Verteilte Algorithmen und Datenstrukturen

Kapitel 6: Informationsorientierte Datenstrukturen

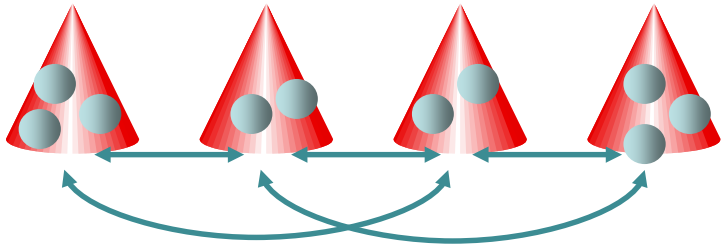
Prof. Dr. Christian Scheideler

SS 2017

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

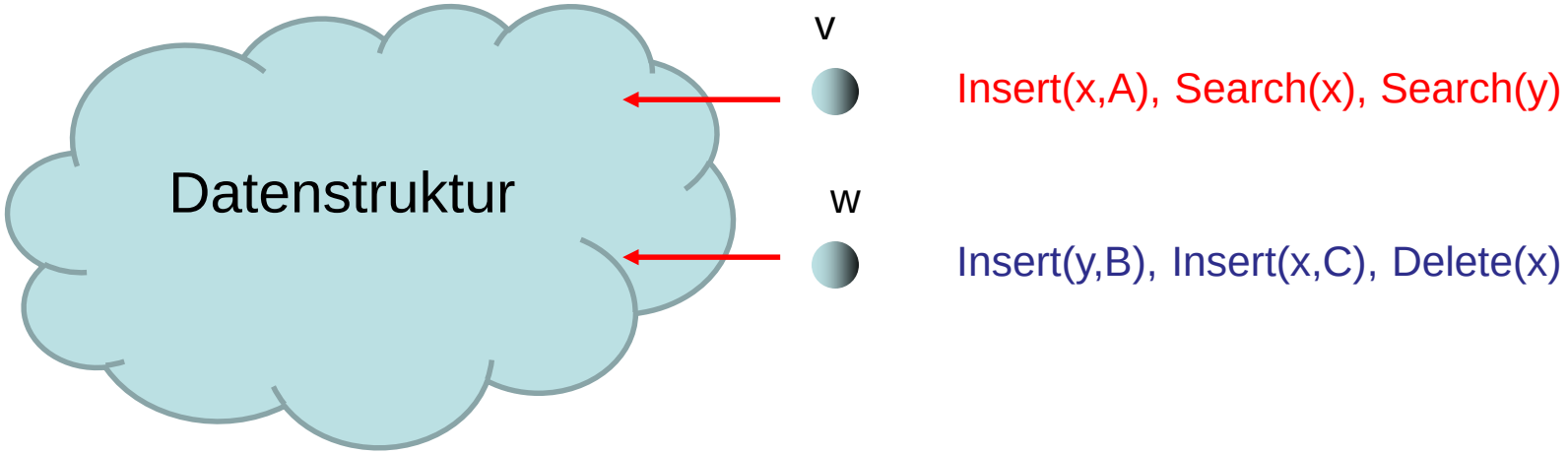
Zur Erinnerung:

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:



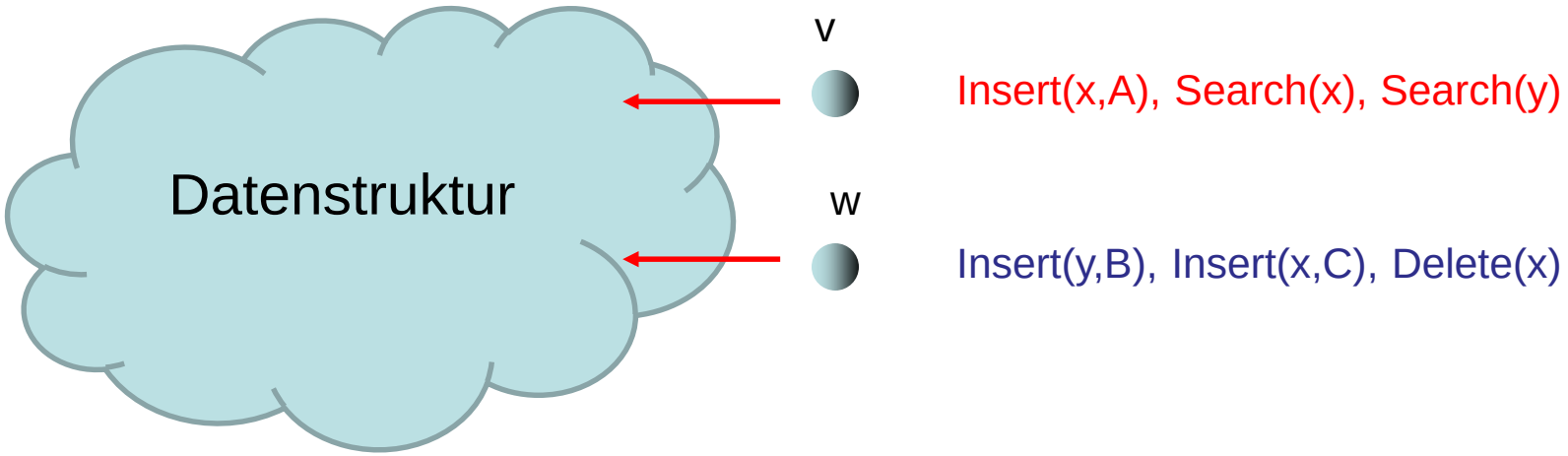
Eine lokal konsistente Ausgabe für v wäre C, B, da diese durch

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

zustande käme.

Informationsorientierte Datenstrukturen

Beispiel:



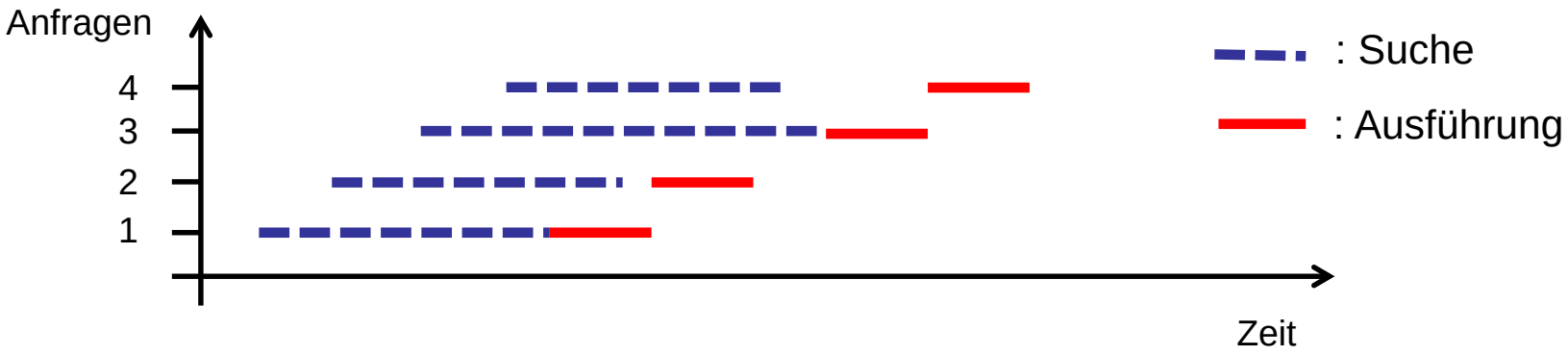
Einfachste Strategie für lokale Konsistenz:

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

Informationsorientierte Datenstrukturen

Lokal sequentielle Ausführung:

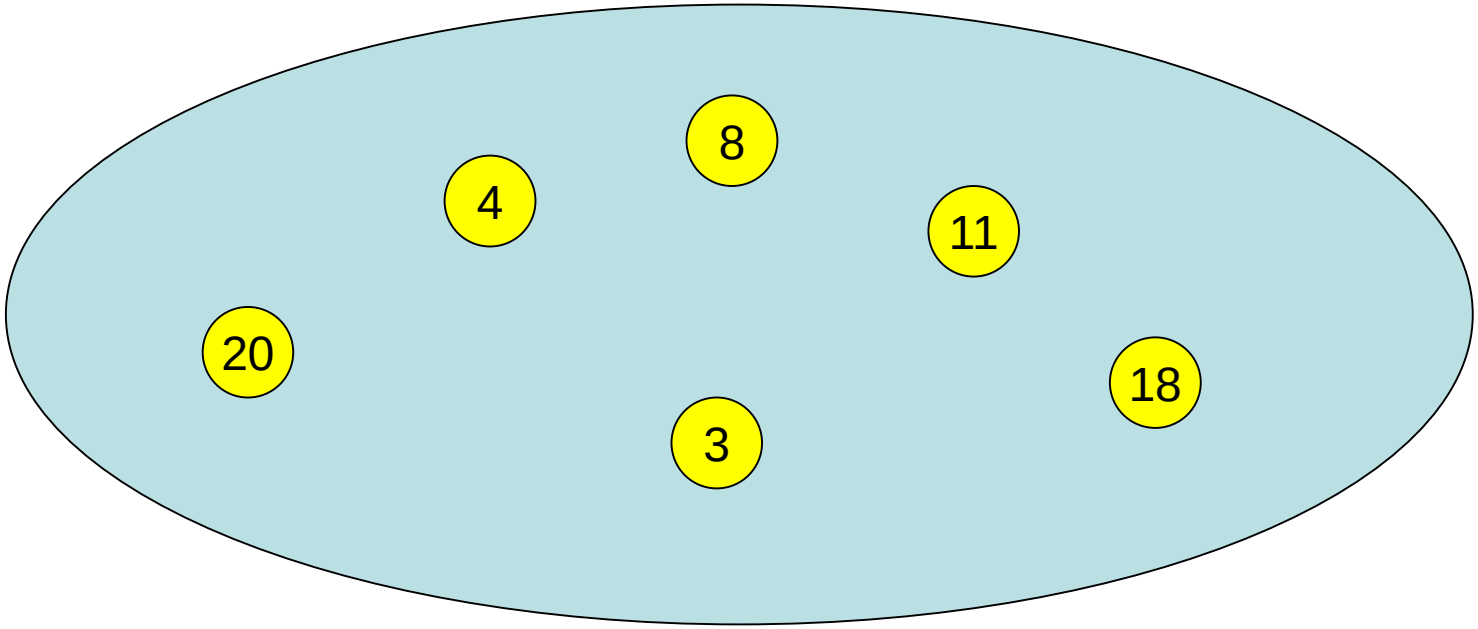
- Stelle für jede Datenanfrage zunächst (über Lookup) fest, mit welchen Prozessen Daten ausgetauscht werden. Das kann oft parallel geschehen.
- Führe dann den Datenaustausch sequentiell in der vorgegebenen Reihenfolge aus.



Übersicht

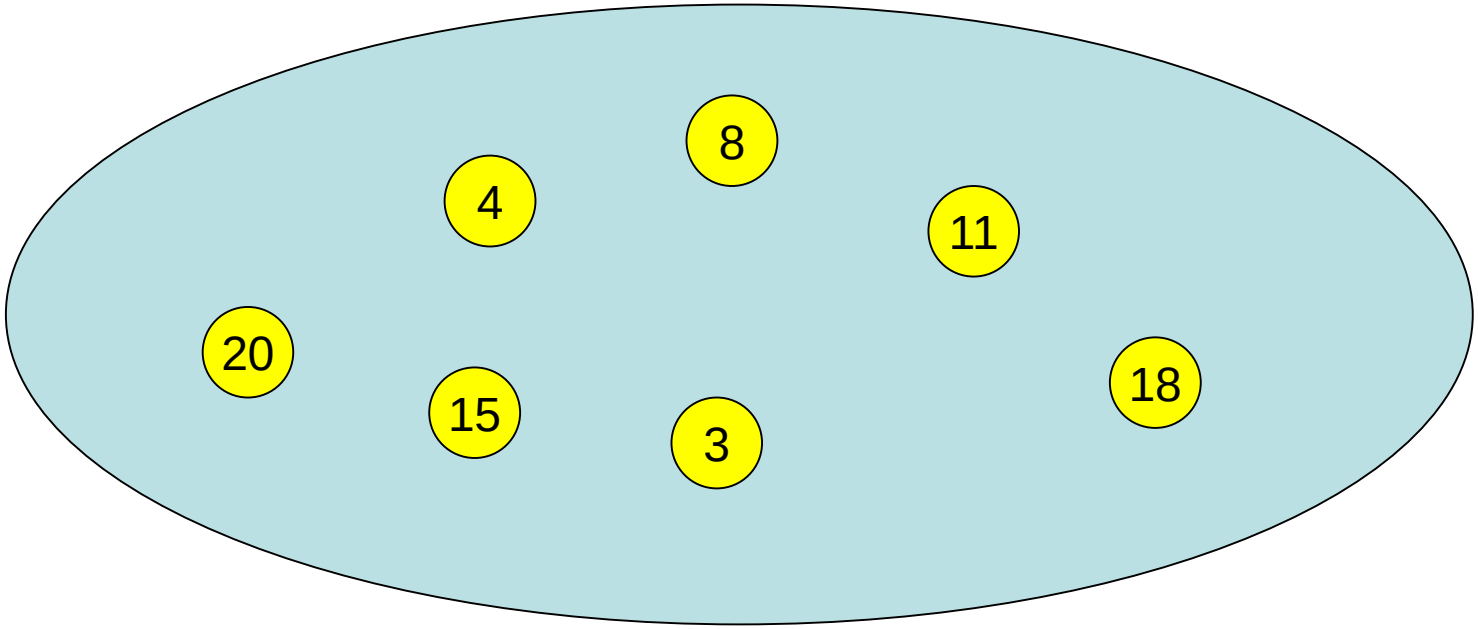
- Verteilte Hashtabelle
- Verteilte Queue
- Verteilter Stack
- Verteilter Heap

Wörterbuch



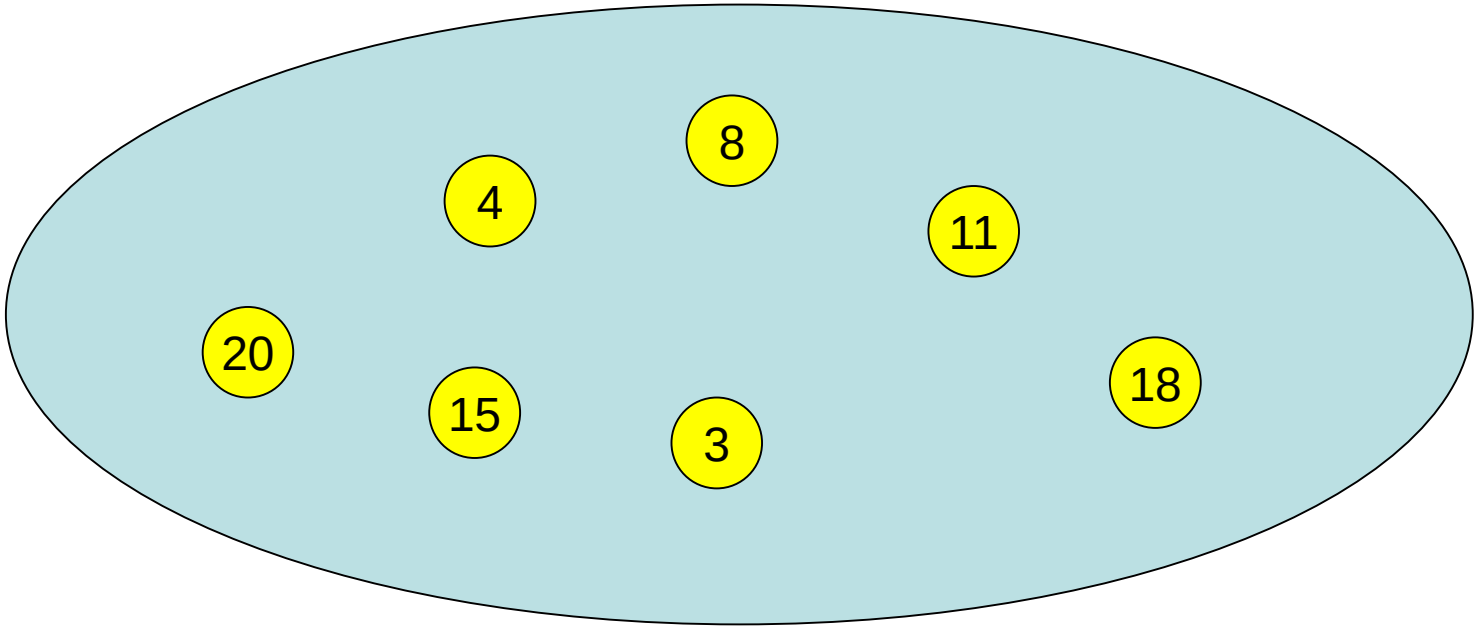
Wörterbuch

insert(15)



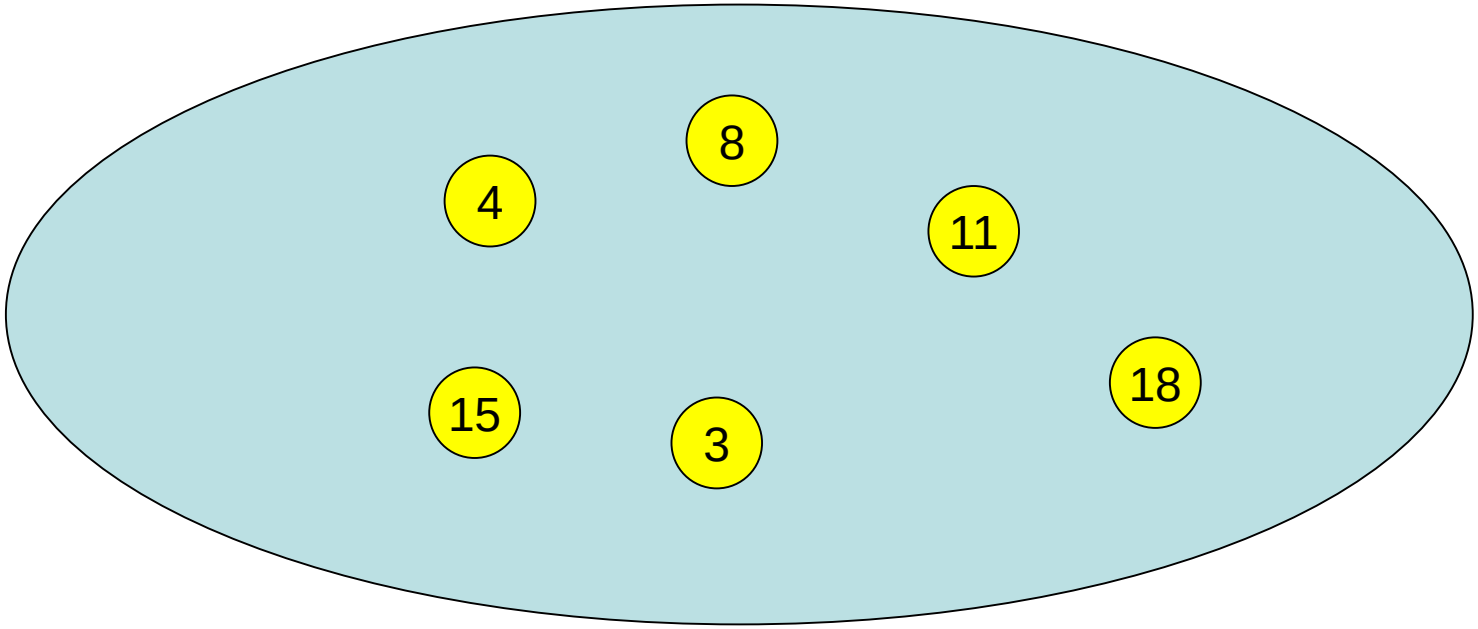
Wörterbuch

delete(20)



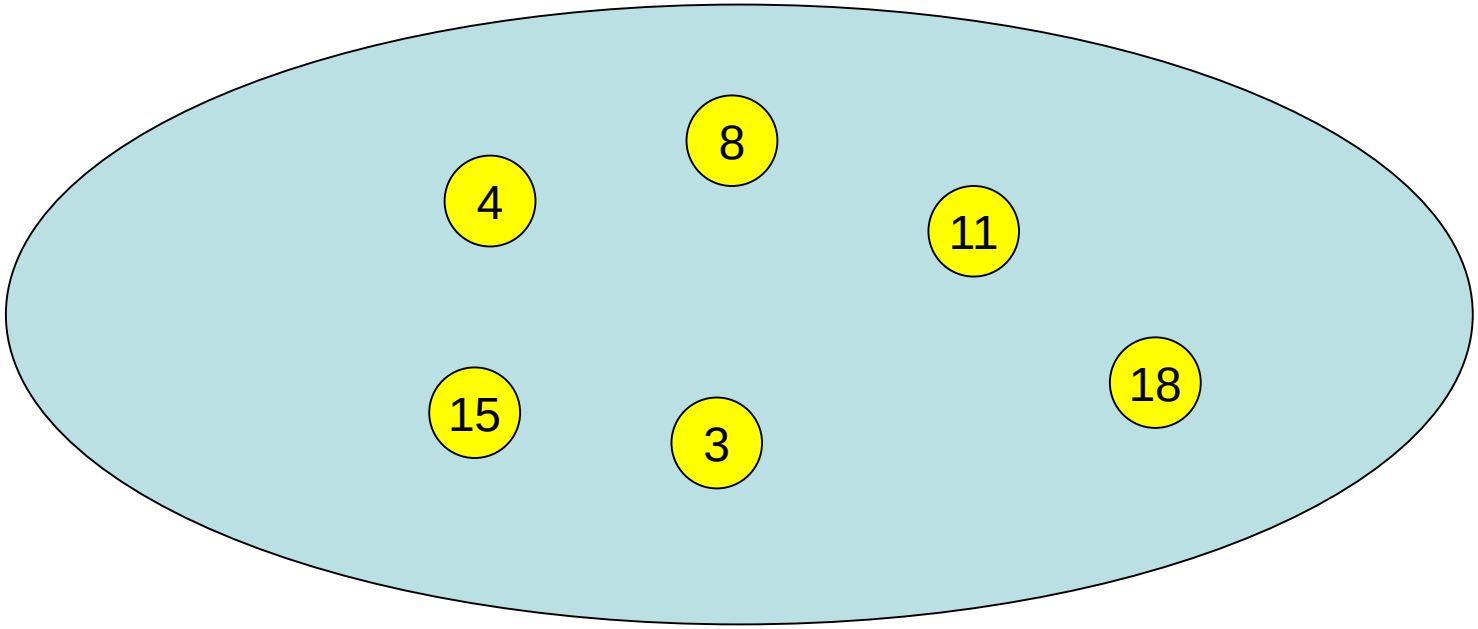
Wörterbuch

lookup(8) ergibt 8



Wörterbuch

lookup(7) ergibt ?



