# Verteilte Algorithmen und Datenstrukturen

## Kapitel 6: Informationsorientierte Datenstrukturen

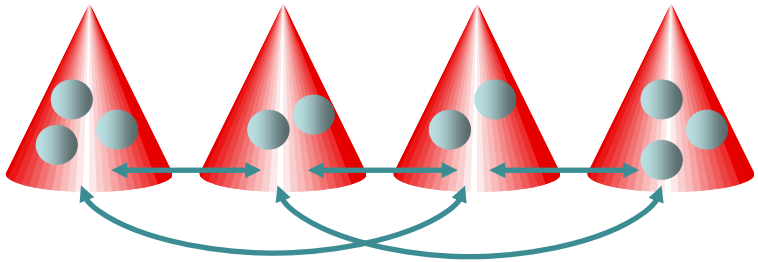
Prof. Dr. Christian Scheideler

SS 2016

# Informationsorientierte Datenstrukturen

- dynamische Menge an Ressourcen (Prozesse)
- dynamische Menge an Informationen (uniforme Datenobjekte)

Beispiel:



Operationen auf Prozessen:

- Join(v): neuer Prozess v kommt hinzu
- Leave(v): Prozess v verlässt das System

Operationen auf Daten: abhängig von Datenstruktur

# Informationsorientierte Datenstrukturen

Grundlegende Ziele:

- Monotone Stabilisierung der Prozessstruktur (so dass monotone Korrektheit aus beliebigem schwachen Zusammenhang heraus gewährleistet ist)
- Lokale Konsistenz für Datenzugriffe

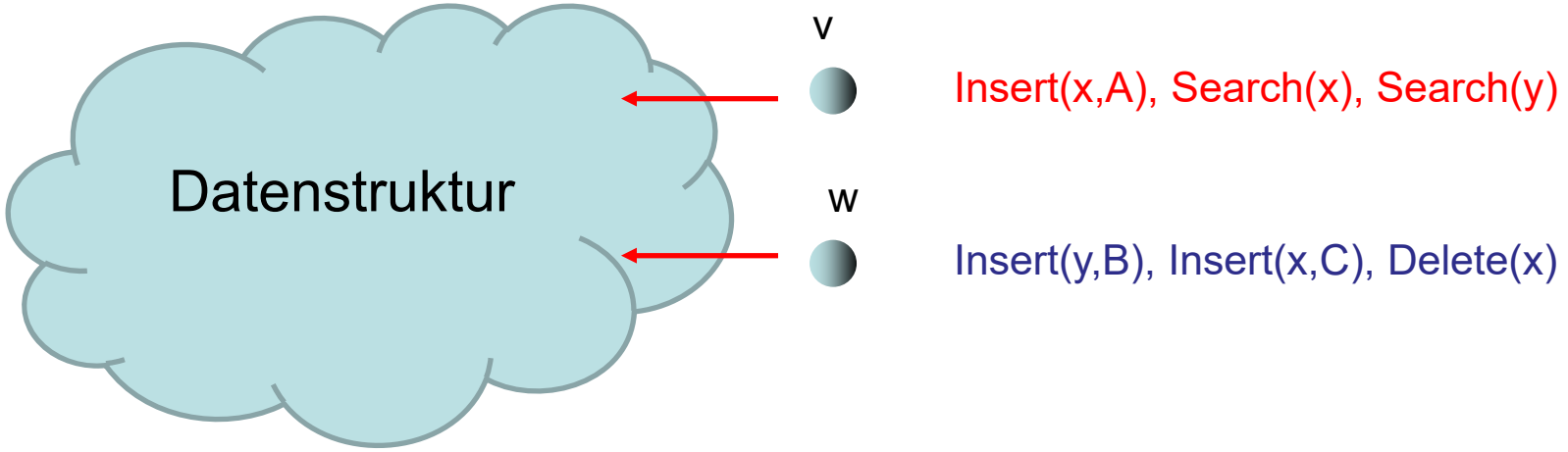
Zur Erinnerung:

Definition 3.10: Eine Linearisierung L(S) einer Rechnung S ist lokal konsistent, wenn für jeden Prozess v gilt, dass die Operationen, die von v initiiert werden, in derselben Reihenfolge in L(S) auftauchen, wie sie von v initiiert worden sind.

Wir wollen sicherstellen, dass für jeden Prozess v des Systems die Operationsfolge von v auf den Daten lokal konsistent ausgeführt wird. Das erlaubt es aber noch, dass die Operationsfolgen der Prozesse beliebig ineinander verzahnt werden dürfen.

# Informationsorientierte Datenstrukturen

Beispiel:




Eine lokal konsistente Ausgabe für v wäre C, B, da diese durch

Insert(x,A), Insert(y,B), Insert(x,C) Search(x), Search(y)

zustande käme.

# Informationsorientierte Datenstrukturen

Beobachtung: monotone Korrektheit ist im Allgemeinen wichtig für lokale Konsistenz.

Begründung:

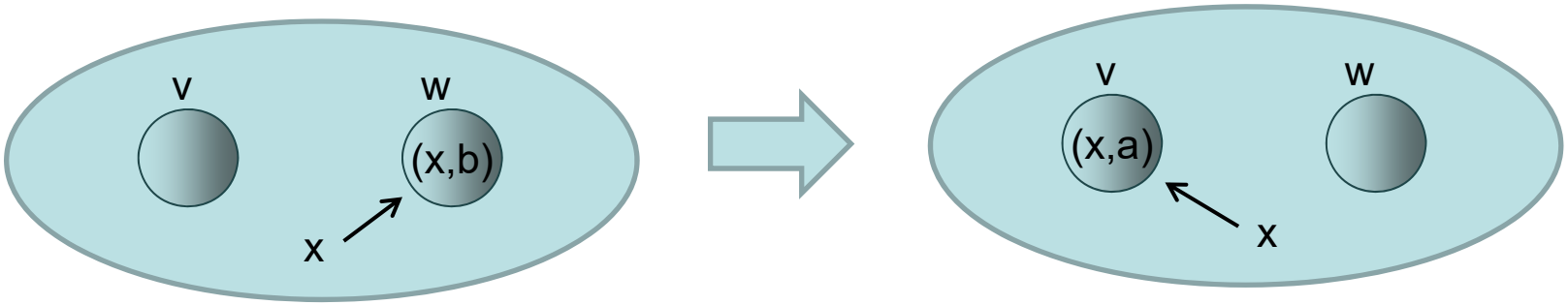
- Betrachte folgende Anfragefolge von Prozess v: push(x), push(y), pop(), pop()
- Falls x und y nicht monoton suchbar wären, könnte es passieren, dass das erste pop() x zurückliefert (da y nicht gefunden wird) während das zweite pop() y zurückliefert, selbst wenn v abwartet, bis push(x) und push(y) abgeschlossen ist (d.h. x und y ihre richtige Position im verteilten Stack erreicht haben)
- Problem: ohne monotone Suchbarkeit könnte es (z.B. durch spätere push oder pop Anfragen) dazu kommen, dass gewisse Elemente temporär nicht suchbar sind.

# Informationsorientierte Datenstrukturen

Problem: monotone Korrektheit nicht einfach sicherzustellen

Mögliche Lösung (Suchstruktur):

- „Capture the Flag“: Wenn Prozess v insert(x,a) ausführt, dann speichert v lokal (x,a), und das x in der verteilten Datenstruktur verweist dann auf dieses (x,a).

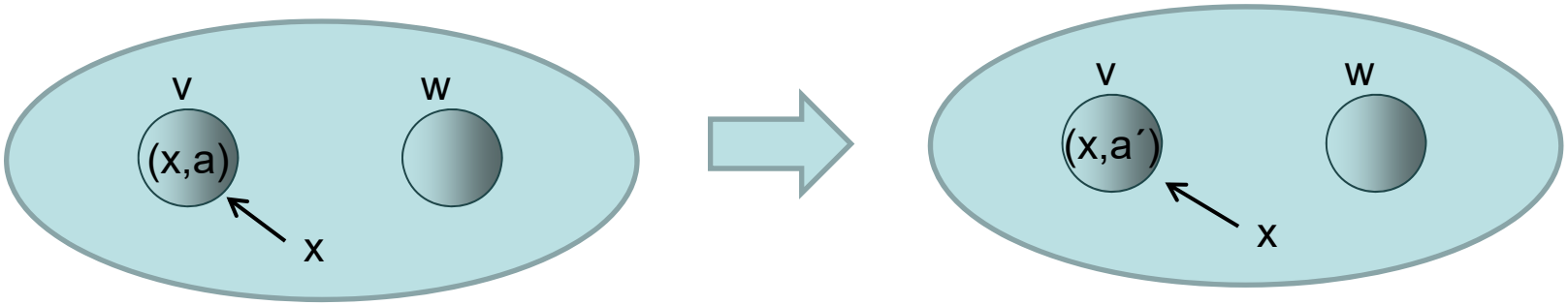


# Informationsorientierte Datenstrukturen

Problem: monotone Korrektheit nicht einfach sicherzustellen

Mögliche Lösung (Suchstruktur):

- „Capture the Flag“: Wenn Prozess v insert(x,a) ausführt, dann speichert v lokal (x,a), und das x in der verteilten Datenstruktur verweist dann auf dieses (x,a). Ein späteres insert(x,a´) überschreibt dann einfach lokal das a mit a´, sofern (x,a) noch in v ist.



# Informationsorientierte Datenstrukturen

Problem: selbst wenn monotone Korrektheit gewährleistet ist, muss auf die Reihenfolge der Ausführungen von Operationen geachtet werden.

Beispiel: bei push(x), push(y), pop() muss für pop() y zurückgegeben werden, ist aber y erst nach x suchbar, könnte trotz monotoner Suchbarkeit für pop() x ausgegeben werden.

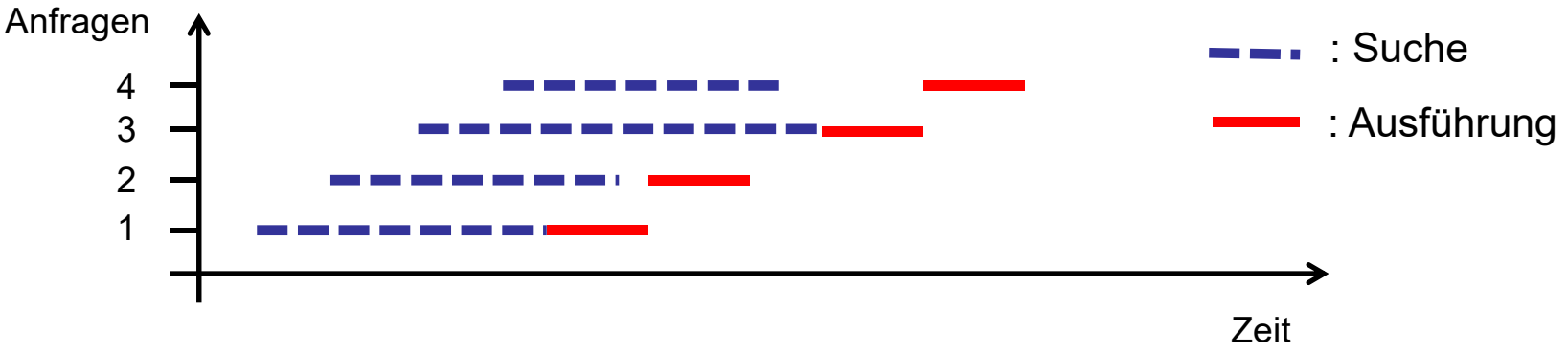
Mögliche Strategie für lokale Konsistenz falls monotone Korrektheit gegeben:

- Lokal sequentielle Ausführung: Quelle startet erst dann neue Anfrage, wenn all ihre vorigen Anfragen abgeschlossen sind.

# Informationsorientierte Datenstrukturen

## Lokal sequentielle Ausführung:

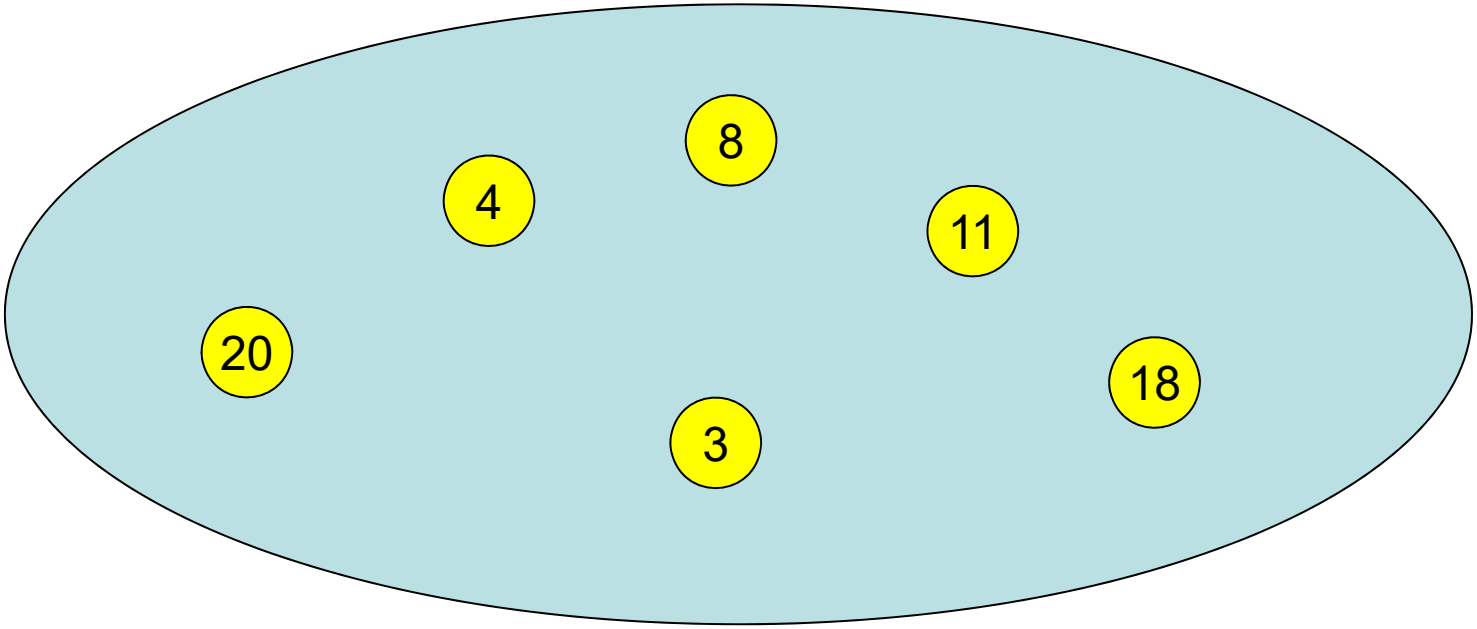
- Stelle für jede Datenanfrage zunächst (über Lookup) fest, mit welchen Prozessen Daten ausgetauscht werden. Das kann oft parallel geschehen.
- Führe dann den Datenaustausch sequentiell in der vorgegebenen Reihenfolge aus.



# Übersicht

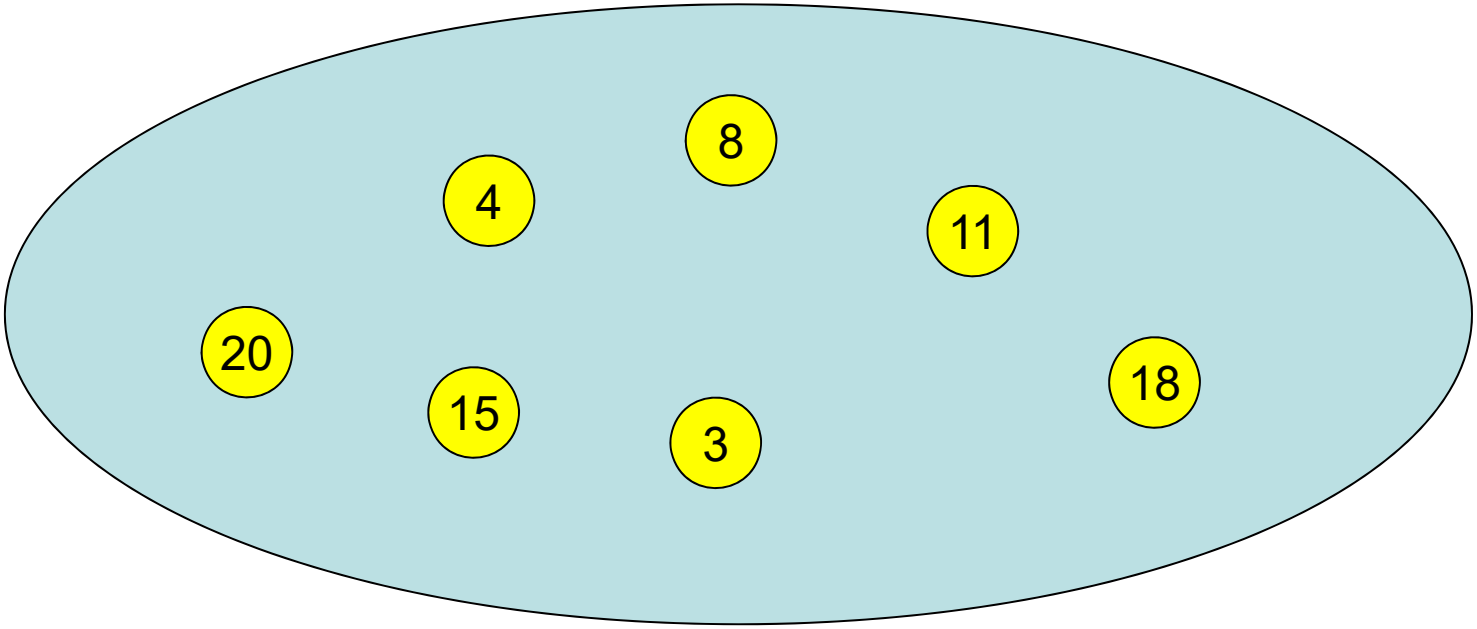
- Verteilte Hashtabelle
- Verteilte Queue
- Verteilter Stack
- Verteilter Heap

# Wörterbuch



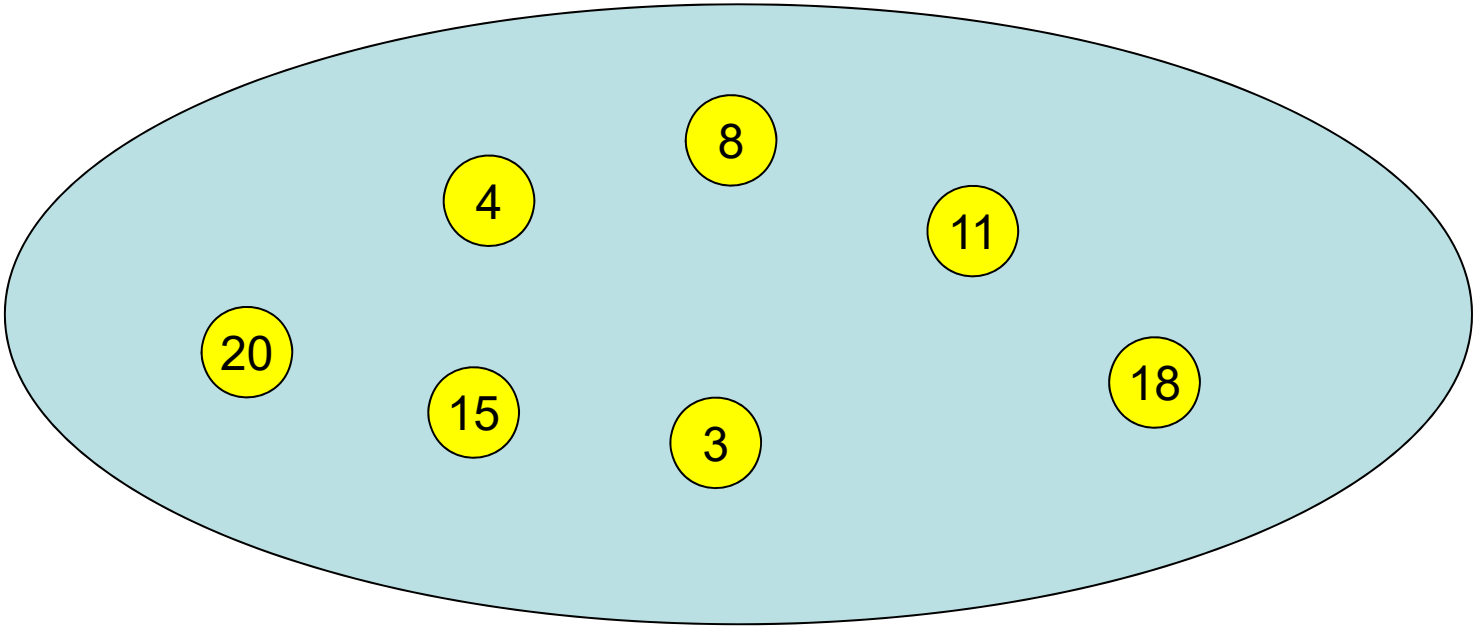
# Wörterbuch

insert(15)



