Verteilte Algorithmen und Datenstrukturen

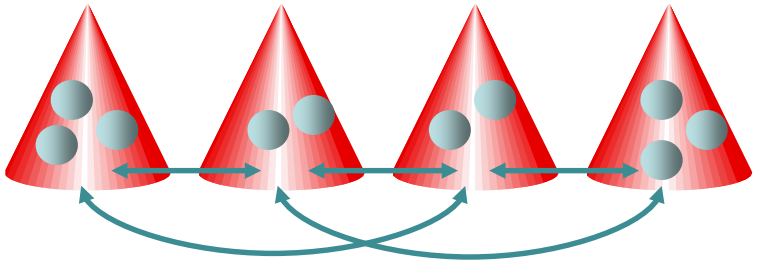
Kapitel 6: Informationsorientierte Datenstrukturen

Prof. Dr. Christian Scheideler

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

Wichtige Ziele:

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

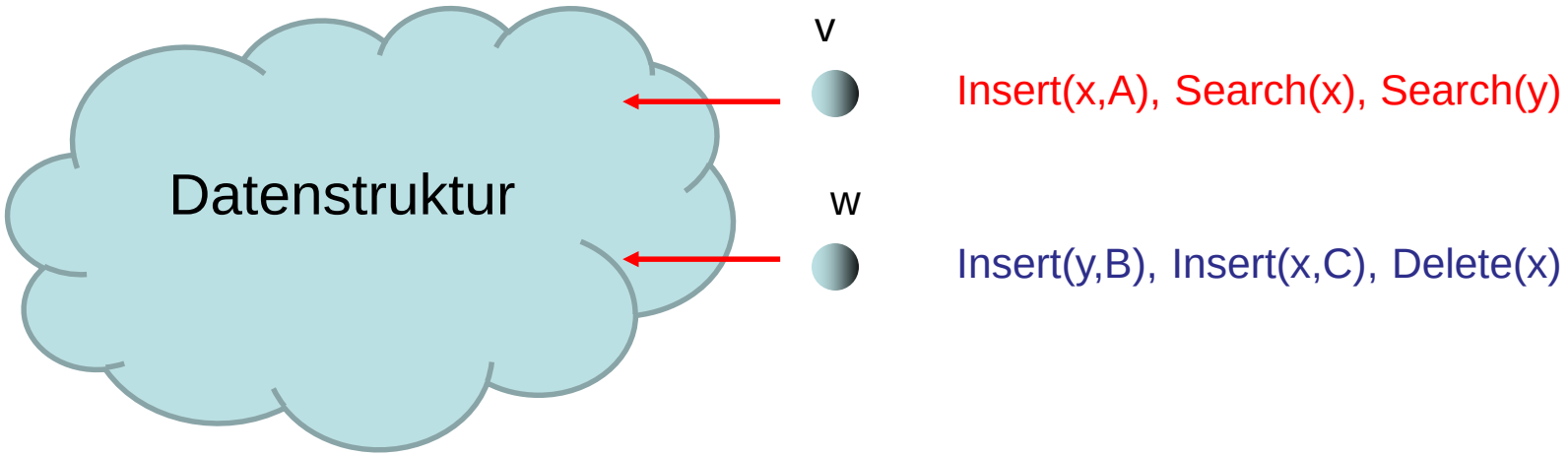
Zur Erinnerung:

Definition 3.10: Eine Linearisierung L(S) einer Rechnung S ist sequentiell 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 sequentiell konsistent ausgeführt wird. Das erlaubt es aber noch, dass die Operationsfolgen der Prozesse beliebig ineinander verzahnt werden dürfen.

Informationsorientierte Datenstrukturen

Beispiel:



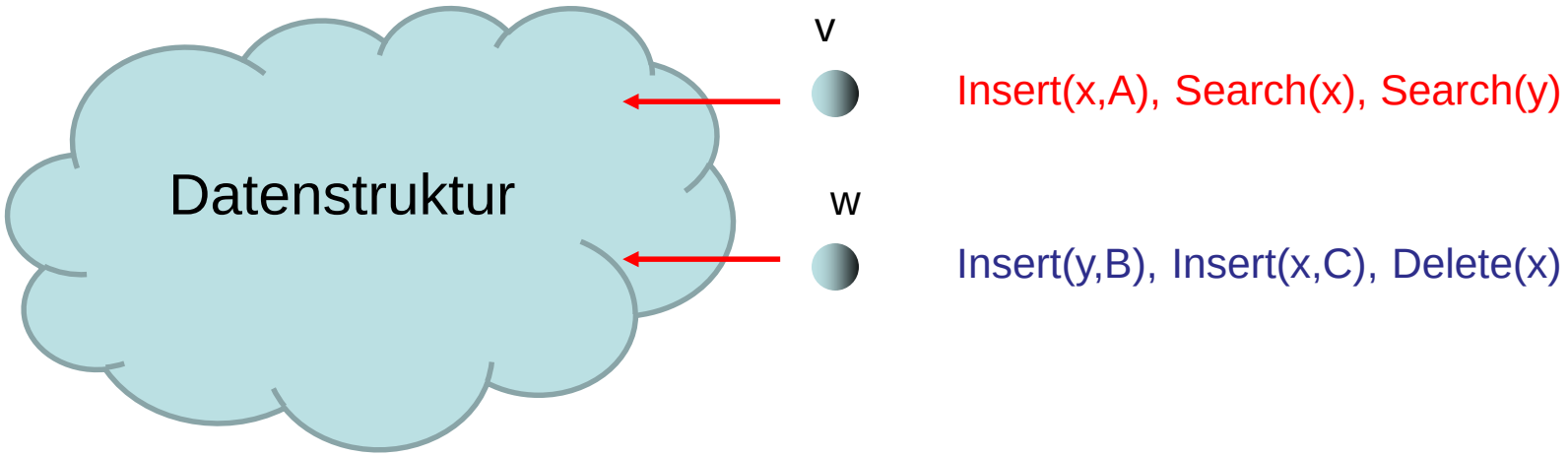
Eine sequ. konsistente Ausgabe für v wäre C, B, da diese durch

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

zustande käme.

Informationsorientierte Datenstrukturen

Beispiel:



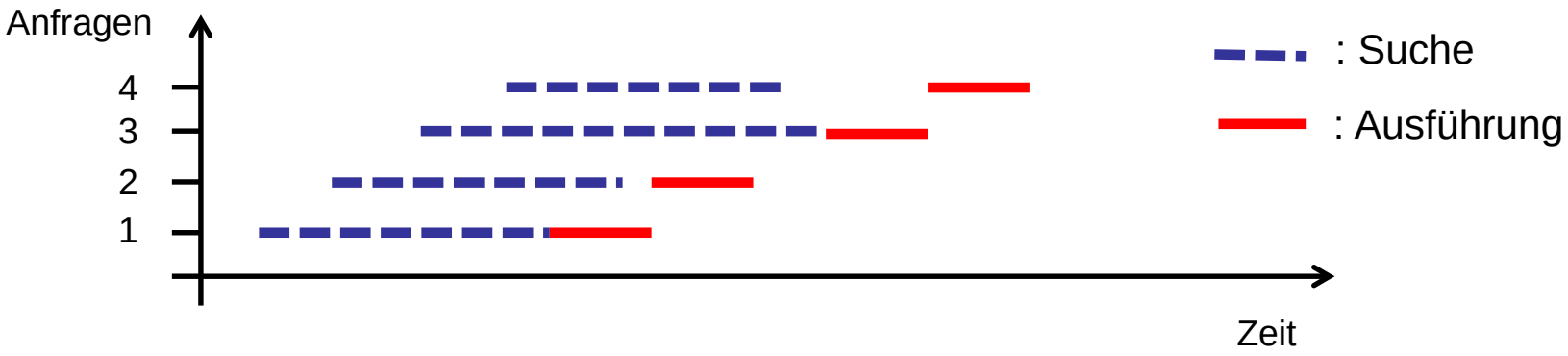
Einfachste Strategie für sequentielle Konsistenz:

- 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.



Informationsorientierte Datenstrukturen

Lokal sequentielle Ausführung:

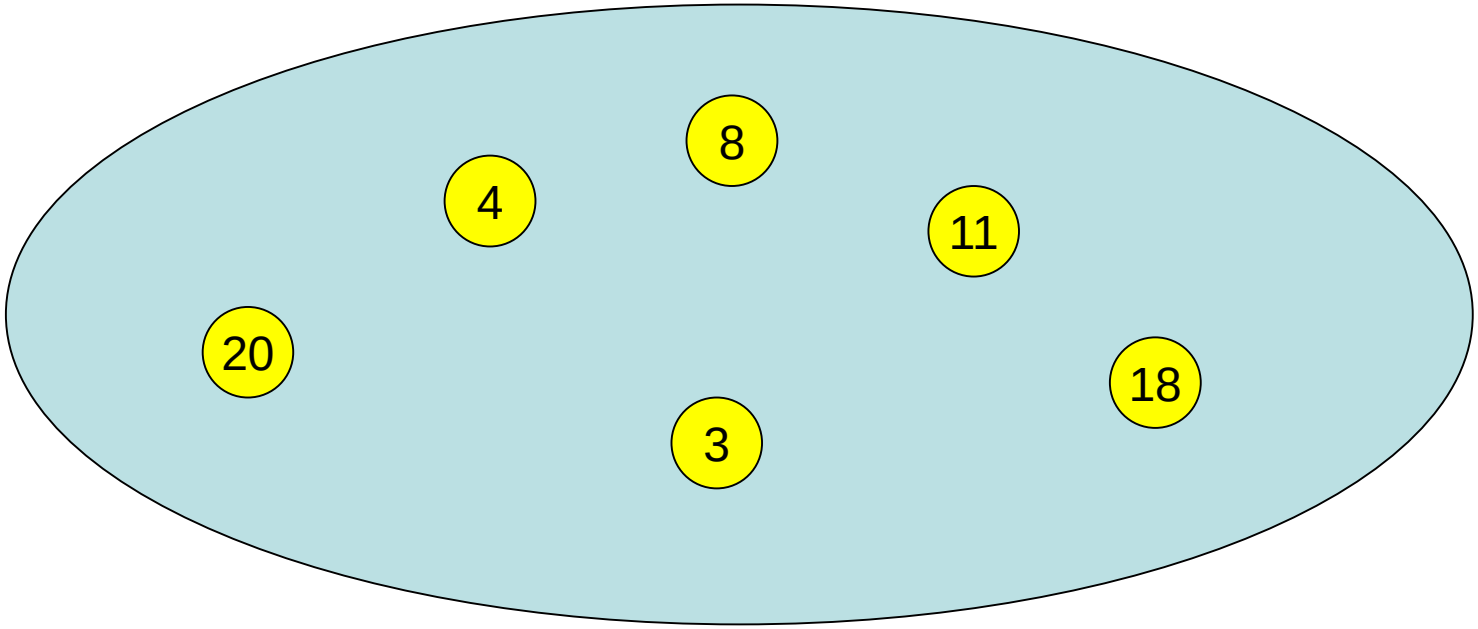
- Stelle für jede Datenanfrage zunächst (über Lookup) fest, mit welchen Prozessen Daten ausgetauscht werden. Das kann oft parallel geschehen.
- Führe dann den Datenaustausch sequentiell in der vorgegebenen Reihenfolge aus.

Wir werden auch andere Strategien in diesem Kapitel kennenlernen.