Wörterbuch-Datenstruktur

S: Menge von Elementen

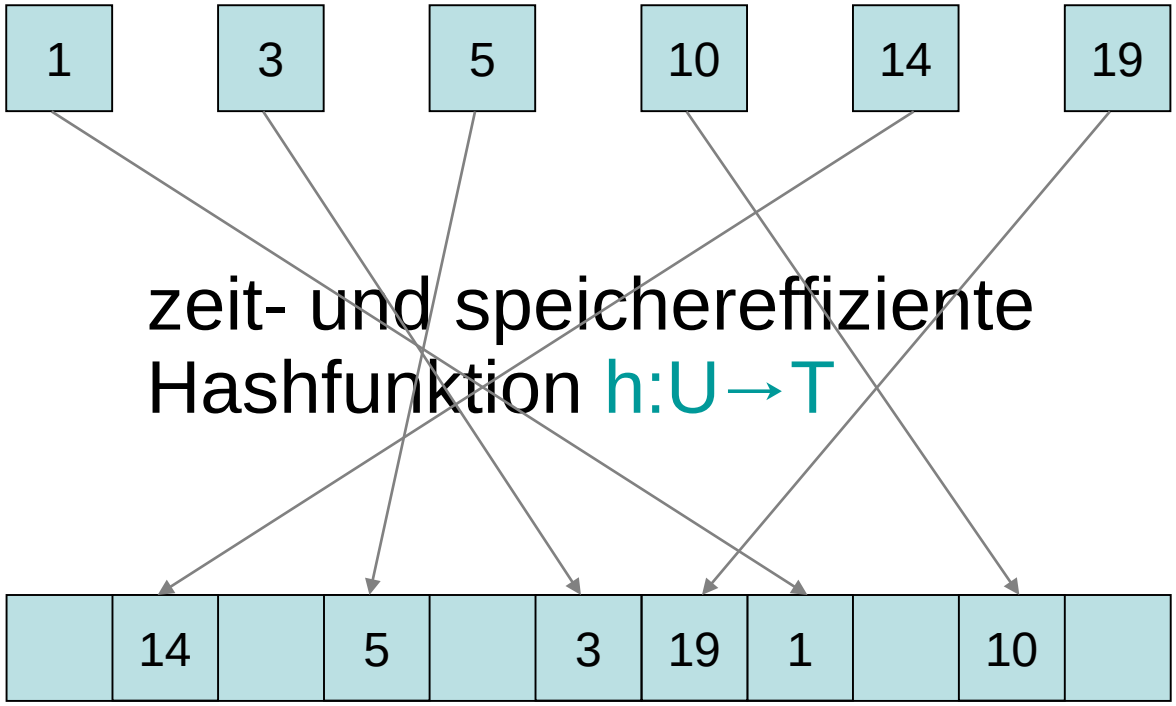
Jedes Element e identifiziert über key(e).

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): $S:=S\setminus\{e\}$, 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

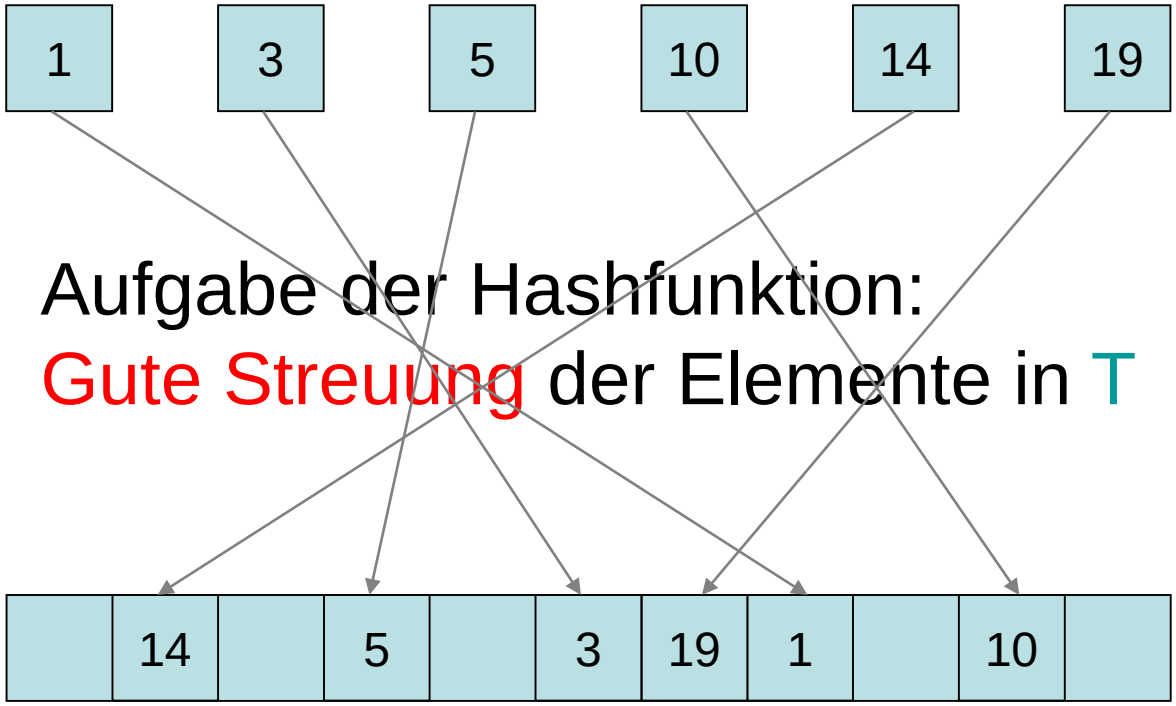
Effiziente Lösung: Hashing

Klassisches Hashing



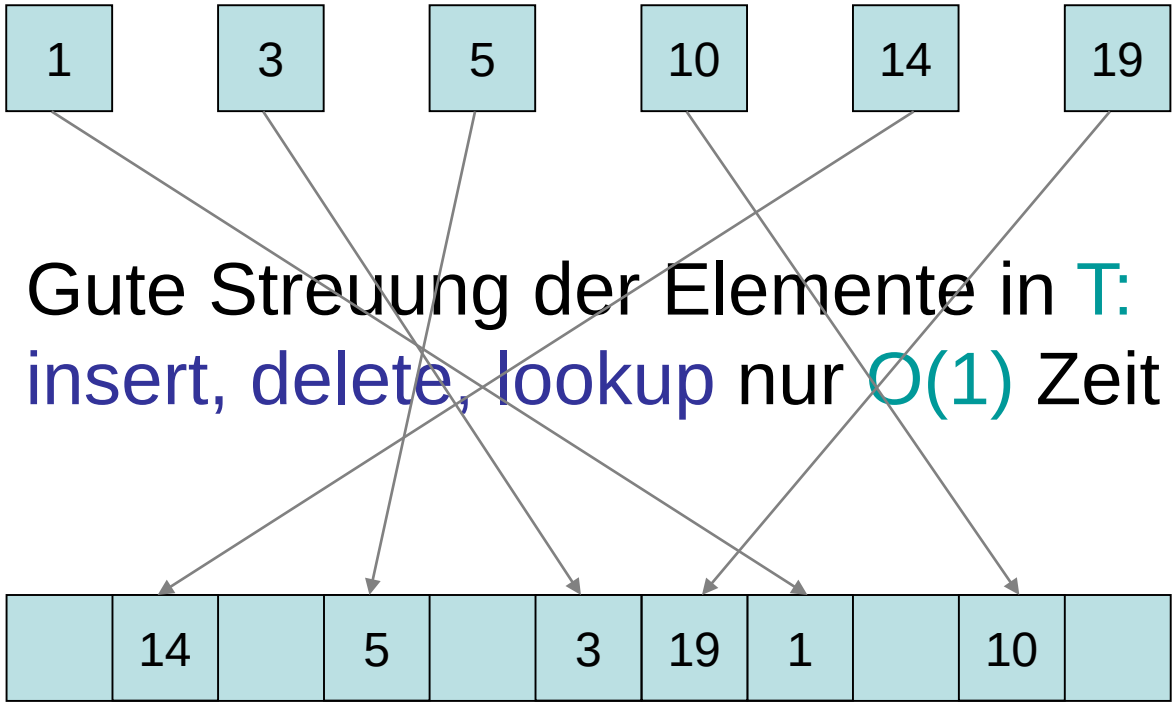
Hashtabelle T

Klassisches Hashing



Hashtabelle T

Klassisches Hashing

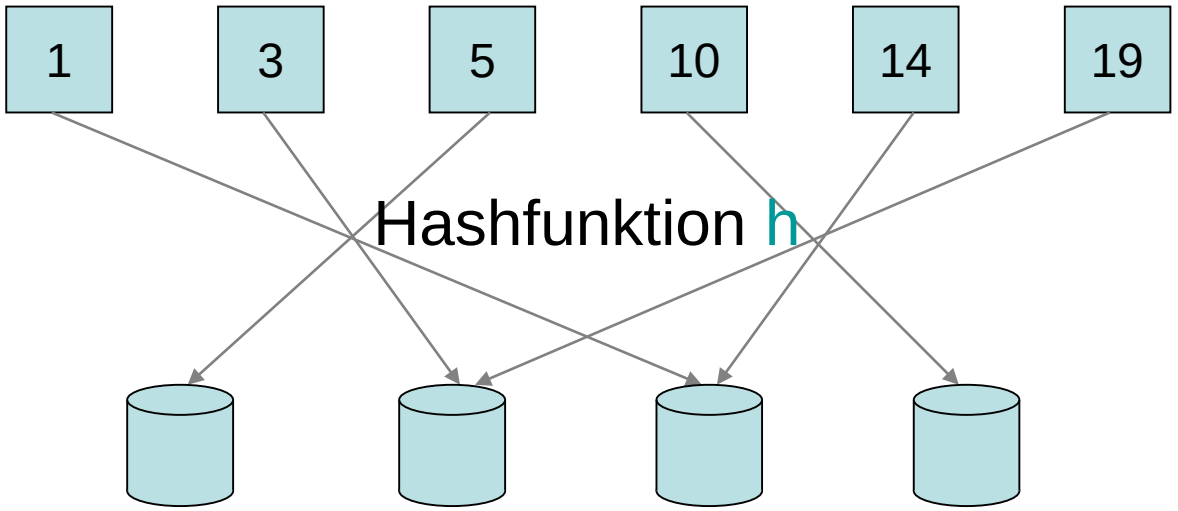


Gute Streuung der Elemente in T:
insert, delete, lookup nur O(1) Zeit

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.

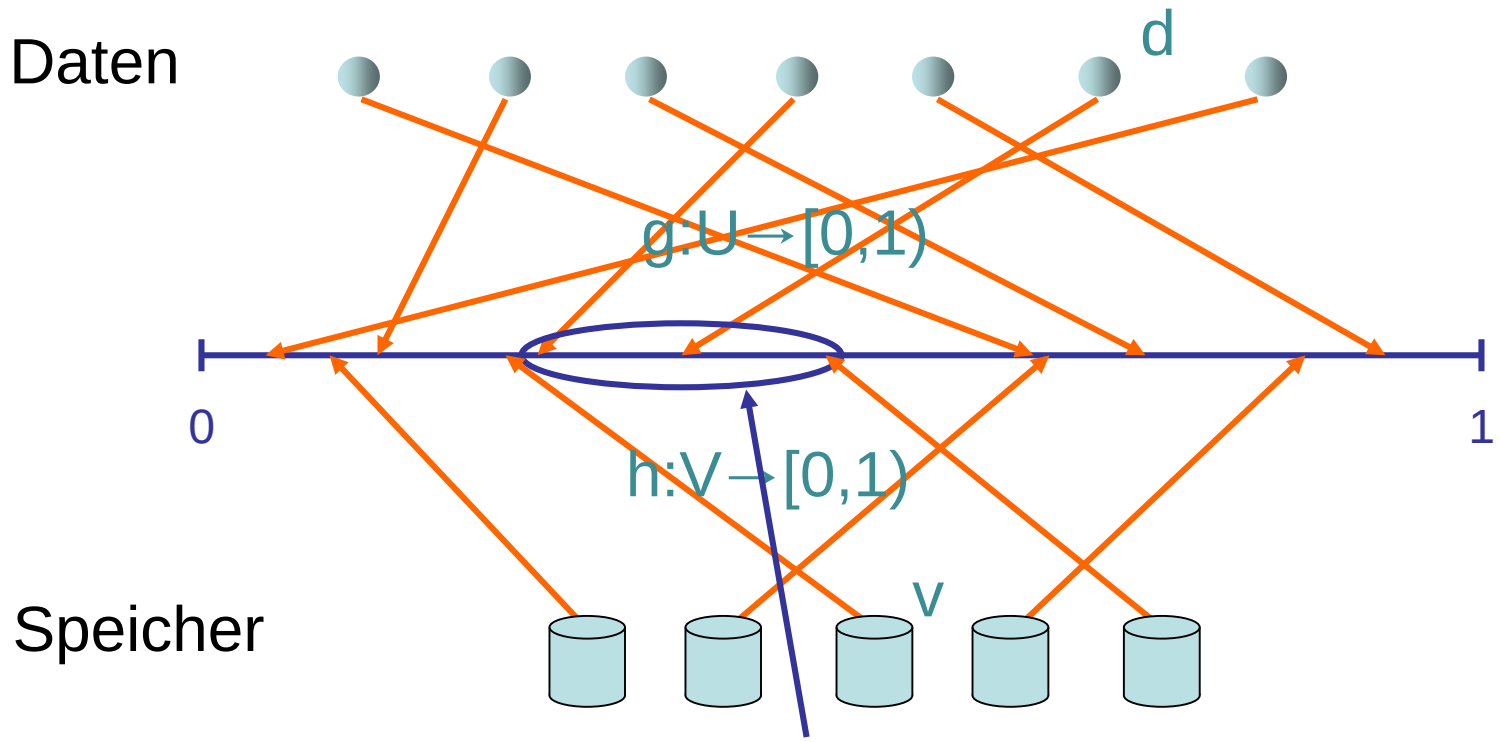
Nichtuniforme Speichersysteme: Kapazitäten können beliebig unterschiedlich sein

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

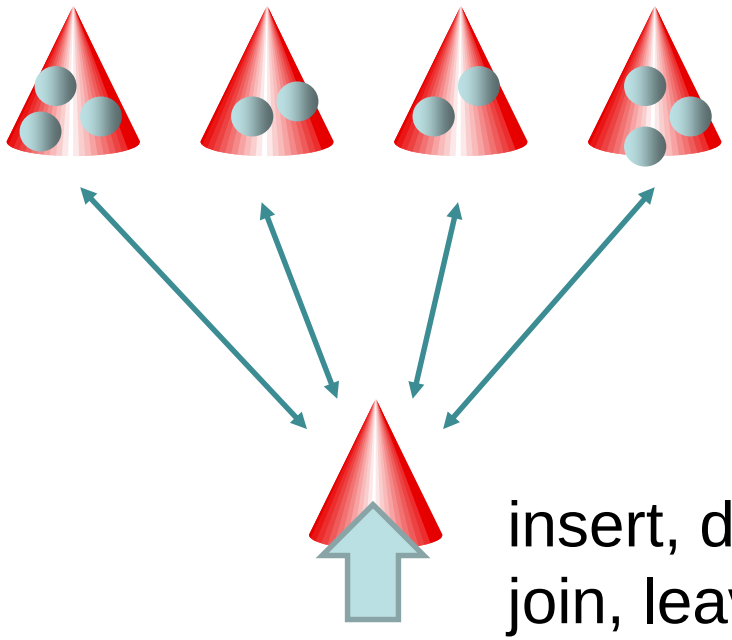
- V: aktuelle Prozessmenge (im folgenden auch Knoten genannt)
- $succ(v)$: nächster Nachfolger von v in V bzgl. Hashfunktion h (wobei $[0,1)$ als Kreis gesehen wird)
- $pred(v)$: nächster Vorgänger von v in V bzgl. Hashfunktion h

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

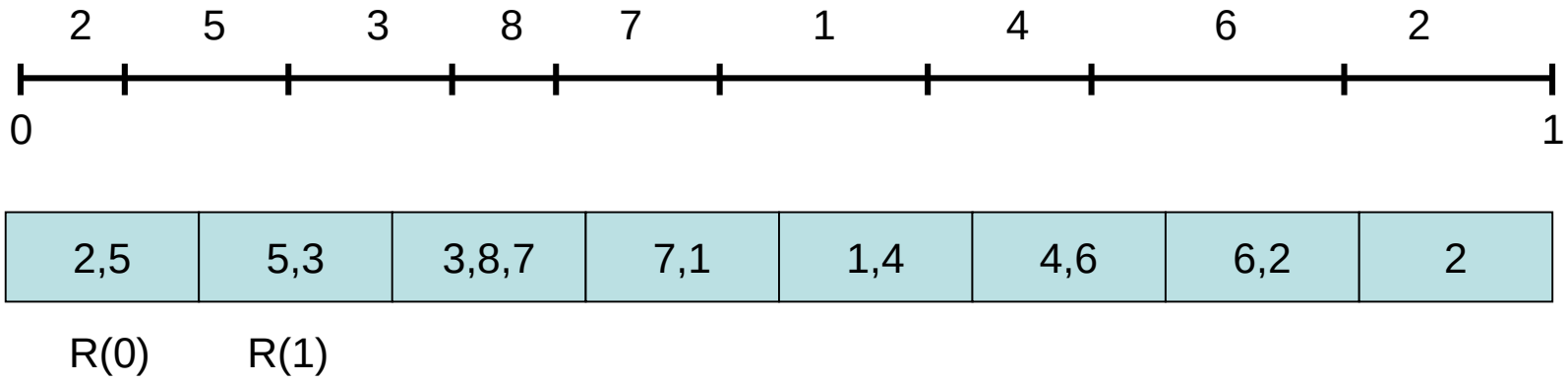


insert, delete, lookup,
join, leave

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:

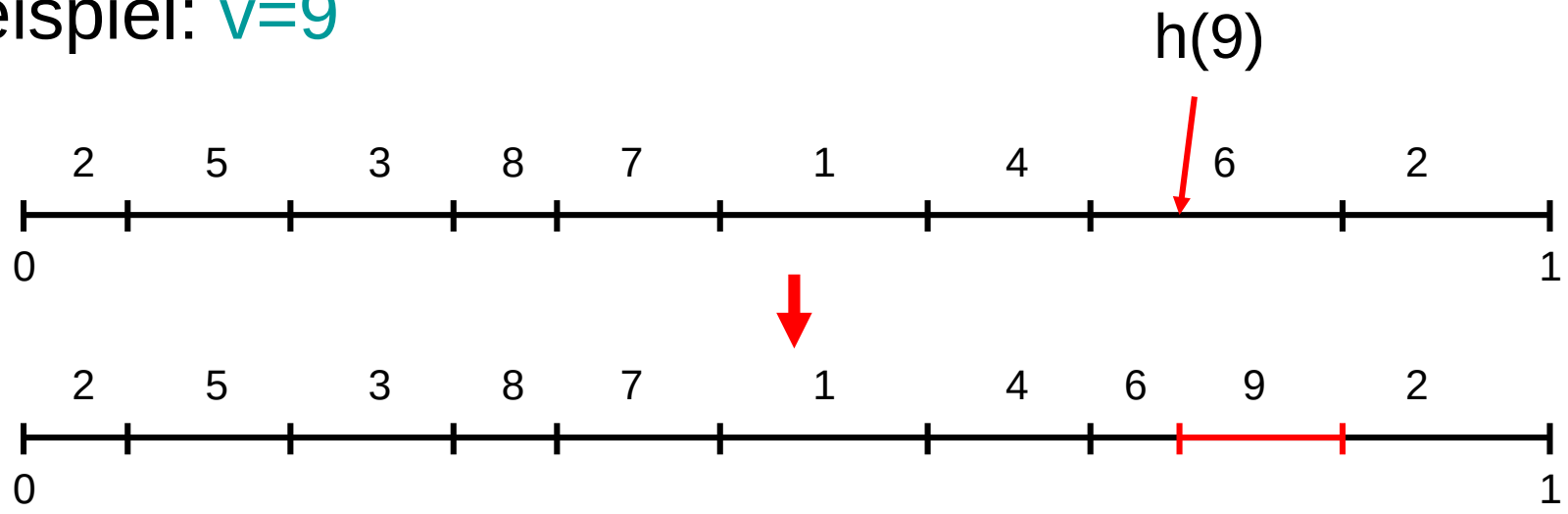
- $T[i]$ enthält für $m\geq n$ erwartet konstant viele Elemente und höchstens $O(\log n / \log\log n)$ mit hoher W.keit.
- D.h. Lookup(k) (die Ermittlung des zuständigen Knotens für einen Schlüssel) benötigt erwartet konstante Zeit