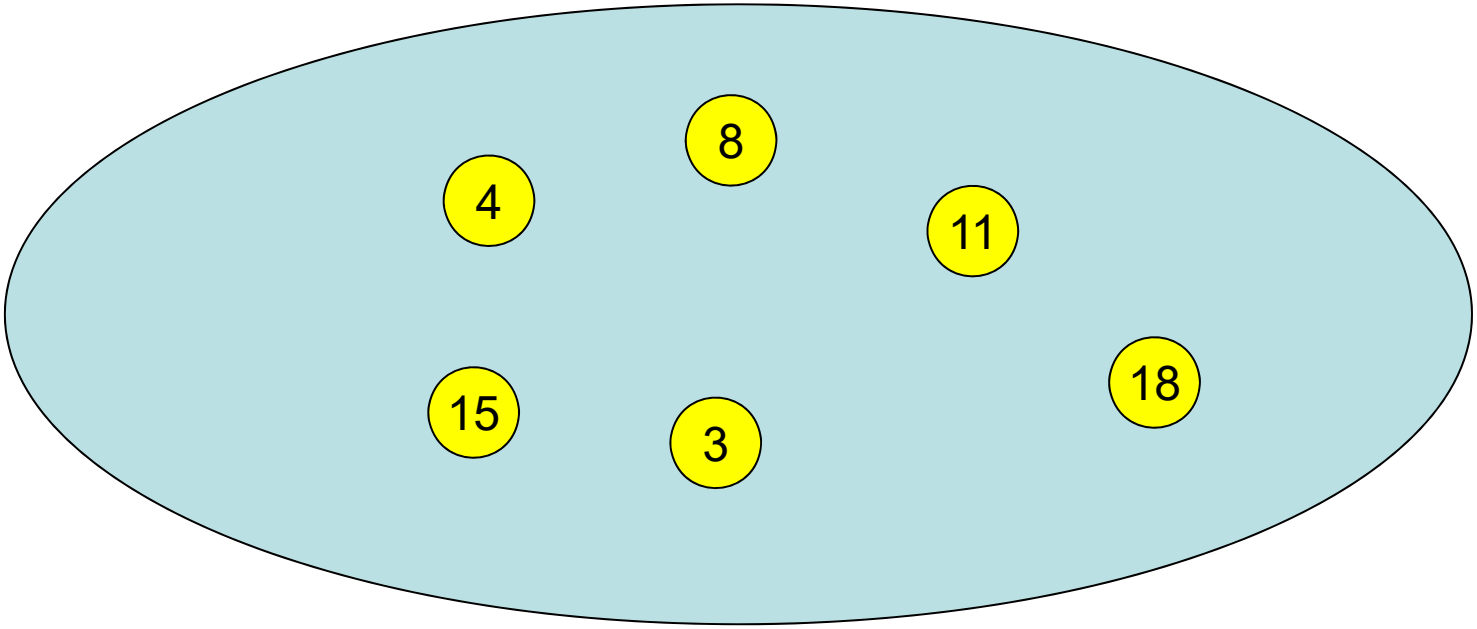
# Wörterbuch

delete(20)



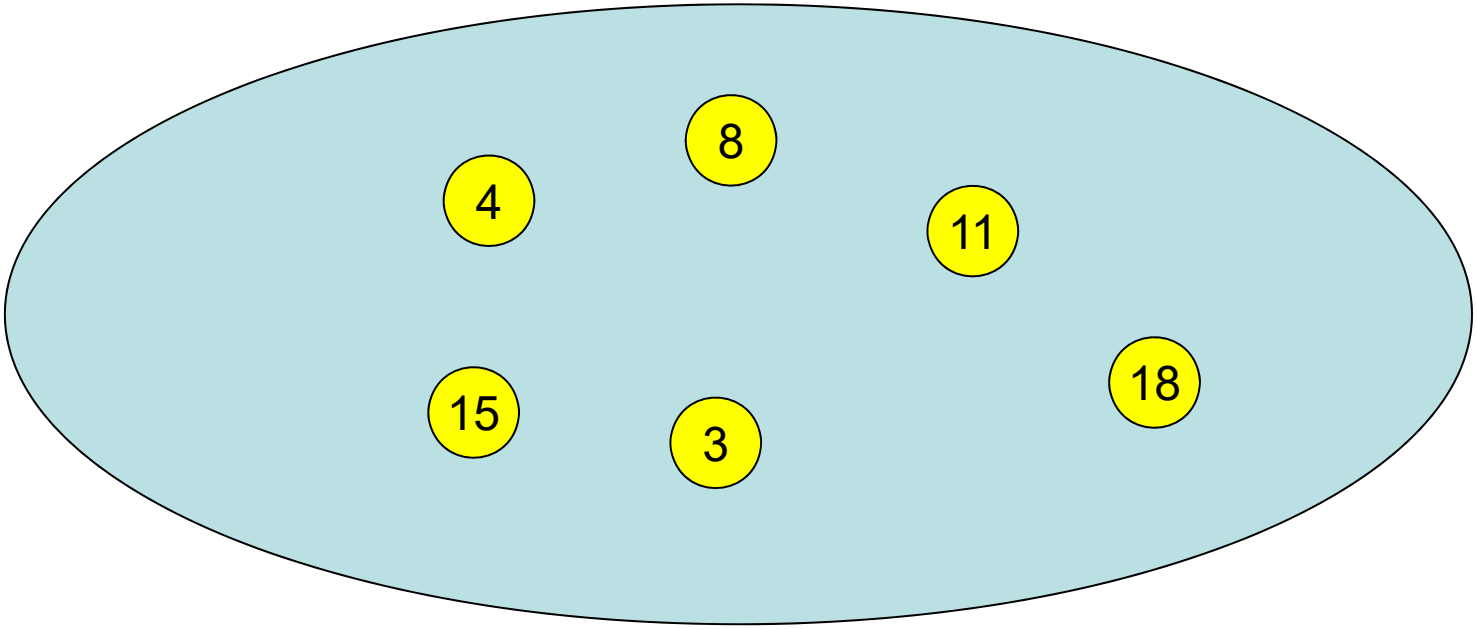
# Wörterbuch

lookup(8) ergibt 8



# Wörterbuch

lookup(7) ergibt ?



# Wörterbuch-Datenstruktur

S: Menge von Elementen

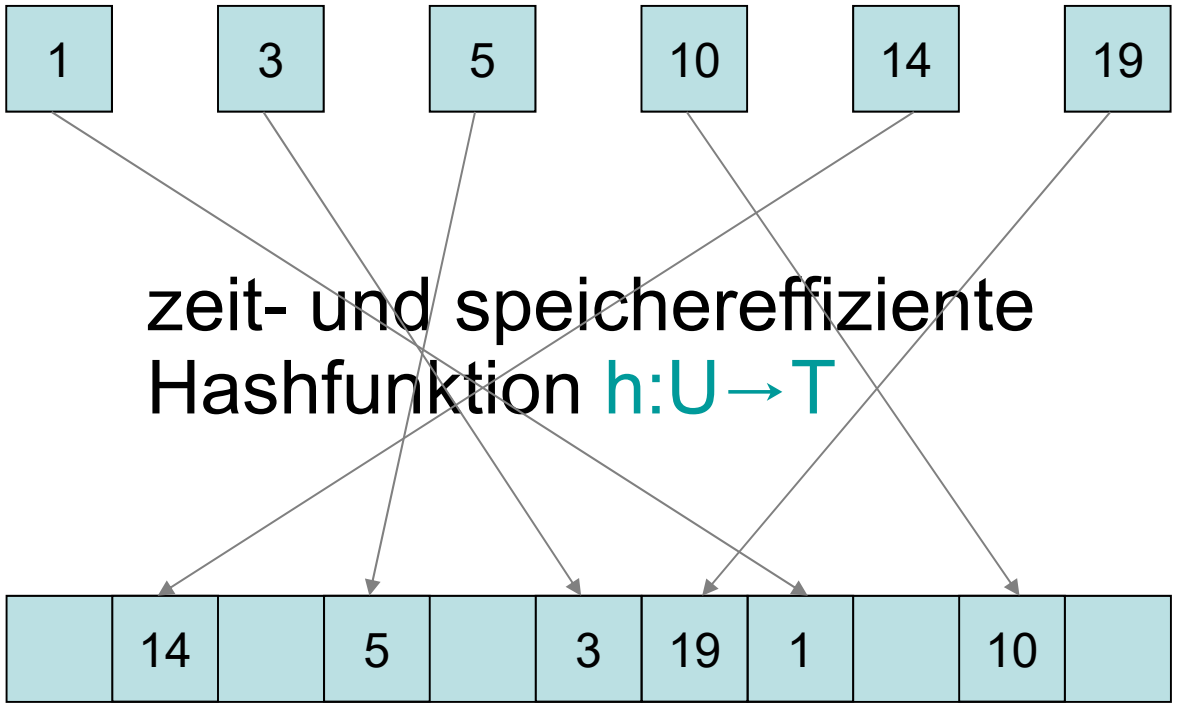
Jedes Element e identifiziert über key(e).

Operationen:

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

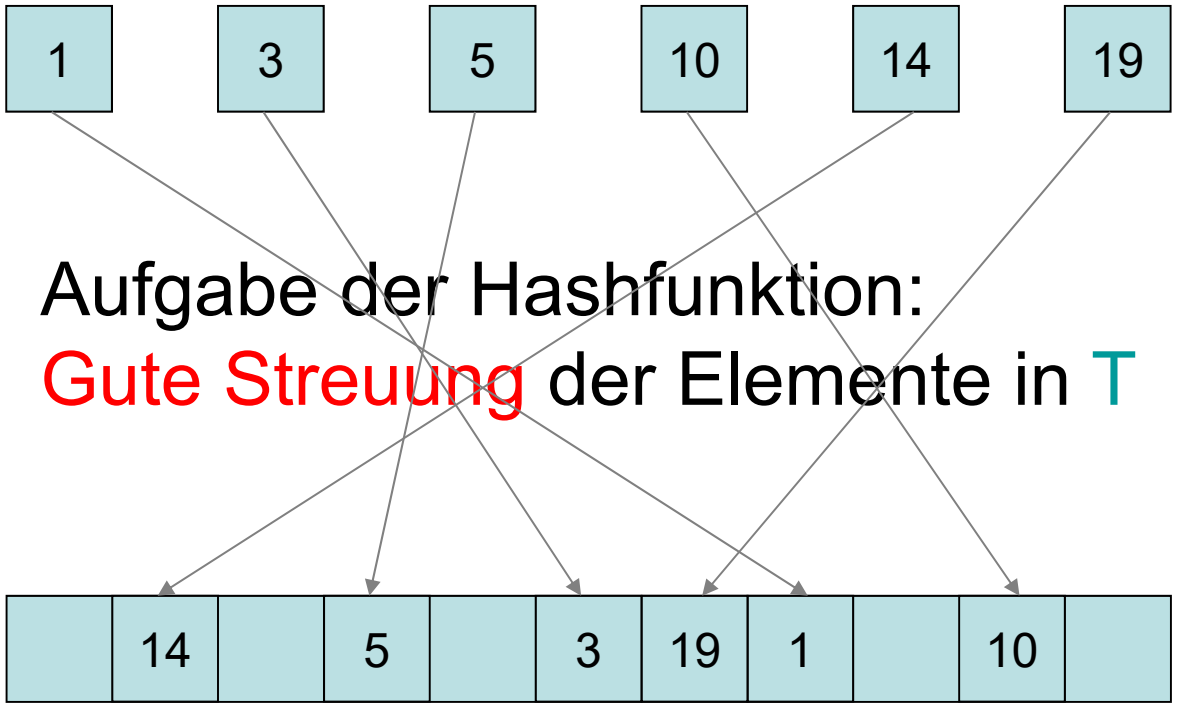
Effiziente Lösung: Hashing

# Klassisches Hashing



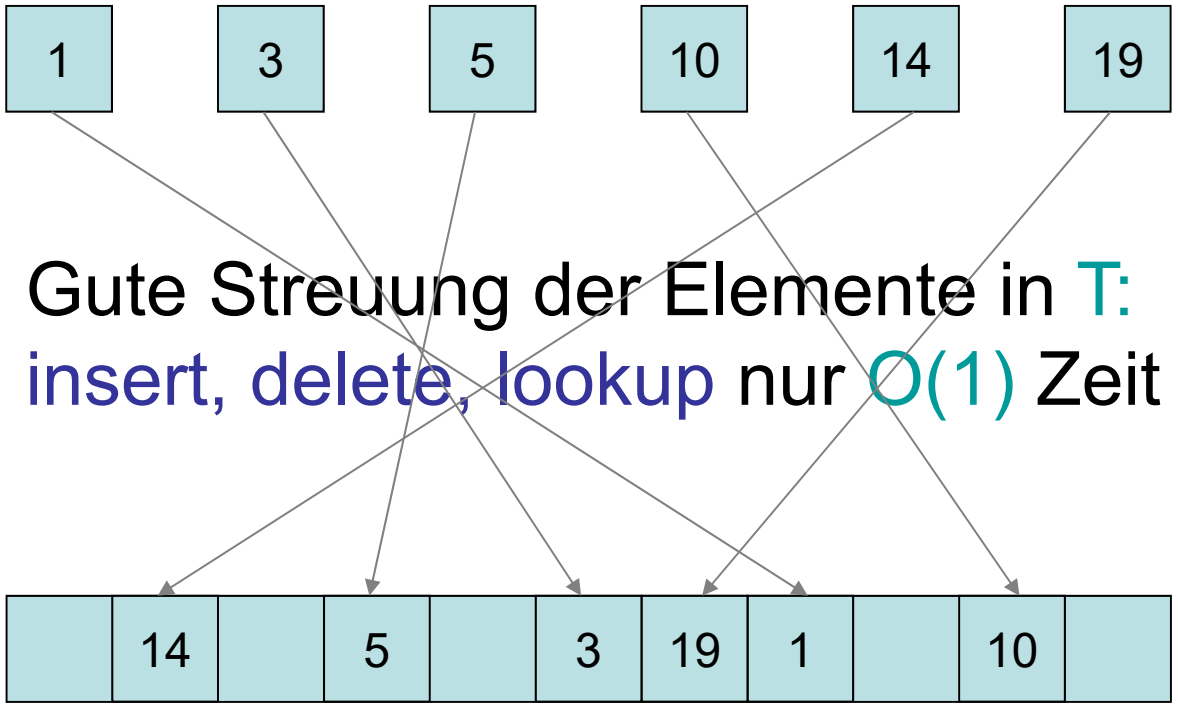

Hashtabelle T

# Klassisches Hashing




Hashtabelle T

# Klassisches Hashing

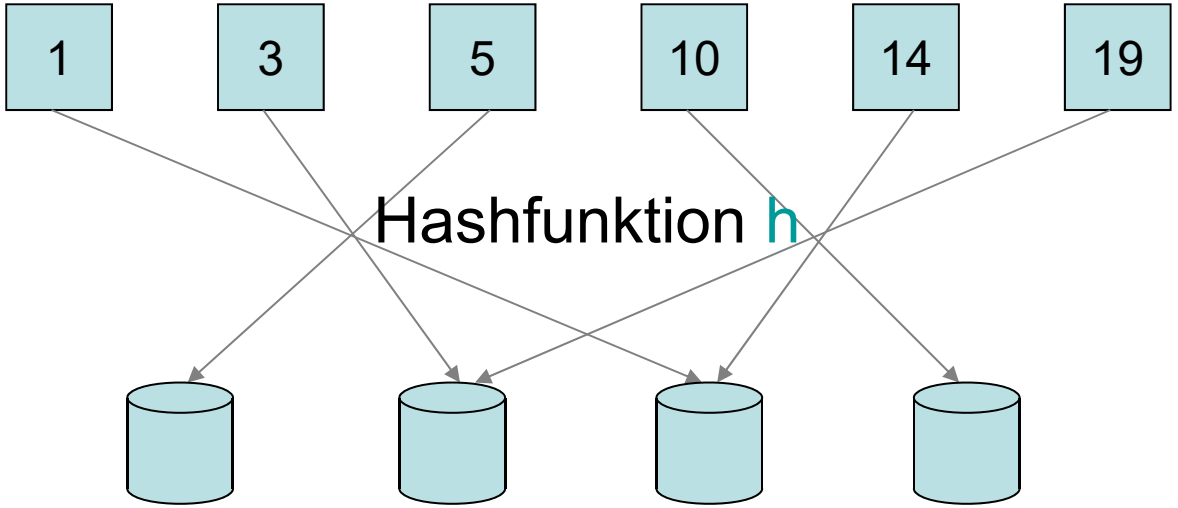



Gute Streuung der Elemente in T:
insert, delete, lookup nur O(1) Zeit

Hashtabelle T

# Verteiltes Hashing

Hashing auch für verteilte Speicher anwendbar:




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

# Verteiltes Wörterbuch

Grundlegende Operationen:

- insert(d): fügt Datum d mit Schlüssel key(d) ein (wodurch eventuell der alte zu key(d) gespeicherte Inhalt überschrieben wird)
- delete(k): löscht Datum d mit key(d)=k
- lookup(k): gibt Datum d zurück mit key(d)=k
- join(v): Prozess (Speicher) v kommt hinzu
- leave(v): Prozess 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 Prozess (Speicher) hat dieselbe Kapazität.