Übersicht

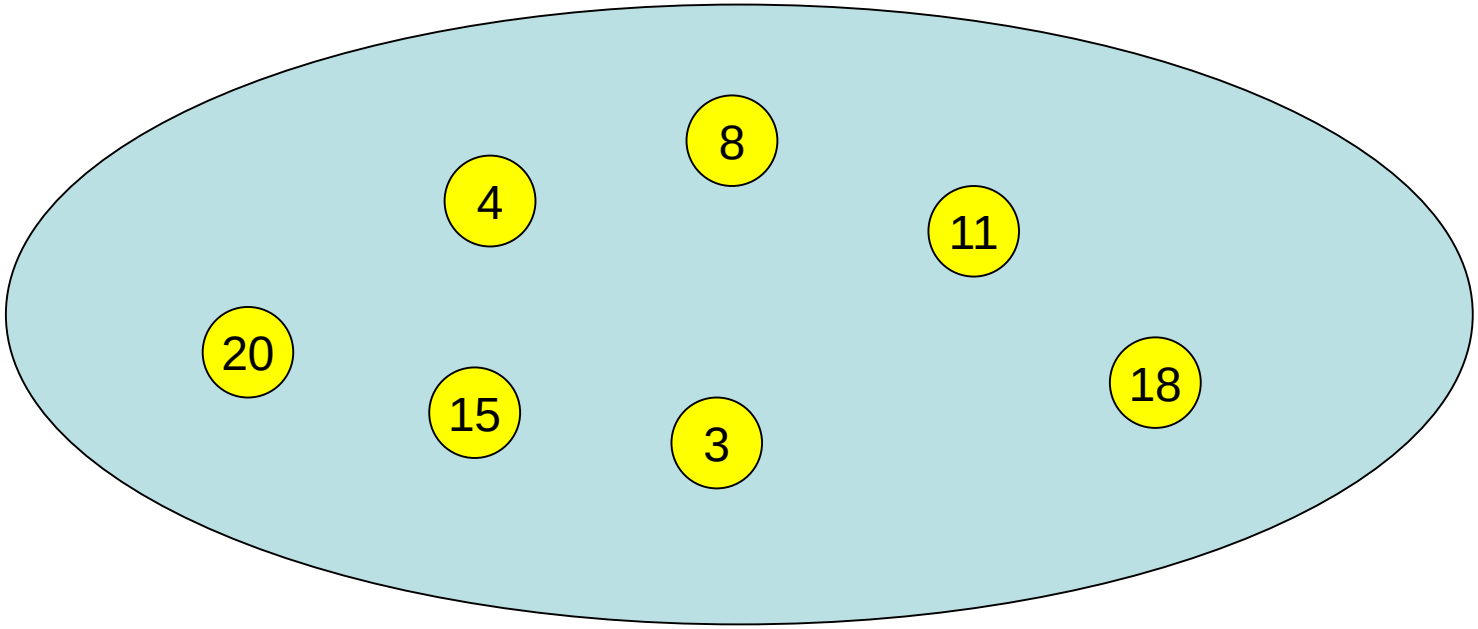
- Verteilte Hashtabelle
- Verteilte Suchstruktur
- Verteilte Queue
- Verteilter Stack
- Verteilter Heap

Wörterbuch



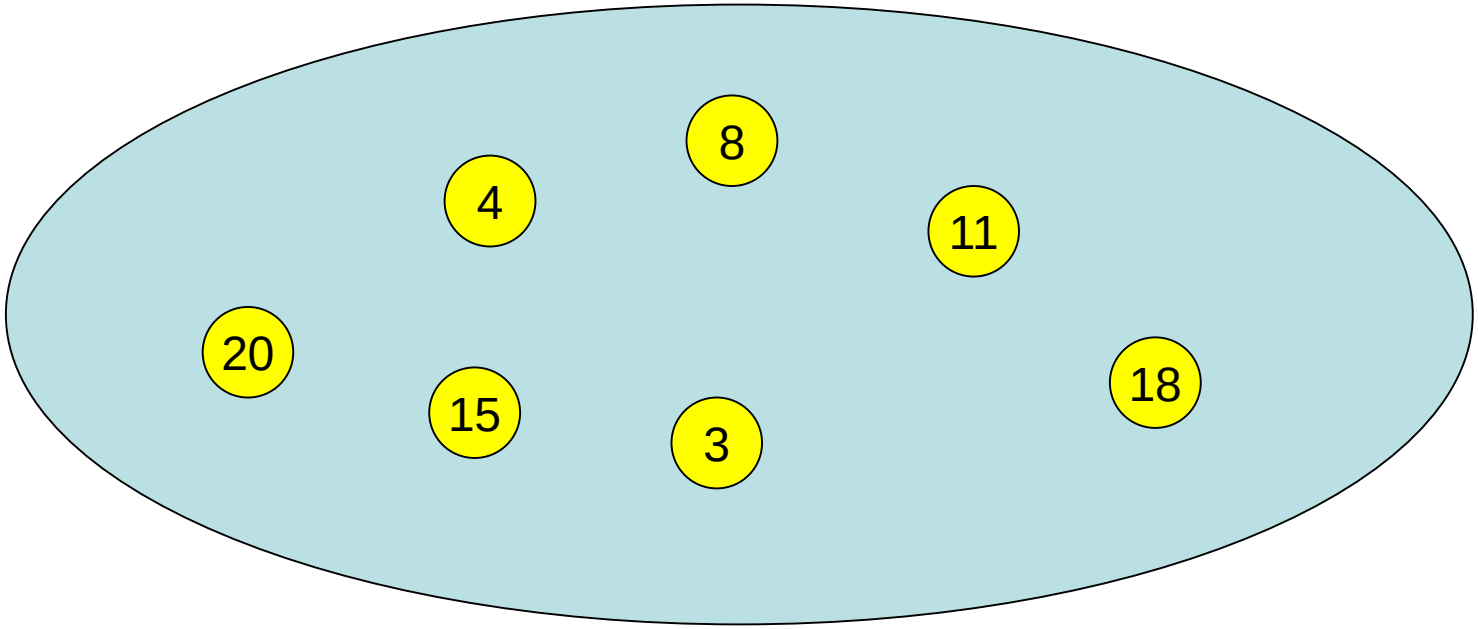
Wörterbuch

insert(15)



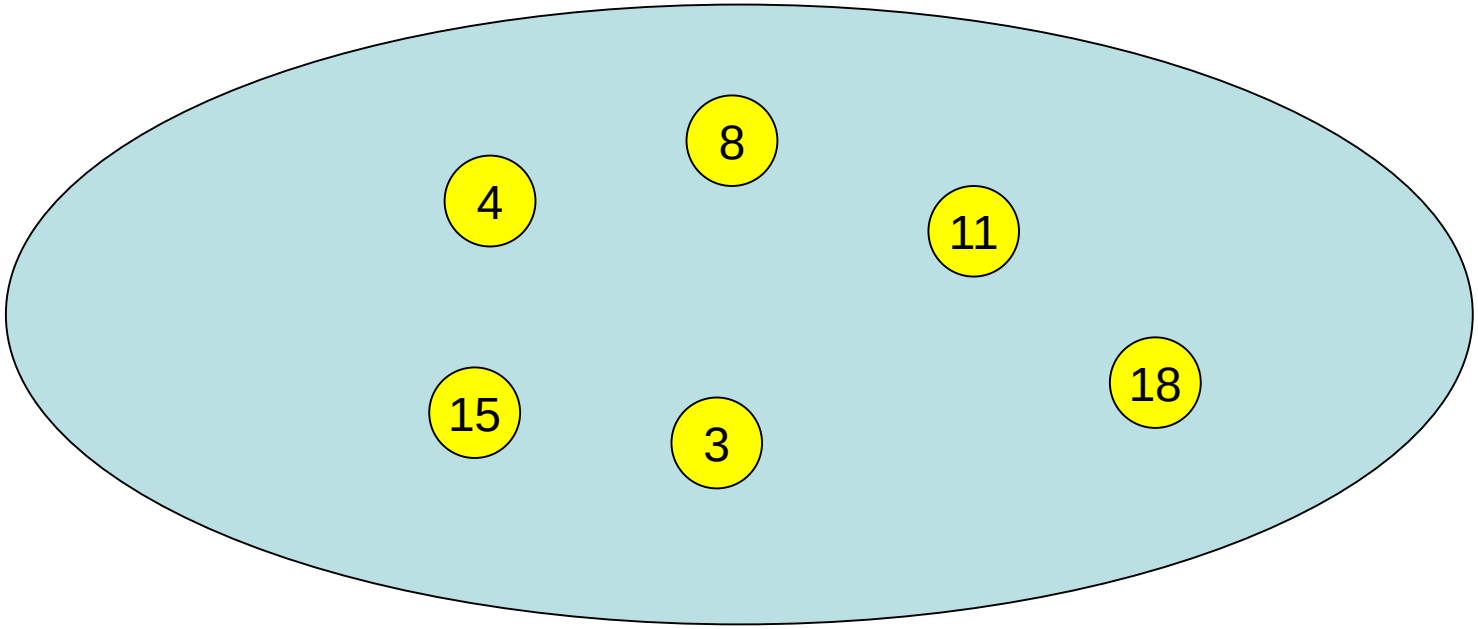
Wörterbuch

delete(20)



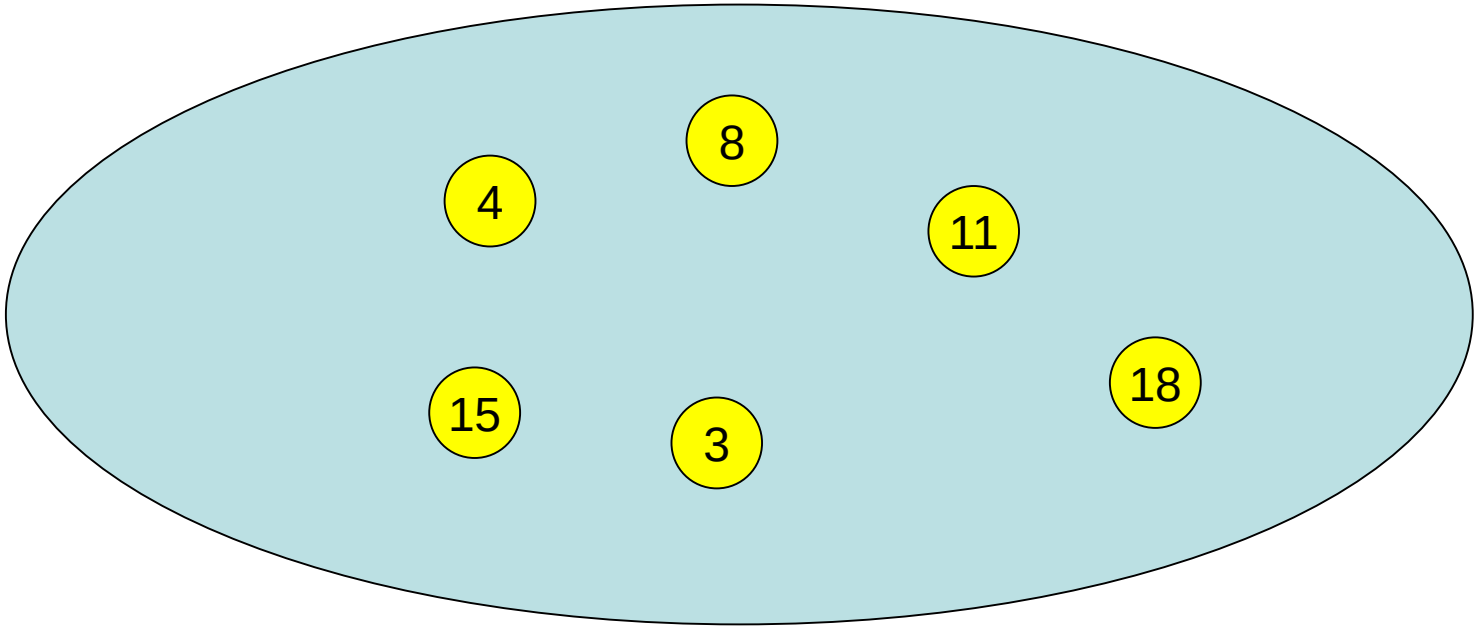
Wörterbuch

lookup(8) ergibt 8



Wörterbuch

lookup(7) ergibt ?