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

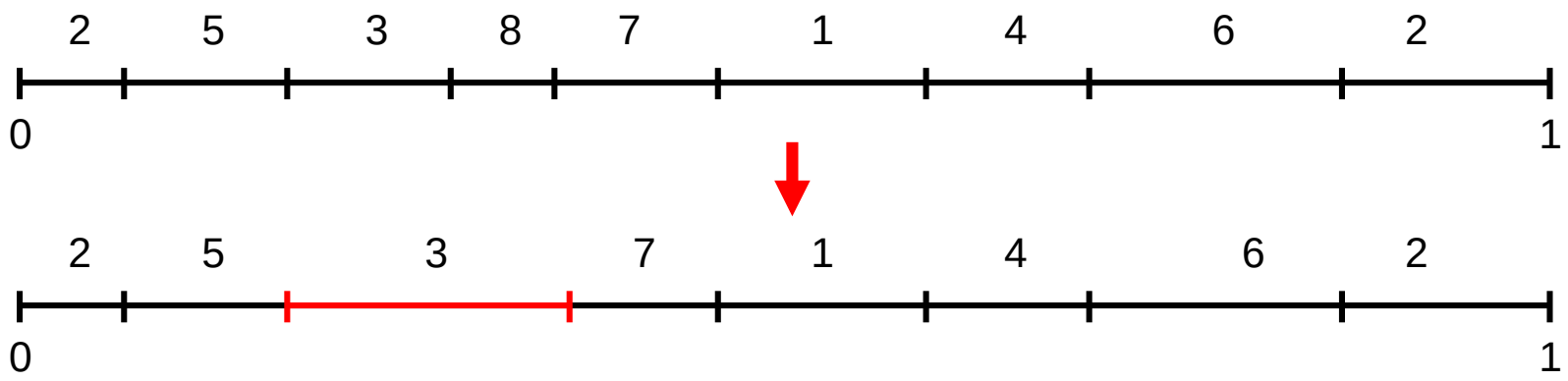


Verteilte Hashtabelle, Fall 1

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

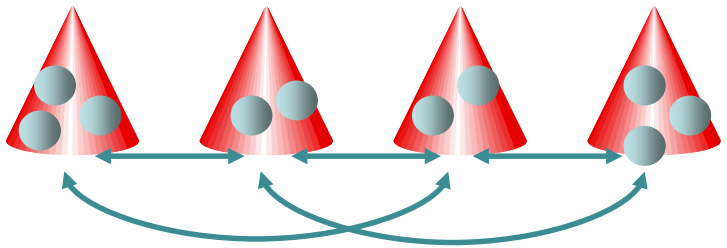
Problem: Schwankung um 1/n hoch!

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



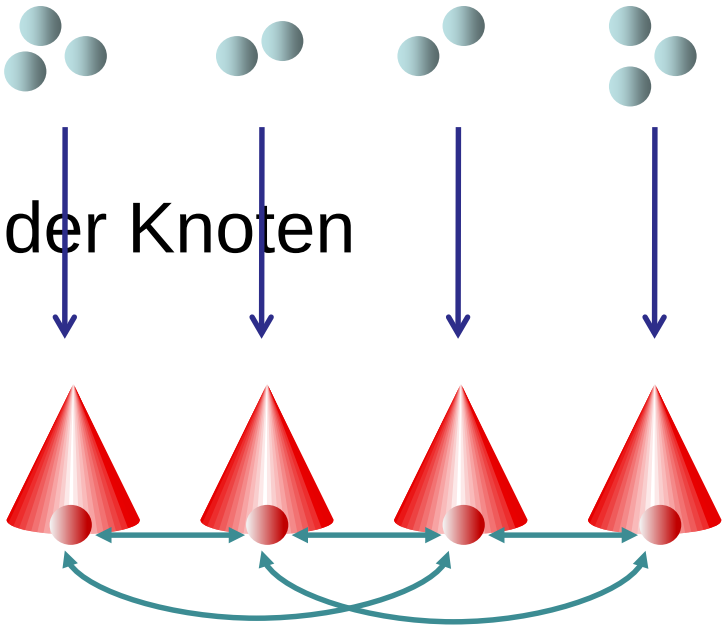
Jeder Knoten kann Anfragen (insert, delete, lookup, join, leave) generieren.

Verteilte Hashtabelle, Fall 2

Konsistentes Hashing: Zerlegung in zwei Probleme.

1. Abbildung der Daten auf die Knoten

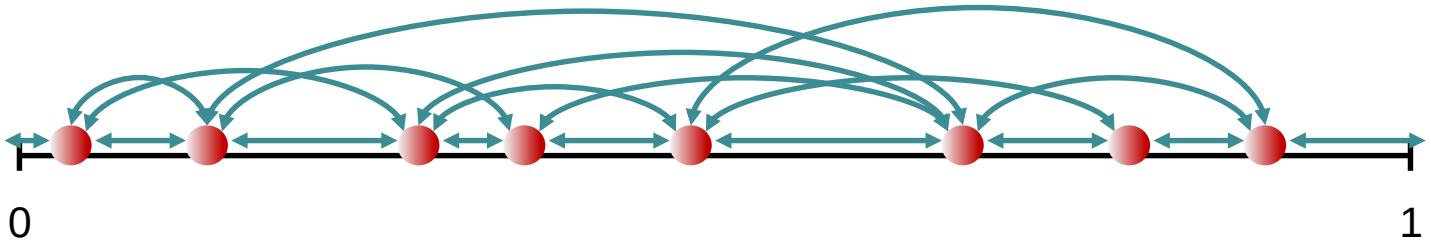
2. Vernetzung der Knoten



Verteilte Hashtabelle, Fall 2

Vernetzung der Knoten:

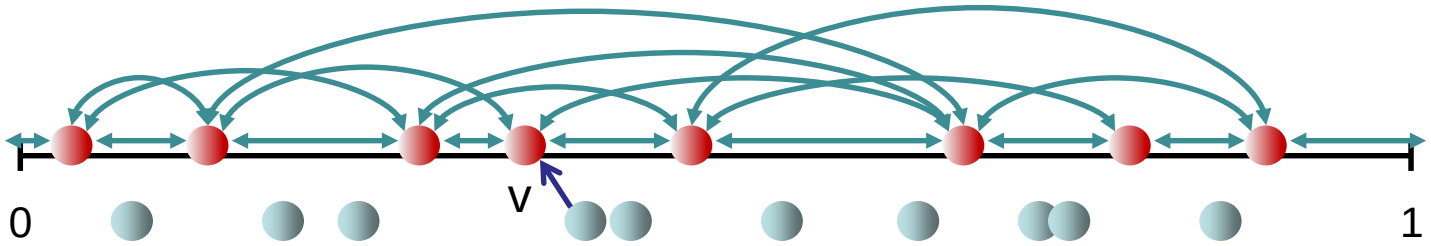
- Jedem Knoten v wird (pseudo-)zufälliger Wert $h(v) \in [0,1)$ zugewiesen.
- Verwende z.B. Skip+ Graph, um Knoten (hier im Kreis!) mittels $h(v)$ zu vernetzen.