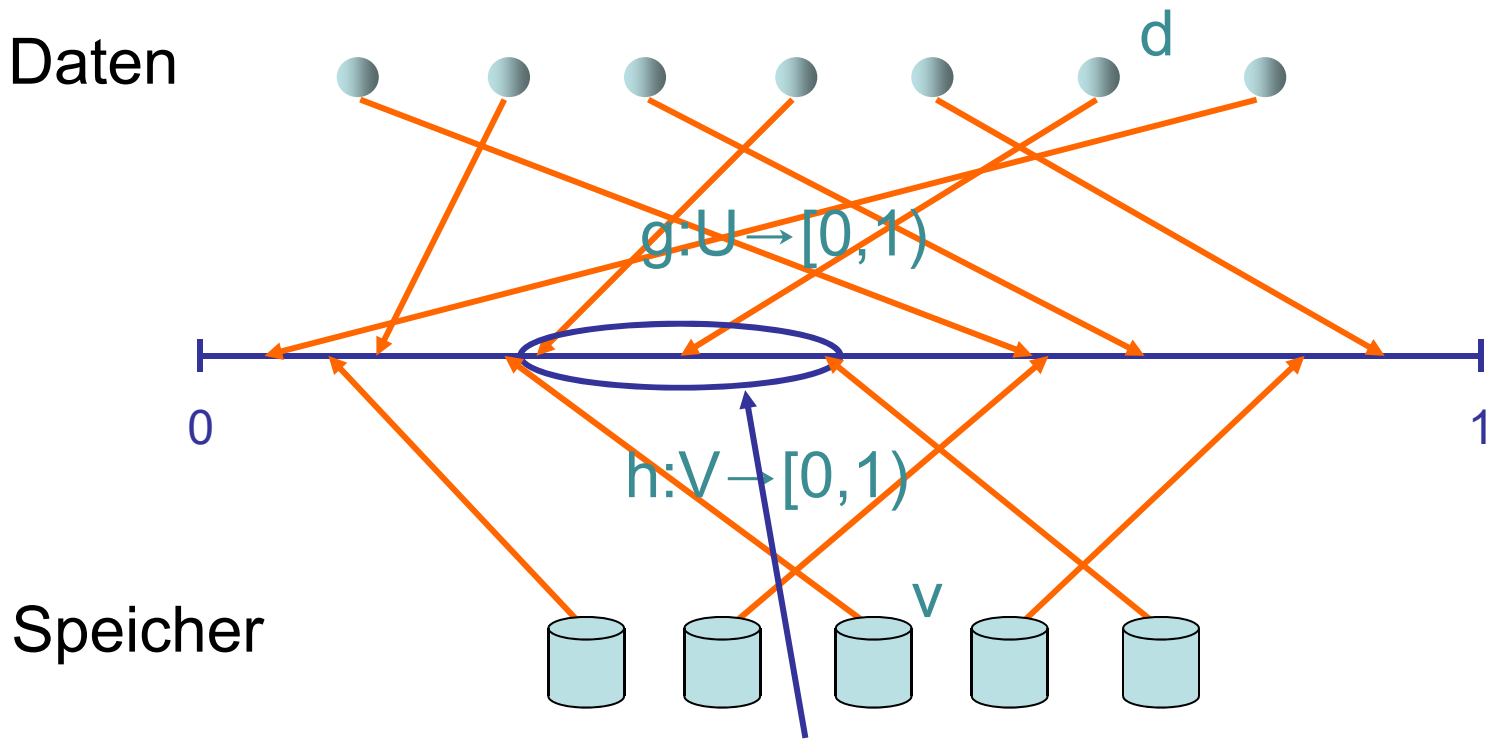
Nichtuniforme Speichersysteme: Kapazitäten können beliebig unterschiedlich sein

Vorgestellte Strategien:

- Uniforme Systeme: konsistentes Hashing
- Nichtuniforme Speichersysteme: SHARE
- Combine & Split

# Konsistentes Hashing

Wähle zwei zufällige Hashfunktionen h, g




Region, für die Speicher v zuständig ist

# Konsistentes Hashing

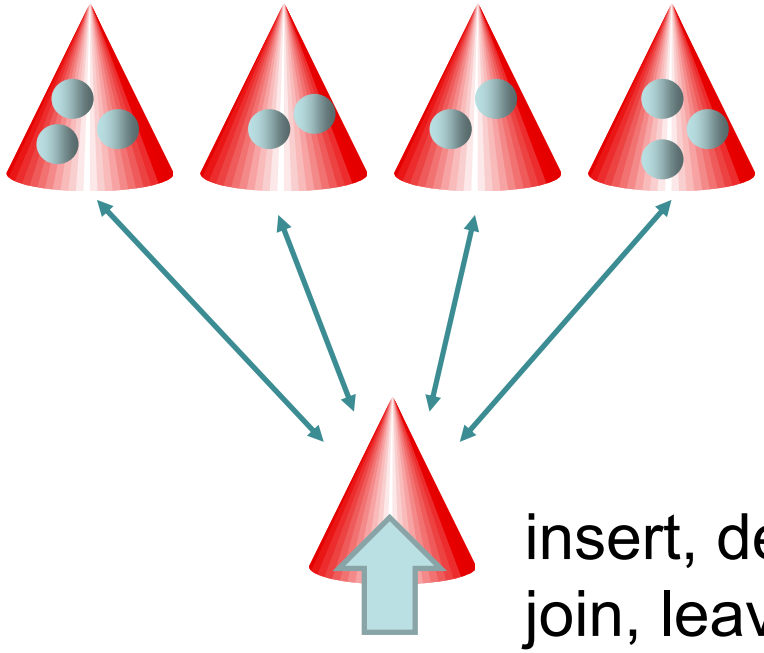
- $V$: aktuelle Prozessmenge (im folgenden auch Knoten genannt)
- $succ(v)$: nächster Nachfolger von $v$ in $V$ bzgl. Hashfunktion $h$ (wobei $[0,1)$ als Kreis gesehen wird)
- $pred(v)$: nächster Vorgänger von $v$ in $V$ bzgl. Hashfunktion $h$

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. $h$

# Verteilte Hashtabelle

Fall 1: Server verwaltet Speicherknoten

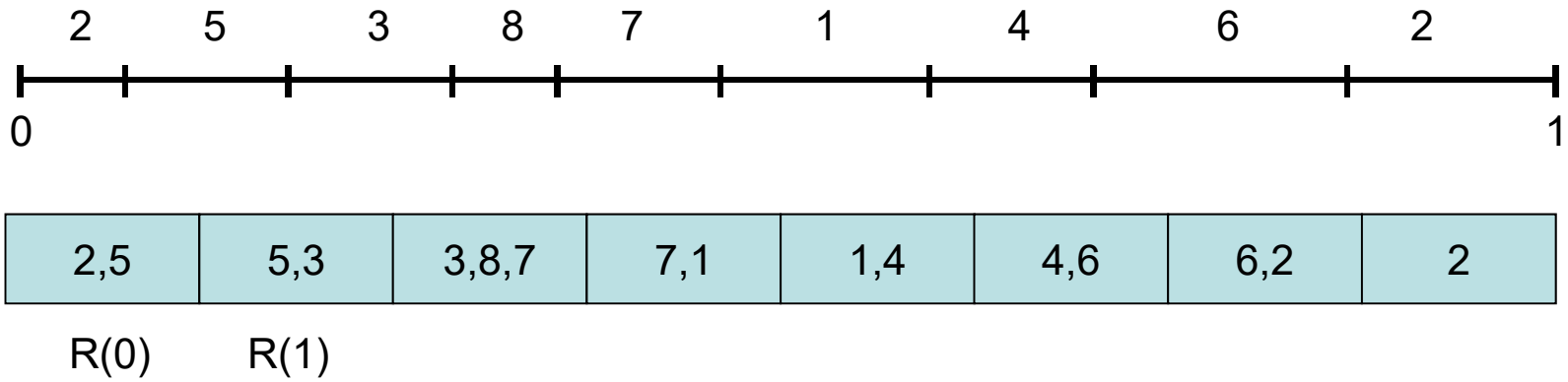


insert, delete, lookup,
join, leave

# Verteilte Hashtabelle, Fall 1

Effiziente Datenstruktur für Server:

- Verwende interne 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 Speicherknoten $v$ mit $I(v)\cap R(i) \neq \emptyset$.



# Verteilte Hashtabelle, Fall 1

Effiziente Datenstruktur für Server:

- 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 Speicherknoten v mit $I(v) \cap R(i) \neq \emptyset$.
- Lookup(k): ermittle das R(i), das g(k) enthält, und bestimme dasjenige v in T[i], dessen h(v) der nächste Vorgänger von g(k) ist. Dieses v ist dann verantwortlich für k und erhält die lookup Anfrage.

| 2,5 | 5,3 | 3,8,7 | 7,1 | 1,4 | 4,6 | 6,2 | 2 |
|---|---|---|---|---|---|---|---|
| R(0) | R(1) | | | | | | |

# Verteilte Hashtabelle, Fall 1

Effiziente Datenstruktur für Server:

- 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 Speicherknoten v mit $I(v) \cap R(i) \neq \emptyset$.
- Insert(d): ermittle zunächst wie bei Lookup dasjenige v, das für d verantwortlich ist und leite dann Insert(d) an dieses v weiter.
- Delete(k): analog

| 2,5 | 5,3 | 3,8,7 | 7,1 | 1,4 | 4,6 | 6,2 | 2 |
|---|---|---|---|---|---|---|---|
| R(0) | R(1) | | | | | | |

# Verteilte Hashtabelle, Fall 1

Effiziente Datenstruktur für Server:

- Verwende interne 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 Speicherknoten $v$ mit $I(v)\cap R(i)\neq\emptyset$.

Einfach zu zeigen:

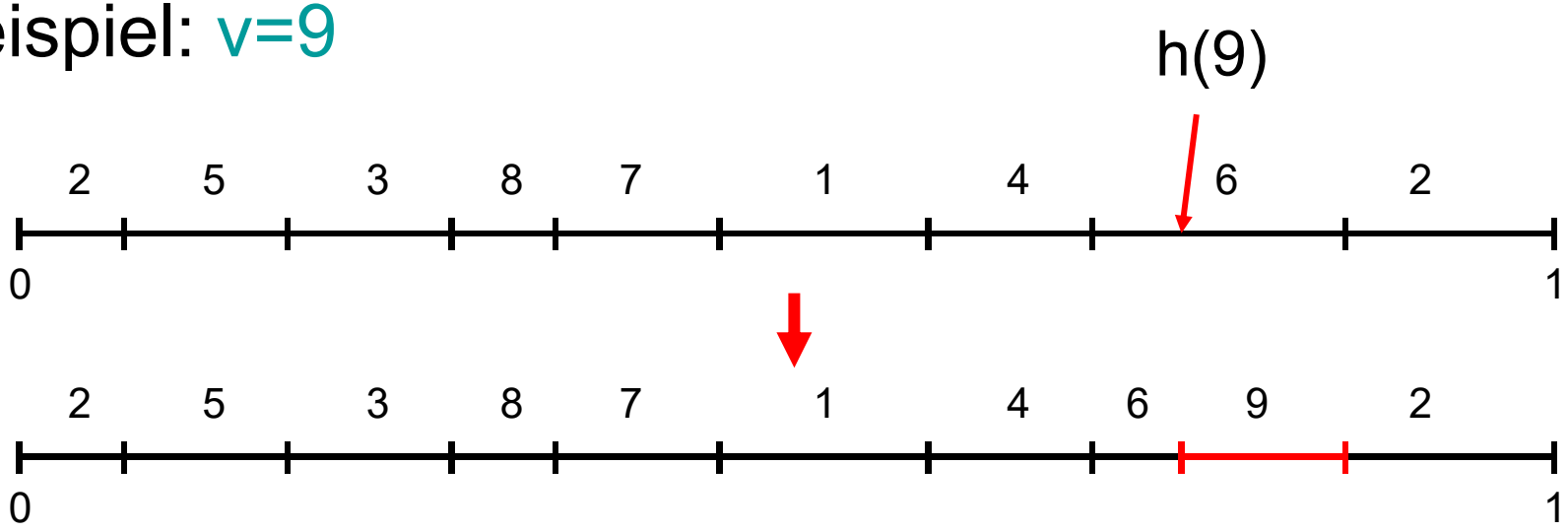
- $T[i]$ enthält für $m\geq n$ erwartet konstant viele Elemente und höchstens $O(\log n / \log \log n)$ mit hoher W.keit.
- D.h. Lookup(k) (die Ermittlung des zuständigen Knotens für einen Schlüssel) benötigt erwartet konstante Zeit

# Verteilte Hashtabelle, Fall 1

Operationen:

- join(v): ermittle Intervall für v (durch Zugriff auf T) und informiere Vorgänger von v, Daten, die nun v gehören, zu v zu leiten

Beispiel: v=9

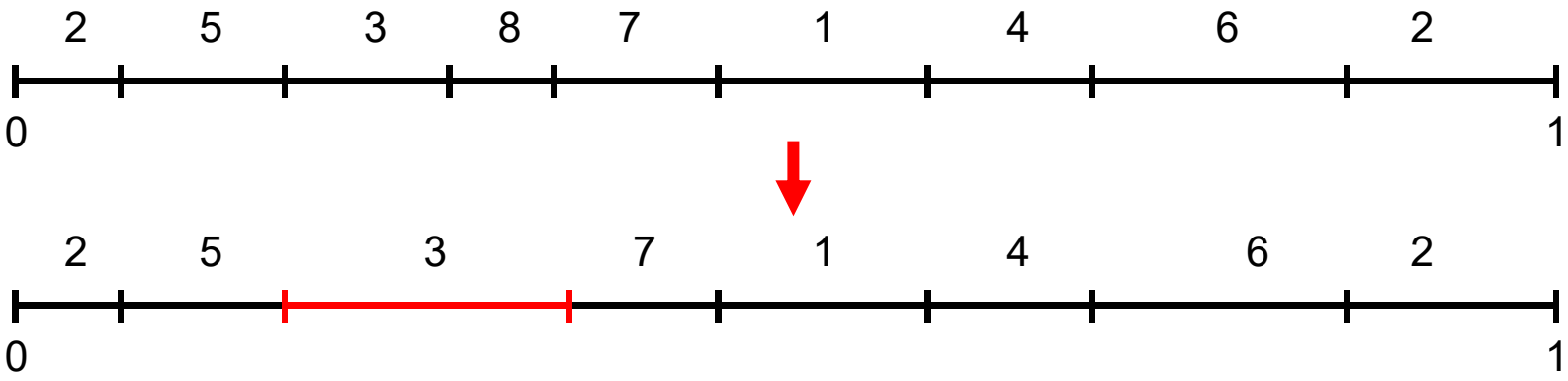


# Verteilte Hashtabelle, Fall 1

Operationen:

- leave(v): errechne über T Knoten w, der Intervall von v beerbt, und weise v an, alle Daten an w zu leiten

Beispiel: v=8



# Verteilte Hashtabelle, Fall 1

Satz 6.1:

- 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

Beweis:

Effizienz und Redundanz: siehe Protokoll

Fairness:

- Für jede Wahl von $h$ gilt, dass $\Sigma_{v \in V} |I(v)| = 1$ und damit $\Sigma_{v \in V} E[|I(v)|] = 1$ (E[]: Erwartungswert).
- Angenommen, wir verwenden eine Klasse von Hashfunktionen $H$, so dass für jedes Paar $v,w \in V$ eine Bijektion $f:H \rightarrow H$ auf der Menge $H$ existiert, so dass für alle $h \in H$, ($|I(v)|$ bzgl. $h$) = ($|I(w)|$ bzgl. $f(h)$). Wenn wir aus $H$ eine Hashfunktion uniform zufällig auswählen, dann gilt, dass $E[|I(v)|] = E[|I(w)|]$.
- Die Kombinierung der beiden Gleichungen ergibt, dass $E[|I(v)|] = 1/n$ für alle $v \in V$.

# Verteilte Hashtabelle, Fall 1

Satz 6.1:

- 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

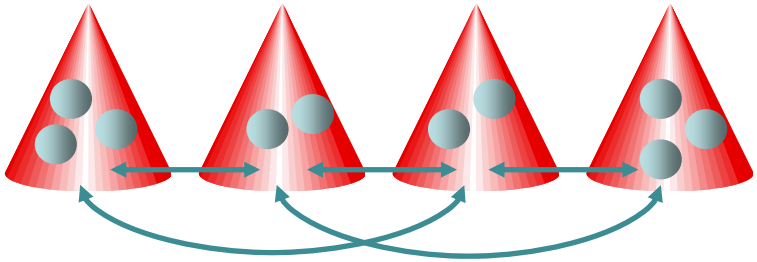
Problem: Schwankung um 1/n hoch!

Mögliche Lösungen:

- zwei alternative Knoten pro Datum über zwei zufällige Hashfunktionen, speichere Datum immer im Ort mit geringerer Last (wird in timeouts überprüft)
- kombiniere konsistentes Hashing mit Linear Probing, d.h. ein Datum wird solange weitergereicht, bis ein Knoten mit weniger als $c \cdot m/n$ Last für eine Konstante $c>1$ gefunden wird, wobei $m$ die aktuelle Anzahl der Daten ist

# Verteilte Hashtabelle

## Fall 2: verteiltes System

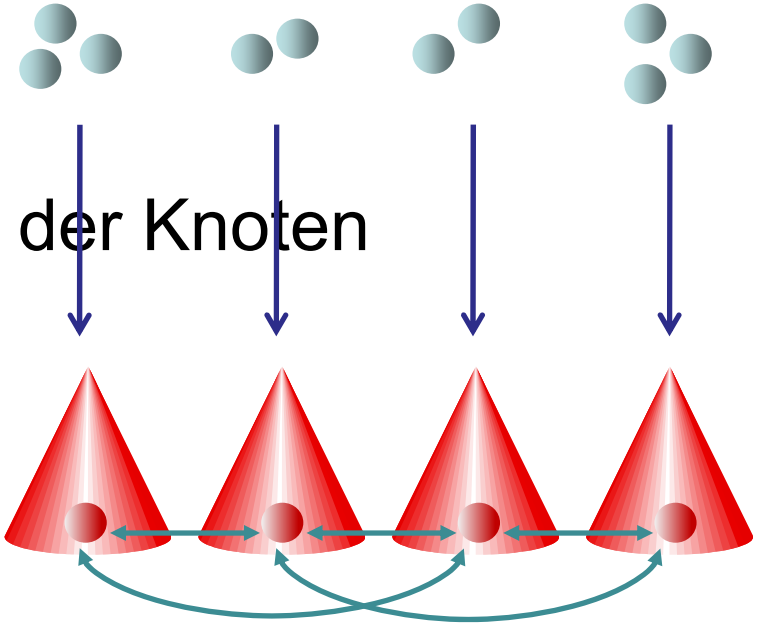


Jeder Knoten kann Anfragen (insert, delete, lookup, join, leave) generieren.

# Verteilte Hashtabelle, Fall 2

Konsistentes Hashing: Zerlegung in zwei Probleme.

1. Abbildung der Daten auf die Knoten

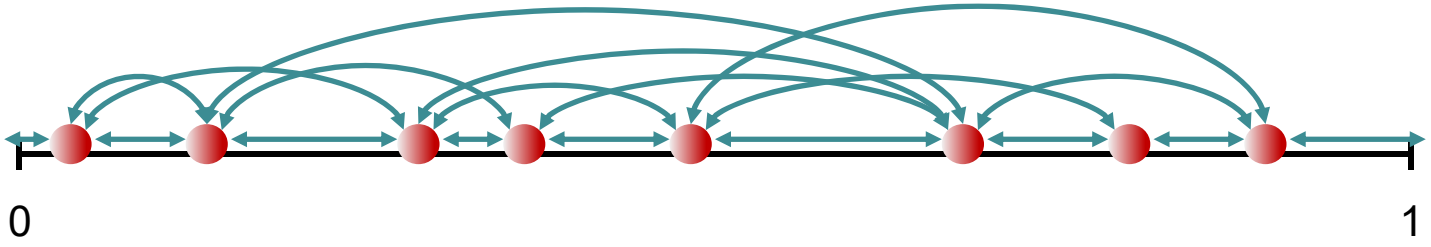


2. Vernetzung der Knoten

# Verteilte Hashtabelle, Fall 2

## Vernetzung der Knoten:

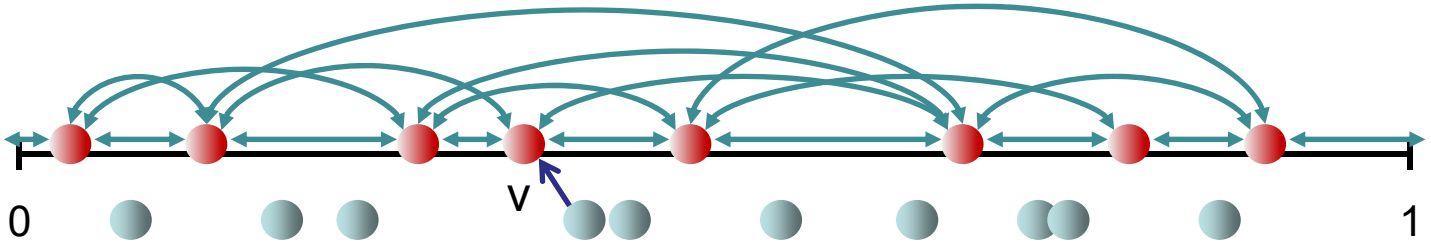
- Jedem Knoten $v$ wird (pseudo-)zufälliger Wert $h(v) \in [0,1)$ zugewiesen.
- Verwende z.B. Skip+ Graph, um Knoten (hier im Kreis!) mittels $h(v)$ zu vernetzen.