Wörterbuch-Datenstruktur

S: Menge von Elementen

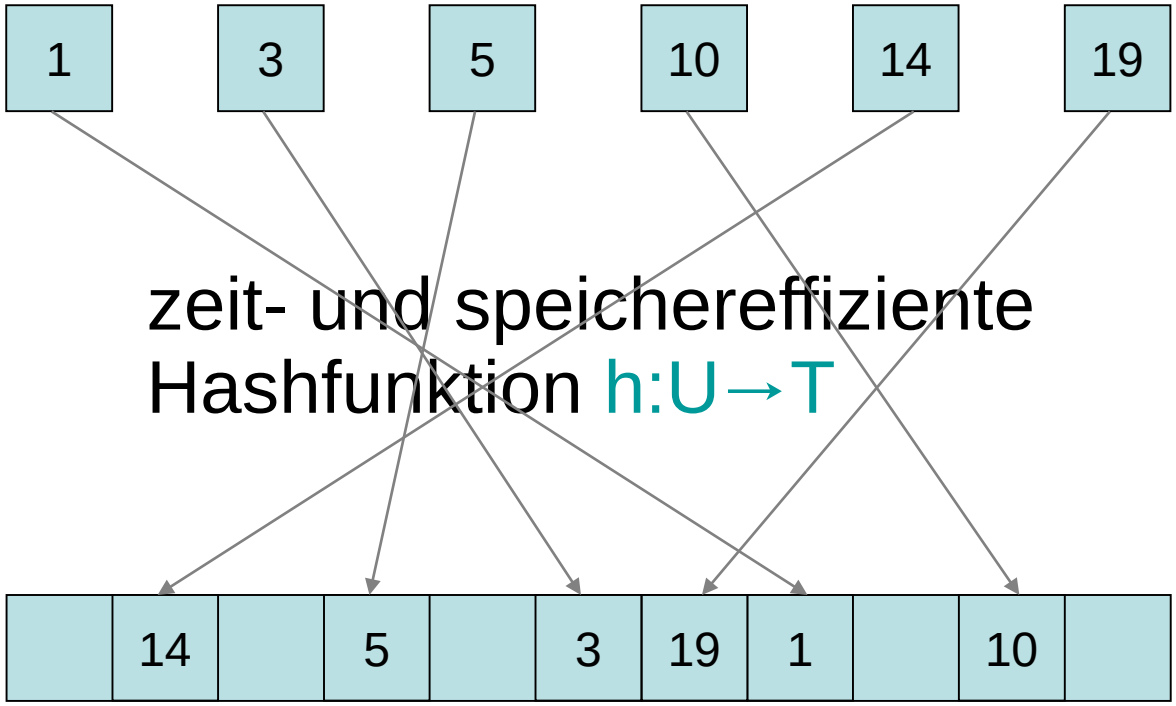
Jedes Element e identifiziert über key(e).

Operationen:

- S.insert(e): falls S bereits ein Element e´ enthält mit key(e´)=key(e), ersetze es durch e, sonst setze S:=S∪{e}
- S.delete(k): S:=S\{e}, wobei e das Element ist mit key(e)=k
- S.lookup(k): Falls es ein e∈S gibt mit key(e)=k, dann gib e aus, sonst gib ⊥ aus

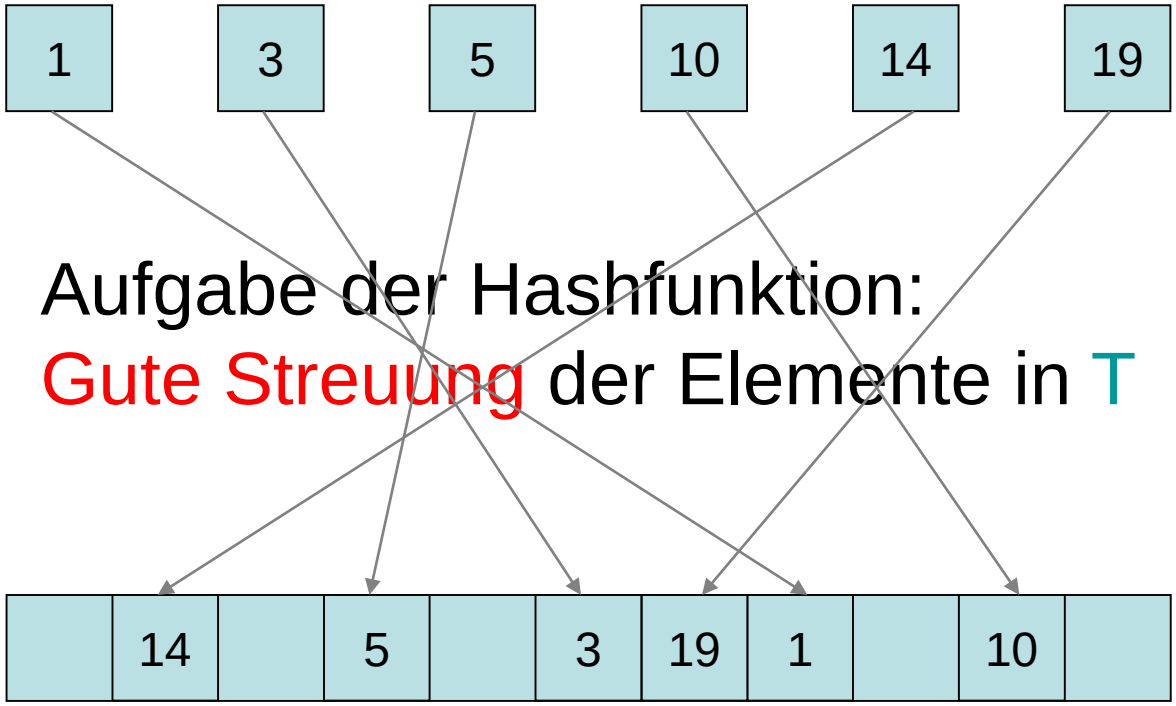
Effiziente Lösung: Hashing

Klassisches Hashing



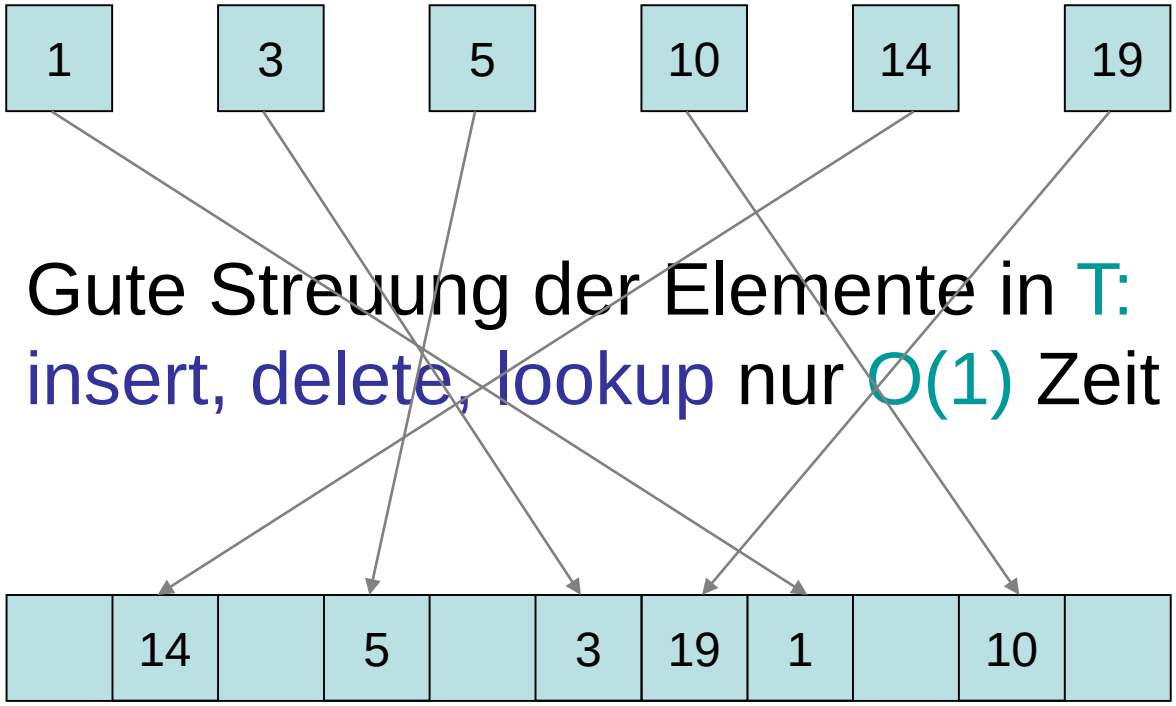
Hashtabelle T

Klassisches Hashing



Hashtabelle T

Klassisches Hashing

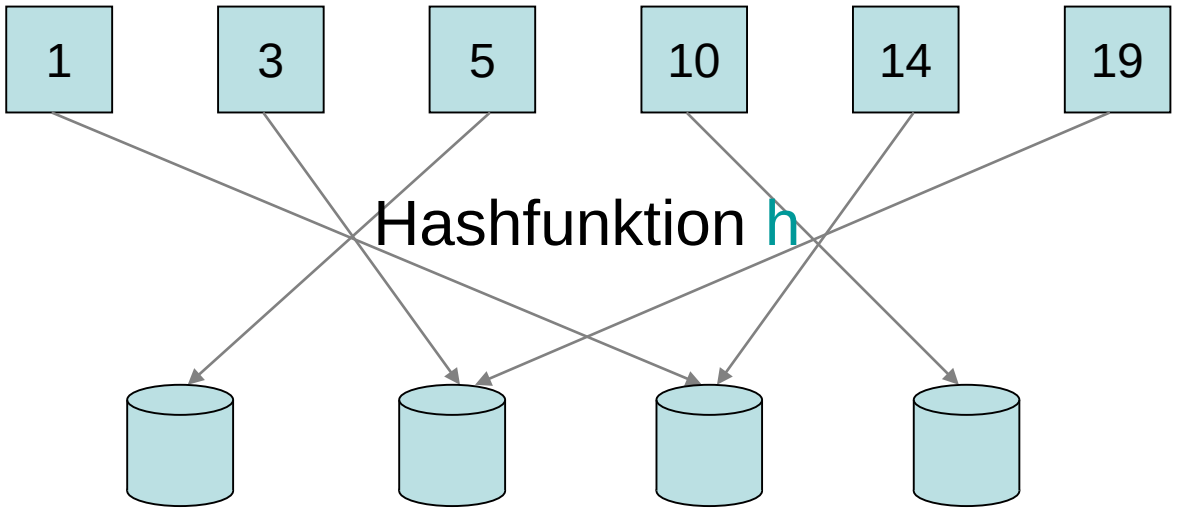


Gute Streuung der Elemente in T:
insert, delete, lookup nur O(1) Zeit

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.