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

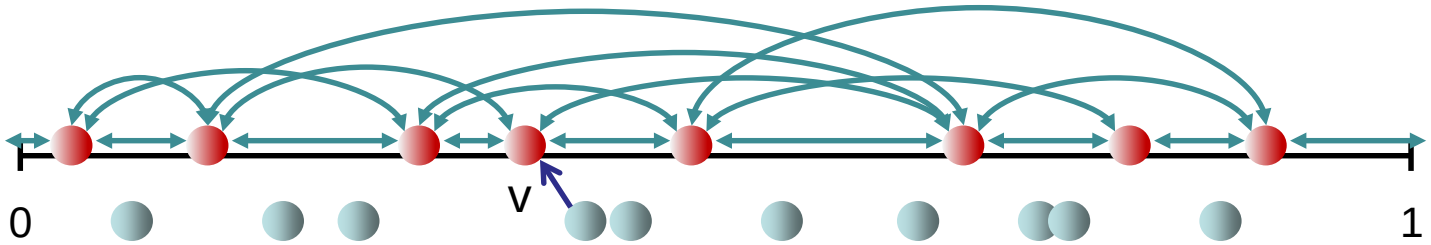
Passende search-Operation im Skip+ Graph für den Einsatz in der verteilten Hashtabelle:

```
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

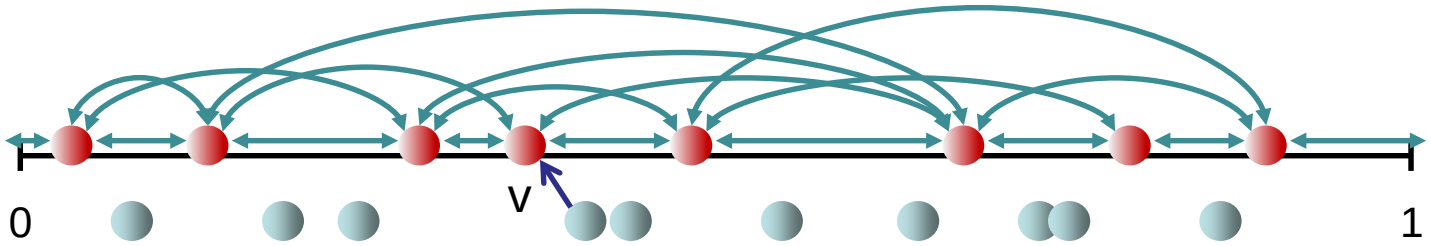


- delete(k): führe search(g(k)) im Skip+ Graph aus und lösche Datum d mit key(d)=k (falls da) im nächsten Vorgänger von g(k)

Verteilte Hashtabelle, Fall 2

Abbildung der Daten auf die Knoten:

- Verwende konsistentes Hashing

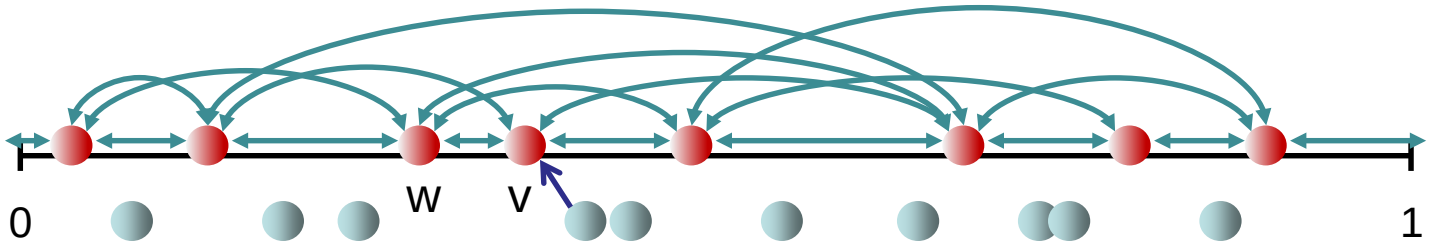


- lookup(k): führe search(g(k)) im Skip+ Graph aus und liefere Datum d mit key(d)=k (falls da) im nächsten Vorgänger von g(k) zurück

Verteilte Hashtabelle, Fall 2

Abbildung der Daten auf die Knoten:

- Verwende konsistentes Hashing

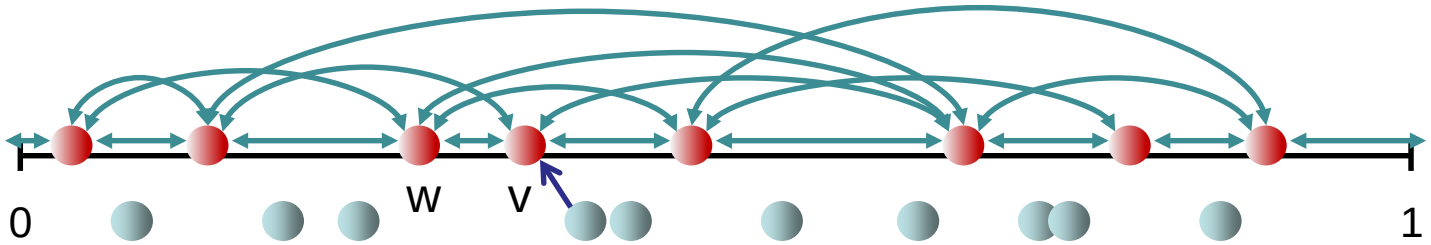


- join(v): nachdem v in den Skip+ Graph integriert ist, reicht es, pred(v)=w zu kontaktieren (mit welchem v direkt verbunden ist), um alle für v relevanten Daten gemäß des konsistenten Hashings zu erhalten

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 dass die Datenzuordnung wieder korrekt ist

Verteilte Hashtabelle, Fall 2

Satz 6.2: Im stabilen Zustand ist der Arbeitsaufwand (d.h. Anzahl Botschaften, die nicht von timeouts getriggert werden bzw. strukturelle Änderungen) 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): erwartet $O(\log n)$ (verbinde v mit beliebigem Knoten im Skip+ Graph, Rest durch Build-Skip)
- Leave(v): erwartet $O(\log n)$ (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: z.B. durch lokal sequentielle Ausführung. Benötigt aber einen legalen Zustand der verteilten Hashtabelle, damit Daten gefunden und damit bei Bedarf aktualisiert werden können.

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

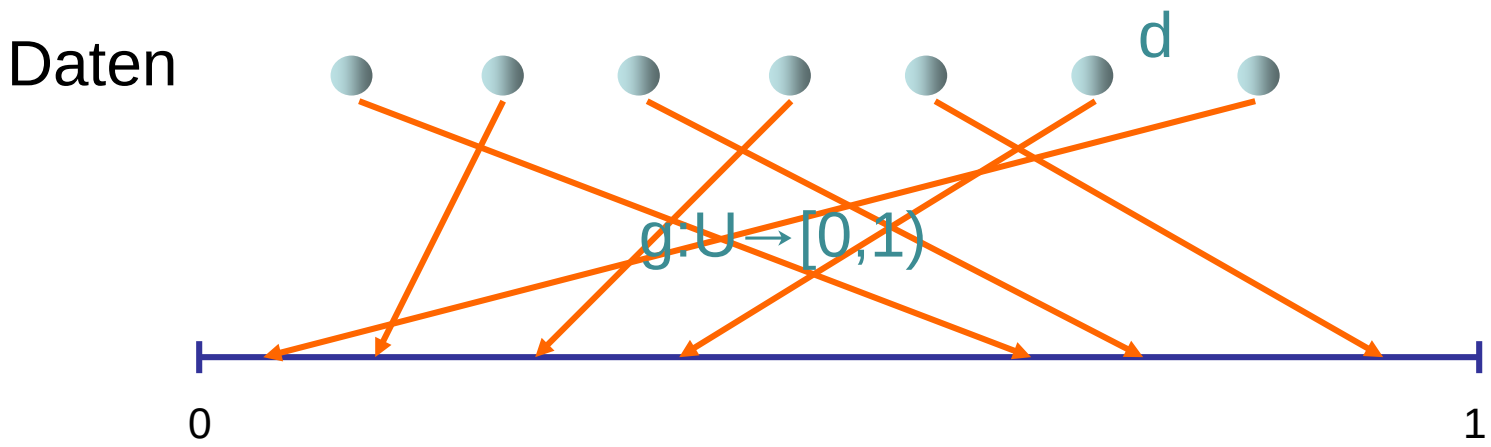
Situation hier: wir haben Knoten mit beliebigen relativen Kapazitäten $c_1,\ldots,c_n$, d.h. $\sum_i c_i = 1$.

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

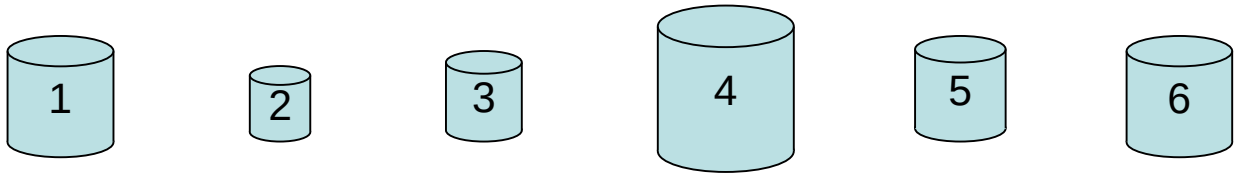
Lösung: SHARE

SHARE

Datenabbildung: wie bei konsistentem Hashing



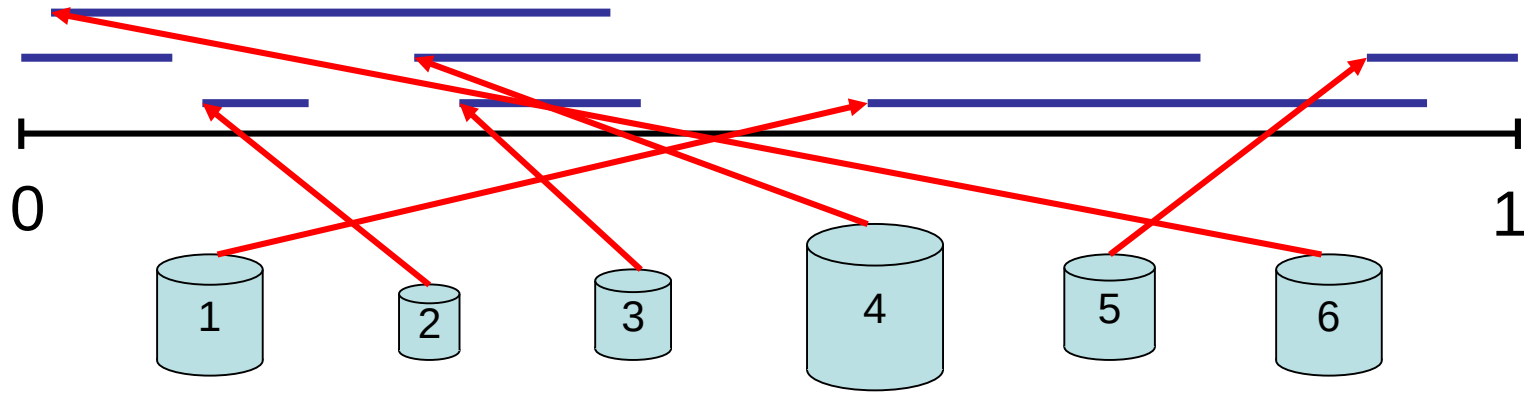
Zuordnung der Speicher: andere Strategie



SHARE

Zuordnung zu Speichern: zweistufiges Verfahren.

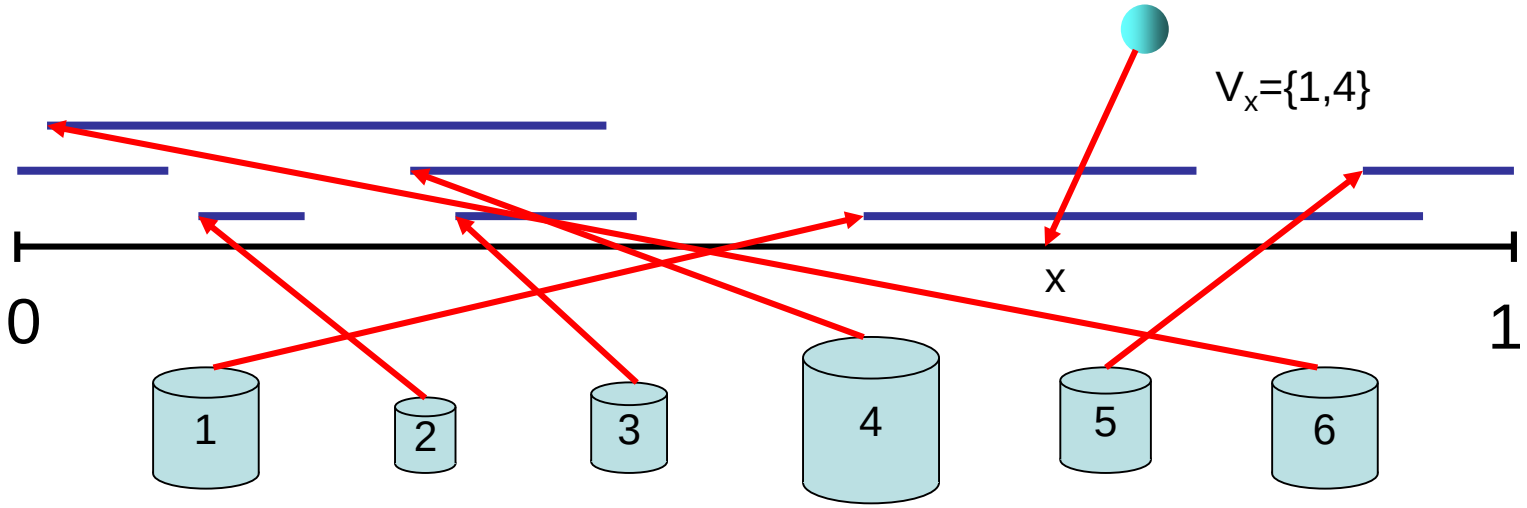
1. Stufe: Jedem Knoten v wird ein Intervall $I(v) \subseteq [0,1)$ der Länge $s \cdot c_v$ zugeordnet, wobei $s = \Theta(\log n)$ ein fester Stretch-Faktor ist. Die Startpunkte der Intervalle sind durch eine Hashfunktion $h: V \rightarrow [0,1)$ gegeben.



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).

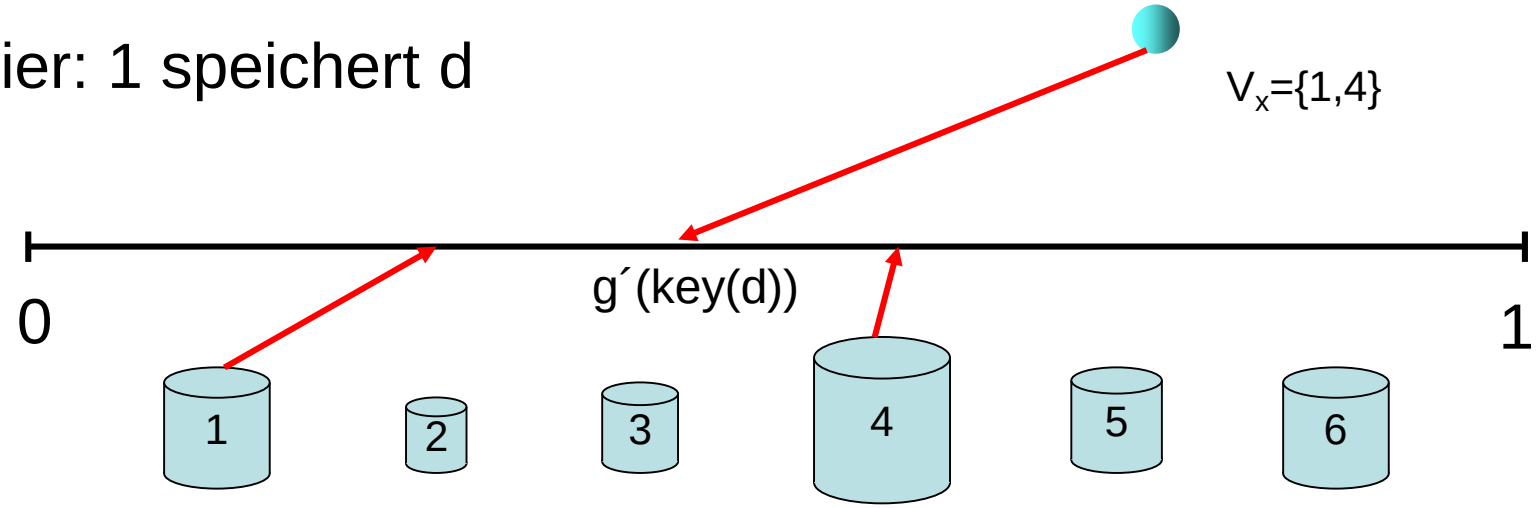


SHARE

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

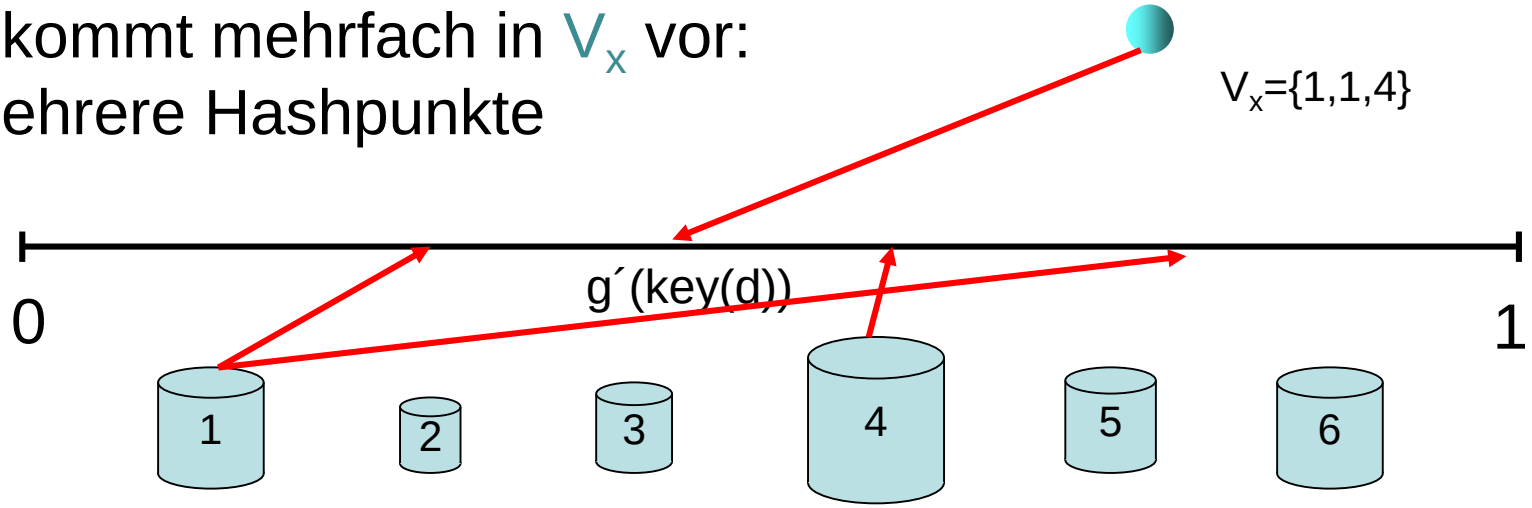


SHARE

Zuordnung zu Speichern: zweistufiges Verfahren.

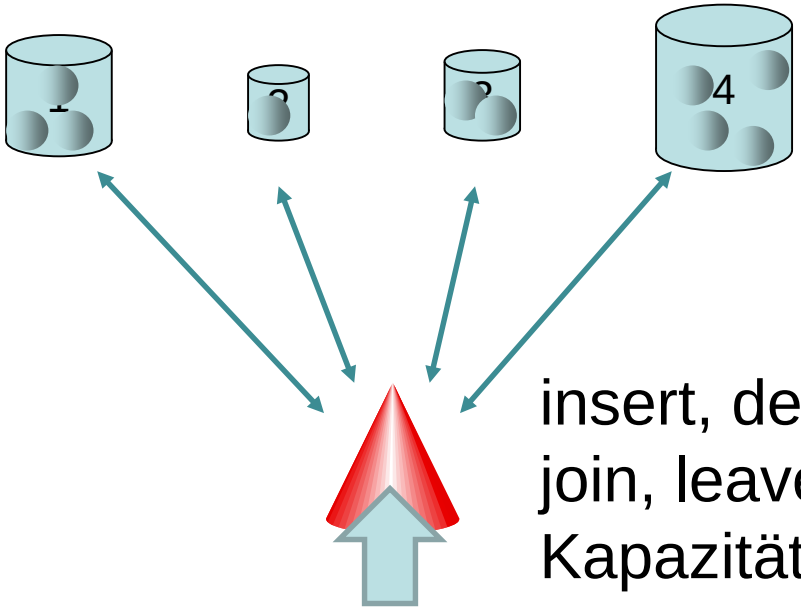
2. Stufe: Für Datum d wird mittels konsistentem Hashing mit Hashfunktionen h' und g' (die für alle Multimengen gleich sind) ermittelt, welcher Knoten in V_x Datum d speichert.

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)

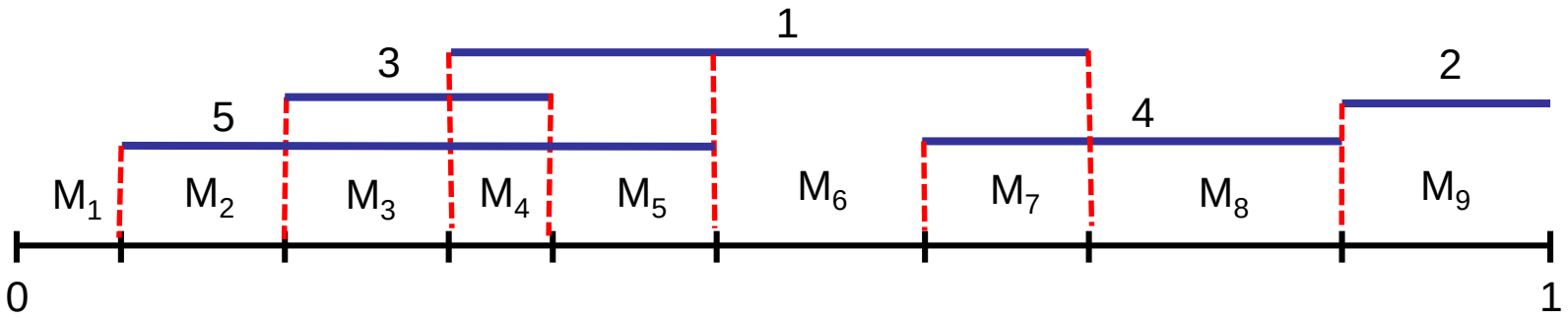
Laufzeit:

- 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.



M_1,M_2	M_2,M_3	M_3,M_4,M_5	M_5,M_6	M_6,M_7	M_7,M_8	M_8,M_9	M_9

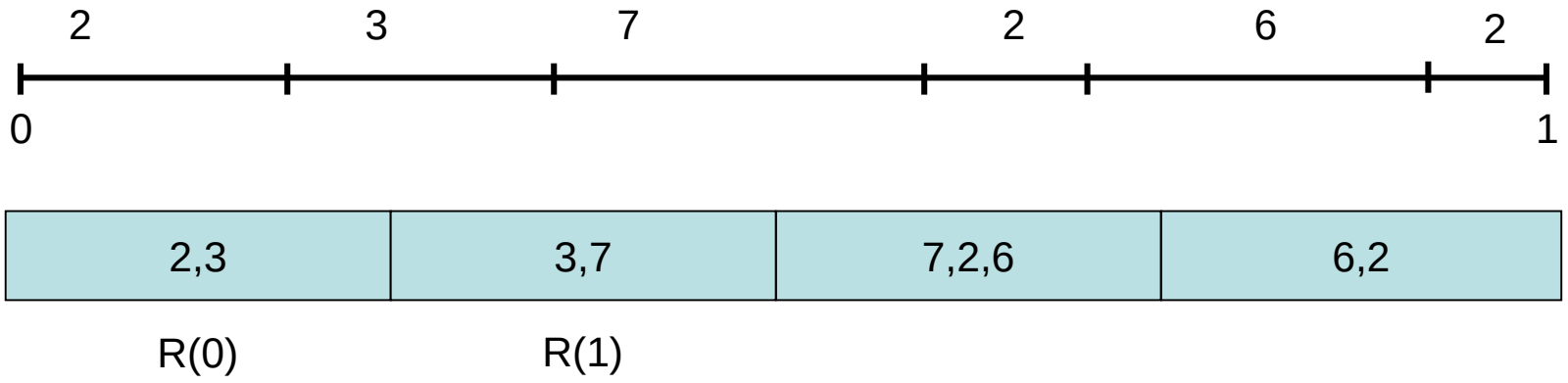
R(0) R(1)

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 $M=\{2,2,3,6,7\}$:



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

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

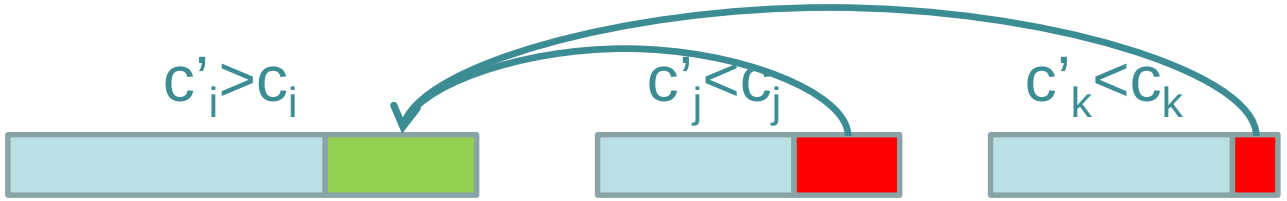
↑ ↑

Phase 1 Phase 2

SHARE

Punkt 3:

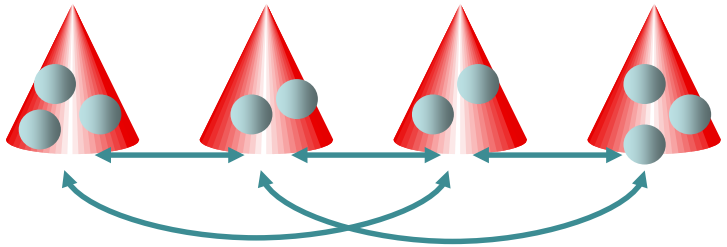
- Betrachte eine Veränderung der Kapazitäten von $(c_1,\ldots,c_n)$ nach $(c'_1,\ldots,c'_n)$
- Unterschied: $c = \sum_i |c_i - c'_i|$
- Optimale Strategie, um Fairness zu bewahren: replaziere einen $c/2$-Anteil der Daten



- 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.

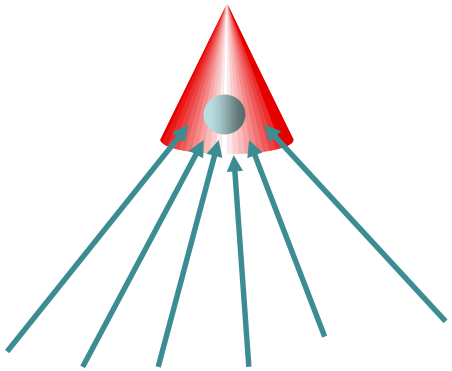
Nichtuniforme Speichersysteme: Kapazitäten können beliebig unterschiedlich sein

Vorgestellte Strategien:

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

Verteilte Hashtabelle

Probleme bei vielen Anfragen auf dasselbe Datum:



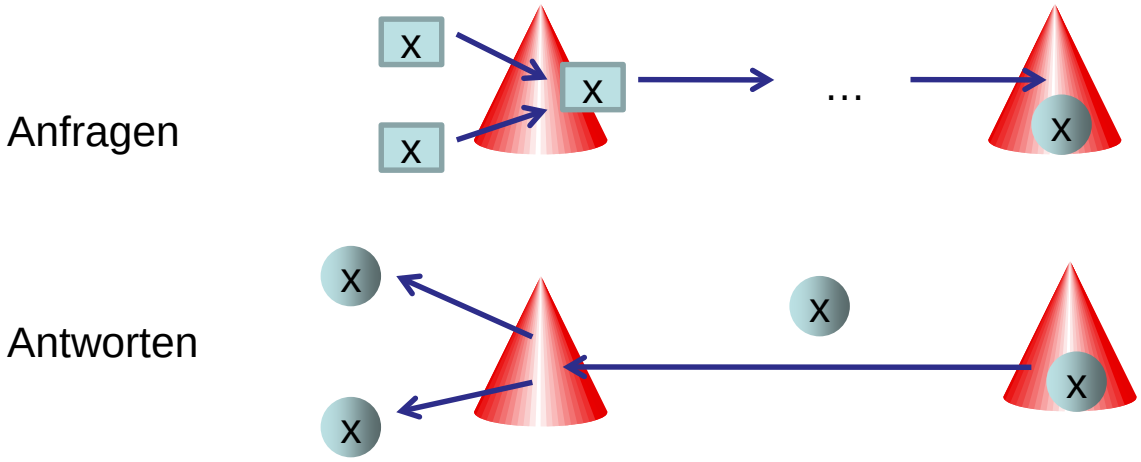
Prozess, der Datum speichert, wird überlastet.

Lösung: Combine & Split

Verteilte Hashtabelle

Combine & Split:

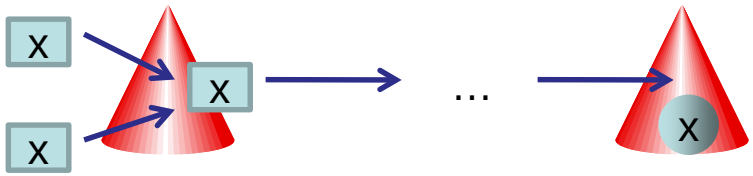
- Jeder Prozess v merkt sich alle Suchanfragen, die bei ihm eintreffen. Hat er für einen Schlüssel x bereits eine Suchanfrage weitergeleitet, dann hält er alle weiteren eingehenden Suchanfragen für x zurück (combine), beantwortet diese (split) sobald er eine Antwort zu x erhält und löscht diese dann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

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 wir combine&split verwenden, dass die maximale Anzahl an Anfragen über einen Knoten bis auf einen konstanten Faktor höchstens so groß ist wie die maximale Anzahl an Zielen, für die Anfragen über einen Knoten laufen wollen.

Beweis: Übung

Bemerkung: Satz 6.4 ist auf Anfragen auf Daten statt Knoten übertragbar, wenn die Daten uniform zufällig auf die Knoten verteilt sind (wie im konsistenten Hashing) und die Wahl der Daten unabhängig von deren Knotenplatzierung ist. Dann passiert es nämlich nur mit sehr kleiner Wahrscheinlichkeit, dass Anfragen auf verschiedene Daten zum gleichen Knoten müssen. Solche Anfragen wären nämlich nicht kombinierbar.

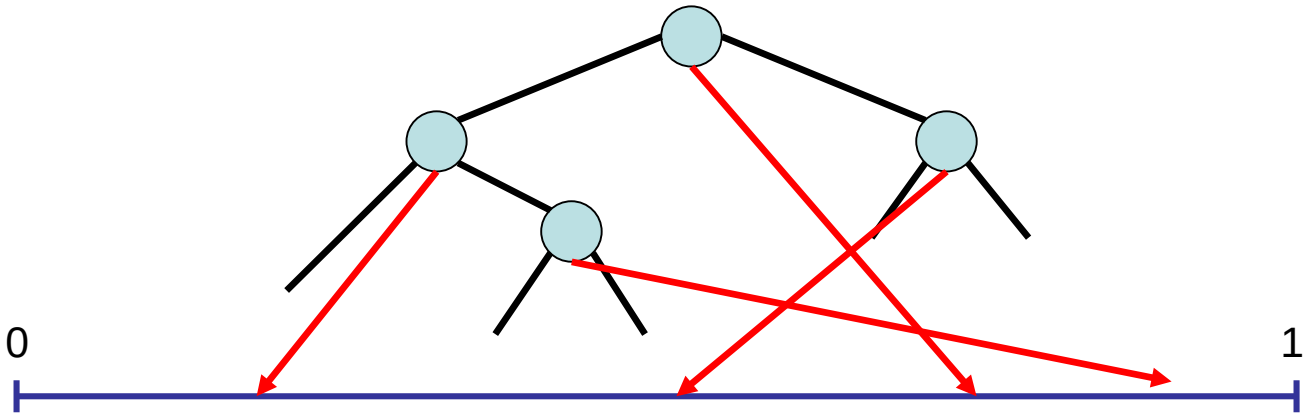
Verteilte Hashtabelle

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.

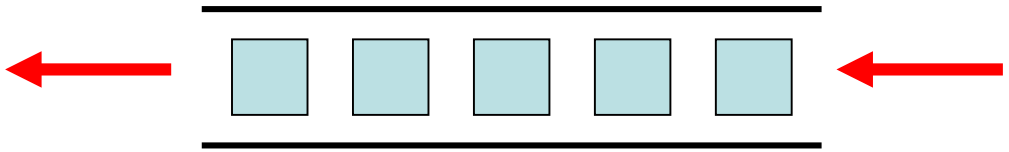
Übersicht

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

Konventionelle Queue

Eine Queue Q unterstützt folgende Operationen:

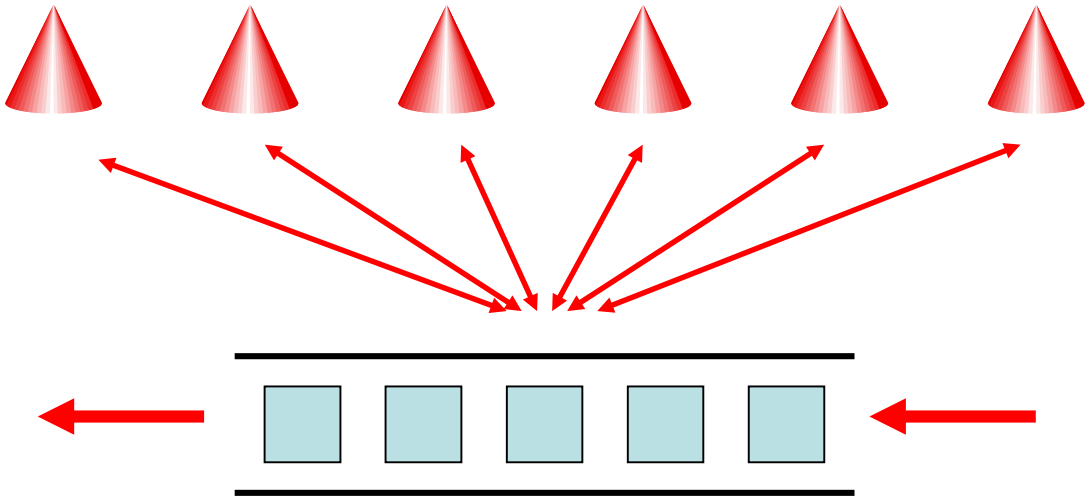
- enqueue(Q,x): fügt Element x hinten an die Queue Q an.
- dequeue(Q): holt das vorderste Element aus der Queue Q heraus und gibt es zurück



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

Verteilte Queue

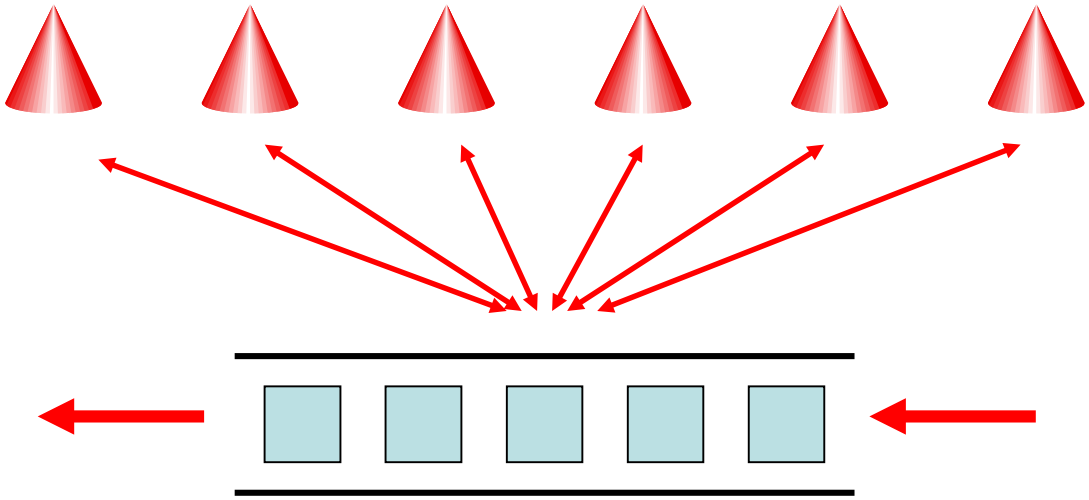
Viele Prozesse agieren auf Queue:



Verteilte Queue

Probleme:

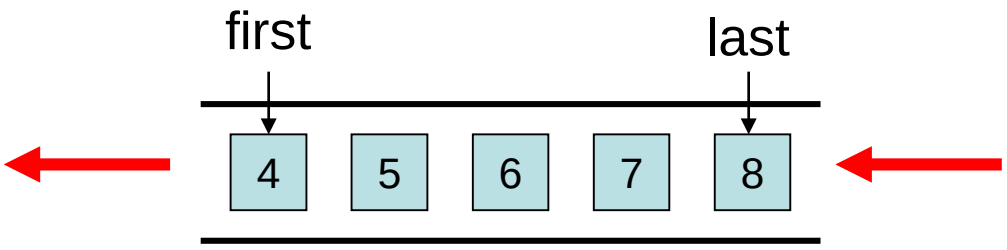
- Speicherung der Queue
- Realisierung von enqueue und dequeue



Verteilte Queue

Speicherung der Queue:

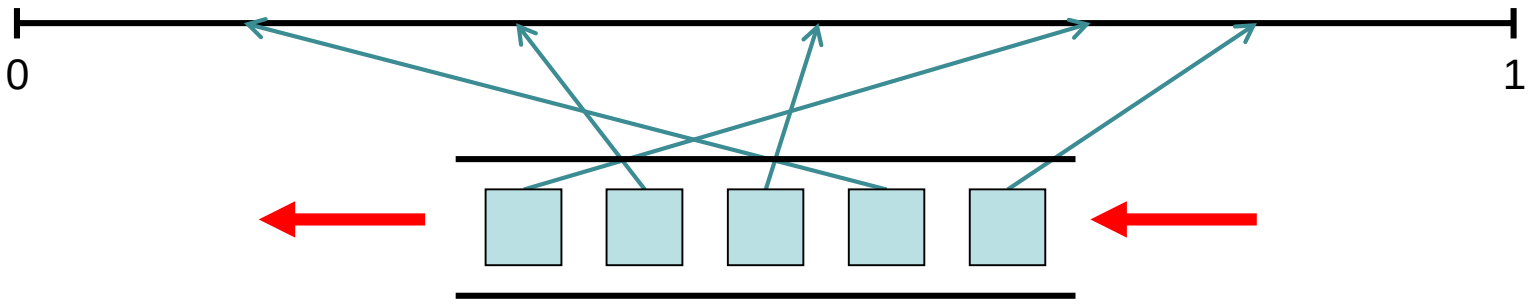
- Jedes Element x besitzt eindeutige Position $pos(x) \geq 1$ in der Queue (das vorderste hat die kleinste und das hinterste die größte Position).



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

Realisierung von dequeue(Q):

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

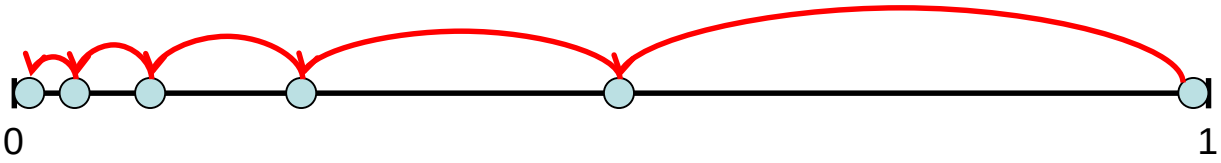
Noch zu klären: Punkt 1 in enqueue und dequeue.

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

Verteilte Queue

Realisierung von enq(Q,1):

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



- Der am nächsten an 0 liegende Knoten v_0 (Anker) 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 enq(Q,i) Anfrage samt Knoten w, der diese an ihn geschickt hat.
- Angenommen, v habe bei Ausführung von timeout die Anfragen enq(Q,i_1), enq(Q,i_2), …, enq(Q,i_k) angesammelt. Dann sendet v eine enq(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 enq(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 enq(Q,i_1), enq(Q,i_2),…,enq(Q,i_k) kombiniert wurde, so schickt v das Intervall [pos, pos+i_1-1] an die Rückadresse von enq(Q,i_1), [pos+i_1, pos+i_1+i_2-1] an die Rückadresse von enq(Q,i_2), usw.
- Am Ende erhält jede enq(Q,1) Anfrage eine eindeutige Position in der Queue.

Verteilte Queue

Realisierung von deq(Q,1):

- Schicke alle deq(Q,1) Anfragen in Richtung des Ankers v_0 mit Hilfe des de Bruijn Routings.
- Die deq(Q,i) Anfragen werden wie die enq(Q,i) Anfragen zu v_0 hin kombiniert.
- Erreicht eine deq(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 deq(Q,i) Anfrage erhält, teilt dieses in Teilintervalle gemäß der deq(Q,j) Anfragen auf, die zur deq(Q,i) Anfrage beigetragen haben, und schickt diese an deren Rückadressen zurück. Sollte das Intervall zu klein sein, wird für einige deq(Q,j) Anfragen nur ein verkleinertes oder leeres Intervall zurückgeschickt.
- Am Ende bekommt dann jede deq(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 Arbeit O(log n)
- dequeue(Q): erwartete Arbeit O(log n)

Verwaltung mehrerer Queues in derselben Hashtabelle: weise jeder Queue statt Punkt 0 einen (pseudo-) zufälligen Punkt in [0,1) zu. Verwende dann besser Skip+ Graph.

Problem: enqueue/dequeue Lösung garantiert nicht die lokale Konsistenz, selbst wenn die verteilte Queue im legalen Zustand ist!

Verteilte Queue

Annahme: verteilte Queue ist im legalen Zustand

Einfache Lösung für lokale Konsistenz:

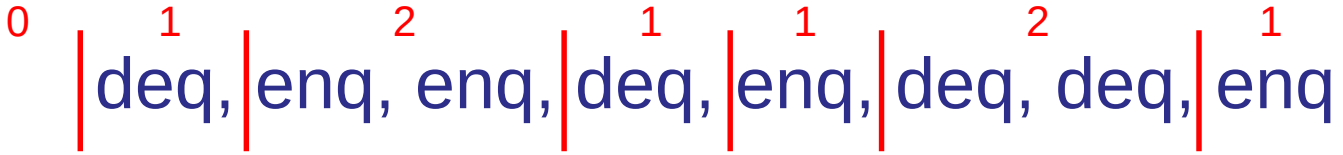
- Jeder Knoten v wartet solange mit der Ausführung einer Operation, bis die vorherige Operation bearbeitet worden ist.

Problem: geringe Bearbeitungsrate der Operationen.

Besser: jeder Knoten bearbeitet ganze Folgen von Operationen und wartet mit der Ausführung einer nachfolgenden Folge bis die Ausführung der vorherigen Folge beendet ist.

Verteilte Queue

- Angenommen, v habe als aktuelle Operationsfolge



- Dann fasst v diese zusammen zu der Anfrage serve(0,1,2,1,1,2,1), wobei für alle $i \geq 1$ die $2i-1$-te Zahl die Länge der i-ten enqueue-Folge und die $2i$-te Zahl die Länge der i-ten dequeue Folge in der Operationsfolge angibt.
- Diese serve-Anfrage wird dann in Richtung des Ankers geschickt und auf ihrem Weg mit anderen serve-Anfragen kombiniert.

Verteilte Queue

Kombinierung von serve-Anfragen in Richtung des Ankers:

- Angenommen, v habe bei Ausführung von timeout die Anfragen $serve(a_1,a_2,\ldots,a_k)$, $serve(b_1,b_2,\ldots,b_k)$ $serve(c_1,c_2,\ldots,c_k),\ldots$ angesammelt (wobei wir fehlende Werte mit Nullen auffüllen).
- Dann sendet v eine $serve(z_1,z_2,\ldots,z_k)$ Anfrage weiter in Richtung v_0, wobei $z_i=a_i+b_i+c_i+\ldots$ für alle i ist.
- Zusätzlich merkt sich v diese Kombinierung und die Rückadressen der einzelnen Anfragen.

Verteilte Queue

Bearbeitung einer serve(a_1,a_2,…,a_k) Anfrage im Anker:

- Der Anker berechnet die Intervalle $[x_1,y_1]$, $[x_2,y_2]$, $[x_3,y_3]$,… wie vorher für separate, aufeinanderfolgende enq(Q, a_1), deq(Q,a_2), enq(Q,a_3),… Anfragen und schickt ($[x_1,y_1]$, $[x_2,y_2]$, $[x_3,y_3]$,…) zurück an die Rücksprungadresse von serve(a_1,a_2,…,a_k).
- Von dort aus werden die Intervalle aufgeteilt wie vorher für separate enqueue und dequeue Anfragen, bis jeder Knoten, der eine serve-Anfrage initiiert hat, seine Intervalle bekommen hat.
- Das erlaubt es dann jedem Knoten, alle Anfragen seiner in einer serve-Anfrage kombinierten enqueue/dequeue Folge auf einen Schlag zu bearbeiten, indem parallel entsprechende insert und get&delete Anfrage ausgeschickt werden.

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. Es könnte mehrere Elemente für eine Position pos geben.
   $\rightarrow$ Behalte nur eines bei.
2. Es könnte Positionen pos$\in$[first,last] geben, für die ein Element fehlt.
   $\rightarrow$ Gib einfach ein leeres Element zurück.
3. Es könnte mehrere Knoten geben, die glauben, dass sie ein Anker sind. Welches [first,last]-Intervall wird dann übernommen?
   $\rightarrow$ Starte mit initialem Intervall und sende ein reset an alle Knoten, um den verfügbaren Elementen neue Positionen zuzuweisen.

Verteilte Queue

Selbststabilisierung:

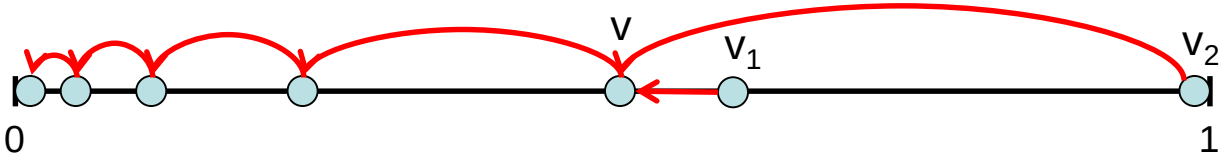
- Verteilte Hashtabelle: bereits vorher betrachtet