# Verteilte Hashtabelle, Fall 2

Abbildung der Daten auf die Knoten:

- Verwende konsistentes Hashing




- insert(d): führe search(g(key(d))) im Skip+ Graph aus und speichere d im nächsten Vorgänger von g(key(d)) im Skip+ Graph

# Verteilte Hashtabelle, Fall 2

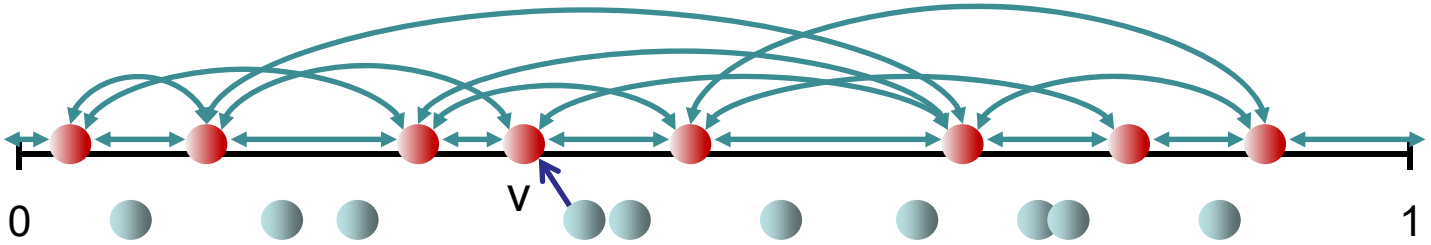
Passende search-Operation im Skip+ Graph für den Einsatz in der verteilten Hashtabelle:

```
search(x∈[0,1]) →
    { ausgeführt in Knoten u }
    if x∉[h(pred(u)),h(succ(u))] then
        { N(u): Nachbarschaft von u
          succ(u): nächster Nachfolger von u bzgl. h in N(u)
          pred(u): nächster Vorgänger von u bzgl. h in N(u) }
        v = Knoten in N(u), der am nächsten zu x liegt,
            ohne dass x übersprungen wird
        v←search(x)
    else
        if x<h(u) then
            pred(u)←search(x)  { hier evtl. Kreiskante notwendig }
        { sonst ist search Request beim Ziel }
```

# Verteilte Hashtabelle, Fall 2

Abbildung der Daten auf die Knoten:

- Verwende konsistentes Hashing

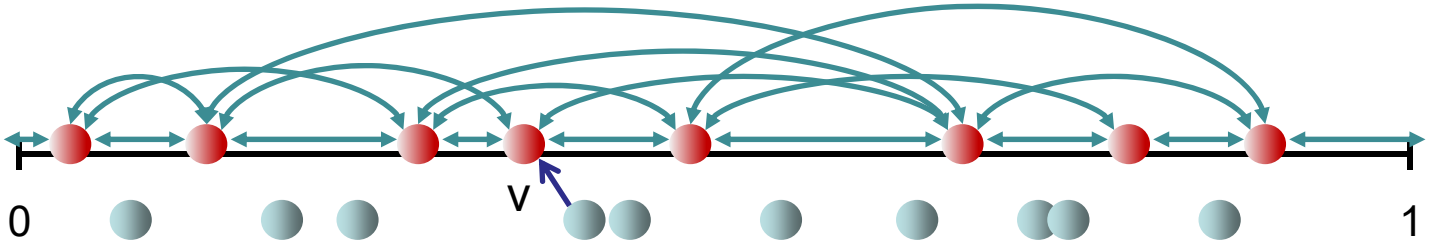



- delete(k): führe search(g(k)) im Skip+ Graph aus und lösche Datum d mit key(d)=k (falls da) im nächsten Vorgänger von g(k)

# Verteilte Hashtabelle, Fall 2

Abbildung der Daten auf die Knoten:

- Verwende konsistentes Hashing

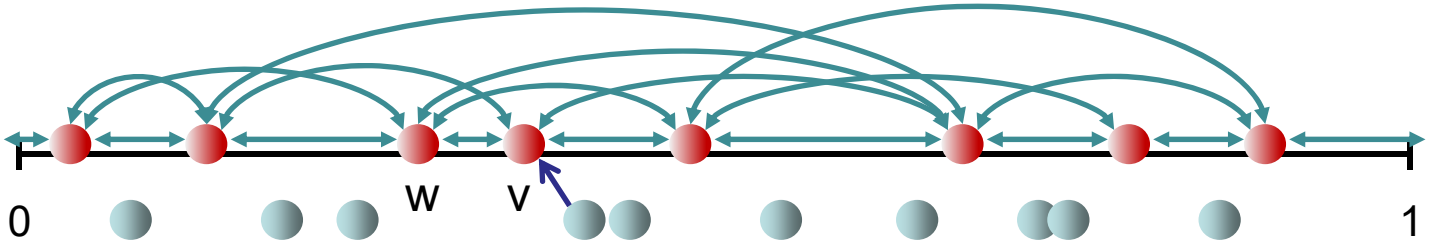



- lookup(k): führe search(g(k)) im Skip+ Graph aus und liefere Datum d mit key(d)=k (falls da) im nächsten Vorgänger von g(k) zurück

# Verteilte Hashtabelle, Fall 2

Abbildung der Daten auf die Knoten:

- Verwende konsistentes Hashing

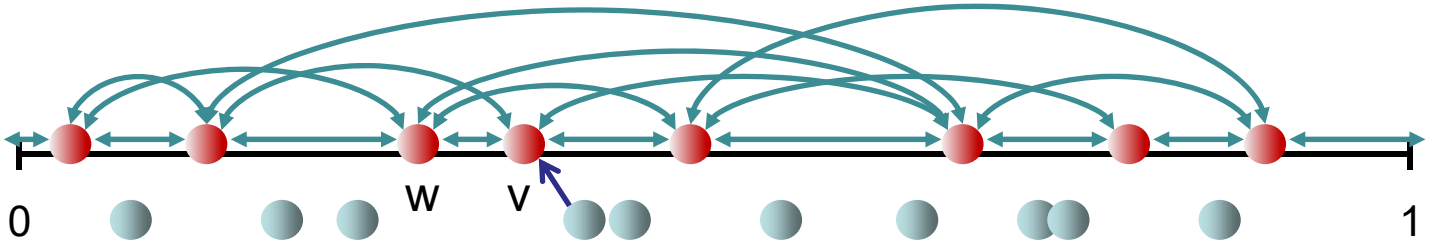



- join(v): nachdem v in den Skip+ Graph integriert ist, reicht es, pred(v)=w zu kontaktieren (mit welchem v direkt verbunden ist), um alle für v relevanten Daten gemäß des konsistenten Hashings zu erhalten

# Verteilte Hashtabelle, Fall 2

Abbildung der Daten auf die Knoten:

- Verwende konsistentes Hashing




- leave(v): hier reicht es (neben der Entfernung von v aus dem Skip+ Graphen), dass v all seine Daten an pred(v)=w (mit dem v direkt verbunden ist) weitergibt, so die Datenzuordnung wieder korrekt ist

# Verteilte Hashtabelle, Fall 2

Satz 6.2: Im stabilen Zustand ist der Arbeitsaufwand für die Operationen ohne den Datenaustausch

- Insert(d): erwartet $O(\log n)$
- Delete(k): erwartet $O(\log n)$
- Lookup(k): erwartet $O(\log n)$
- Join(v): $O(\log^2 n)$ (verbinde v mit beliebigem Knoten im Skip+ Graph, Rest durch Build-Skip)
- Leave(v): $O(\log^2 n)$ (verlasse Skip+ Graph, Rest durch Build-Skip)

Beweis:

Folgt aus Analyse des Skip+ Graphen

# Verteilte Hashtabelle, Fall 2

Selbststabilisierung:

- Selbststabilisierender Skip+ Graph: gelöst
- Selbststabilisierende Datenplazierung: bewege Daten in timeout Aktion analog zur search Operation, falls der Ort des Datums falsch ist.

Lokale Konsistenz: „Capture the Flag“ mit lokal sequentieller Ausführung kombinieren.

Übung: kläre die Details dazu (z.B. Vorsicht bei mehreren Insert Anfragen auf dasselbe Datum).

# Verteiltes Wörterbuch

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

Nichtuniforme Speichersysteme: Kapazitäten können beliebig unterschiedlich sein

Vorgestellte Strategien:

- Uniforme Systeme: konsistentes Hashing
- Nichtuniforme Speichersysteme: SHARE
- Combine & Split

# SHARE

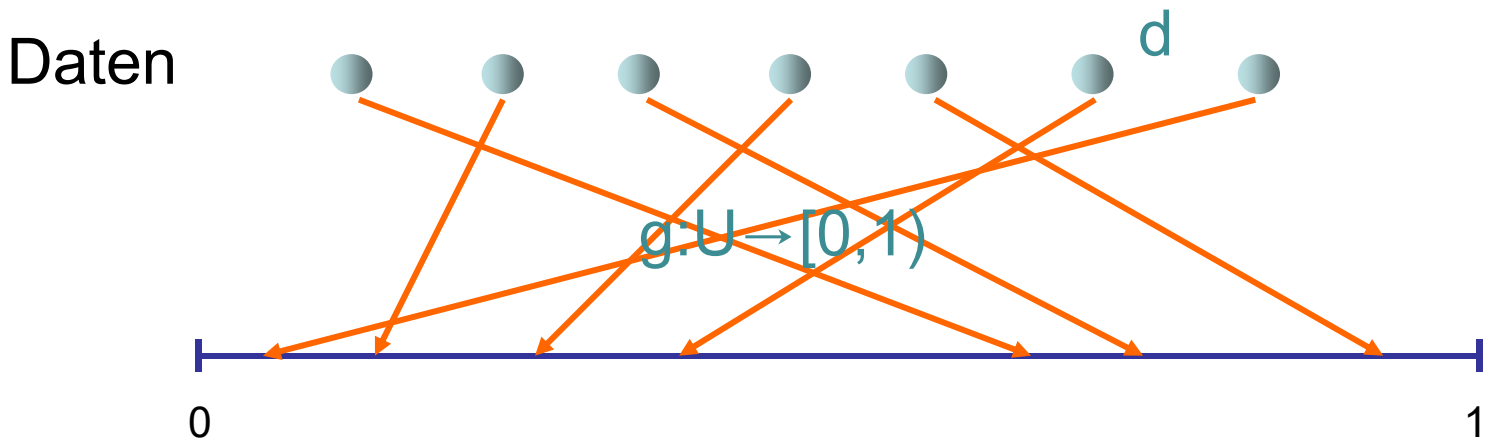
Situation hier: wir haben Knoten mit beliebigen relativen Kapazitäten $c_1,\ldots,c_n$, d.h. $\sum_i c_i = 1$.

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

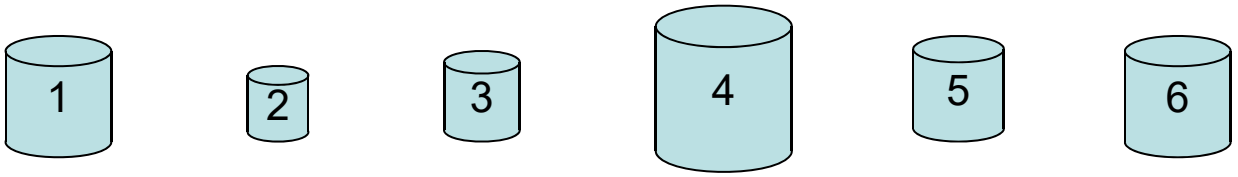
Lösung: SHARE

# SHARE

Datenabbildung: wie bei konsistentem Hashing



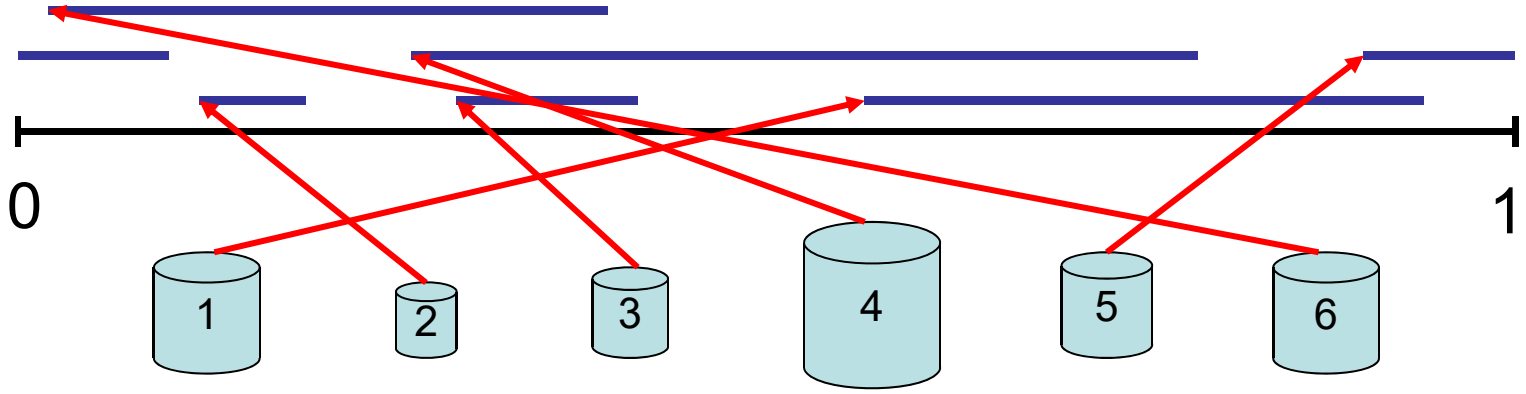

Zuordnung der Speicher: andere Strategie



# SHARE

Zuordnung zu Speichern: zweistufiges Verfahren.

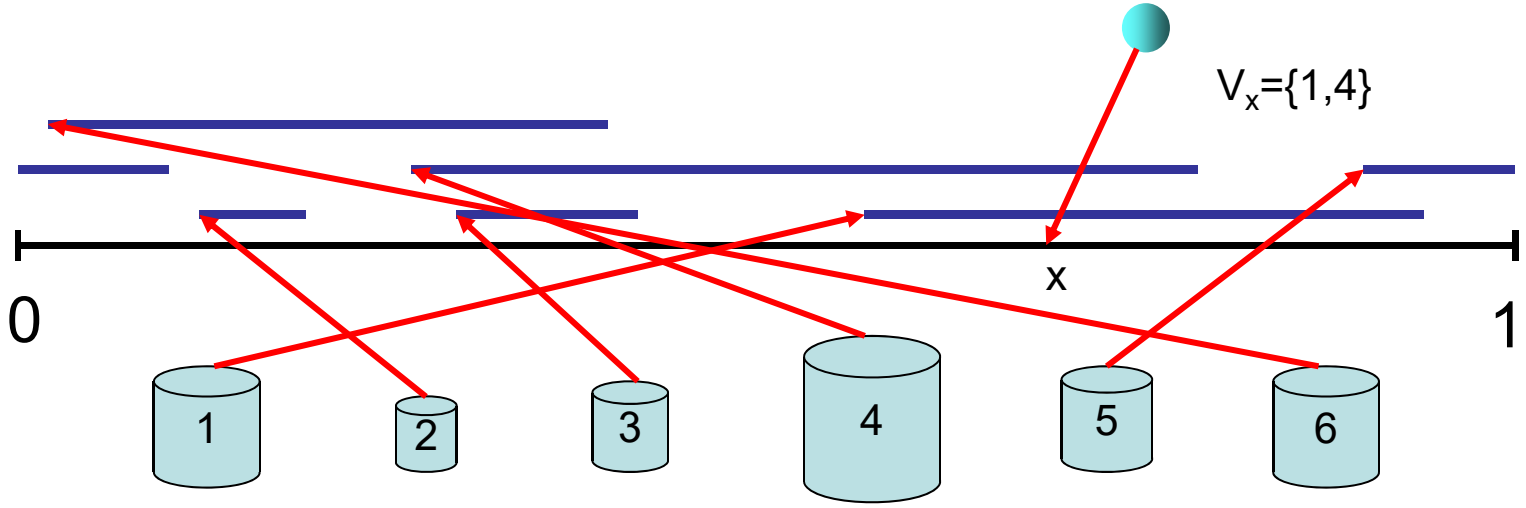
1. Stufe: Jedem Knoten $v$ wird ein Intervall $I(v) \subseteq [0,1)$ der Länge $s \cdot c_v$ zugeordnet, wobei $s = \Theta(\log n)$ ein fester Stretch-Faktor ist. Die Startpunkte der Intervalle sind durch eine Hashfunktion $h: V \rightarrow [0,1)$ gegeben.



# SHARE

Zuordnung zu Speichern: 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 $|I(v)|>1$ kommt v sooft in $V_x$ vor wie $I(v)$ x enthält).

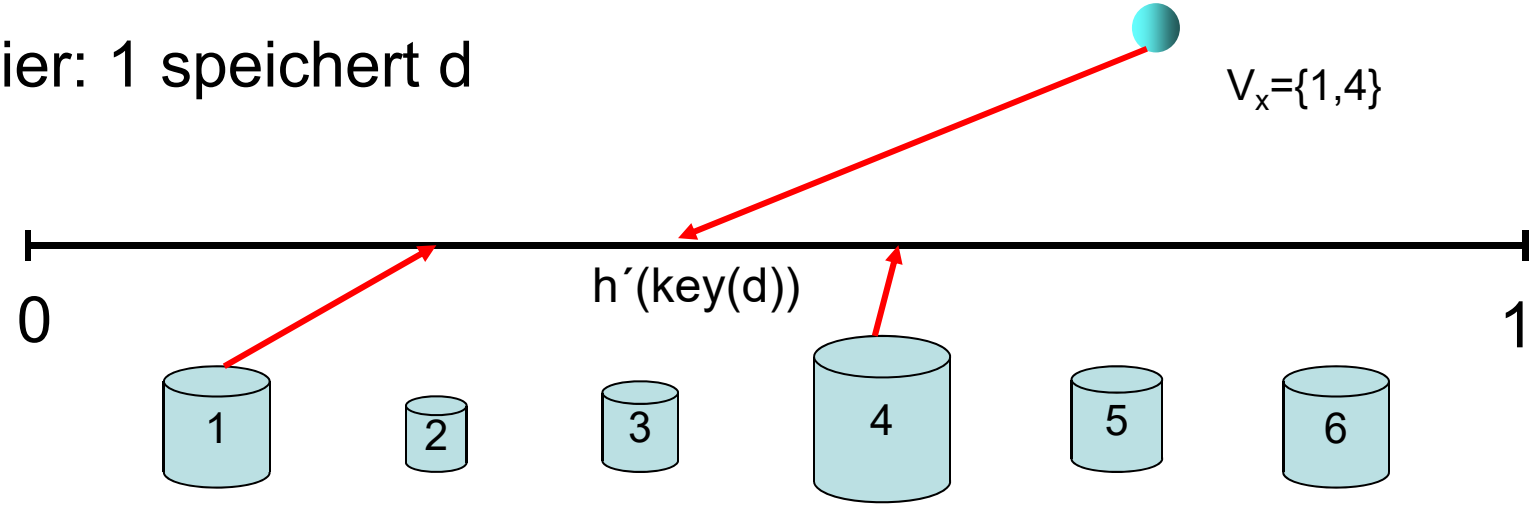


# SHARE

Zuordnung zu Speichern: zweistufiges Verfahren.

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

Hier: 1 speichert d

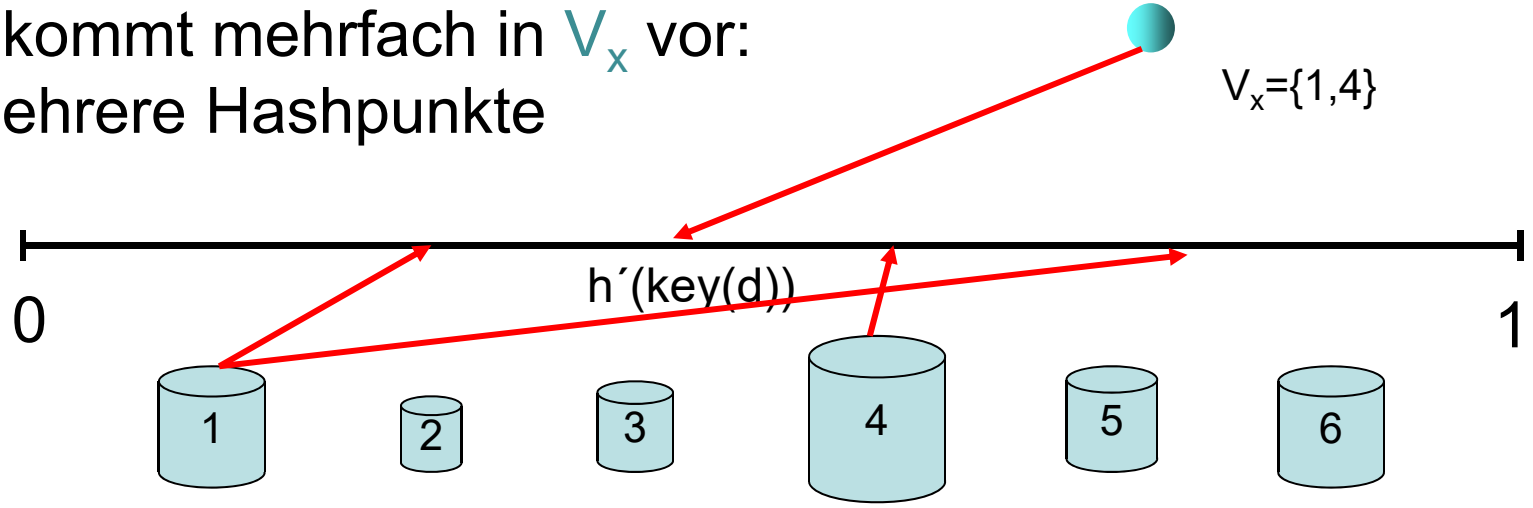


# SHARE

Zuordnung zu Speichern: zweistufiges Verfahren.