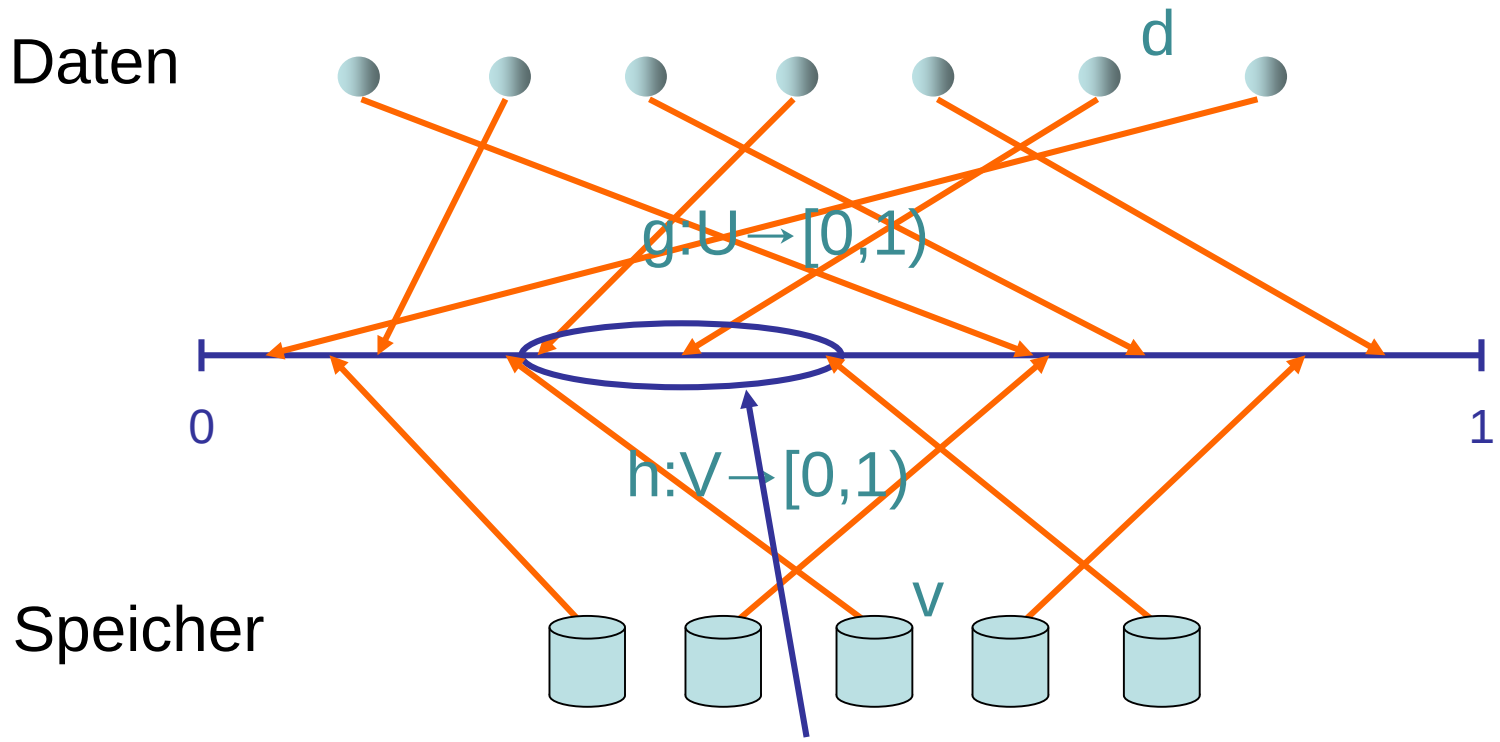
Nichtuniforme Speichersysteme: Kapazitäten können beliebig unterschiedlich sein

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

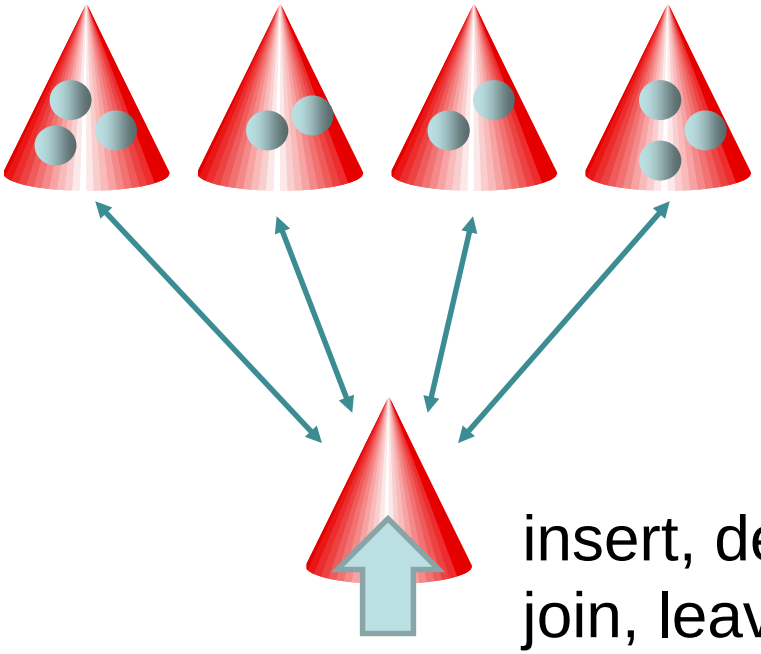
- V: aktuelle Prozessmenge (im folgenden auch Knoten genannt)
- $succ(v)$: nächster Nachfolger von v in V bzgl. Hashfunktion h (wobei $[0,1)$ als Kreis gesehen wird)
- $pred(v)$: nächster Vorgänger von v in V bzgl. Hashfunktion h

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

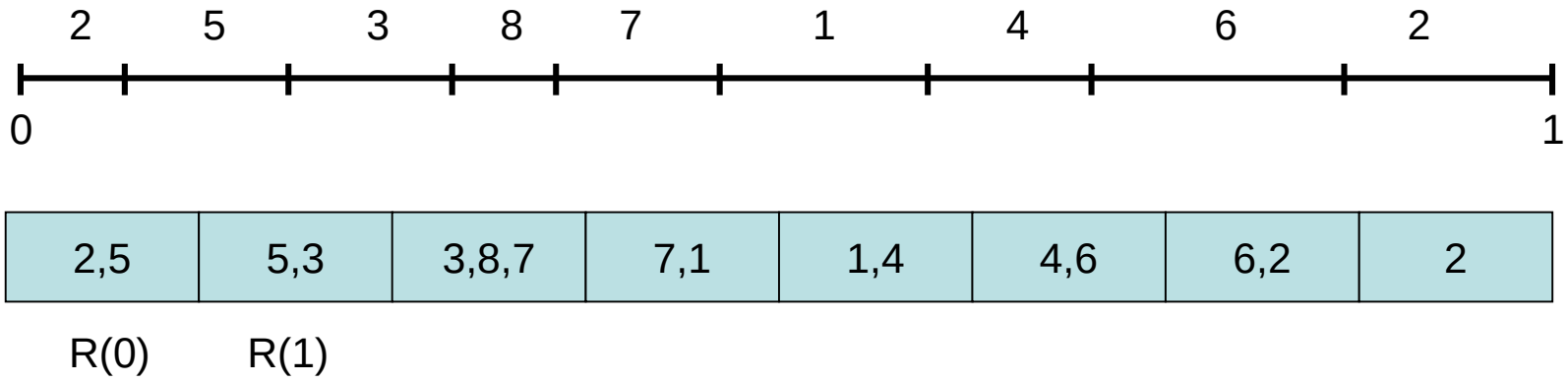


insert, delete, lookup,
join, leave

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:

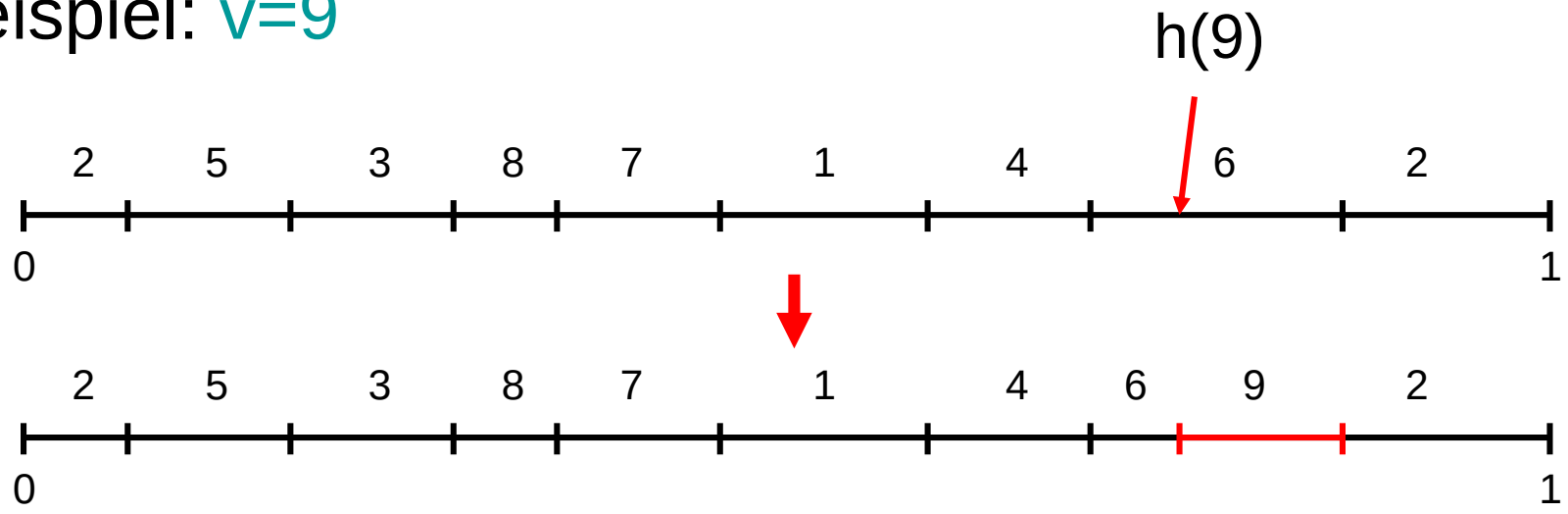
- $T[i]$ enthält für $m\geq n$ erwartet konstant viele Elemente und höchstens $O(\log n / \log\log n)$ mit hoher W.keit.
- D.h. Lookup(k) (die Ermittlung des zuständigen Knotens für einen Schlüssel) benötigt erwartet konstante Zeit

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

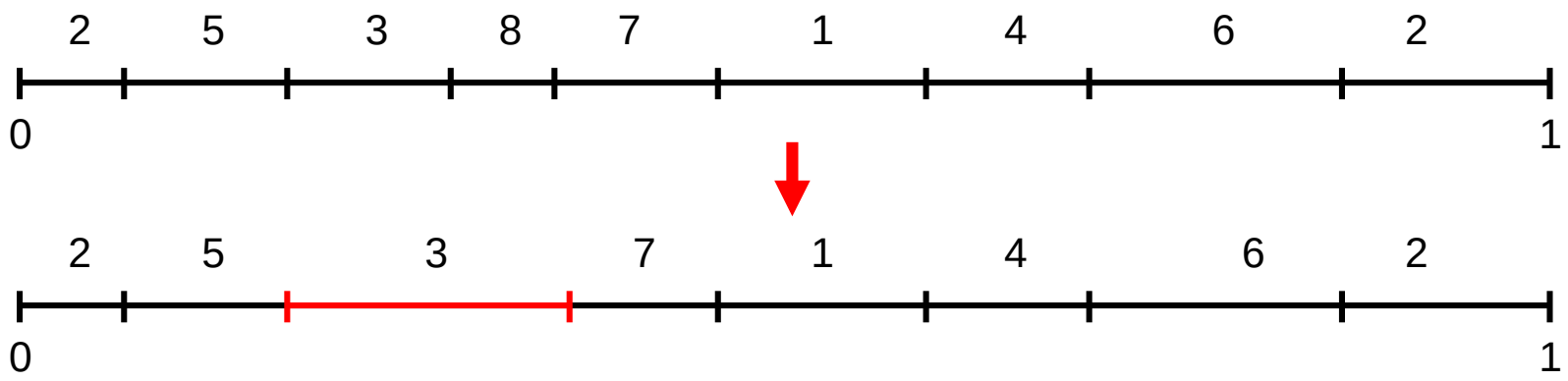


Verteilte Hashtabelle, Fall 1

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

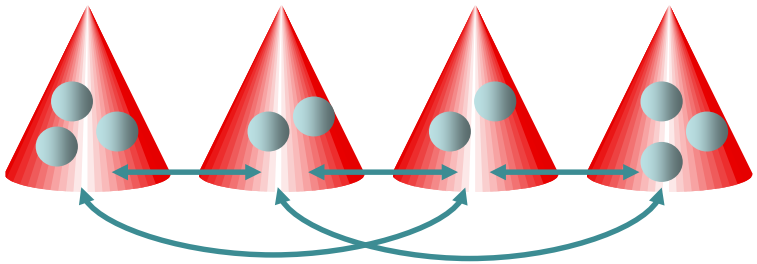
Problem: Schwankung um 1/n hoch!