- Anker v_0: Knoten ist Anker, solange er keinen linken Vorgänger hat. Sonst gibt er die Ankerfunktion auf (und transferiert gegebenenfalls first und last).

Probleme:

4. Wie überprüfen wir, ob ein [first,last]-Intervall korrekt gesetzt ist?

Mögliche Lösung:

- Alle Knoten v mit Elementen halten senden periodisch [min(v),max(v)] in Richtung des Ankers, wobei min(v) die minimale Position eines Elementes in v und max(v) die maximale Position eines Elementes in v angibt.



- Treffen Intervalle $[min(v_1),max(v_1)],\ldots,[min(v_k),max(v_k)]$ bei v ein, gibt v das Intervall [min(v),max(v)] weiter in Richtung des Ankers, wobei $min(v)=min(min(v_1),\ldots,min(v_k))$ und $max(v)=max(max(v_1),\ldots,max(v_k))$ ist.
- Der Anker kennt dann irgendwann die Elemente mit kleinstem und größtem pos Wert.

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:

5. enqueue bzw. dequeue Operation wartet vergebens auf Rückantwort mit Position x.
6. 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

Problem: enqueue bzw. dequeue Operation wartet vergebens auf Rückantwort mit Position x.

Mögliche Lösung:

- Jede noch nicht bearbeitete enq(Q,i) und deq(Q,i) Anfrage hinterlässt eine „Spur“ in dem Sinne, dass sich jeder Knoten merkt, an welchen Knoten er diese weitergeleitet hat.
- Periodisch überprüft jeder Knoten in timeout durch check Anfragen, ob seine Spuren korrekt sind, indem er die Knoten kontaktiert, an die er die Anfragen weitergereicht haben will.
- Falls ein Knoten, der eine check Anfrage erhält, keine Aufzeichnung der entsprechenden enq(Q,i) oder deq(Q,i) Anfrage hat, schickt dieser eine NACK-Antwort und sonst eine ACK-Antwort zurück.
- Falls eine NACK-Antwort zu einer enq(Q,i) oder deq(Q,i) Anfrage empfangen wird, wird diese im Speicher des Knotens gelöscht.
- Falls der Initiator einer enq(Q,1) oder deq(Q,1) Anfrage ein NACK empfängt, sendet er diese nochmal aus.

Bemerkung: wir brauchen natürlich eindeutige Ids für die Anfragen, damit diese Überprüfungen möglich sind.

Verteilte Queue

Problem: dequeue Operation wird Position zugewiesen, bei der sie kein Element findet.

Einfachste Lösung (Knoten arbeiten halbwegs synchron):

- Eine get(pos) Anfrage wartet maximal O(log n) Runden bei dem zugewiesenen Knoten. Ist bis dahin kein Element eingetroffen, wird pos in diesem Knoten durch einen Marker als gelöscht markiert.
- Trifft im Rahmen der Selbststabilisierung irgendwann das durch put(pos,x) eingefügte Element x auf einen Löschmarker von pos, wird x zusammen mit dem Marker gelöscht, um die Einträge zu bereinigen.

Viele weitere Details müssen zur vollständigen Selbststabilisierung der verteilten Queue beachtet werden, auf die wir im Rahmen der Vorlesung nicht näher eingehen werden. Hier bieten sich interessante Softwareprojekte an, die zumindest einen Teil dieser Probleme lösen.

Verteilte Queue

Monotone Korrektheit: wir müssen erfüllen:

- Ist einmal eine Anfrage von v zu v_0 gelangt, ist das auch in Zukunft so
- Der Rückweg ist garantiert, da die Knoten sich die Referenzen für die Rücksprünge merken.

Problem: was ist mit der Bearbeitung von insert und get&delete Anfragen in der verteilten Hashtabelle???

Hier scheint es schwer zu sein, monotone Korrektheit sicherzustellen in dem Sinne, dass wenn erstmal eine enqueue oder dequeue Anfrage korrekt bearbeitet wurde, jede nachfolgende Anfrage auch korrekt bearbeitet wird.

Zu klären wird erstmal sein, was das genau heißt!

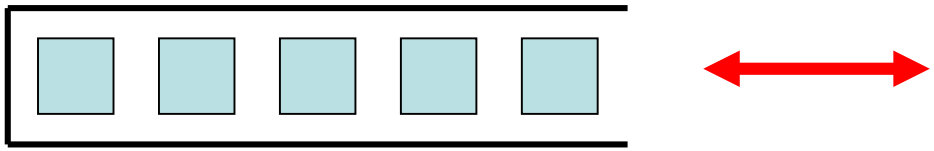
Übersicht

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

Konventioneller Stack

Ein Stack S unterstützt folgende Operationen:

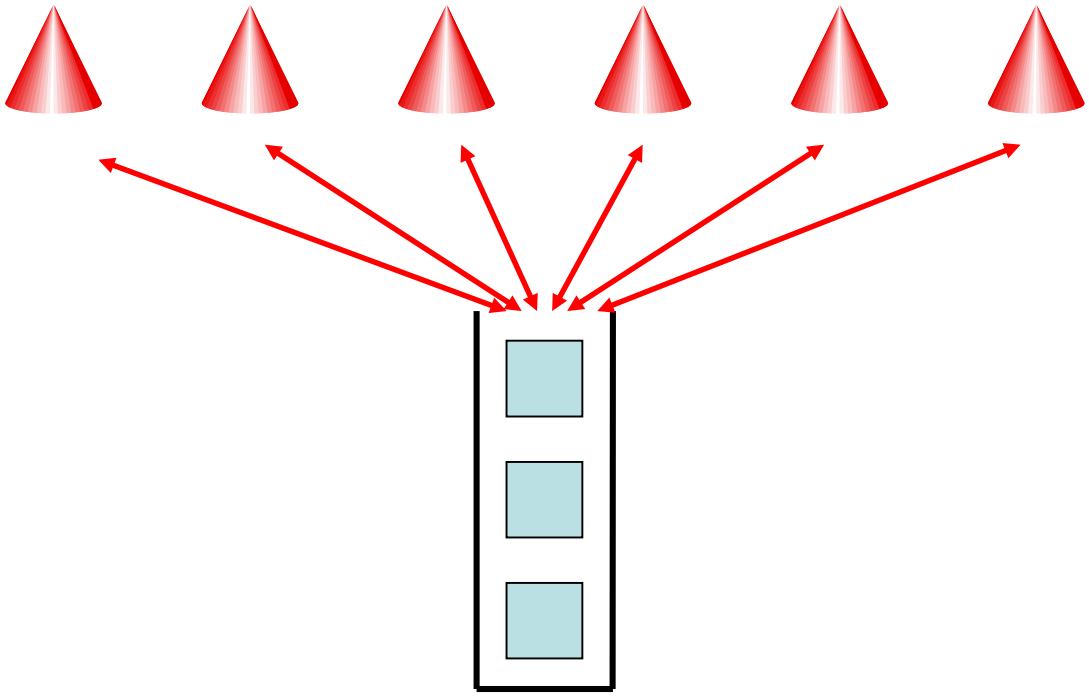
- push(S,x): legt Element x oben auf dem Stack ab.
- pop(S): holt das oberste Element aus dem Stack S heraus und gibt es zurück



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

Verteilter Stack

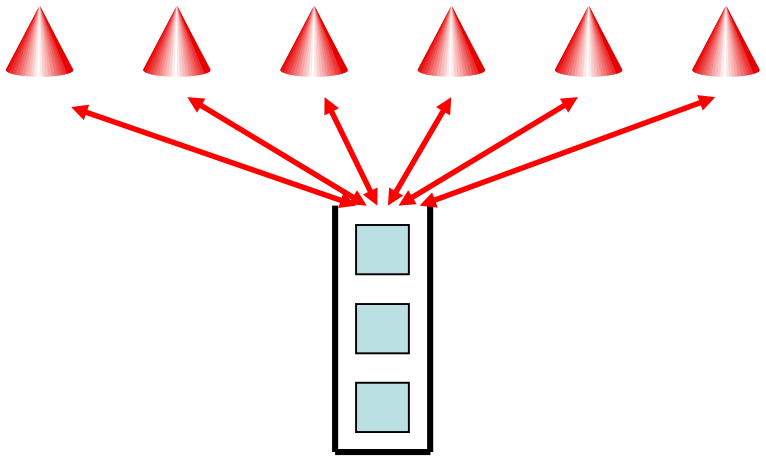
Viele Prozesse agieren auf Stack:



Verteilter Stack

Probleme:

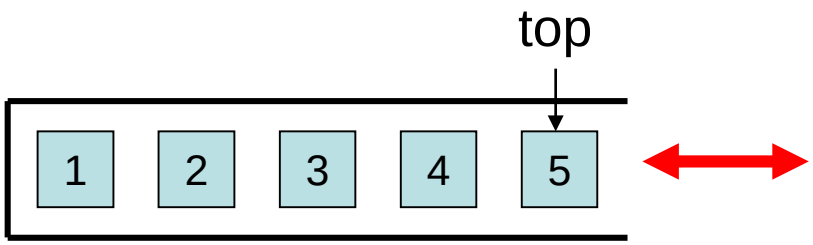
- Speicherung des Stacks
- Realisierung von push und pop



Verteilter Stack

Speicherung des Stacks:

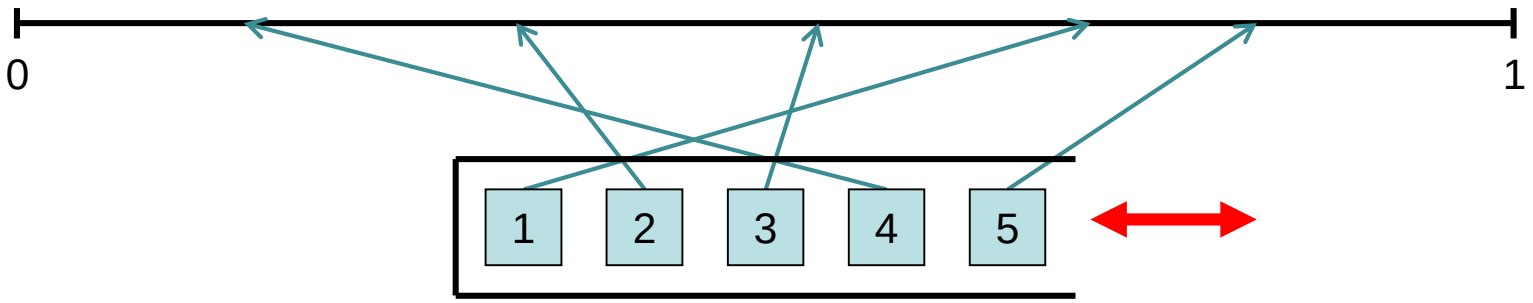
- Jedes Element x besitzt eindeutige Position $pos(x) \geq 1$ im Stack (das oberste hat die größte Position).



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

Realisierung von pop(S):

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

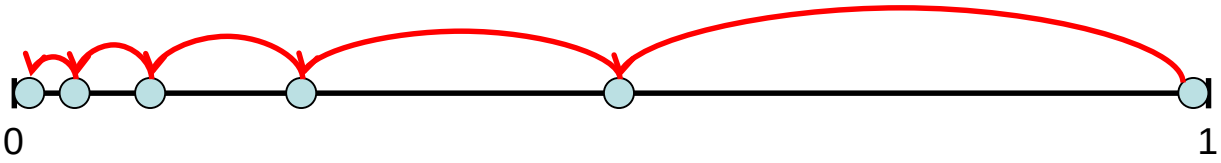
Noch zu klären: Punkt 1.

Hier können wir ähnlich wie für enq und deq vorgehen.

Verteilter Stack

Realisierung von pushall(S,1):

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



- Der am nächsten an 0 liegende Knoten v_0 (Anker) merkt sich einen Zähler top. top speichert die Position des obersten Elements in S.
- Jeder Knoten v merkt sich zunächst jede erhaltene pushall(S,i) Anfrage samt Knoten w, der diese an ihn geschickt hat.
- Angenommen, v habe bei Ausführung von timeout die Anfragen pushall(S,i_1), pushall(S,i_2),…,pushall(S,i_k) angesammelt. Dann sendet v eine pushall(S,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 pushall(S,i) Anfrage v_0, dann schickt v_0 das Intervall [top+1,top+i] an die Rückadresse und setzt top:=top+i.
- Erhält ein Knoten v ein Intervall [pos,pos+i-1], das zu einer push(S,i) Anfrage gehört, die aus den Anfragen pushall(S,i_1), pushall(S,i_2),…,pushall(S,i_k) kombiniert wurde, so schickt v das Intervall [pos, pos+i_1-1] an die Rückadresse von pushall(S,i_1), [pos+i_1, pos+i_1+i_2-1] an die Rückadresse von pushall(S,i_2), usw.
- Am Ende erhält jede pushall(S,1) Anfrage eine eindeutige Position im Stack.

Verteilter Stack

Realisierung von push(S,x):

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