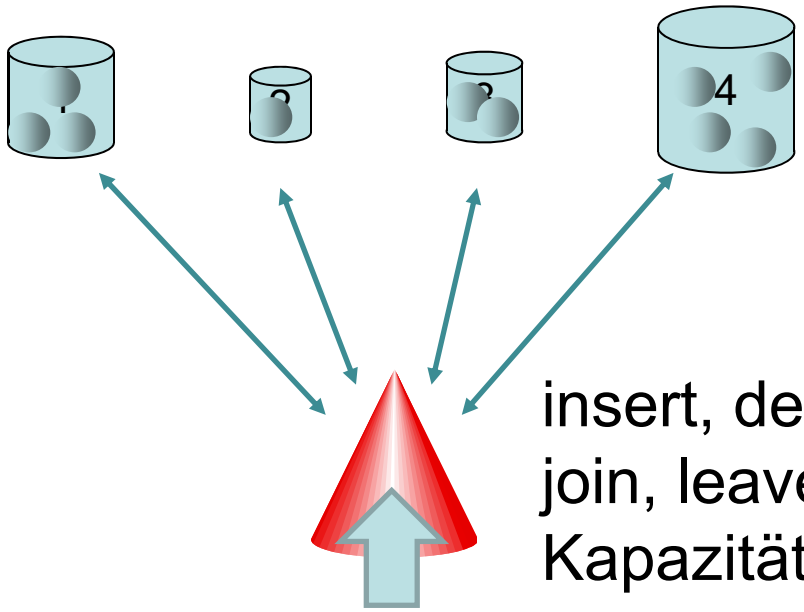
2. Stufe: Für Datum d wird mittels konsistentem Hashing mit Hashfunktionen h' und g' (die für alle Multimengen gleich sind) ermittelt, welcher Knoten in $V_x$ Datum d speichert.

1 kommt mehrfach in $V_x$ vor:
mehrere Hashpunkte



# SHARE

Realisierung:




insert, delete, lookup,
join, leave und
Kapazitätsveränderungen

# SHARE

Effiziente Datenstruktur im Server:

- 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)

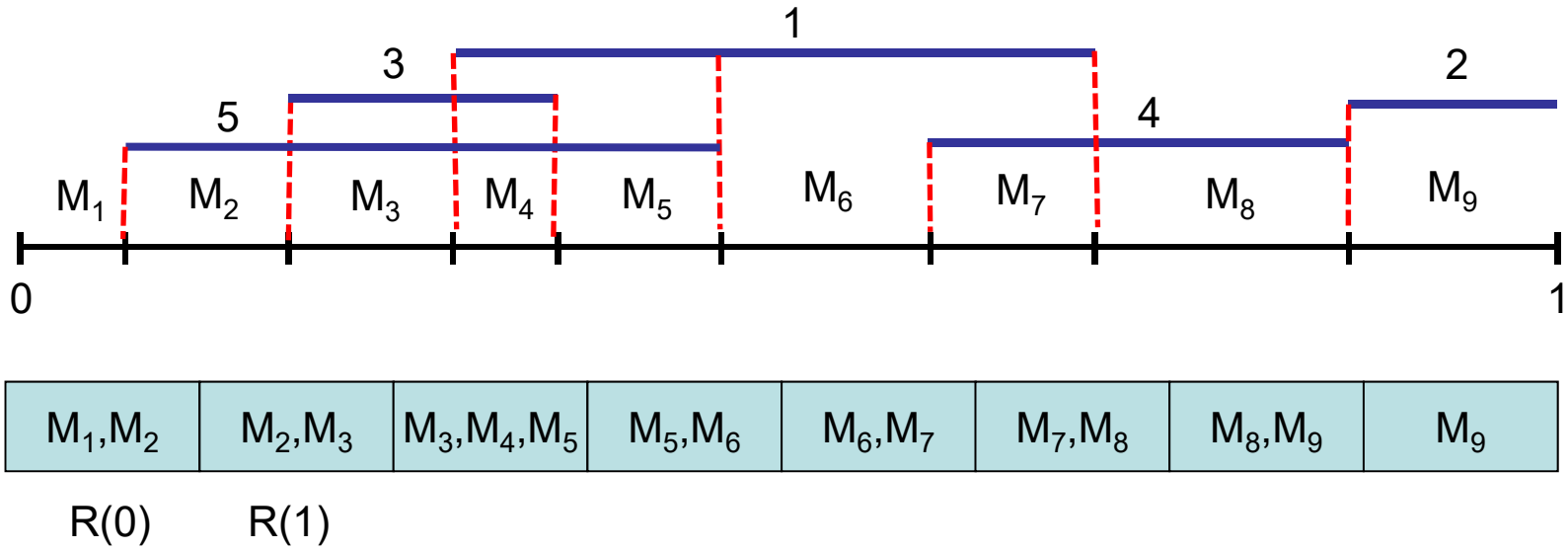
Laufzeit:

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

# SHARE

Effiziente Datenstruktur im Server:

- 1. Stufe: verwende Hashtabelle wie für konsistentes Hashing, um alle möglichen Multimengen $M_i$ für alle Bereiche [i/m,(i+1)/m) zu speichern.

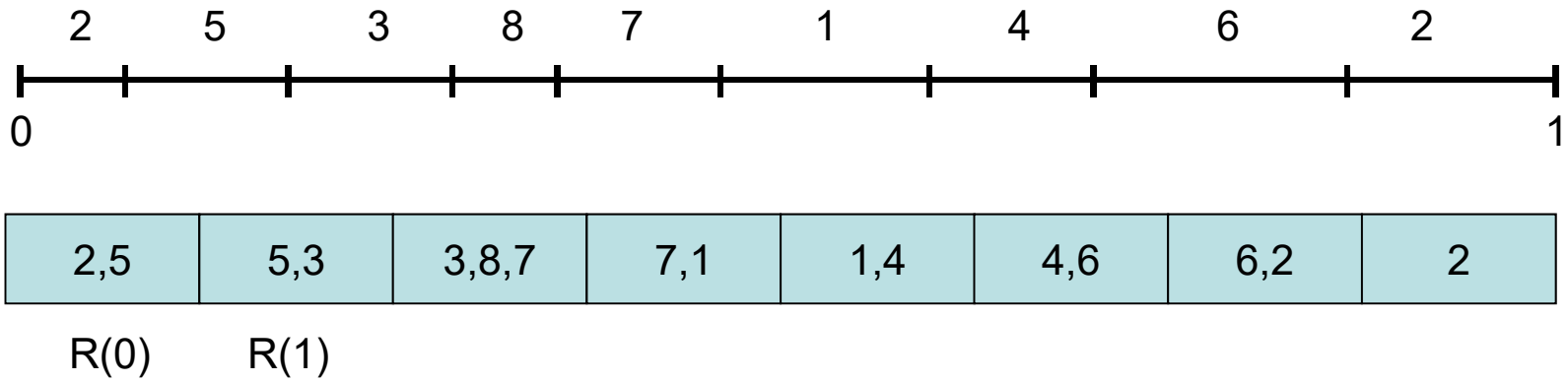


# SHARE

Effiziente Datenstruktur im Server:

- 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)

Beispiel für Multimenge $M=\{1,2,3,4,5,6,7,8\}$:



# SHARE

Satz 6.3:

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 (SODA 2008, recht komplex)

# SHARE

Beweis:

Punkt 2:

- $s=\Theta(\log n)$: Da dann $\Sigma_{v \in V} |I(v)|=s$, ist die erwartete Anzahl Intervalle über jeden Punkt in $[0,1)$ gleich $s$, und Abweichungen davon sind klein mit hoher W.keit falls $s$ genügend groß.
- Knoten $i$ hat Intervall der Länge $s \cdot c_i$
- Erwarteter Anteil Daten in Knoten $i$:

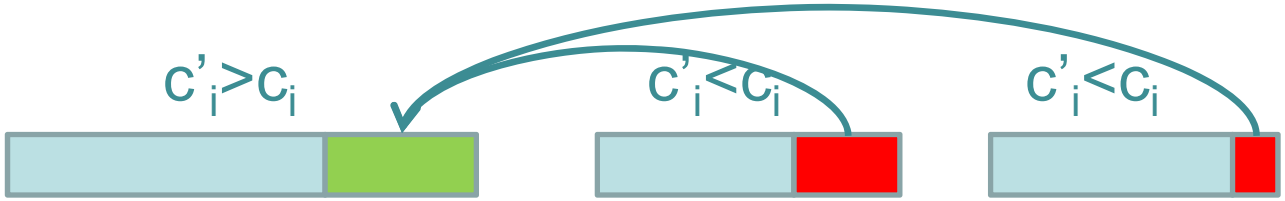
$$(s \cdot c_i) \cdot (1/s) = c_i$$

Phase 1 Phase 2

# SHARE

Punkt 3:

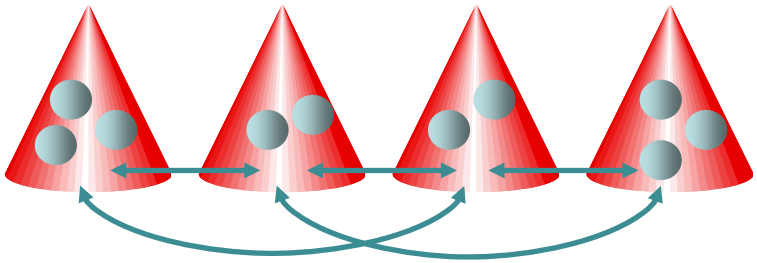
- Betrachte eine Veränderung der Kapazitäten von $(c_1,\ldots,c_n)$ nach $(c'_1,\ldots,c'_n)$
- Unterschied: $c = \sum_i |c_i - c'_i|$
- Optimale Strategie, um Fairness zu bewahren: replaziere einen $c/2$-Anteil der Daten




- SHARE: Veränderung der Intervalle $\sum_i |s(c_i - c'_i)| = s \cdot c$
- Erwarteter Anteil der replazierten Daten: $(s \cdot c) / s = c$, also max. doppelt so groß wie optimal

# SHARE

Gibt es auch eine effiziente verteilte Variante?



Jeder Knoten kann Anfragen (insert, delete, lookup, join, leave) generieren und Kapazität beliebig verändern.

Ja, Cone Hashing (evtl. Master-Level Kurs).

# Verteiltes Wörterbuch

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

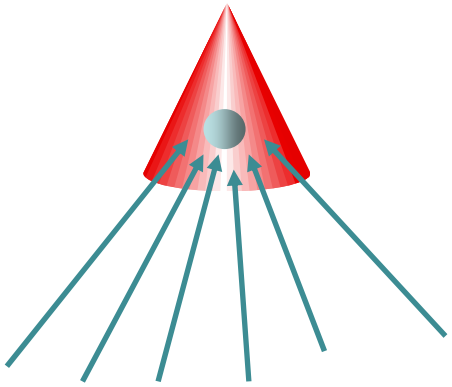
Nichtuniforme Speichersysteme: Kapazitäten können beliebig unterschiedlich sein

Vorgestellte Strategien:

- Uniforme Systeme: konsistentes Hashing
- Nichtuniforme Speichersysteme: SHARE
- Combine & Split

# Verteilte Hashtabelle

Probleme bei vielen Anfragen auf dasselbe Datum:



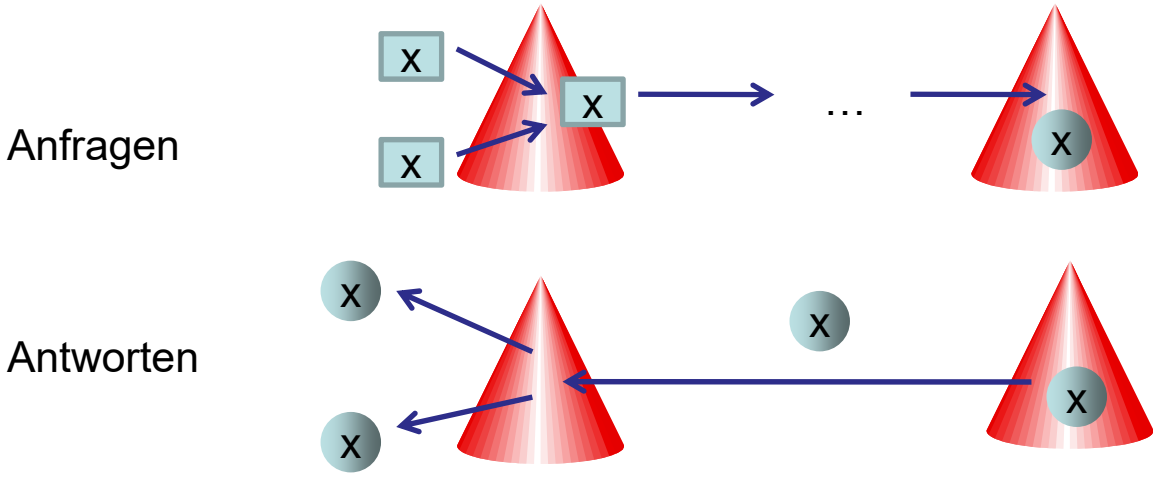
Prozess, der Datum speichert, wird überlastet.

Lösung: Combine & Split

# Verteilte Hashtabelle

Combine & Split:

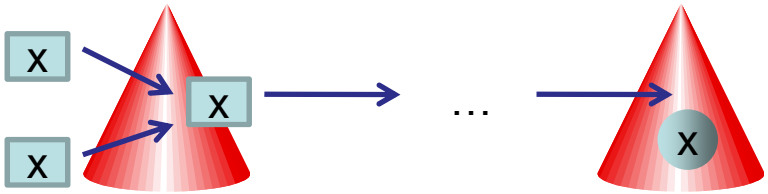
- Jeder Prozess v merkt sich alle Suchanfragen, die bei ihm eintreffen. Hat er für einen Schlüssel x bereits eine Suchanfrage weitergeleitet, dann hält er alle weiteren eingehenden Suchanfragen für x zurück (combine), beantwortet diese (split) sobald er eine Antwort zu x erhält und löscht diese dann aus seinem lokalen Speicher.



# Verteilte Hashtabelle

Combine & Split:

- Bei mehreren insert (bzw. delete) Anfragen zu demselben Schlüssel gewinnt die erste (d.h. es wird so getan, als sei die erste Anfrage als letzte bearbeitet worden, was denselben Effekt hätte).




Beobachtung: Mit der Combine & Split Regel ist die Congestion in der Größenordnung der Congestion, falls nur Anfragen auf verschiedene Daten unterwegs sind.

# Verteilte Hashtabelle

Satz 6.4: Sei G=(V,E) ein beliebiges Netzwerk konstanten Grades und P ein beliebiges Wegesystem für G. Dann gilt für ein beliebiges Routingproblem mit einer Anfrage pro Quellknoten (d.h. die Ziele sind beliebig), bei dem Wege zum selben Ziel beim ersten Aufeinandertreffen zu einem Weg verschmolzen werden, dass die Congestion (bis auf einen konstanten Faktor) höchstens so groß ist die maximale Congestion, die man erhält, wenn man im gegebenen Routingproblem maximal eine Anfrage pro Ziel zulässt (d.h. jeder Knoten ist Quelle und Ziel maximal einer Anfrage).

Beweis: Übung

Bemerkung: Falls die Daten zufällig und unabhängig über die Knoten verteilt werden, dann gilt für jedes Datenanfrageproblem mit einer Anfrage pro Quellknoten, in dem die angefragten Daten unabhängig von der Datenverteilung ausgewählt werden, dass es erwartungsgemäß nur konstant viele Daten mit Anfragen pro Zielknoten gibt. Satz 6.5 impliziert dann, dass bei Verwendung von combine&split die erwartete Congestion kaum schlimmer ist als wenn wir für jede Anfrage das Ziel zufällig und unabhängig von den anderen Anfragen aussuchen, unabhängig davon, wieviele Anfragen auf dasselbe Datum zugreifen wollen.

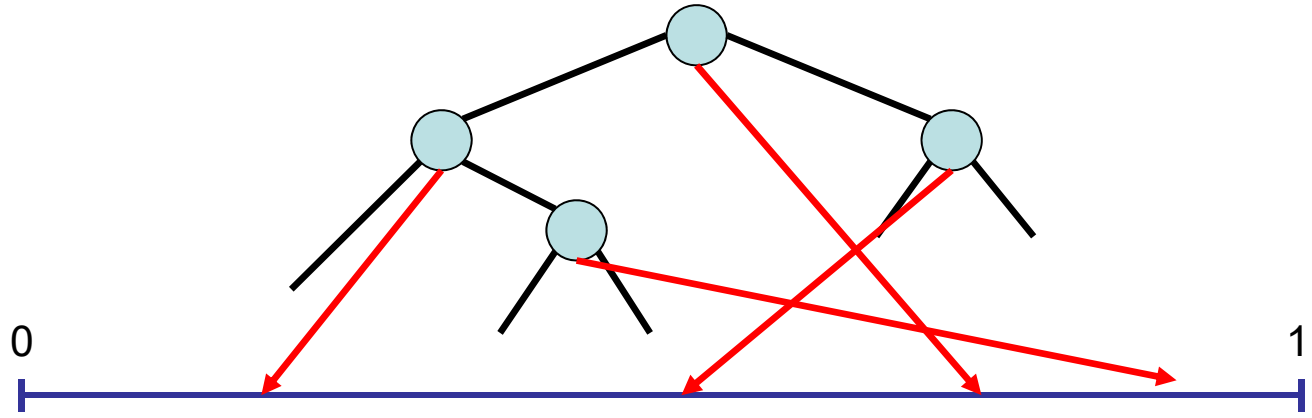
# Verteilte Hashtabelle

Weitere Vorzüge von combine&split:

- Verteilte Zugriffe auf sequentielle Datenstrukturen können effizient mittels verteilter Hashtabelle simuliert werden, sofern nur Lesezugriffe zu bearbeiten sind.

Beispiel: Suchbaum.

- Mittels konsistentem Hashing kann dieser einfach in einer verteilten Hashtabelle abgelegt werden.



# Verteilte Hashtabelle

Weitere Vorzüge von combine&split:

- Verteilte Zugriffe auf sequentielle Datenstrukturen können effizient mittels verteilter Hashtabelle simuliert werden, sofern nur Lesezugriffe zu bearbeiten sind.

Beispiel: Suchbaum.

- Anfangs starten alle Anfragen in der Wurzel: Routingproblem mit n Anfragen, die alle dasselbe Ziel haben, was laut Satz 6.4 durch combine&split effizient gelöst werden kann.

Problem: Bei einem Suchbaum der Tiefe T sind insgesamt T Anfragen auf die verteilte Hashtabelle pro Leseanfrage notwendig, um nach dem gesuchten Element im Suchbaum zu suchen. Das dauert eventuell zu lange.

Bessere Lösungen bekannt: Hashed Patricia Tries.