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



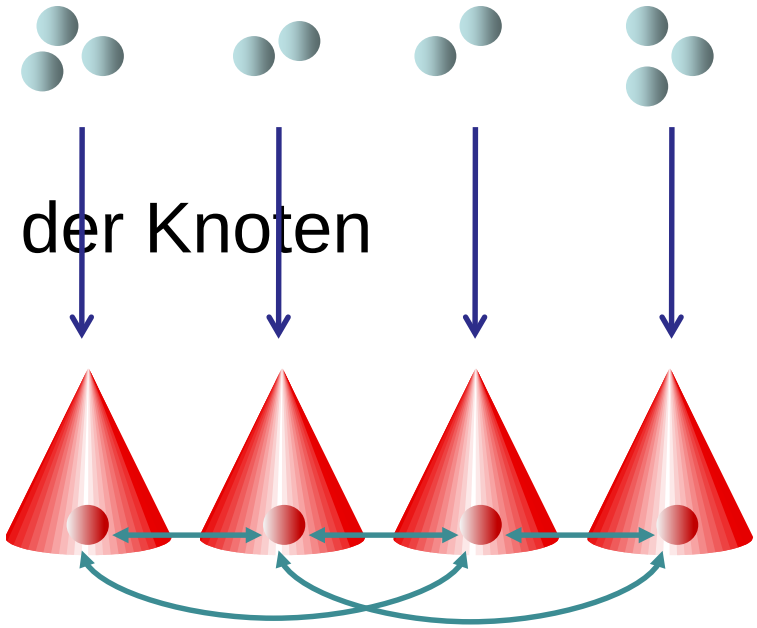
Jeder Knoten kann Anfragen (insert, delete, lookup, join, leave) generieren.

Verteilte Hashtabelle, Fall 2

Konsistentes Hashing: Zerlegung in zwei Probleme.

1. Abbildung der Daten auf die Knoten

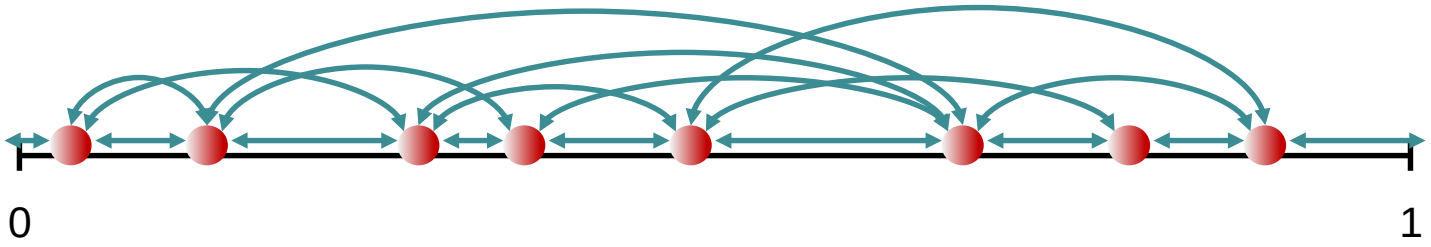
2. Vernetzung der Knoten



Verteilte Hashtabelle, Fall 2

Vernetzung der Knoten:

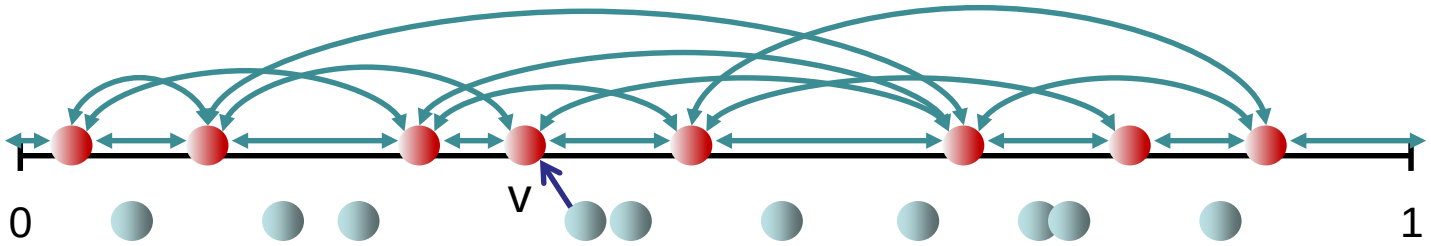
- Jedem Knoten v wird (pseudo-)zufälliger Wert $h(v) \in [0,1)$ zugewiesen.
- Verwende z.B. Skip+ Graph, um Knoten (hier im Kreis!) mittels $h(v)$ zu vernetzen.



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

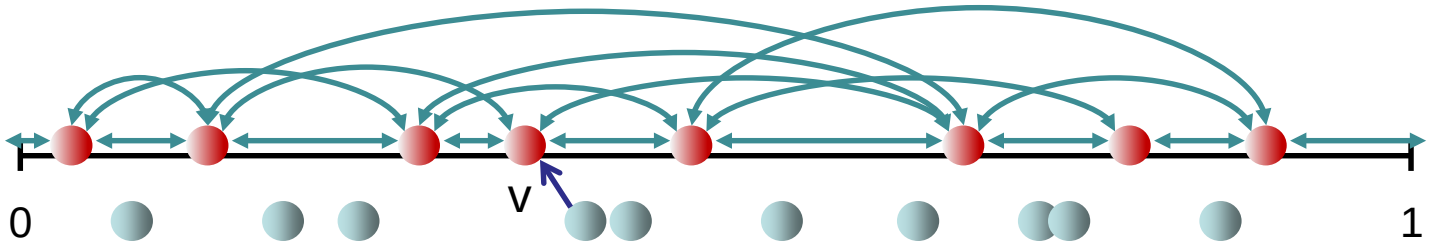
Passende search-Operation im Skip+ Graph für den Einsatz in der verteilten Hashtabelle:

```
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 Anfrage beim Ziel }
```

Verteilte Hashtabelle, Fall 2

Abbildung der Daten auf die Knoten:

- Verwende konsistentes Hashing

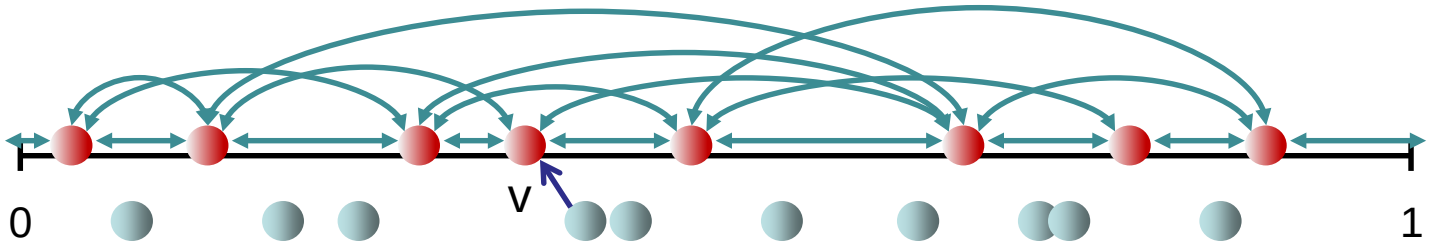


- delete(k): führe search(g(k)) im Skip+ Graph aus und lösche Datum d mit key(d)=k (falls da) im nächsten Vorgänger von g(k)

Verteilte Hashtabelle, Fall 2

Abbildung der Daten auf die Knoten:

- Verwende konsistentes Hashing

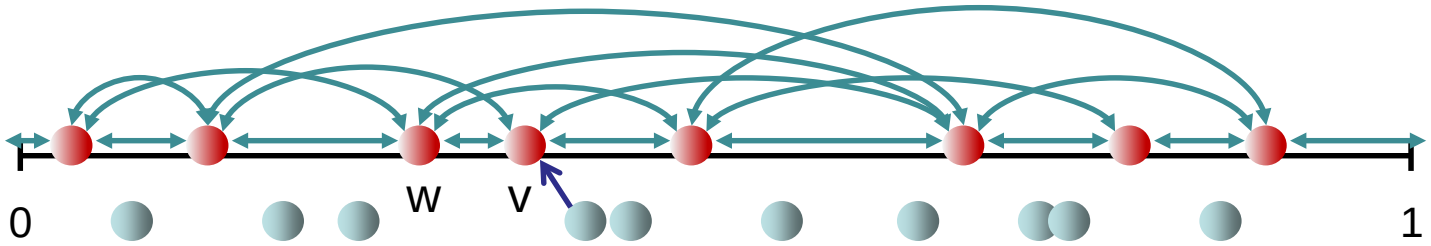


- lookup(k): führe search(g(k)) im Skip+ Graph aus und liefere Datum d mit key(d)=k (falls da) im nächsten Vorgänger von g(k) zurück

Verteilte Hashtabelle, Fall 2

Abbildung der Daten auf die Knoten:

- Verwende konsistentes Hashing

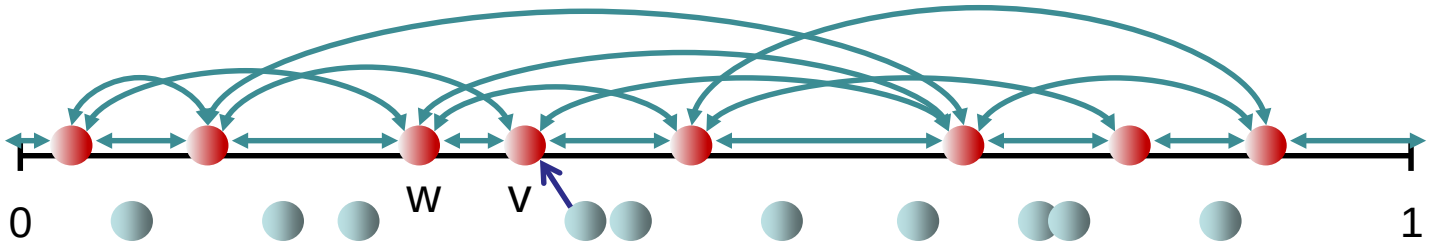


- join(v): nachdem v in den Skip+ Graph integriert ist, reicht es, pred(v)=w zu kontaktieren (mit welchem v direkt verbunden ist), um alle für v relevanten Daten gemäß des konsistenten Hashings zu erhalten

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

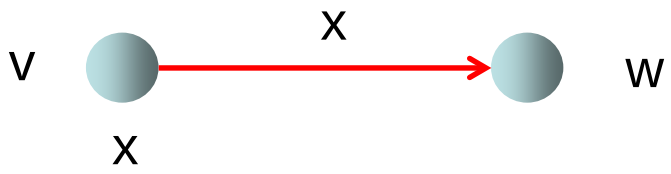
Verteilte Hashtabelle, Fall 2

Selbststabilisierung:

- Selbststabilisierender Skip+ Graph: gelöst
- Selbststabilisierende Datenplazierung: bewege Daten in timeout Aktion analog zur search Operation, falls der Ort des Datums falsch ist gemäß der konsistenten Hashing Regel. Sobald sich der Skip+ Graph stabilisiert hat, wird das die Daten an den richtigen Ort bewegen.