Realisierung von pop(S):

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

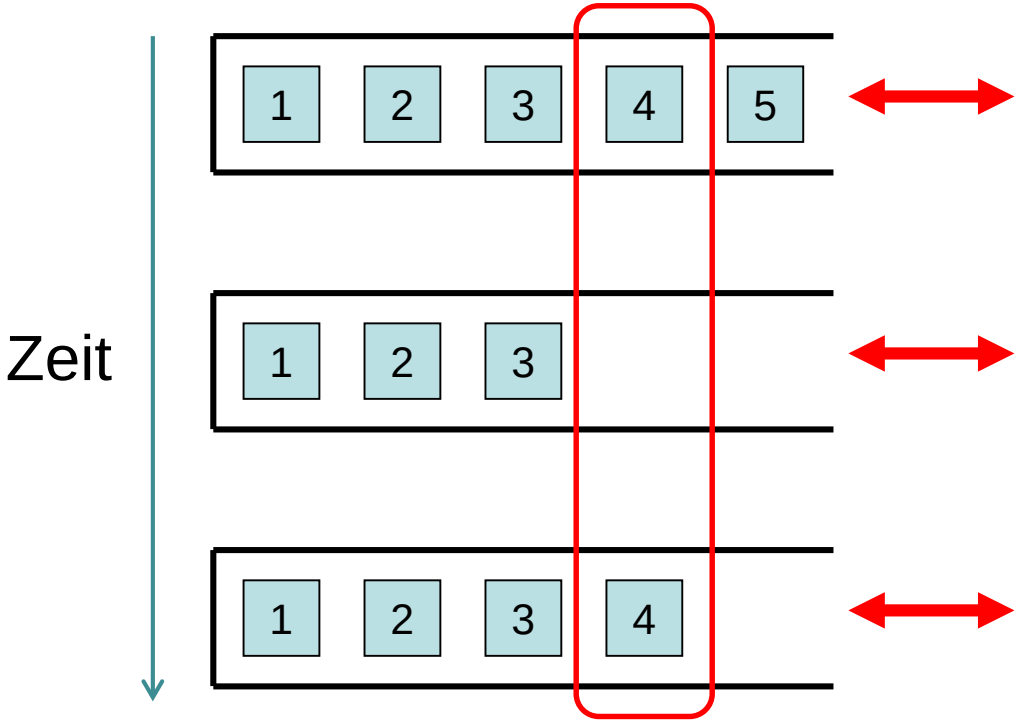
Noch zu klären: Punkt 1.

Hier können wir ähnlich wie für enq und deq vorgehen.

Neues Problem: push Anfragen könnten dieselbe Position zugewiesen bekommen.

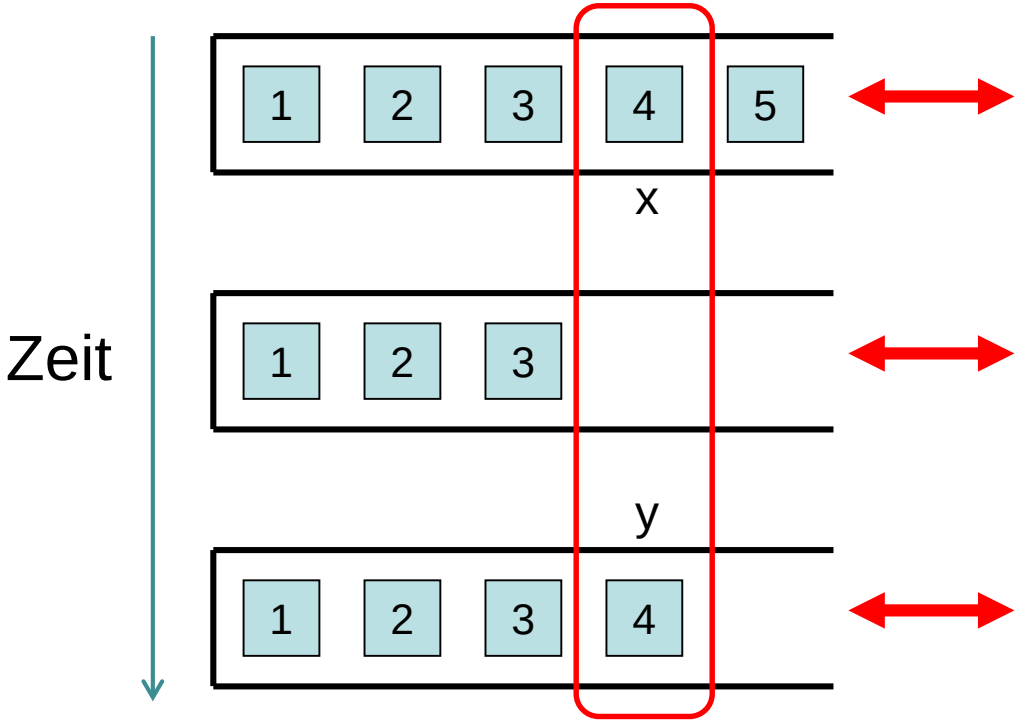
Verteilter Stack

Neues Problem: push Anfragen könnten dieselbe Position zugewiesen bekommen.



Verteilter Stack

Es kann passieren, dass y vor x im Speicher von 4 eintrifft, so dass x statt y gepullt wird, was die Stackregel verletzt.



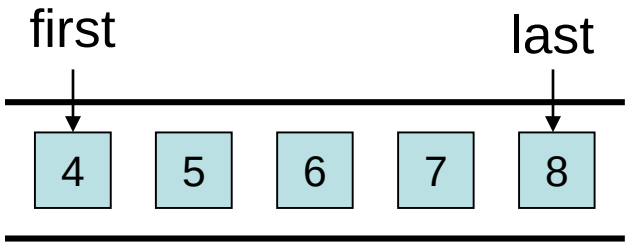
Verteilter Stack

Neues Problem: push Anfragen könnten dieselbe Position zugewiesen bekommen.

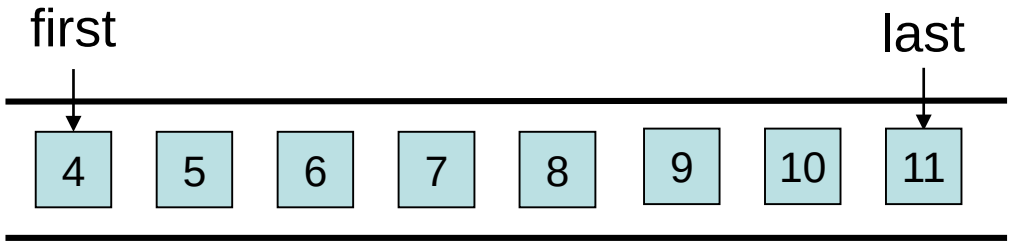
Mögliche Lösung (einfacher Fall):

- Implementiere Stack ähnlich zur Queue mit first und last Wert. Allerdings benötigen wir mehrere [first,last]-Intervalle, wie wir in einem Beispiel veranschaulichen werden.

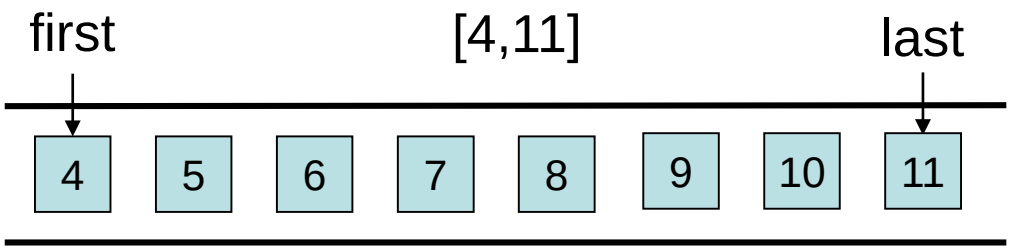
Verteilter Stack



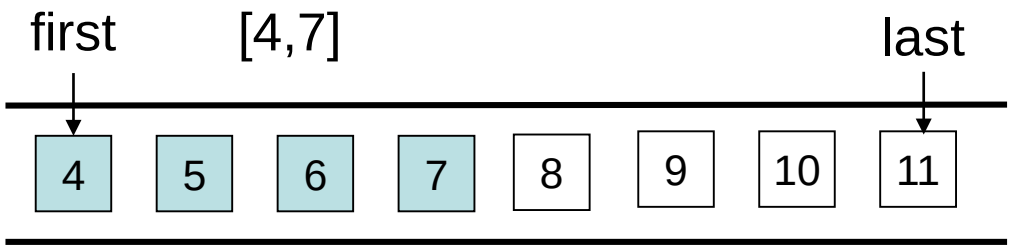
pushall(S,3):



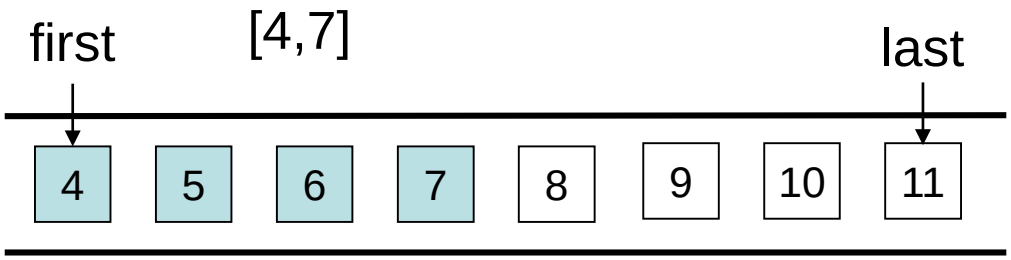
Verteilter Stack



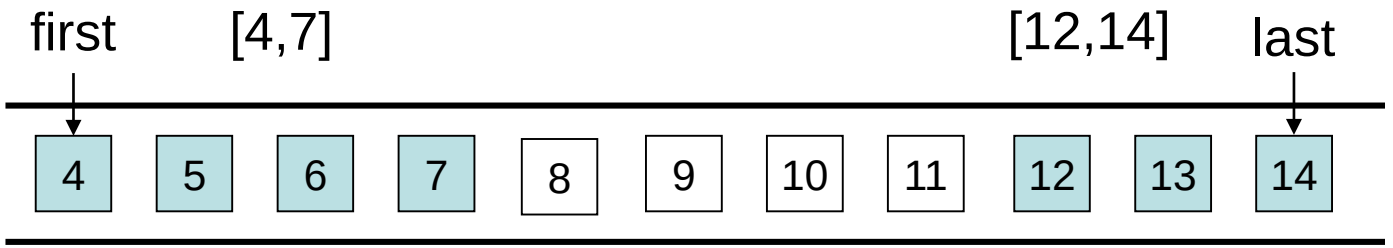
popall(S,4):



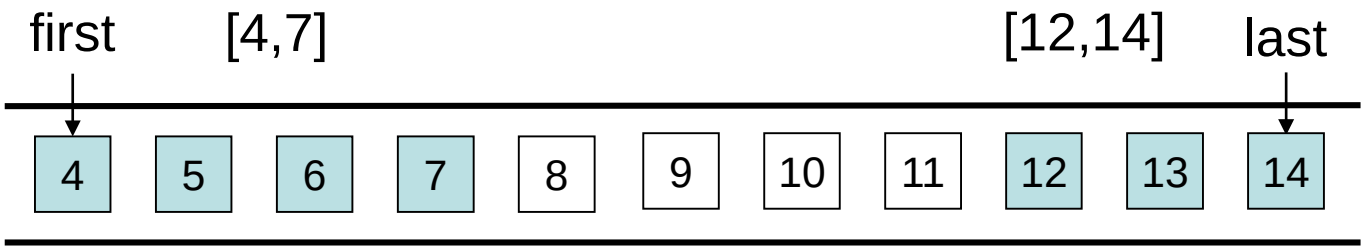
Verteilter Stack



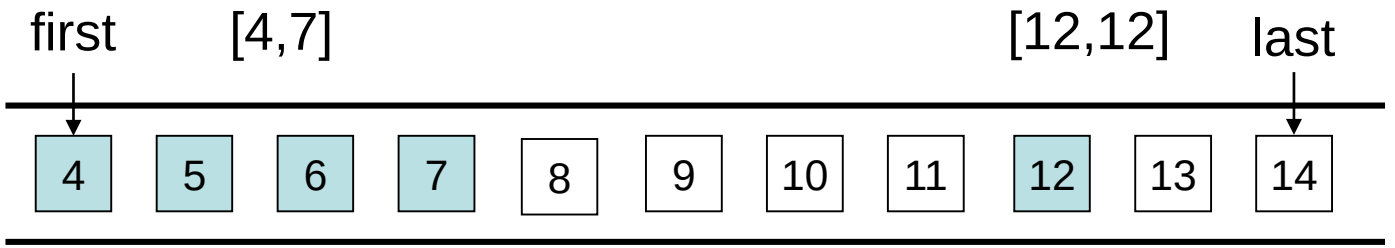
pushall(S,3):



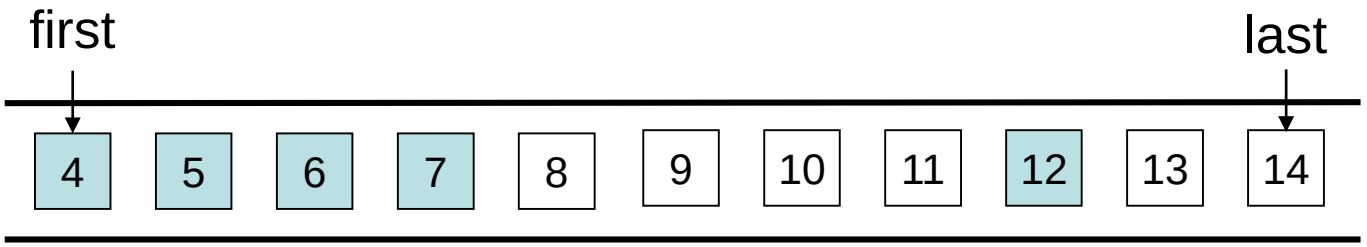
Verteilter Stack



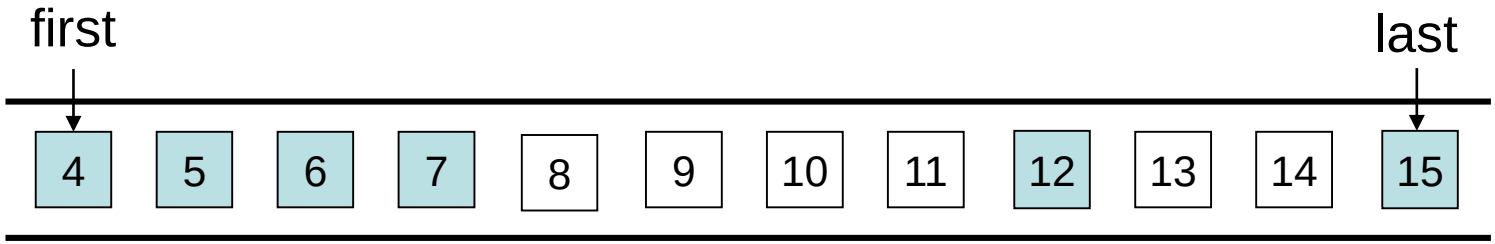
popall(S,2):



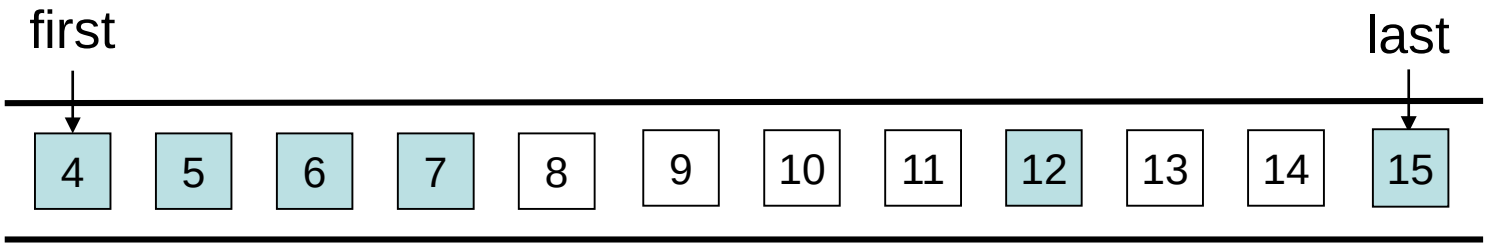
Verteilter Stack



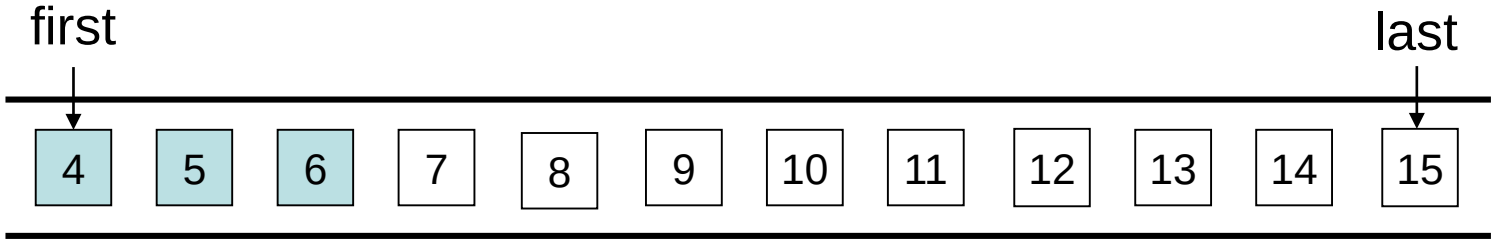
pushall(S,1):



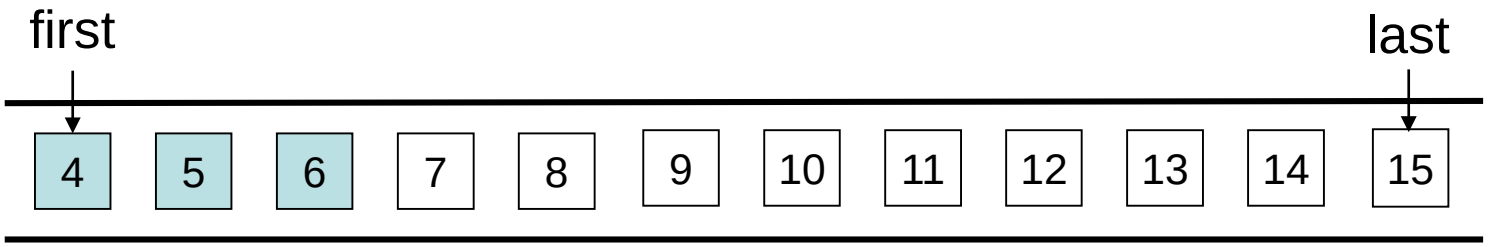
Verteilter Stack



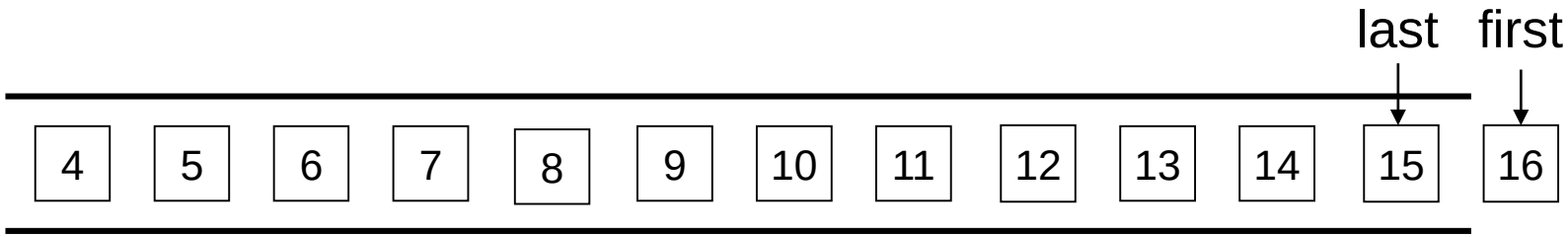
popall(S,3):



Verteilter Stack



popall(S,3):



Verteilter Stack

Neues Problem: push Anfragen könnten dieselbe Position zugewiesen bekommen.

Mögliche Lösung:

- Implementiere Stack ähnlich zur Queue. Allerdings müssen beim Stack eventuell mehrere [l,r]-Intervalle im Anker gespeichert werden, um sich zusammenhängende Element-folgen zu merken.
- Für jedes pushall(S,i) wird [last+1,last+i] zurückgegeben, das [last+1,last+i] –Intervall dann entweder mit einem [l,r]-Intervall mit r=last verschmolzen oder ein neues [last+1,last+i] –Intervall angelegt und danach last:=last+i gesetzt.
- Für jedes popall(S,i) werden die [l,r]-Intervalle mit den höchsten i Positionen zurückgegeben.

Verteilter Stack

Warum müssen wir für die Eindeutigkeit der Stackpositionen eventuell mehrere Intervalle verwenden und damit mehr speichern als für die verteilte Queue?

Indikator dafür, dass der Stack mehr Speicher für den Zustand benötigt als die Queue:

- Im Gegensatz zur Queue ist die Menge der Elemente im Stack abhängig von der Verzahnung der push und pop Operationen und daher nicht nur von der Anzahl der push und pop Operationen, selbst wenn der Stack nie leer ist.
- In der Queue hingegen ist die Menge der Elemente nur abhängig von der Anzahl der enqueue und dequeue Operationen und nicht deren Verzahnung, falls die Queue niemals leer wird.

Beispiel: Tafel

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 Arbeit O(log n)
- pop(S): erwartete Arbeit O(log n)

Verwaltung mehrerer Stacks in derselben Hashtabelle:
weise jedem Stack statt Punkt 0 einen (pseudo-) zufälligen Punkt in [0,1) zu, verwende dann Skip+ Graph

Verteilter Stack

Optimierungsmöglichkeit:

- Aufeinandertreffende pushall(S,i) und popall(S,i) Anfragen können gematcht werden (d.h. den i popall Anfragen werden die Elemente der i pushall Anfragen zugeordnet), so dass nur eine Folge von pushall oder eine Folge von popall Anfragen gleichzeitig beim Anker bearbeitet werden muss.

Warum ist das in Ordnung?

Lokale Konsistenz:

- Bearbeite Folgen von push und pop Operationen (mittels serve Anfragen wie in der Queue) anstatt nur einer.
- Falls die Optimierungsmöglichkeit oben genutzt wird, können push und pop Operationen lokal soweit möglich gematcht werden, so dass nur noch eine pushall(S,i) oder popall(S,i) oder pop&pushall(S,i,j) Anfrage zum Anker geschickt werden muss, wobei pop&pushall(S,i,j) i pop Anfragen gefolgt von j push Anfragen repräsentiert.

Verteilter Stack

Selbststabilisierung: ähnliche Probleme wie bei der Queue, nur aufgrund mehrerer Intervalle noch komplexer…

Monotone Korrektheit: auch noch ungeklärt.

Auch hier bieten sich interessante Softwareprojekte an, um einige Aspekte zu beleuchten.

Übersicht

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

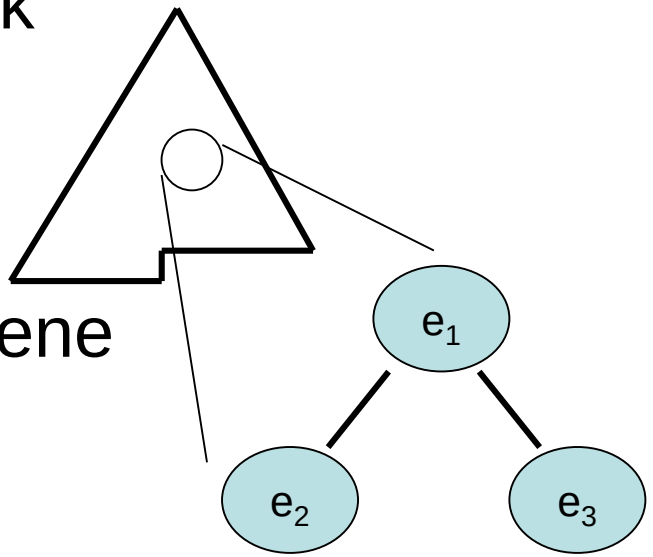
Konventioneller Heap

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

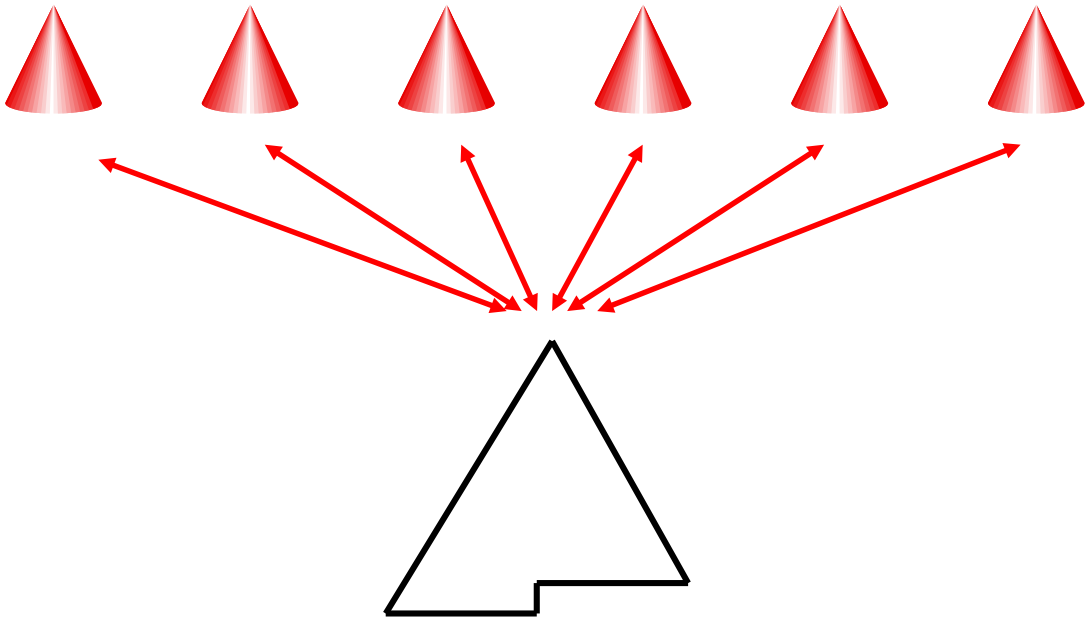
Zwei Invarianten:

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



Verteilter Heap

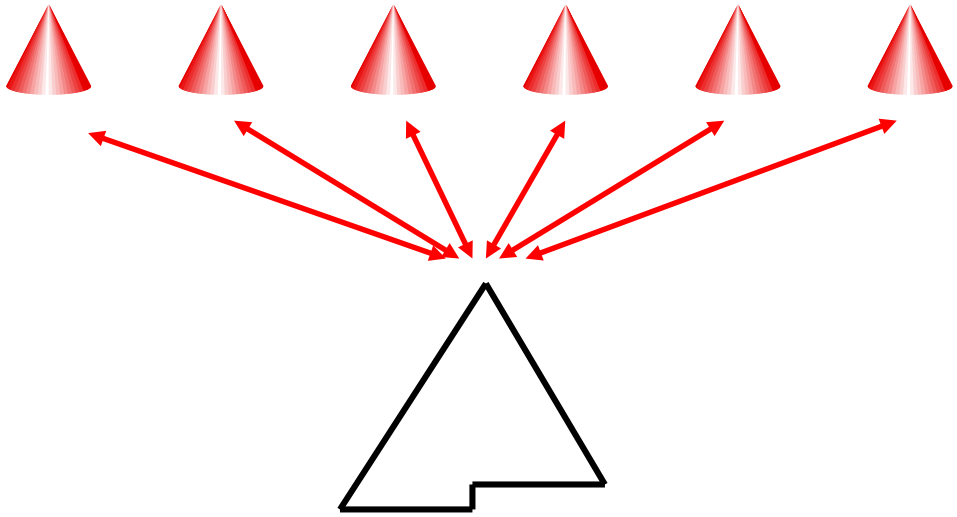
Viele Subjekte agieren auf Heap:



Verteilter Heap

Probleme:

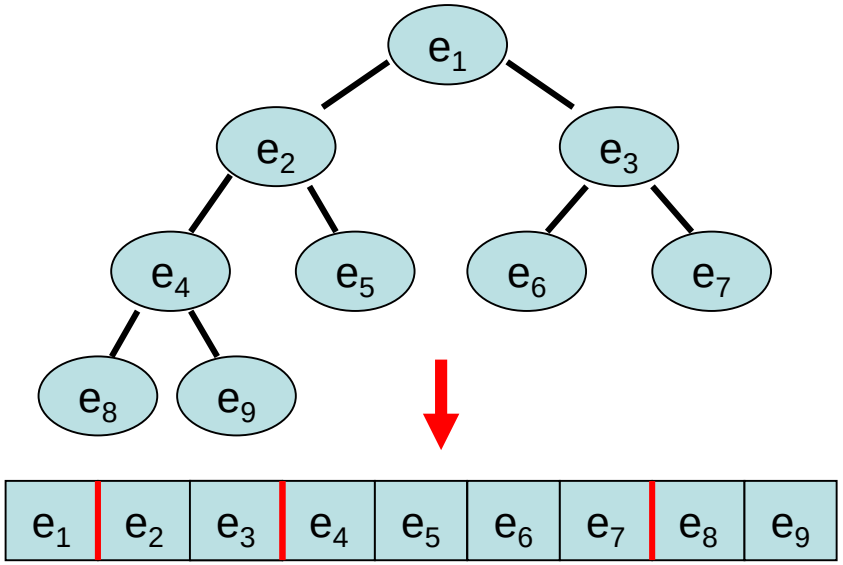
- Speicherung des Heaps
- Realisierung von insert und deleteMin



Verteilter Heap

Speicherung des Heaps:

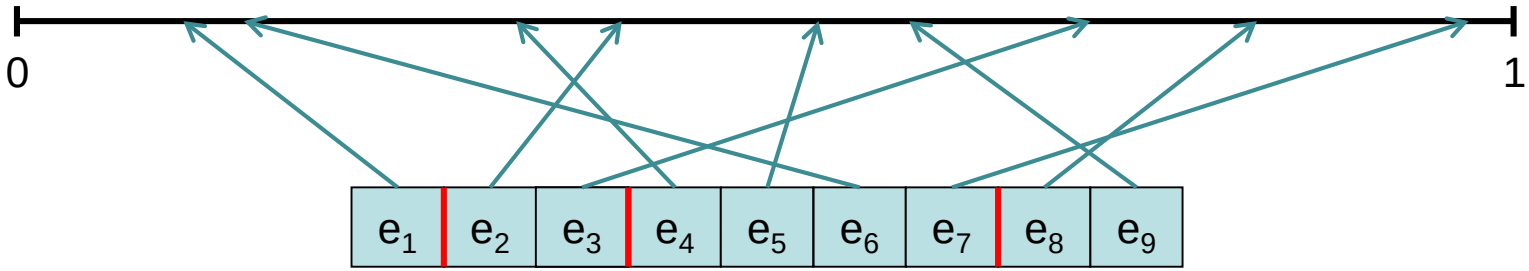
- Jedes Element x besitzt eine eindeutige Position pos(x)≥1 im Heap wie bei der Feldrealisierung des konventionellen Heaps



Verteilter Heap

Speicherung des Heaps:

- Jedes Element x besitzt eine eindeutige Position $pos(x) \geq 1$ 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 ins(H,1)-Anfrage, um eine Nummer pos zu erhalten.
2. Führe put(pos,x) auf der verteilten Hashtabelle aus, um x unter pos zu speichern.

Realisierung von deleteMin(H):

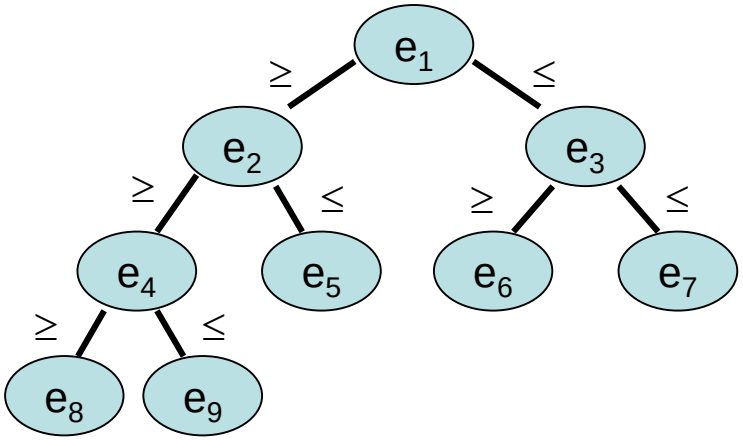
1. Stelle delmin(H,1)-Anfrage, um eine Nummer pos zu erhalten.
2. Führe get(pos) auf der verteilten Hashtabelle aus, um das unter pos gespeicherte Element x zu löschen und zu erhalten.

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.



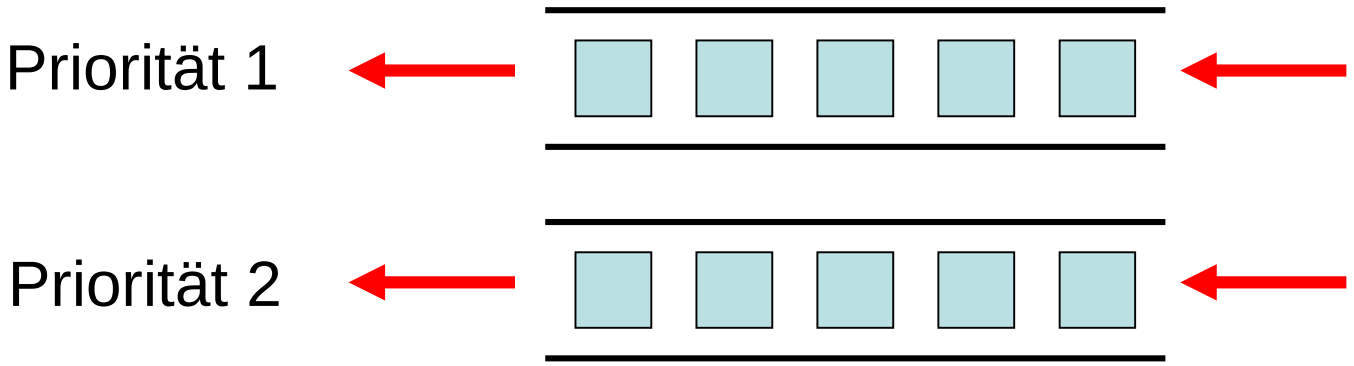