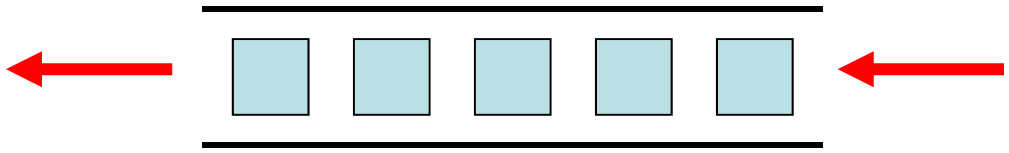
# Übersicht

- Verteilte Hashtabelle
- Verteilte Queue
- Verteilter Stack
- Verteilter Heap

# Konventionelle Queue

Eine Queue Q unterstützt folgende Operationen:

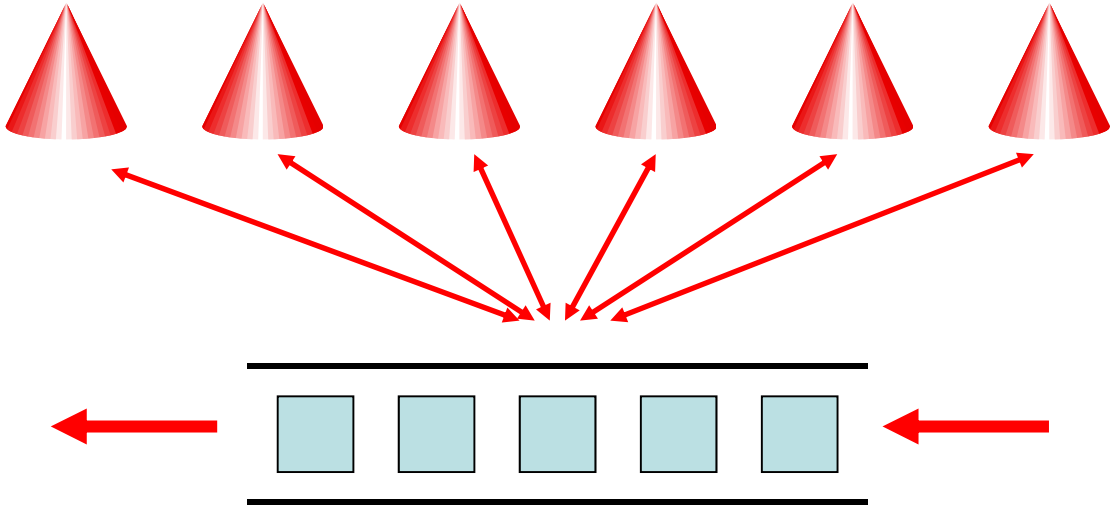
- enqueue(Q,x): fügt Element x hinten an die Queue Q an.
- dequeue(Q): holt das vorderste Element aus der Queue Q heraus und gibt es zurück



D.h. eine Queue Q implementiert die FIFO-Regel (FIFO: first in first out).

# Verteilte Queue

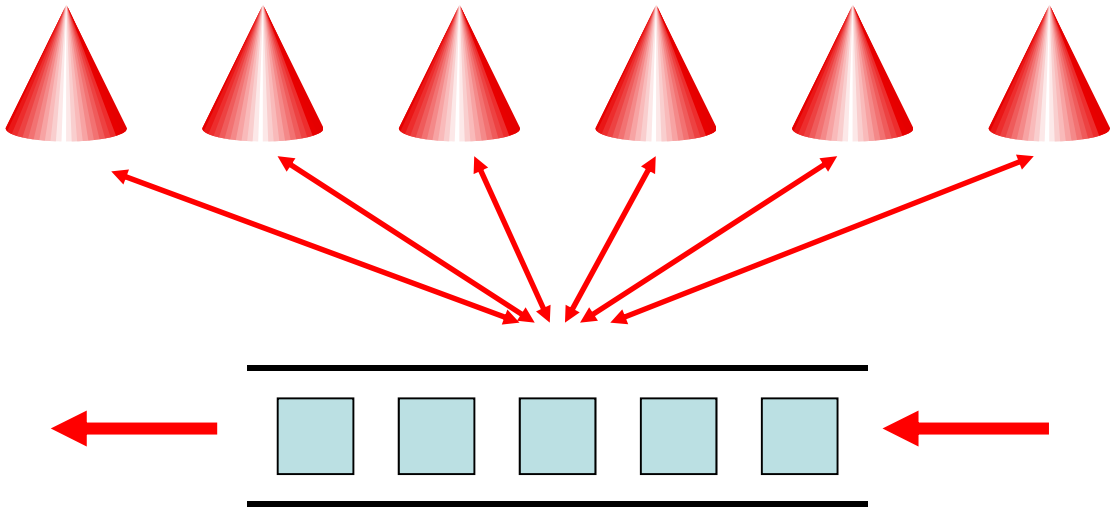
Viele Prozesse agieren auf Queue:



# Verteilte Queue

Probleme:

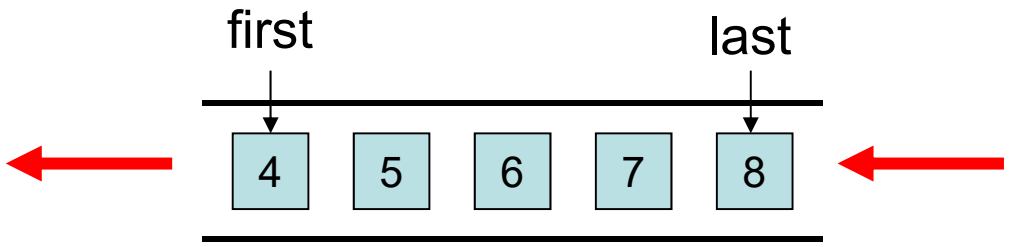
- Speicherung der Queue
- Realisierung von enqueue und dequeue



# Verteilte Queue

## Speicherung der Queue:

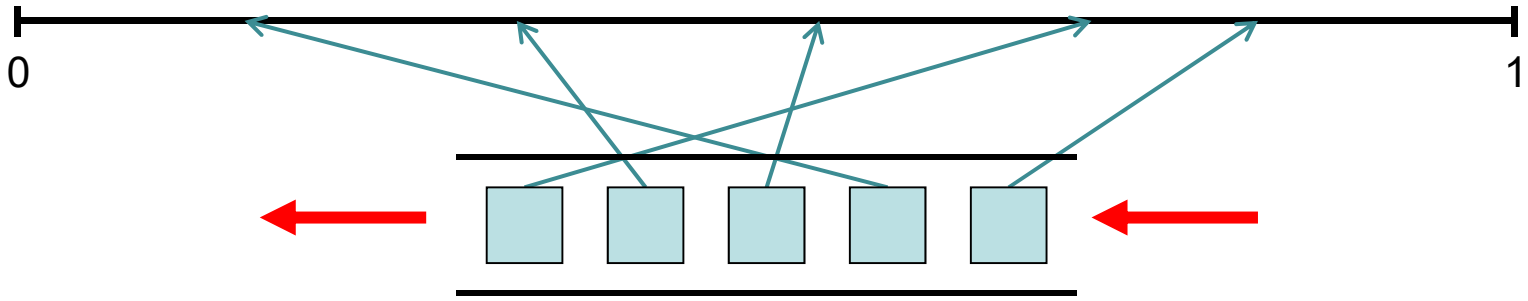
- Jedes Element x besitzt eindeutige Position $pos(x) \geq 1$ in der Queue (das vorderste hat die kleinste und das hinterste die größte Position).



# Verteilte Queue

## Speicherung der Queue:

- Jedes Element x besitzt eindeutige Position $pos(x) \geq 1$ in der Queue (das vorderste hat die kleinste und das hinterste die größte Position).
- Verwende eine verteilte Hashtabelle, um die Elemente x gleichmäßig mit Schlüsselwert pos(x) zu speichern.



# Verteilte Queue

Realisierung von enqueue(Q,x):

1. Stelle enqueue(Q,1)-Anfrage, um eine Nummer pos zu erhalten.
2. Führe insert(pos,x) auf der verteilten Hashtabelle aus, um x unter pos zu speichern.

Realisierung von dequeue(Q):

1. Stelle dequeue(Q,1)-Anfrage, um eine Nummer pos zu erhalten.
2. Führe delete(pos) auf der verteilten Hashtabelle aus, um das unter pos gespeicherte Element x zu löschen und zu erhalten.

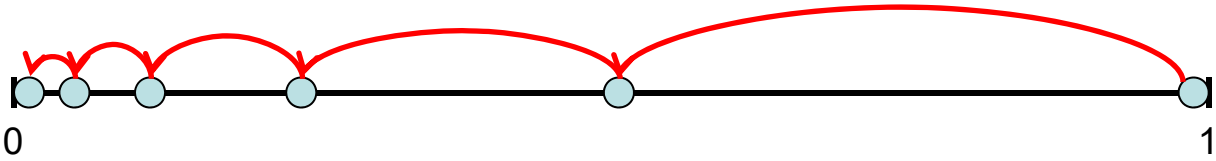
Noch zu klären: Punkt 1 in enqueue und dequeue.

Hier kann uns z.B. die de Bruijn Topologie zu Hilfe kommen.

# Verteilte Queue

Realisierung von enqueue(Q,1):

- Schicke alle enqueue(Q,1) Anfragen zu Punkt 0 im [0,1)-Raum mit Hilfe des de Bruijn Routings.




- Der für Punkt 0 zuständige Knoten $v_0$ merkt sich zwei Zähler first und last. first speichert die Position des ersten und last die Position des letzten Elements in Q.
- Jeder Knoten v merkt sich zunächst jede erhaltene enqueue(Q,i) Anfrage samt Knoten w, der diese an ihn geschickt hat.
- Angenommen, v habe bei Ausführung von timeout die Anfragen enqueue(Q,$i_1$), enqueue(Q,$i_2$),…,enqueue(Q,$i_k$) angesammelt. Dann sendet v eine enqueue(Q,i) Anfrage weiter in Richtung $v_0$, wobei $i=i_1+i_2+\ldots+i_k$ ist. Zusätzlich merkt sich v diese Kombinierung und die Rückadressen der einzelnen Anfragen.
- Erreicht eine enqueue(Q,i) Anfrage $v_0$, dann schickt $v_0$ das Intervall [last+1,last+i] an die Rückadresse und setzt last:=last+i.
- Erhält ein Knoten v ein Intervall [pos,pos+i-1], das zu einer enqueue(Q,i) Anfrage gehört, die aus den Anfragen enqueue(Q,$i_1$), enqueue(Q,$i_2$),…,enqueue(Q,$i_k$) kombiniert wurde, so schickt v das Intervall [pos, pos+$i_1$] an die Rückadresse von enqueue(Q,$i_1$), [pos+$i_1$, pos+$i_1$+$i_2$-1] an die Rückadresse von enqueue(Q,$i_2$), usw.
- Am Ende erhält jede enqueue(Q,1) Anfrage eine eindeutige Position in der Queue.

# Verteilte Queue

Realisierung von dequeue(Q,1):

- Schicke alle dequeue(Q,1) Anfragen zu Punkt 0 im [0,1)-Raum mit Hilfe des de Bruijn Routings.
- Der für Punkt 0 zuständige Knoten $v_0$ merkt sich zwei Zähler first und last.
- Die dequeue(Q,i) Anfragen werden wie die enqueue(Q,i) Anfragen zu $v_0$ hin kombiniert.
- Erreicht eine dequeue(Q,i) Anfrage $v_0$, dann schickt $v_0$ das Intervall [first, min{first+i-1,last}] an die Rückadresse und setzt first:=min{first+i,last+1}.
- Jeder Knoten, der ein Intervall für eine von ihm ausgeschickte dequeue(Q,i) Anfrage erhält, teilt dieses in Teilintervalle gemäß der dequeue(Q,j) Anfragen auf, die zur dequeue(Q,i) Anfrage beigetragen haben, und schickt diese an deren Rückadressen zurück. Sollte das Intervall zu klein sein, wird für einige dequeue(Q,j) Anfragen nur ein verkleinertes oder leeres Intervall zurückgeschickt.
- Am Ende bekommt dann jede dequeue(Q,1) Anfrage eine eindeutige Position oder $\perp$ zurück.

# Verteilte Queue

Satz 6.5: Die verteilte Queue benötigt (mit einer verteilten Hashtabelle auf Basis des de Bruijn Graphen) für die Operationen

- enqueue(Q,x): erwartete Zeit $O(\log n)$
- dequeue(Q): erwartete Zeit $O(\log n)$

Verwaltung mehrerer Queues in derselben Hashtabelle:
weise jeder Queue statt Punkt 0 einen (pseudo-) zufälligen Punkt in $[0,1)$ zu.

# Verteilte Queue

- $Op_v(i)$: i-te Operation in Knoten v
- $Enq_v(i)$: i-te Enqueue Operation in v
- $Deq_v(i)$: i-te Dequeue Operation in v
- M: Menge der Zuordnungen $(Enq_v(i), Deq_w(j))$ , d.h. das j-te Dequeue in w hat das i-te Enqueue von v ausgegeben

Gesucht: eine globale Ordnung „<“ auf den Operationen, so dass die folgenden Forderungen erfüllt sind.

Linearisierbarkeit:

1. Für alle $(Enq_v(i), Deq_w(j)) \in M$ ist $Enq_v(i) < Deq_w(j)$. (Ein Dequeue kann nur ein bereits eingefügtes Element zurückgeben.)
2. Für alle $(Enq_u(i), Deq_v(j)) \in M$ gilt: es gibt kein unzugeordnetes $Deq_w(k)$ mit $Enq_u(i) < Deq_w(k) < Deq_v(j)$ und es gibt kein unzugeordnetes $Enq_w(k)$ mit $Enq_w(k) < Enq_u(i) < Deq_v(j)$. (Operationen werden soweit möglich bedient.)
3. Für alle $(Enq_u(i), Deq_v(j)), (Enq_w(k), Deq_x(l)) \in M$ gilt nicht: $Enq_u(i) < Enq_w(k) < Deq_x(l) < Deq_v(j)$ oder $Enq_w(k) < Enq_u(i) < Deq_v(j) < Deq_x(l)$ (Die Queue-Eigenschaft gilt.)

Lokale Konsistenz:

- Für alle $v \in V$ und $i \in \mathbb{N}$ ist $Op_v(i) < Op_v(i+1)$.

Lösung: Bearbeitungsreihenfolge durch $v_0$ und Intervallaufteilung gibt Ordnung „<“ vor.

# Verteilte Queue

Selbststabilisierung:

- Verteilte Hashtabelle: bereits vorher betrachtet
- Anker $v_0$: Knoten ist Anker, solange er keinen linken Vorgänger hat. Sonst gibt er die Ankerfunktion auf (und transferiert gegebenenfalls first und last).

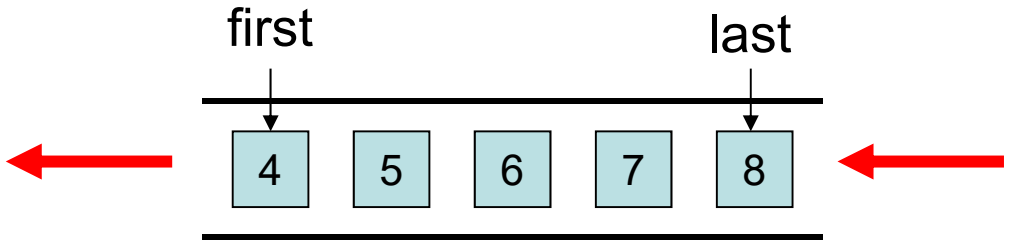
Probleme:

1. enqueue bzw. dequeue Operation wartet vergebens auf Rückantwort mit Position x.
2. dequeue Operation wird Position zugewiesen, bei der sie kein Element findet (entweder weil es noch dahin unterwegs ist, oder weil dieser Position eigentlich kein Element zugewiesen wurde dadurch dass, z.B., last korrumpiert ist)

# Verteilte Queue - Alternative

Speicherung der Queue:

- Jeder Prozess v verwaltet seine eigene Queue $Q_v$ bestehend aus Elementen seiner eigenen enqueue Anfragen.
- Jedes Element x besitzt eindeutige Position $pos(x) \geq 1$ in dieser Queue (das vorderste hat die kleinste und das hinterste die größte Position).
- Jeder Prozess v merkt sich über first und last die erste und letzte besetzte Position seiner Queue.



# Verteilte Queue

Realisierung von enqueue(Q,x) durch Prozess v:

1. Füge x in $Q_v$ (die lokale Queue von v) ein, d.h. last wird um 1 erhöht und pos(x) wird last zugewiesen.
2. Schicke eine enqueue(Q,v)-Botschaft aus, um das neue Element im System bekannt zu machen.

Realisierung von dequeue(Q) durch Prozess v:

1. Schicke eine dequeue(Q,v)-Anfrage aus, um nach einer enqueue(Q,w)-Botschaft zu suchen.
2. Sobald eine solche Botschaft gefunden wurde, wird diese gelöscht und eine dequeue($Q_w$,v)-Anfrage an w weitergeleitet.
3. Prozess w wird dann ein dequeue($Q_w$) auf $Q_w$ durchführen und das entfernte Element an v schicken.

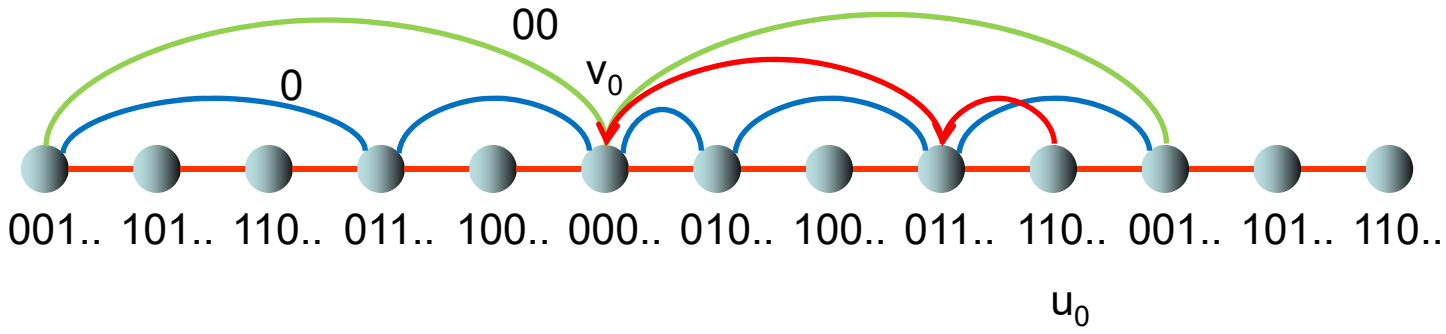
Noch zu klären: Punkt 2 in enqueue und Punkt 1 in dequeue.

Hier kann z.B. der Skip+ Graph verwendet werden

# Verteilte Queue

Realisierung der fehlenden Punkte in enqueue(Q,u) und dequeue(Q,u):

- Phase 1: Wähle für eine neue enqueue(Q,u) und dequeue(Q,u) Anfrage zunächst eine zufällige Bitfolge r aus und schicke die Anfrage zu dem Knoten $u_0$ mit dem längsten Präfix von $r(u_0)$ mit r (bei mehreren solchen Knoten zu dem mit kleinster ID) mit Hilfe des SKIP+ Routings.
- Phase 2: Von $u_0$ aus wird die Anfrage zu dem Knoten $v_0$ mit dem längsten 0-Präfix im zufälligen Bitstring $r(v_0)$ (bei mehreren Knoten demjenigen mit kleinster ID) mit Hilfe des SKIP+-Routings geschickt (siehe Folie 184 in Kapitel 5 und beispielhaft den roten Weg von $u_0$ nach $v_0$).