Selbststabilisierende Bewegung eines Datums:

- Angenommen, v muss Datum x an w weiterreichen.
- v behält Kopie von x und leitet in timeout solange x an w weiter, bis v eine Bestätigung von w erhalten hat. Dann erst löscht v x.



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 gemäß der konsistenten Hashing Regel. Sobald sich der Skip+ Graph stabilisiert hat, wird das die Daten an den richtigen Ort bewegen.

Suche nach einem Datum x:

- Eine Suchanfrage wird solange weitergeleitet, bis sie ihren Zielort (gemäß des konsistenten Hashings) erreicht hat.
- Sollte die Suchanfrage zwischendurch auf eine Kopie von x stoßen, wird diese in der Suchanfrage vermerkt (bzw. das bereits dort vermerkte x durch das neu gefundene x ersetzt), diese aber trotzdem weitergeleitet, bis das Ziel erreicht ist.

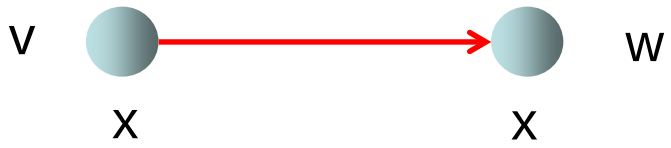
Verteilte Hashtabelle, Fall 2

Selbststabilisierung:

- Selbststabilisierender Skip+ Graph: gelöst
- Selbststabilisierende Datenplazierung: bewege Daten in timeout Aktion analog zur search Operation, falls der Ort des Datums falsch ist gemäß der konsistenten Hashing Regel. Sobald sich der Skip+ Graph stabilisiert hat, wird das die Daten an den richtigen Ort bewegen.

Insert-Operation für ein Datum x:

- Die Insert-Anfrage wird solange weitergeleitet, bis sie ihren Zielort (gemäß des konsistenten Hashings) erreicht hat.
- Sollte die Anfrage zwischendurch auf eine Kopie von x stoßen, wird diese auf den Insert-Wert aktualisiert, die Anfrage aber trotzdem weitergeleitet, bis das Ziel erreicht ist.



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 gemäß der konsistenten Hashing Regel. Sobald sich der Skip+ Graph stabilisiert hat, wird das die Daten an den richtigen Ort bewegen.

Delete-Operation für ein Datum x:

- Die Delete-Anfrage wird solange weitergeleitet, bis sie ihren Zielort (gemäß des konsistenten Hashings) erreicht hat.
- Alle Kopien von x, auf die die Delete-Anfrage auf ihrem Weg stößt, werden gelöscht.

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 gemäß der konsistenten Hashing Regel. Sobald sich der Skip+ Graph stabilisiert hat, wird das die Daten an den richtigen Ort bewegen.

Problem: bei der Bewegung eines Datums x kann es passieren, dass x zwischendurch aktualisiert (oder gelöscht) wird, aber später wieder durch eine veraltete Version von x, die noch in einer Nachricht unterwegs war, überschrieben wird (wir haben keine FIFO Kanäle!).

Mögliche Lösung: wir verwenden Zeitstempel und bevorzugen immer die Kopie mit aktuellstem Stempel.

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 gemäß der konsistenten Hashing Regel. Sobald sich der Skip+ Graph stabilisiert hat, wird das die Daten an den richtigen Ort bewegen.

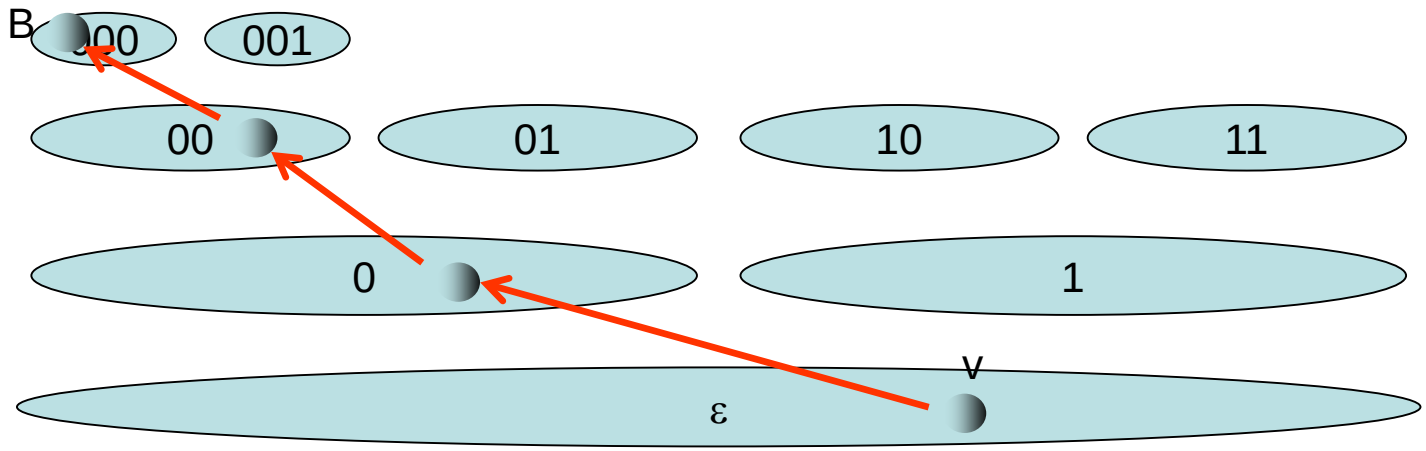
Mögliche Lösung: wir verwenden Zeitstempel und bevorzugen immer die Kopie mit aktuellstem Stempel.

Problem: es könnte einen sehr hohen, korrumpierten Zeitstempel geben!

Aber: sofern Insert(x)-Anfragen in genügend großen Zeitabständen gestellt werden, werden kene Zeitstempel benötigt, da am Ende nur der Zielknoten eine Kopie haben wird, und dieses wird die aktuelle Kopie sein (Warum?)

Monotone Suchbarkeit

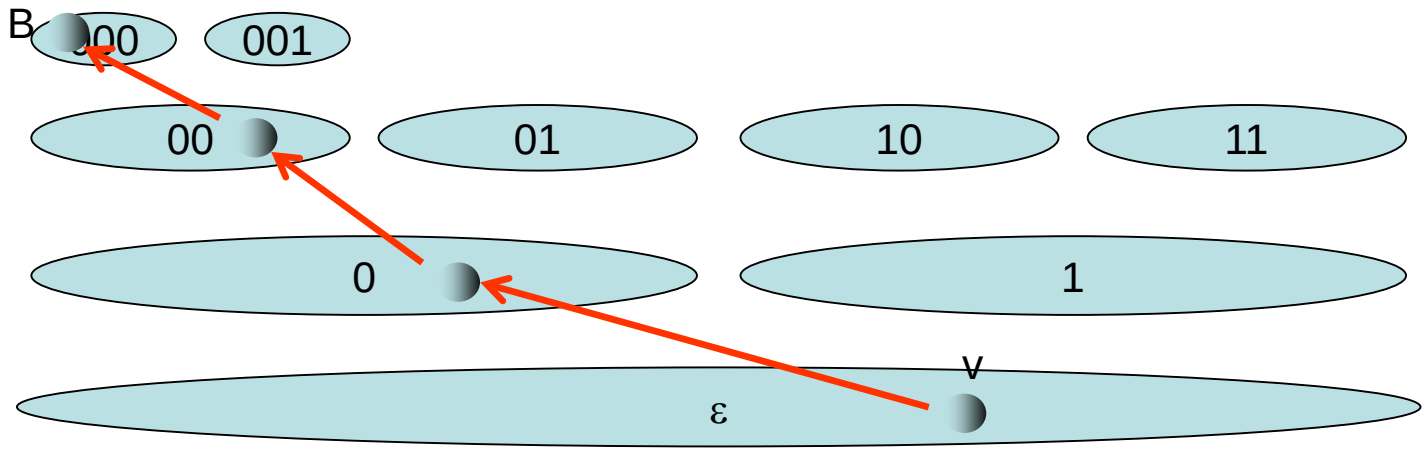
- Wir wissen, dass wir monotone Suchbarkeit für den Skip+ Graphen erzielen können. Die Idee der Suchoperation war dabei, bitweise den Bitstring des Ziels anzupassen.



- Angenommen, wir verwenden jetzt folgende Regel für das Speichern der Daten:
  Für jedes Datum x sei b(x) ein (pseudo-)zufälliger Bitstring.

Monotone Suchbarkeit

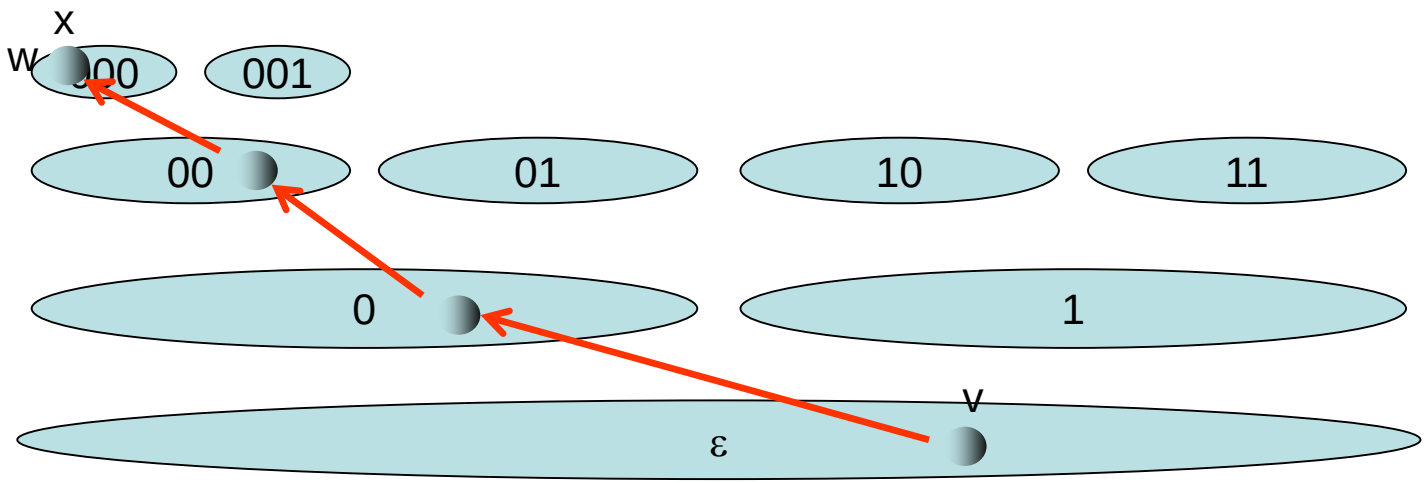
- Wir wissen, dass wir monotone Suchbarkeit für den Skip+ Graphen erzielen können. Die Idee der Suchoperation war dabei, bitweise den Bitstring des Ziels anzupassen.



