweichung davon klein m.h.W.
- Knoten i hat Intervall der Länge $s \cdot c_i$
- Erwarteter Anteil Daten in Knoten i:

$$(s \cdot c_i) \cdot (1/s) = c_i$$

Phase 1 Phase 2

SHARE

Punkt 3:

- Betrachte Kapazitätsveränderung von $(c_1,\ldots,c_n)$ nach $(c'_1,\ldots,c'_n)$
- Differenz: $c = \sum_i |c_i - c'_i|$
- Optimale Strategie, die Fairness sicherstellt: Mindestaufwand $c/2$
- SHARE: Intervalldifferenz $\sum_i |s(c_i - c'_i)| = s \cdot c$
- Erwarteter Anteil umverteilter Daten: $(s \cdot c) / s = c$, also max. doppelt so viel wie optimal

Nächste Woche

Weiter mit Kapitel 5:
Sortieren und Selektieren