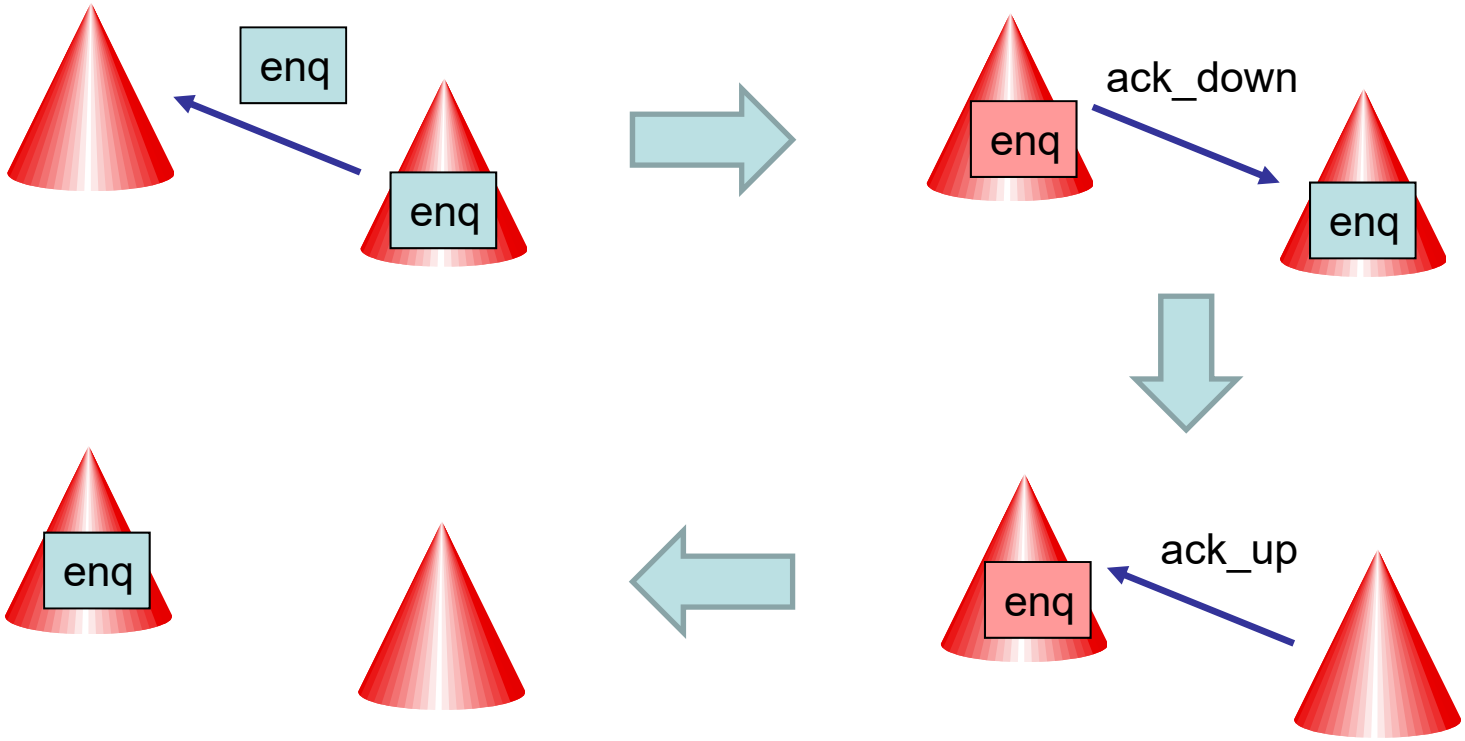
# Verteilte Queue

Phase 2 von enqueue(Q,u):

- Jeder Knoten v hat einen Zähler $s_v$. Jeder bestätigten enqueue(Q,u) Anfrage wird ein eindeutiger Rang mittels $s_v$ zugewiesen (und danach $s_v$ um 1 erhöht). Zusätzlich merkt sich v in $m_v$ die aktuelle Anzahl bestätigter enqueue Anfragen, die in ihm gespeichert sind. („bestätigt“ wird unten definiert)
- Bei jedem timeout sendet v alle (aber maximal K für eine feste, genügend große Konstante K) derzeit in v gespeicherten bestätigten enqueue Anfragen samt ihrer Ränge (mit Priorität auf den niedrigsten Rängen) in einer Botschaft zum nächsten Knoten in Richtung $v_0$, merkt sich aber noch Kopien dieser enqueue Anfragen.
- Bekommt ein Knoten v eine Botschaft bestehend aus enqueue Anfragen mit Rängen im Intervall [a,b], dann behält er die Anfragen mit Rängen in $[a,\max\{b,a+f_v\}]$ und markiert diese zunächst als unbestätigt, wobei $f_v$ angibt, wieviele enqueue Anfragen v noch aufnehmen kann, bis $m_v=K$ erreicht ist. Die restlichen Anfragen löscht v. Weiterhin schickt v $ack\_down[a,\max\{b,a+f_v\}]$ an den Absender zurück.
- Bekommt ein Knoten v die Antwort ack_down[a,c] und ist sein momentanes Intervall [a´,b] ($a´\geq a$, $b\geq c$), dann löscht er alle enqueue Anfragen mit Rängen in [a´,c] und schickt nack_up[v,a,a´-1] und ack_up[v,a´,c] an den Absender zurück.
- Bekommt ein Knoten die Antwort nack_up[v,a,b], dann löscht er alle unbestätigten enqueue Anfragen von v mit Rängen im Bereich [a,b].
- Bekommt ein Knoten die Antwort ack_up[v,a,b], dann markiert er die enqueue Anfragen von v mit Rängen im Bereich [a,b] als bestätigt.

# Verteilte Queue

Phase 2 von enqueue(Q,u) bildlich:



# Verteilte Queue

Phase 2 von dequeue(Q,u):

- Empfängt ein Knoten v eine dequeue(Q,u) Anfrage von w, und er besitzt weder unbestätigte enqueue Anfragen von w noch bestätigte enqueue Anfragen, dann leitet er die dequeue(Q,u) Anfrage zusammen mit $s_v$ in Richtung $v_0$ weiter.
- Hat v mindestens eine unbestätigte enqueue Anfrage von w und ist die dequeue(Q,u) Anfrage zusammen mit dem Rang s von w geschickt worden, dann wird zunächst nack_up[w,0,s-1] wie beim enqueue beschrieben durchgeführt. Hat v danach noch mindestens eine unbestätigte enqueue Anfrage von w, dann weist er der dequeue Anfrage die enqueue Anfrage mit niedrigstem Rang zu und löscht diese aus seinem Speicher.
- Hat v ansonsten mindestens eine bestätigte enqueue Anfrage, dann weist er der dequeue Anfrage die enqueue Anfrag mit niedrigstem Rang zu und löscht diese aus seinem Speicher. Sonst leitet v die dequeue(Q,u) Anfrage zusammen mit $s_v$ in Richtung $v_0$ weiter.

Satz 6.6: Solange mindestens eine enqueue Anfrage auf dem Pfad einer dequeue Anfrage in Phase 2 liegt, wird diese irgendwann einer enqueue Anfrage zugewiesen werden. Weiterhin kann eine enqueue Anfrage nie doppelt vergeben werden.

Beweis: Übung.

# Verteilte Queue

- $Op_v(i)$: i-te Operation in Knoten v
- $Enq_v(i)$: i-te Enqueue Operation in v
- $Deq_v(i)$: i-te Dequeue Operation in v
- M: Menge der Zuordnungen $(Enq_v(i), Deq_w(j))$ , d.h. das j-te Dequeue in w hat das i-te Enqueue von v ausgegeben

Gesucht: eine globale Ordnung „<“ auf den Operationen, so dass die folgenden Forderungen erfüllt sind.

Linearisierbarkeit:

1. Für alle $(Enq_v(i), Deq_w(j)) \in M$ ist $Enq_v(i) < Deq_w(j)$. (Ein Dequeue kann nur ein bereits eingefügtes Element zurückgeben.)
2. Für alle $(Enq_u(i), Deq_v(j)) \in M$ gilt: es gibt kein unzugeordnetes $Deq_w(k)$ mit $Enq_u(i) < Deq_w(k) < Deq_v(j)$ und es gibt kein unzugeordnetes $Enq_w(k)$ mit $Enq_w(k) < Enq_u(i) < Deq_v(j)$. (Operationen werden soweit möglich bedient.)
3. Für alle $(Enq_u(i), Deq_v(j)), (Enq_w(k), Deq_x(l)) \in M$ gilt nicht: $Enq_u(i) < Enq_w(k) < Deq_x(l) < Deq_v(j)$ oder $Enq_w(k) < Enq_u(i) < Deq_v(j) < Deq_x(l)$ (Die Queue-Eigenschaft gilt.)

Lokale Konsistenz:

- Für alle $v \in V$ und $i \in \mathbb{N}$ ist $Op_v(i) < Op_v(i+1)$.

Lösung: noch unklar.

# Verteilte Queue

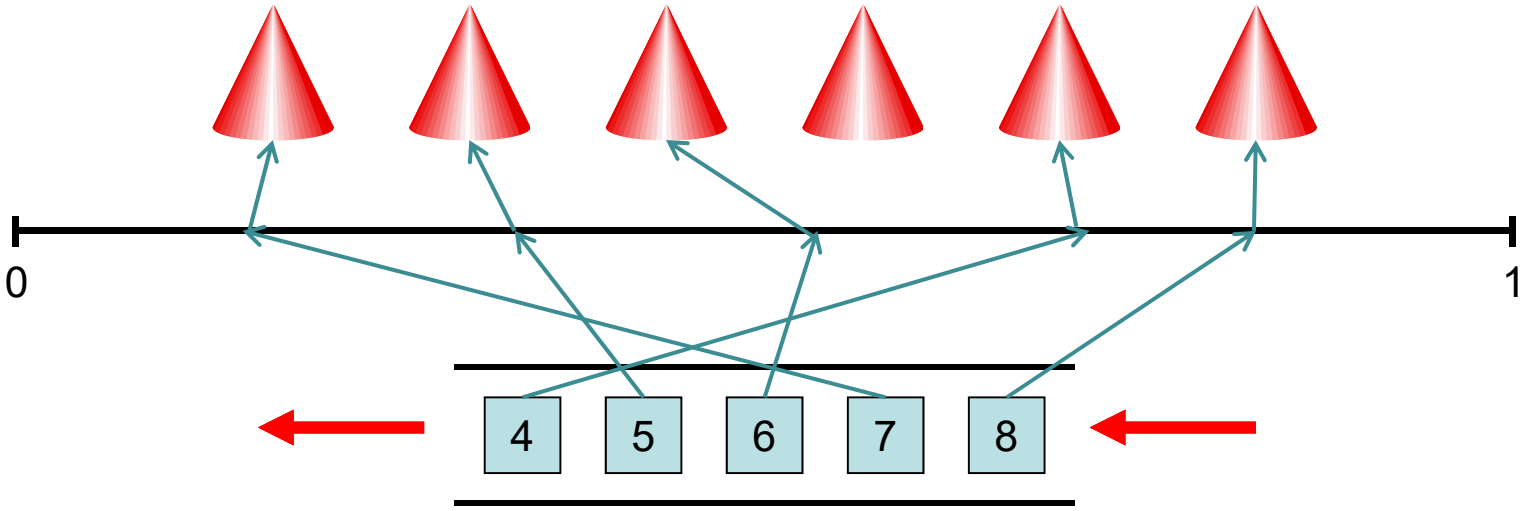
Weiteres Problem: Große Lastdifferenzen, da die Queues mancher Knoten viel größer als die anderer sind.

Lösung: Jeder Knoten $v$ speichert die Einträge von $Q_v$ nicht bei sich selbst sondern in einer verteilten Hashtabelle.

# Verteilte Queue

Speicherung der Queue:

- Jeder Prozess v merkt sich über first und last die erste und letzte Position seiner Queue und verwendet eine verteilte Hashtabelle, um Element x in seiner Queue unter dem eindeutigen Schlüssel (v,pos(x)) zu speichern.



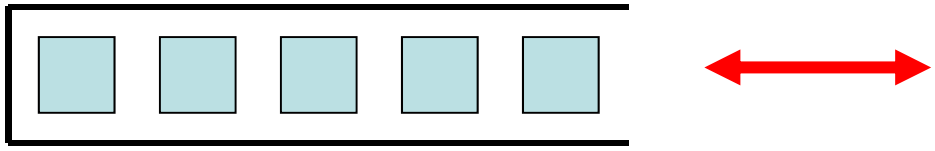
# Übersicht

- Verteilte Hashtabelle
- Verteilte Queue
- Verteilter Stack
- Verteilter Heap

# Konventioneller Stack

Ein Stack S unterstützt folgende Operationen:

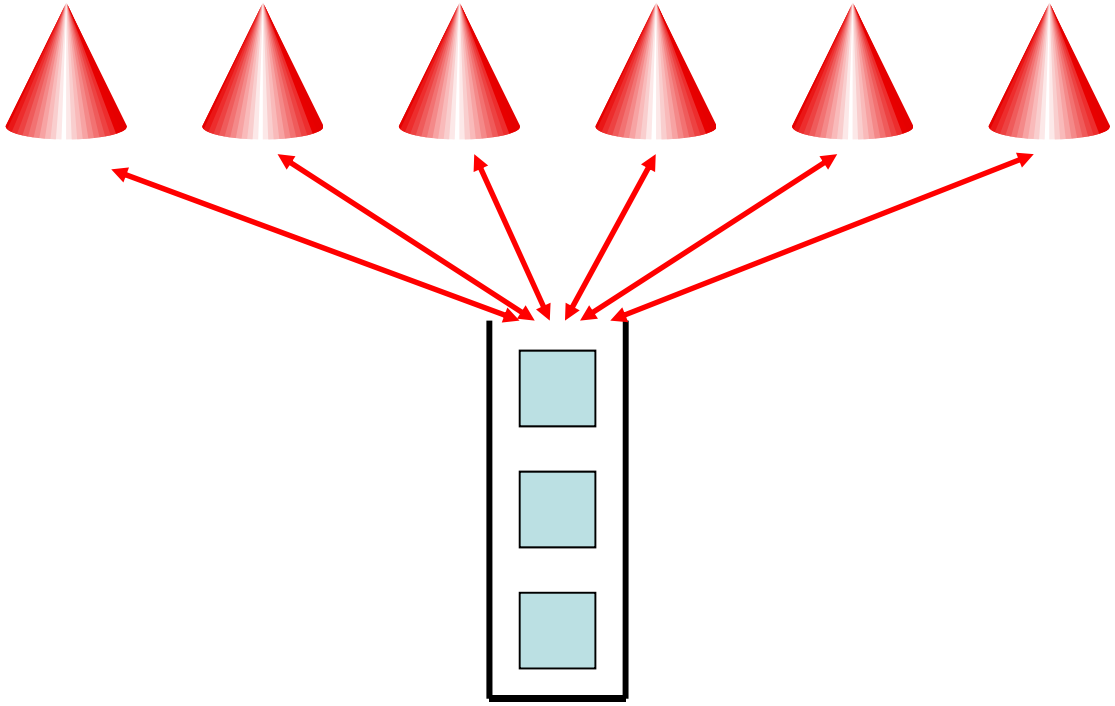
- push(S,x): legt Element x oben auf dem Stack ab.
- pop(S): holt das oberste Element aus dem Stack S heraus und gibt es zurück



D.h. ein Stack S implementiert die LIFO-Regel (LIFO: last in first out).

# Verteilter Stack

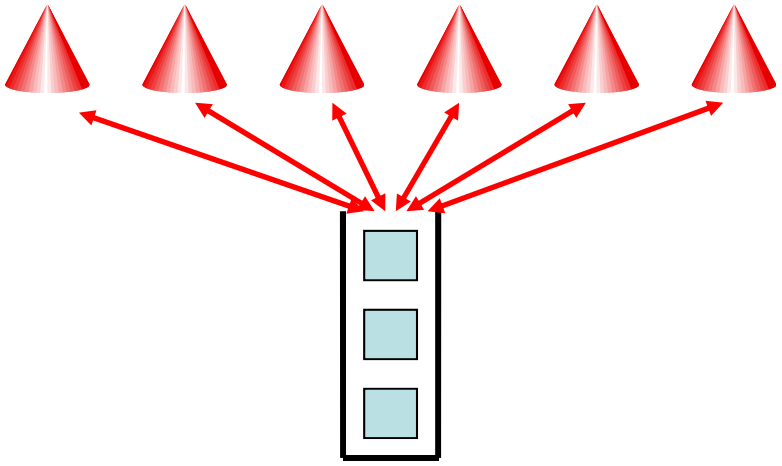
Viele Prozesse agieren auf Stack:



# Verteilter Stack

Probleme:

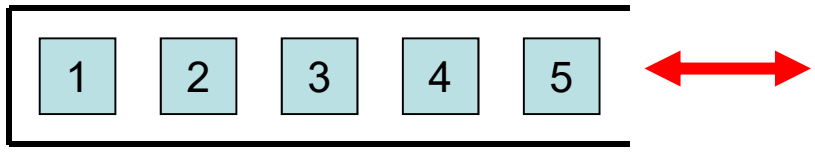
- Speicherung des Stacks
- Realisierung von push und pop



# Verteilter Stack

## Speicherung des Stacks:

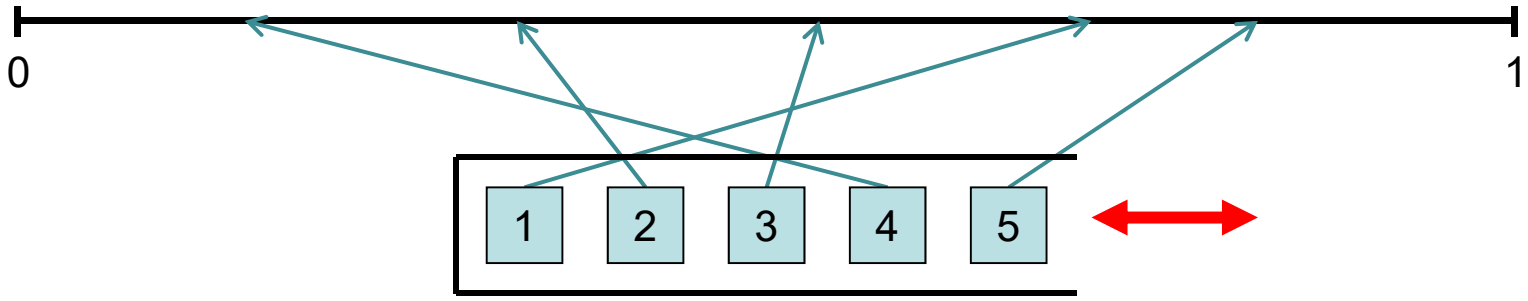
- Jedes Element x besitzt eindeutige Position pos(x)≥1 im Stack (das oberste hat die größte Position).



# Verteilter Stack

## Speicherung des Stacks:

- Jedes Element x besitzt eindeutige Position $pos(x) \geq 1$ im Stack (das oberste hat die größte Position).
- Verwende eine verteilte Hashtabelle, um die Elemente x gleichmäßig mit Schlüsselwert pos(x) zu speichern.



# Verteilter Stack

Realisierung von push(S,x):

1. Stelle push(S,1)-Anfrage, um eine Nummer pos zu erhalten.
2. Führe insert(pos,x) auf der verteilten Hashtabelle aus, um x unter pos zu speichern.

Realisierung von pop(S):

1. Stelle pop(S,1)-Anfrage, um eine Nummer pos zu erhalten.
2. Führe delete(pos) auf der verteilten Hashtabelle aus, um das unter pos gespeicherte Element x zu löschen und zu erhalten.

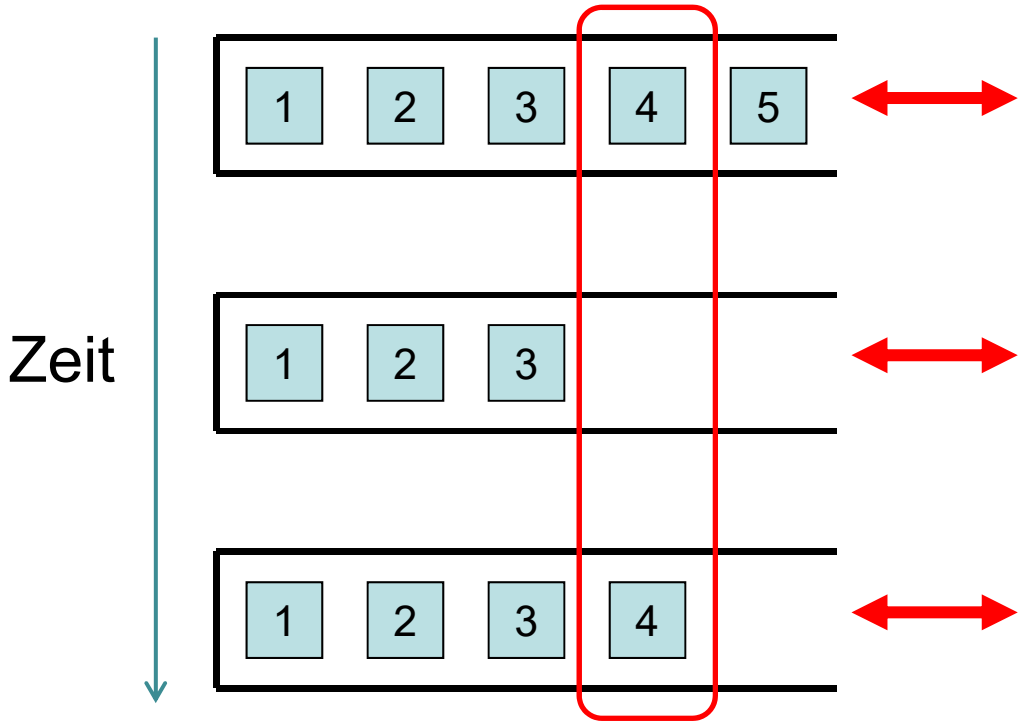
Noch zu klären: Punkt 1.