- Speichere x bei dem Knoten w mit größtem gemeinsamen Präfix von r(w) und b(x). (Wir nehmen zunächst vereinfachend an, es gibt einen eindeutigen Knoten in V mit größtem gemeinsamen Präfix.)

Monotone Suchbarkeit

- Eine Anfrage von v für Datum x wird dann so geroutet, dass wir bitweise den Bitstring von b(x) anpassen, was uns bei einem stabilen Skip+ Graph zum eindeutigen Knoten w führt, der für x verantwortlich ist.



Monotone Suchbarkeit

Lookup Operation für Datum x: gestartet wird mit Lookup(x,0).

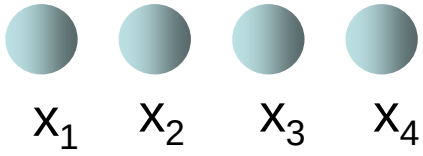
Lookup(x,i) $\rightarrow$
  if $\exists v \in marked(N_{i+1})$: $r_{i+1}(v)=b_{i+1}(x)$ then
    v:=oldest node in $marked(N_{i+1})$ with $r_{i+1}(v)=b_{i+1}(x)$
    v$\leftarrow$Lookup(sid,i+1)
  else
    serve Lookup request at current node

Dieses Lookup erfüllt die monotone Suchbarkeit in dem Sinne, dass wenn v in der Lage ist, eine Lookup Anfrage zum eindeutigen Knoten mit größter Präfixübereinstimmung mit x zu schicken, schafft v das in der Zukunft immer. (Beweis: Übung)

Weiterhin gilt: solange x nicht gelöscht wird, findet v mindestens eine Kopie von x (bei Verwendung der vorsichtigen Weiterleitungsstrategie für x, bei der x erst dann gelöscht wird, wenn eine Bestätigung erhalten wurde).

Monotone Suchbarkeit

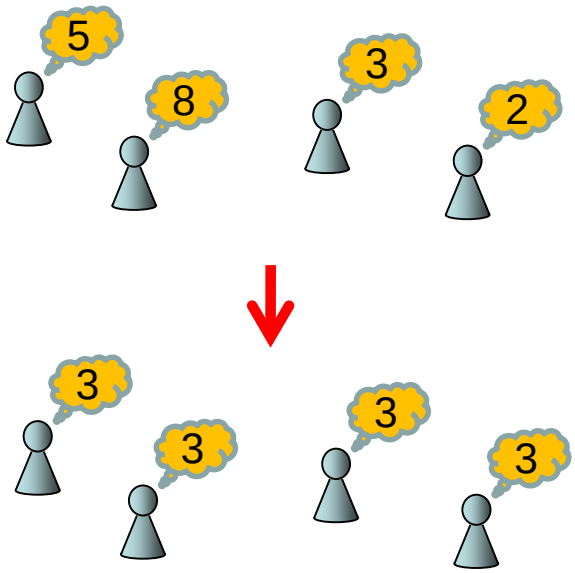
Was machen wir, wenn es keinen eindeutigen Knoten in V mit größter Präfixübereinstimmung mit x gibt?



Die Knoten müssen sich dann auf einen der Werte einigen.
Dazu benötigen wir ein sogenanntes Konsensusprotokoll.

Verteilter Konsensus

Konsensusproblem: Prozesse müssen sich für denselben Wert entscheiden.

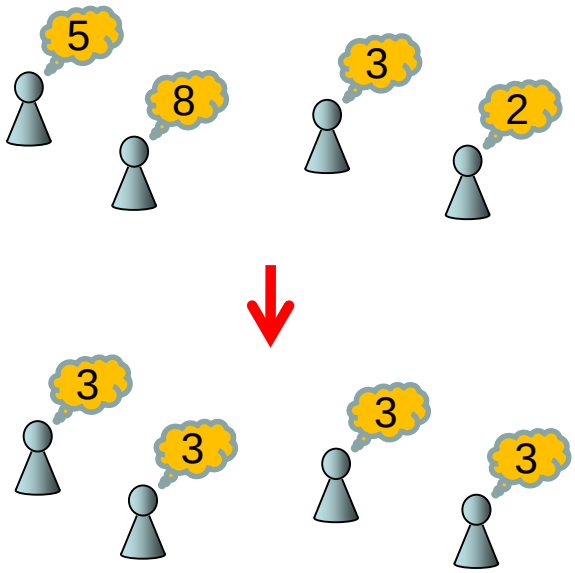


Formale Anforderungen:

- Terminierung: Alle korrekten Prozesse müssen sich irgendwann für einen Ausgabewert entscheiden.
- Übereinstimmung: Alle korrekten Prozesse müssen denselben Ausgabewert haben.
- Validität: Für jeden korrekten Prozess muss die Ausgabe gleich einer der Eingabewerte sein.

Verteilter Konsensus

Stabiler Konsensus Problem: Prozesse müssen irgendwann denselben Wert haben.



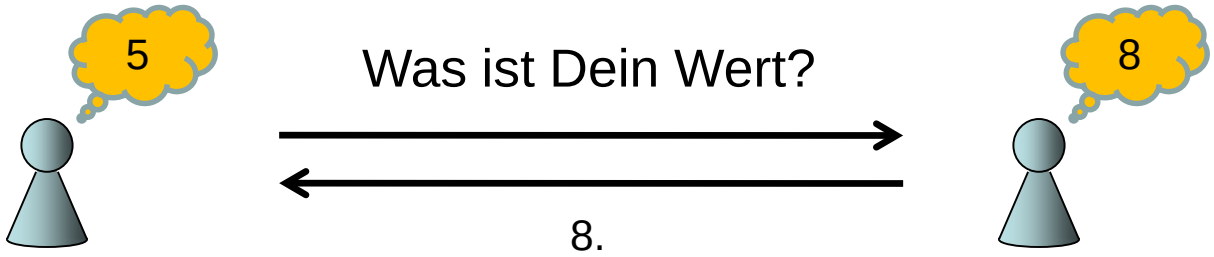
Formale Anforderungen:

- Stabilität: Alle korrekten Prozesse haben irgendwann einen stabilen Ausgabewert.
- Übereinstimmung: Alle korrekten Prozesse haben denselben stabilen Ausgabewert.
- Validität: Für jeden korrekten Prozess muss der stabile Wert gleich einer der Eingabewerte sein.

Stabiler Konsensus

Modell:

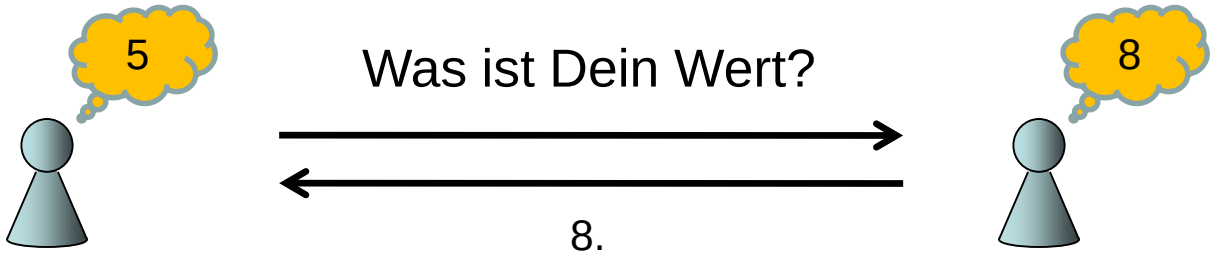
- n Prozesse
- Jeder Prozess kennt alle anderen (Clique)
- Zeit verläuft in synchronen Runden
- In jeder Runde kontaktiert jeder Prozess eine feste Anzahl anderer Prozesse.



Stabiler Konsensus

Komplexitätsmaße:

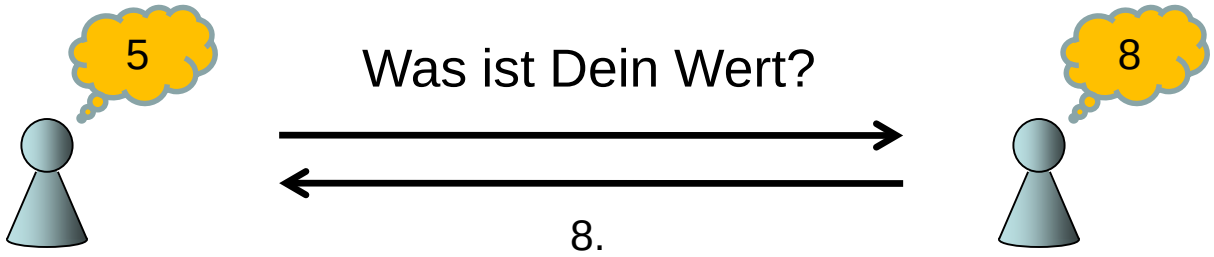
- Zeit: minimiere Anzahl Kommunikationsrunden bis stabiler Konsensus erreicht ist.
- Arbeit: minimiere maximale Arbeit (d.h. Anzahl an Nachrichten) die ein Prozess bis zum stabilen Konsensus benötigt.



Stabiler Konsensus

Grundlegende Fragen:

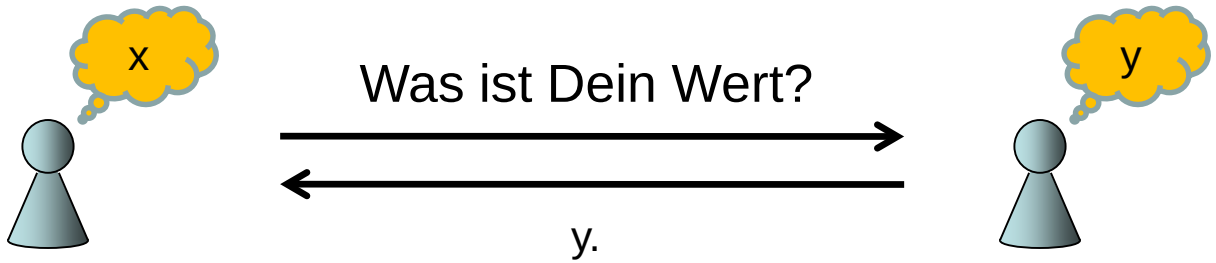
- Ist es möglich, einen stabilen Konsensus in logarithmischer Zeit und Arbeit von einem beliebigen Zustand aus zu erreichen?
- Falls ja, wieviele gegnerische Prozesse können toleriert werden?