Verteilter Heap

Einfacher Fall: nur k verschiedene Prioritäten

→ Anker verwaltet k Queues, eine für jede Priorität

→ insert: wie enqueues in separate Queues

→ deleteMin: dequeue auf Queues kleinster Prio.



Verteilter Heap

Allgemeiner Fall: Prioritäten $\in \{0,1\}^k$ für eine vorgegebene Konstante k

$\rightarrow$ verwalte Prioritäten in einem Trie

Trie

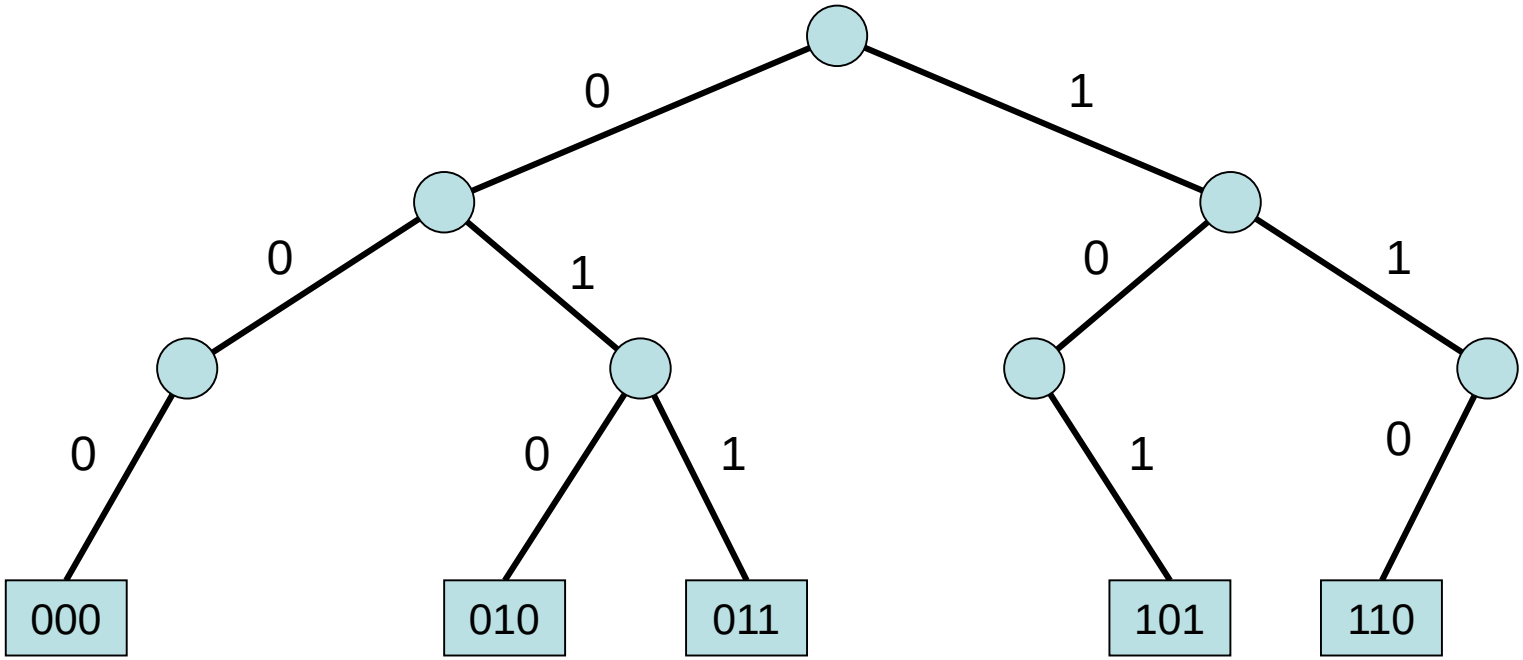
Ein Trie ist ein Suchbaum über einem Alphabet Σ, der die folgenden Eigenschaften erfüllt:

- 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: $\Sigma=\{0,1\}$, Schlüssel repräsentieren Prioritäten

Trie

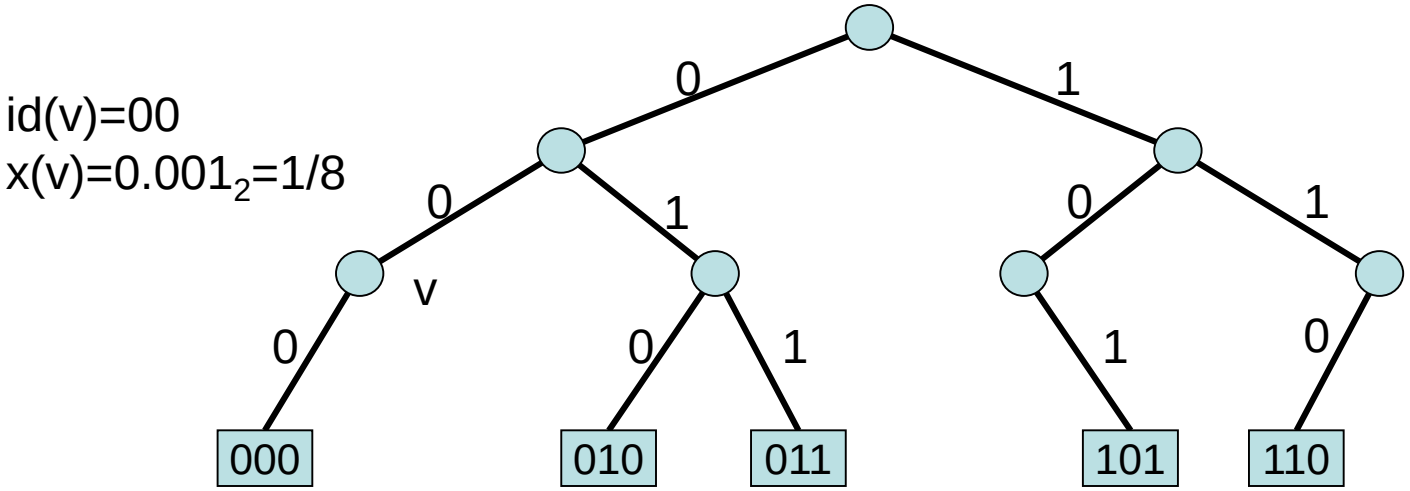
Trie aus Beispielprioritäten:



Trie

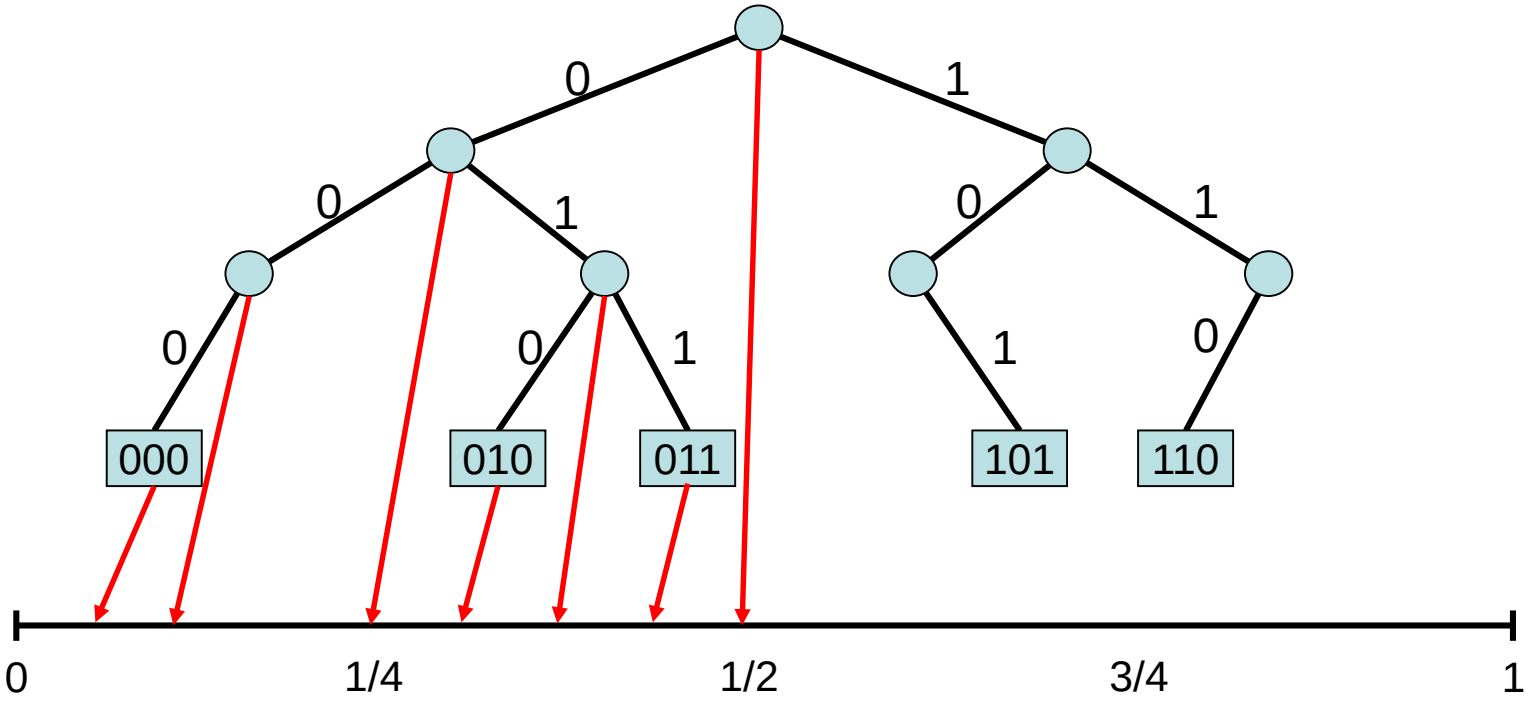
Einbettung des Tries in [0,1]:

- Knotenlabel id(v): Bitfolge zum Knoten v
- Position von v: $x \in [0,1]$ mit $x=(0.\mathrm{id}(v)\circ 1)_2$



Trie

Einbettung des Tries in [0,1]:



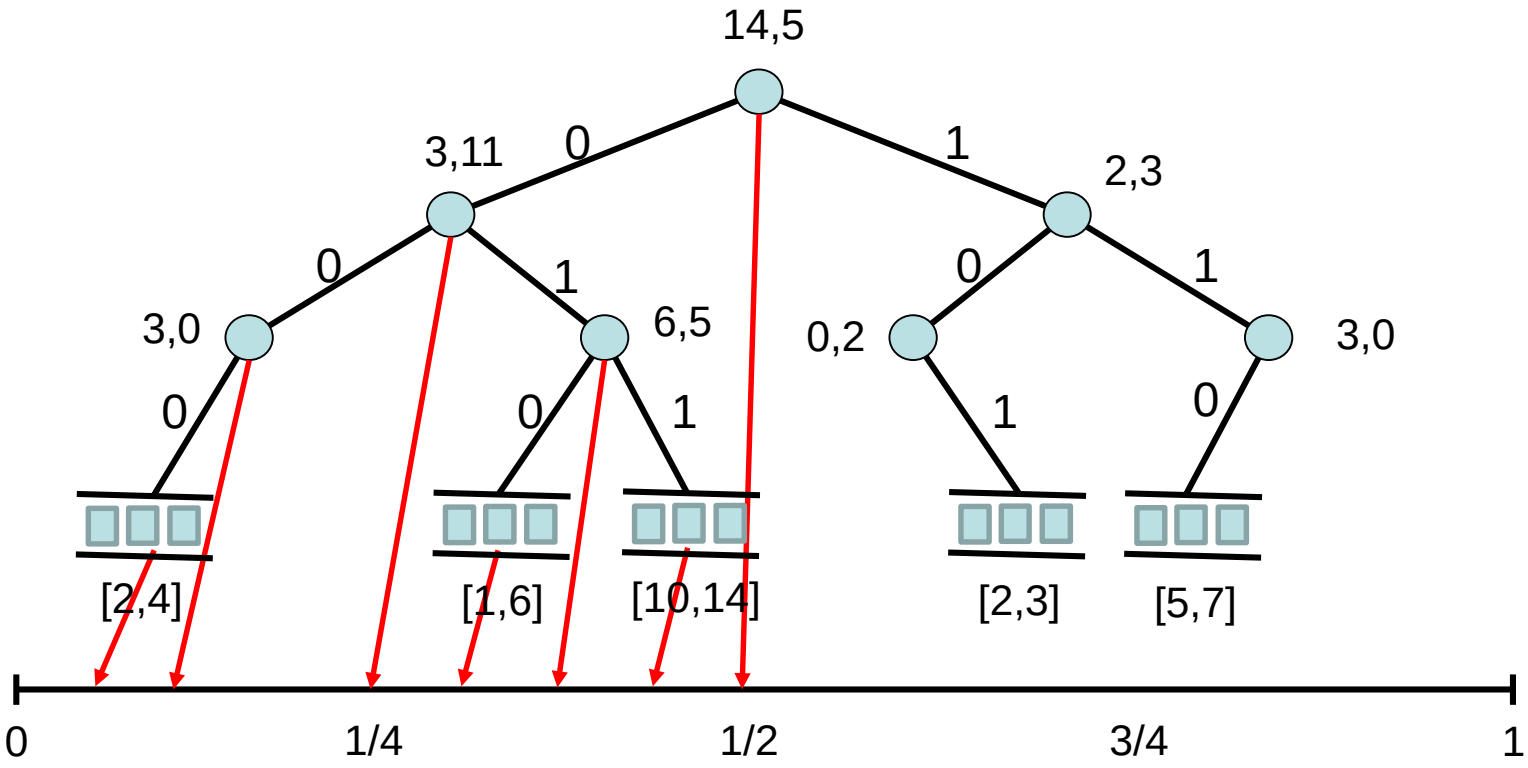
Verteilter Heap

Speicherung des Tries:

- Weise die Knoten des Tries den Prozessen gemäß des konsistenten Hashings zu (d.h. der Prozess v mit $1/2 \in I(v)$ verwaltet die Wurzel des Tries)
- Jeder innere Knoten des Tries merkt sich die Anzahl der gespeicherten Elemente in den Teilbäumen seiner Kinder v und w in den Variablen $m(v)$ und $m(w)$.
- Jedes Blatt des Tries verwaltet eine Queue für die entsprechende Priorität, indem es sich ein $[first,last]$-Intervall merkt.

Trie

Beispiel:



Heap-Operationen

Realisierung von insert(H,x):

1. Stelle ins(H,prio(x),1)-Anfrage, um eine Nummer pos in der Queue von prio(x) zu erhalten, wobei prio(x) die Priorität von x angibt.
2. Führe put((prio(x),pos),x) auf der verteilten Hashtabelle aus, um x unter (prio(x),pos) zu speichern.

Realisierung von deleteMin(H):

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

Noch zu klären: Punkt 1 in insert und deleteMin.

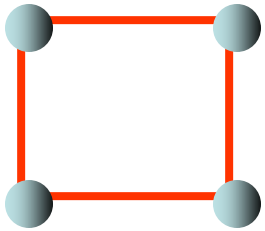
Hier benötigen wir eine hypercubische Topologie.

Hypercube

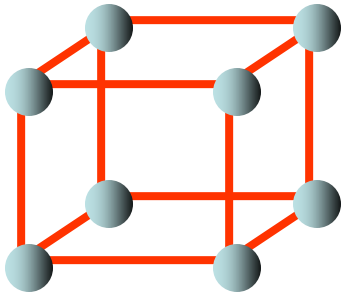
Klassischer Hypercube:

- Knoten: $(x_1,\ldots,x_d) \in \{0,1\}^d$
- Kanten: $\forall i: (x_1,\ldots,x_d) \rightarrow (x_1,..,1\text{-}x_i,..,x_d)$

d=1



d=2



d=3

Hypercube

Klassischer Hypercube:

- V: Knoten mit Labeln $(x_1,\ldots,x_d)\in\{0,1\}^d$
- E: $\forall i$: $(x_1,\ldots,x_d)\rightarrow(x_1,..,1\text{-}x_i,..,x_d)$

Kontinuierliche Version des Hypercube:

- Interpretiere $(x_1,\ldots,x_d)$ als $z=\sum_i x_i/2^i$

$d\rightarrow\infty$:

- $V=[0,1)$
- $E=\{ \{x,y\} \mid y=x+1/2^i \pmod 1$ oder $y= x-1/2^i \pmod 1$ für ein $i>0 \}$
- E deckt Kanten in klassischer Kantenmenge E ab

Hypercube

Kontinuierlicher Hypercube:

- $V=[0,1)$
- $E=\{ \{x,y\} \mid y=x+1/2^i \pmod 1$ oder $y= x-1/2^i \pmod 1$ für ein $i>0 \}$

Routingstrategie für zwei Punkte $x,y \in [0,1)$ interpretiert als Bitfolgen:

$(x_1,x_2,x_3,\ldots) \rightarrow (y_1,x_2,x_3,\ldots) \rightarrow (y_1,y_2,x_3,\ldots)$
$\rightarrow (y_1,y_2,y_3,x_4,\ldots) \rightarrow \ldots$

Dynamischer Hypercube

Kontinuierlicher Hypercube:

- $V=[0,1)$
- $E=\{ \{x,y\} \mid y=x+1/2^i \pmod 1$ oder $y= x-1/2^i \pmod 1$ für ein $i>0 \}$

Kontinuierlich-diskreter Ansatz:

- Betrachte eine beliebige Prozessmenge V.
- Wir weisen jedem Prozess $v \in V$ ein Intervall $I(v) \subseteq [0,1)$ gemäß des konsistenten Hashings zu.
- Jeder Prozess v ist mit allen Prozessen w verbunden, für die es ein $i \in \mathbb{N}$ gibt mit

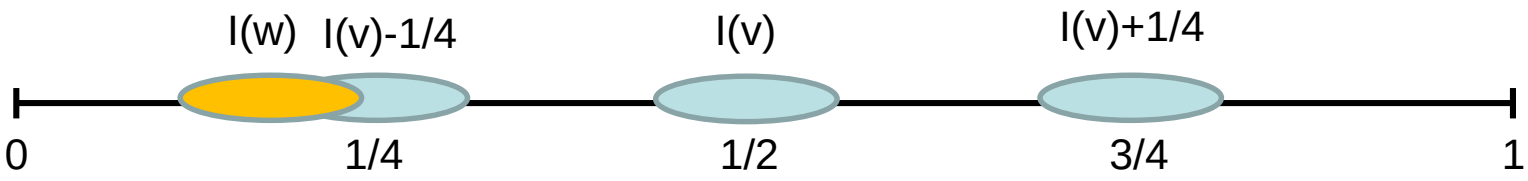
  $(I(v)+1/2^i) \cap I(w) \neq \emptyset$ oder $(I(v)-1/2^i) \cap I(w) \neq \emptyset$

Dynamischer Hypercube

- Jeder Prozess v ist mit allen Prozessen w verbunden, für die es ein $i \in \mathbb{N}$ gibt mit

  $(I(v)+1/2^i) \cap I(w) \neq \emptyset$ oder $(I(v)-1/2^i) \cap I(w) \neq \emptyset$

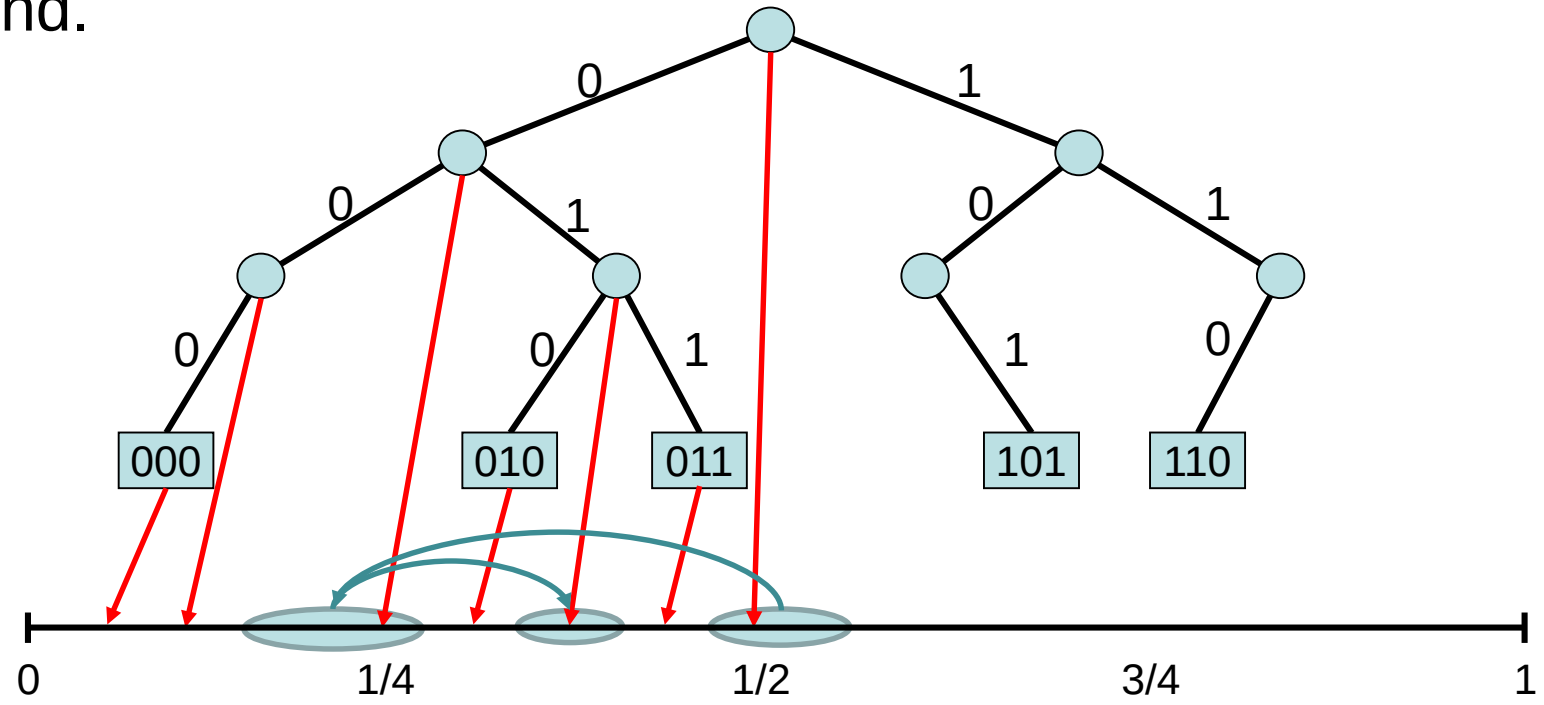
Anschaulich:



Prozess v hat eine Kante zu Prozess w.

Trie

Mithilfe des Hypercubes kann Trie effizient durchlaufen werden, da Nachbarn des Tries mit HC-Kanten verbunden sind.



Heap Operationen

Realisierung von ins(H,p,1):

- Anker ist jetzt der Prozess v mit $1/2 \in I(v)$.
- Die ins(H,p,i) Anfragen werden für jedes p wie vorher zum Anker hin kombiniert.
- Die kombinierten Anfragen werden ab der Wurzel in FIFO-Ordnung entlang der Kanten des Tries weitergeleitet (muss im Protokoll erzwungen werden, da die zugrundeliegende Kommunikationsschicht die Anfragen eventuell nicht in FIFO-Ordnung weiterleitet).
- Erhält ein innerer Knoten eine ins(H,p,i) Anfrage und gehört p zum Teilbaum seines Kindes v, dann erhöht er m(v) um i und leitet ins(H,p,i) an v weiter.
- Erhält ein Blatt eine ins(H,p,i) Anfrage, so bearbeitet es diese wie der Anker eine enq(Q,i) Anfrage in der verteilten Queue.

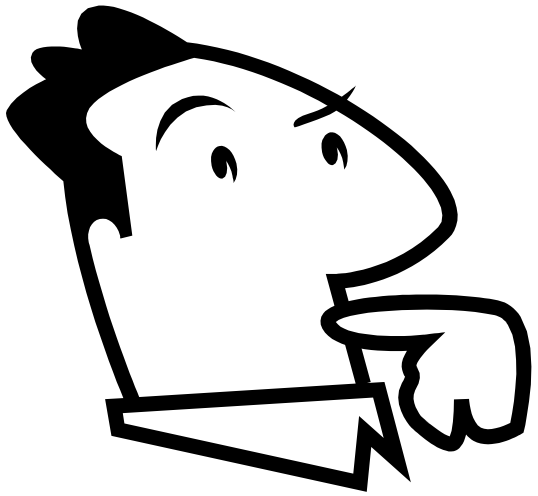
Heap Operationen

Realisierung von delmin(H,1):

- Die delmin(H,i) Anfragen werden für jedes p wie vorher zum Anker hin kombiniert.
- Die kombinierten Anfragen werden ab der Wurzel in FIFO-Ordnung entlang der Kanten des Tries weitergeleitet (muss im Protokoll erzwungen werden, da die zugrundeliegende Kommunikationsschicht die Anfragen eventuell nicht in FIFO-Ordnung weiterleitet).
- Erhält ein innerer Knoten linkem Kind v und rechtem Kind w eine delmin(H,i) Anfrage, und ist $m(v) \leq i \leq m(v)+m(w)$, dann leitet dieser die Anfrage delmin(H,m(v)) an v und delmin(H,m(w)-(i-m(v))) an w weiter und setzt $m(v)=0$ und $m(w)=m(w)-(i-m(v))$., d.h. v wird gemäß der Definition von deleteMin wegen der geringeren Prioritäten unter ihm bevorzugt. Die anderen Fälle für i sind ähnlich.
- Erhält ein Blatt die Anfrage delmin(H,i), behandelt es diese wie der Anker eine deq(Q,i) Anfrage für eine Queue.

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?