Hier können wir ähnlich wie für enqueue und dequeue vorgehen.

Neues Problem: push Anfragen könnten dieselbe Position zugewiesen bekommen.

# Verteilter Stack

Neues Problem: push Anfragen könnten dieselbe Position zugewiesen bekommen.



# Verteilter Stack

Neues Problem: push Anfragen könnten dieselbe Position zugewiesen bekommen.

Warum ist das ein Problem? Lokale Konsistenz könnte dadurch verloren gehen.

Beispiel:

- Prozess v führt folgende Operationen aus: push(x), push(y), pop(), pop()
- Werden Positionen in der Reihenfolge push(x), pop(), push(y), pop() vom Anker $v_0$ geholt, kann es passieren, dass alle Operationen dieselbe Position im Stack zugewiesen bekommen (falls z.B kein anderer Prozess zu dieser Zeit Operationen ausführt).
- Dann könnte sich das erste pop() das x und das zweite das y holen, was die lokale Konsistenz verletzen würde.

# Verteilter Stack

Mögliche Auswege:

- lokal sequentielle Ausführung der Operationen. Aber wie setzt man das am effizientesten um?
- Eindeutige Positionen der Queue können als Ticket-System verwendet werden. Die Kombination (pos,ticket) vermeidet dann Überholungen.
- Ticket-System wäre auch für lokale Konsistenz für Hashtabelle anwendbar.

# Verteilter Stack

Satz 6.6: Der verteilte Stack benötigt (mit einer verteilten Hashtabelle, z.B. auf Basis des Skip+ Graphen) für die Operationen

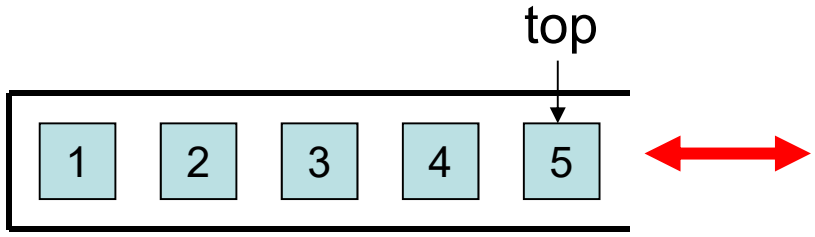
- push(S,x): erwartete Zeit $O(\log n)$
- pop(S): erwartete Zeit $O(\log n)$

Verwaltung mehrerer Stacks in derselben Hashtabelle:
weise jedem Stack statt Punkt 0 einen (pseudo-) zufälligen Punkt in $[0,1)$ zu.

# Verteilter Stack - Alternative

Speicherung des Stacks:

- Jeder Prozess v verwaltet seinen eigenen Stack $S_v$ bestehend aus Elementen seiner eigenen push Anfragen.
- Jedes Element x besitzt eindeutige Position $pos(x) \geq 1$ in diesem Stack (das älteste hat die kleinste und das neueste die größte Position).
- Jeder Prozess v merkt sich über top die oberste besetzte Position seines Stacks.



# Verteilter Stack

Realisierung von push(S,x) durch Prozess v:

1. Füge x in $S_v$ (den lokalen Stack von v) ein, d.h. top wird um 1 erhöht und pos(x) wird top zugewiesen.
2. Schicke eine push(S,v)-Botschaft aus, um das neue Element im System bekannt zu machen.

Realisierung von pop(Q) durch Prozess v:

1. Schicke eine pop(Q,v)-Anfrage aus, um nach einer push(S,w)-Botschaft zu suchen.
2. Sobald eine solche Botschaft gefunden wurde, wird diese gelöscht und eine pop($Q_w$,v)-Anfrage an w weitergeleitet.
3. Prozess w wird dann ein pop($Q_w$) auf $Q_w$ durchführen und das entfernte Element an v schicken.

Noch zu klären: Punkt 2 in push und Punk 1 in pop.

Auch hier kann z.B. der Skip+ Graph verwendet werden

# Übersicht

- Verteilte Hashtabelle
- Verteilte Queue
- Verteilter Stack
- Verteilter Heap

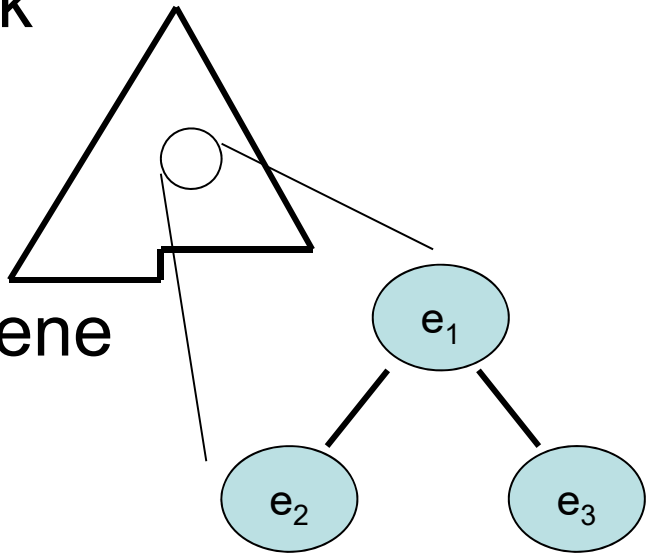
# Konventioneller Heap

Ein Heap H unterstützt folgende Operationen:

- insert(H,x): fügt Element x in den Heap H ein (die Priorität wird über key(x) bestimmt).
- deleteMin(H): entfernt das Element x mit kleinstem key(x) aus H und gibt es zurück

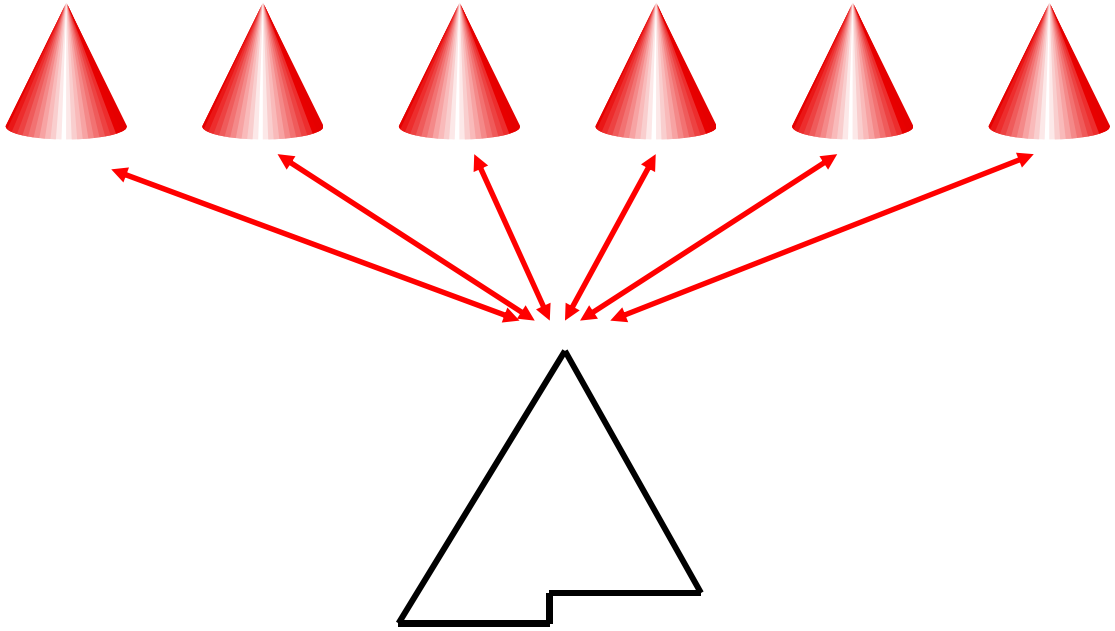
Zwei Invarianten:

- Form-Invariante:vollständiger Binärbaum bis auf unterste Ebene
- Heap-Invariante: $key(e_1) \leq \min\{key(e_2), key(e_3)\}$



# Verteilter Heap

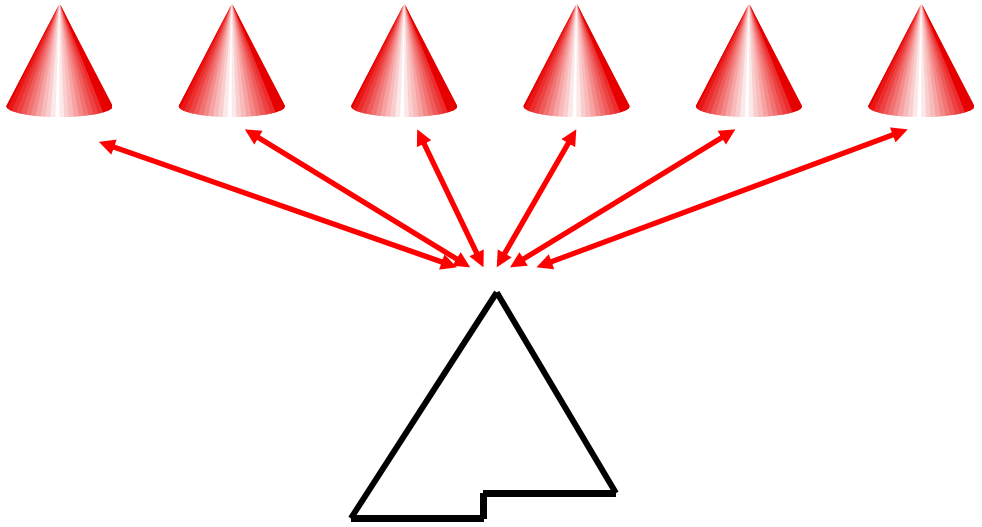
Viele Subjekte agieren auf Heap:



# Verteilter Heap

Probleme:

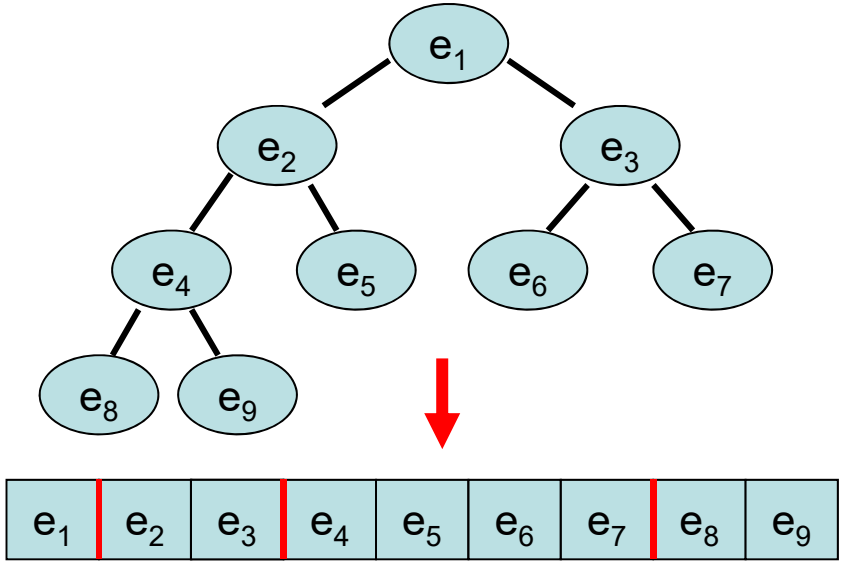
- Speicherung des Heaps
- Realisierung von insert und deleteMin



# Verteilter Heap

## Speicherung des Heaps:

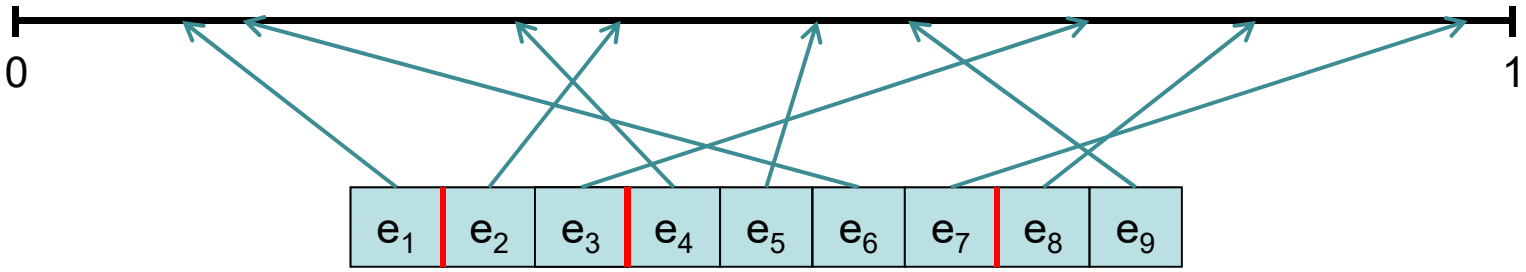
- Jedes Element x besitzt eine eindeutige Position pos(x)≥1 im Heap wie bei der Feldrealisierung des konventionellen Heaps



# Verteilter Heap

Speicherung des Heaps:

- Jedes Element x besitzt eine eindeutige Position pos(x)≥1 im Heap wie bei der Feldrealisierung des konventionellen Heaps
- Verwende eine verteilte Hashtabelle, um die Elemente x gleichmäßig mit Schlüsselwert pos(x) zu speichern.



# Verteilter Heap

Realisierung von insert(H,x):

1. Stelle insert(H,1)-Anfrage, um eine Nummer pos zu erhalten.
2. Führe insert(pos,x) auf der verteilten Hashtabelle aus, um x unter pos zu speichern.

Realisierung von deleteMin(H):

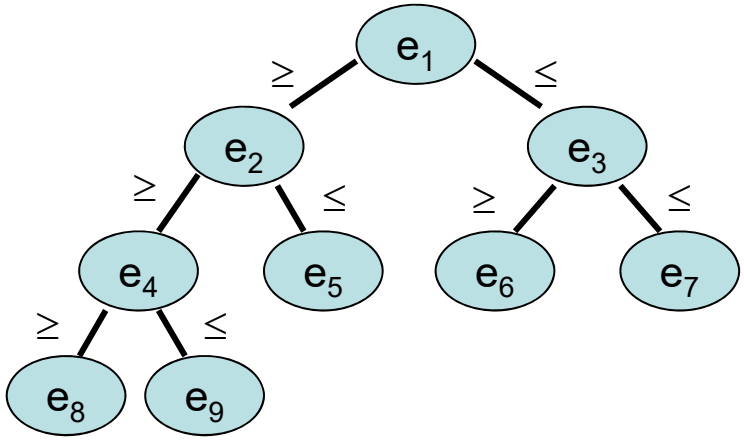
1. Stelle deleteMin(H,1)-Anfrage, um eine Nummer pos zu erhalten.
2. Führe delete(pos) auf der verteilten Hashtabelle aus, um das unter pos gespeicherte Element x zu löschen und zu erhalten.

Problem: Punkt 2. in deleteMin nicht gut parallelisierbar!

# Verteilter Heap

Problem: Punkt 2. in deleteMin nicht gut parallelisierbar!

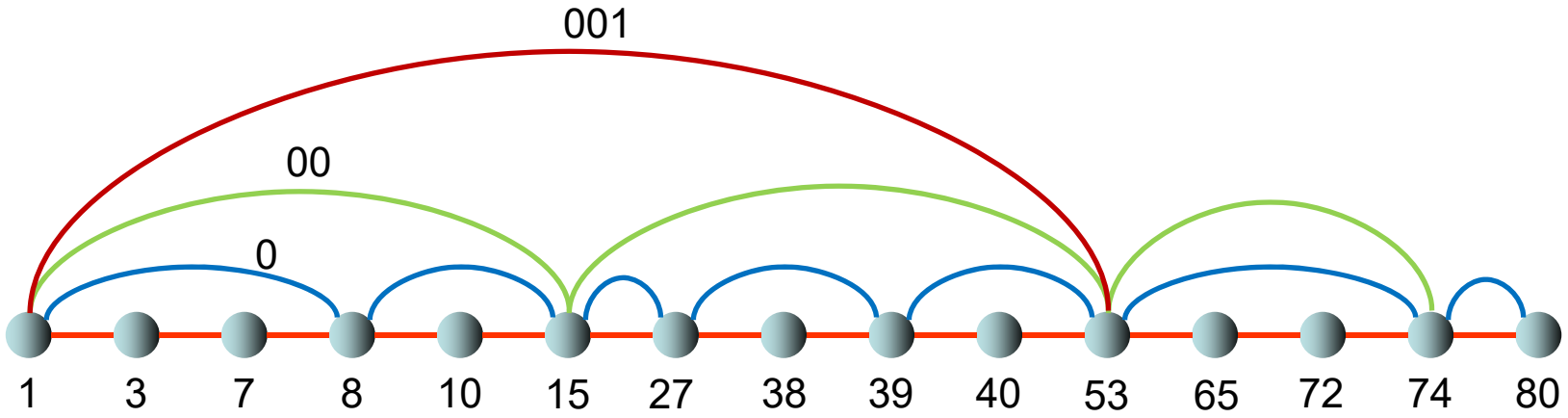
Warum? Im binären Heap können Minima nur sequentiell ermittelt werden, da zu wenig Ordnungsinformation über Elemente.




Lösung: verwende stattdessen Suchstruktur.

# Verteilter Heap

Angenommen, Elemente sind in Skip+ Graph organisiert, der in verteilter Hashtabelle gespeichert wird.




deleteMin(H,10): nimm 10 kleinste Elemente aus Skip+ Graph, was einfach durch Broadcast in O(log n) Zeit zu bewerkstelligen ist

# Verteilter Heap

Grundlegende Idee: Anker $v_0$ verwendet zwei Variaben first und last wie für die Queue:

- insert(H,k): setzt last:=last+k
- deleteMin(H,k): schickt [first,first+k-1] zurück und setzt first:=first+k

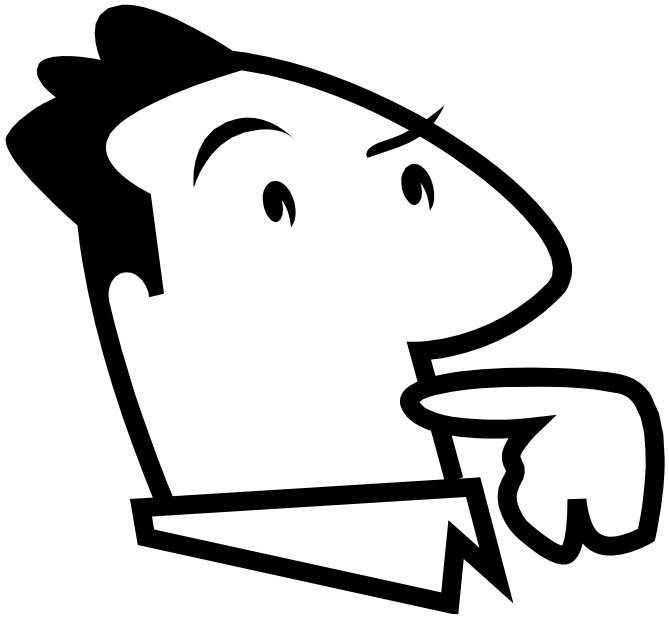
Zuweisung der k kleinsten Elemente in Priority Queue an k Knoten, die deleteMin ausgeführt haben:

1. Knoten erhalten über zurückgeschicktes Intervall eindeutige Werte aus [first,first+k-1] (die wie Adressen von Postfächern fungieren, in denen diese ihr minimales Elementen finden können)
2. Anker schickt weiterhin [first,first+k-1] an die kleinsten k Elemente im Skip+ Graph der Elemente, so dass jedes Element eine eindeutige Adresse in [first,first+k-1] zugewiesen bekommt. Diese Adressen werden dafür verwendet, die kleinsten k Elemente unter diesen Adressen in der verteilten Hashtabelle abzuspeichern, so dass die anfragenden Knoten diese dort in Empfang nehmen können.

Kurs leider hier zuende…

# Referenzen


- John Byers, Jeffrey Considine, and Michael Mitzenmacher. Simple Load Balancing for Distributed Hash Tables. IPTPS 2003.
- Petra Berenbrink, Andre Brinkmann, Tom Friedetzky, and Lars Nagel. Balls into non-uniform bins. Journal of Parallel and Distributed Computing 74(2), 2014.
- David Karger and Matthias Ruhl. Simple Efficient Load-Balancing Algorithms for Peer-to-Peer Systems.Theory of Computing Systems 39(6), 2006.



Fragen?