Stabiler Konsensus

Alle n Prozesse sind ehrlich:

- Minimum Regel:



x:=min{x,y}

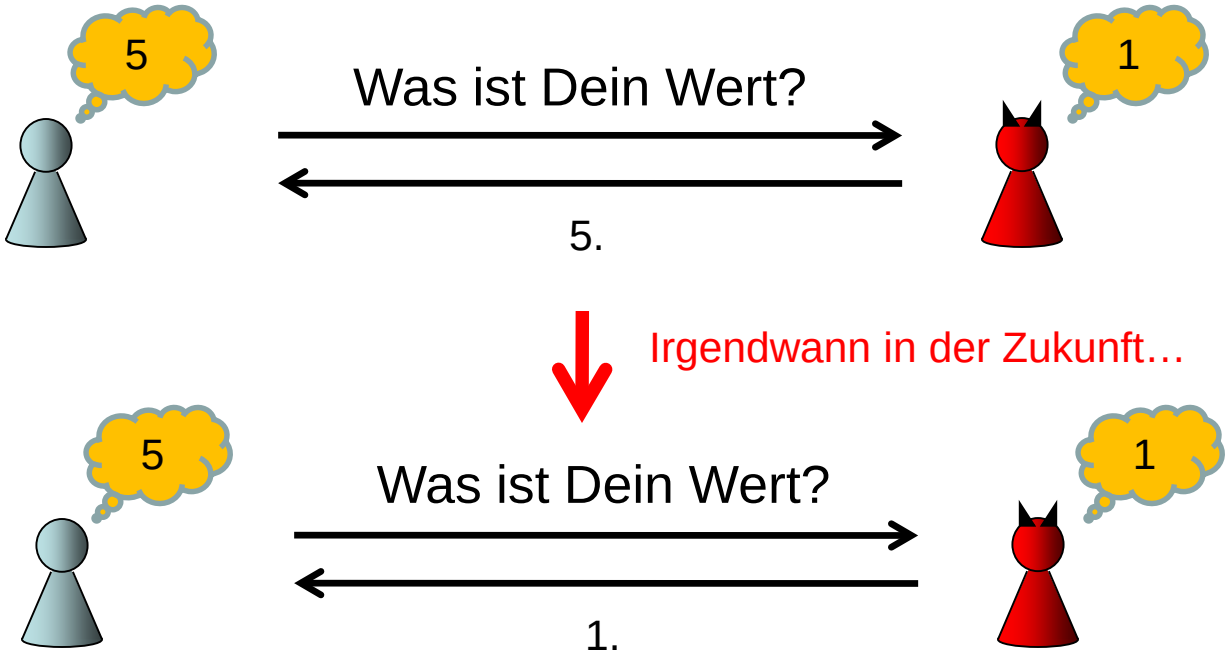
Zufälliger Prozess

- Nach $O(\log n)$ Runden (und daher $O(\log n)$ Arbeit) haben alle Prozesse mit hoher Wahrscheinlichkeit (d.h. Wahrscheinlichkeit $1-1/n^{\Omega(1)}$) denselben Wert

Stabiler Konsensus

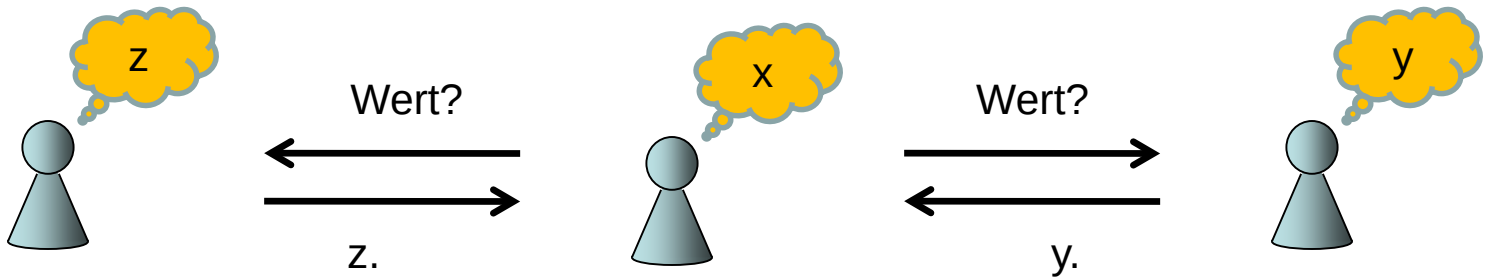
Ein Prozess gegnerisch:

- Minimum Regel: unbegrenzte Laufzeit.



Stabiler Konsensus

Besser: Median Regel



Zufälliger Proz. x:=median{x,y,z} Zufälliger Proz.

- O(log n) Runden m.h.W.: alle korrekt
- O(log n loglog n) m.h.W.: $<\sqrt{n}$ Gegner

Stabiler Konsensus

Problem bei Median Regel: neue Werte (durch Insert Anfrage) werden nicht akzeptiert (da Median Regel nicht mehr vom Konsensus auf altem Wert abweicht)!

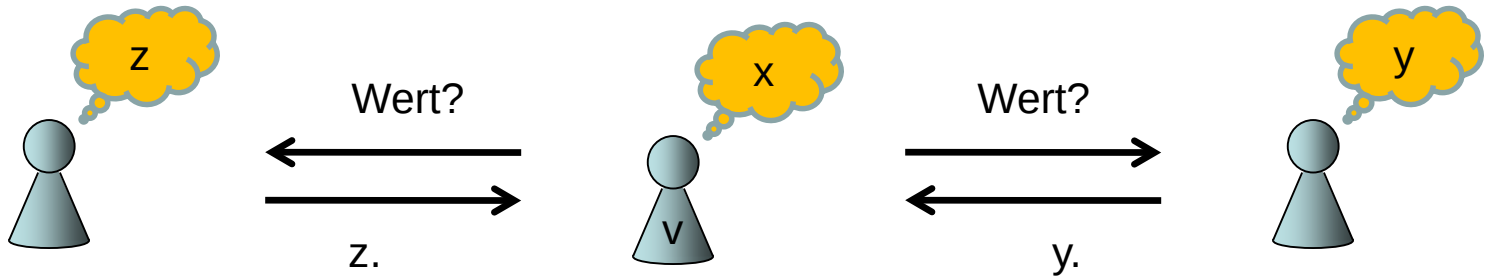
Lösungen:

- Verwende Zeitstempel, Maximum Regel basierend auf Zeitstempel. Das ist aber wieder leicht angreifbar und auch für die Selbststabilierung problematisch, da anfangs ein sehr hoher Zeitstempel existieren könnte!
- Verwende Medianregel auf dem Log der Werte, durch den eine Variable x gegangen ist.

Log: (x_1,x_2,x_3,x_4,x_5), wobei x_1 die älteste und x_5 die jüngste Kopie ist. Der Log wird für die Medianregel als Binärzahl $0.x_1x_2x_3x_4x_5$ interpretiert. Neue Werte werden hinten an den Log angehängt.

Stabiler Konsensus

Median Regel mit Logs:



- Prozess v ergänzt zunächst x,y,z hinten um fehlende Werte, die in anderen Logs vorkommen, bis x,y,z alle Werte aller Logs enthalten.
- Erst dann wird die Median Regel auf x,y und z angewendet.

Übung: erklären Sie, warum sich mit dieser Regel neue Werte durchsetzen können, und warum bei konkurrierenden neuen Werten sich ein eindeutiger Wert durchsetzen wird.

Stabiler Konsensus

Beispiel für die Ergänzung von Logs:

Angenommen, v habe folgende Logs:

- $x=(x_1,x_3,x_5,x_4)$
- $y=(x_2,x_4,x_5)$
- $z=(x_1,x_4,x_2,x_3)$

Dann erweitert v diese zunächst zu

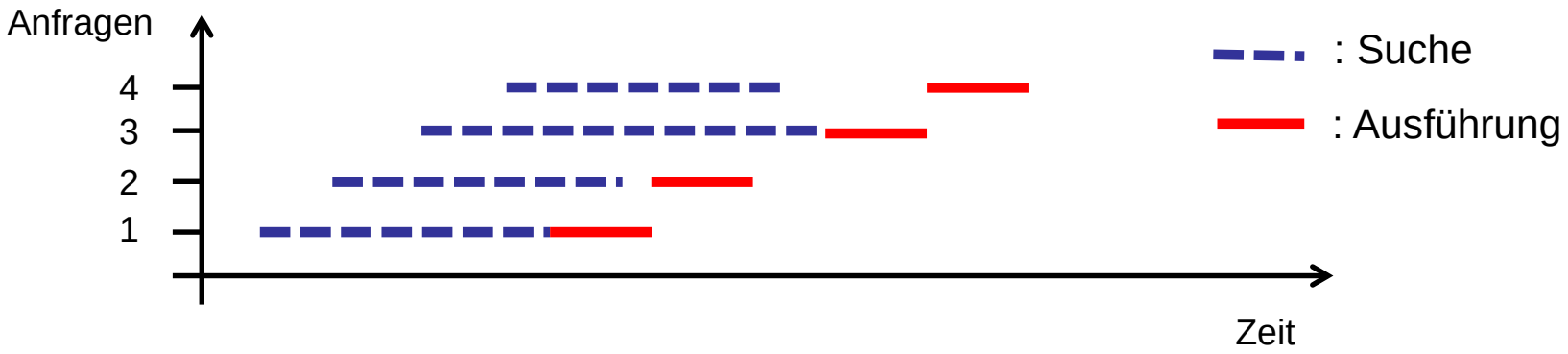
- $x'=(x_1,x_3,x_5,x_4,x_2)$
- $y'=(x_2,x_4,x_5,x_1,x_3)$
- $z'=(x_1,x_4,x_2,x_3,x_5)$

und wendet dann die Medianregel auf die binären Kodierungen von x´, y´ und z´ (d.h. $0.x_1x_3x_5x_4x_2$, $0.x_2x_4x_5x_1x_3$ und $0.x_1x_4x_2x_3x_5$) an.

Ist der Median z.B. $0.x_2x_4x_5x_1x_3$, dann speichert v y´ als neuen Log bei sich ab.

Sequentielle Konsistenz

Kann z.B. durch lokal sequentielle Ausführung garantiert werden. Benötigt aber einen legalen Zustand der verteilten Hashtabelle oder monotone Suchbarkeit, 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

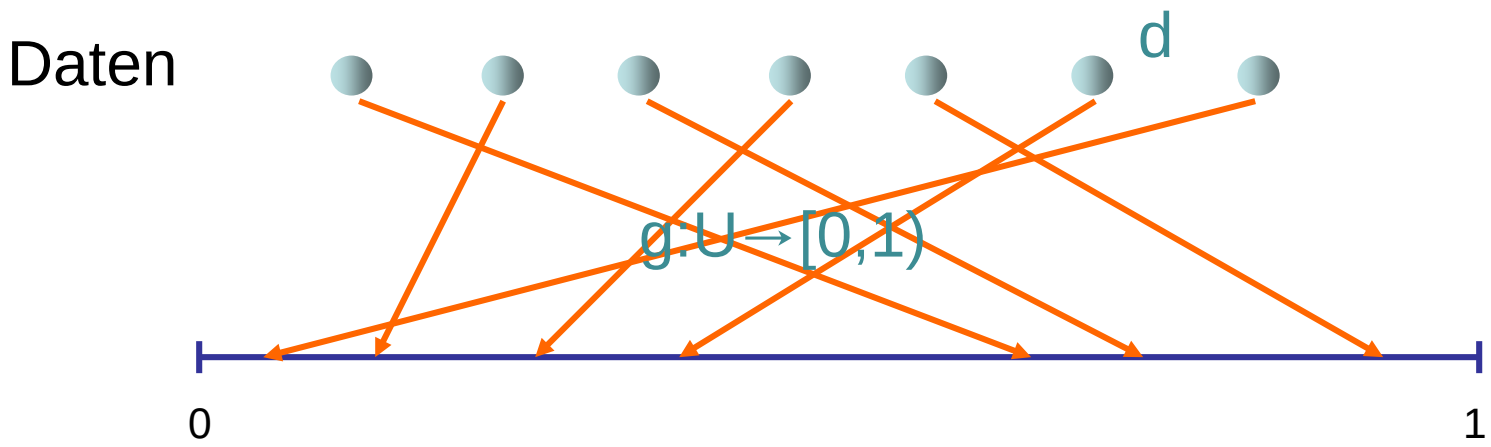
Situation hier: wir haben Knoten mit beliebigen relativen Kapazitäten $c_1,\ldots,c_n$, d.h. $\sum_i c_i = 1$.

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

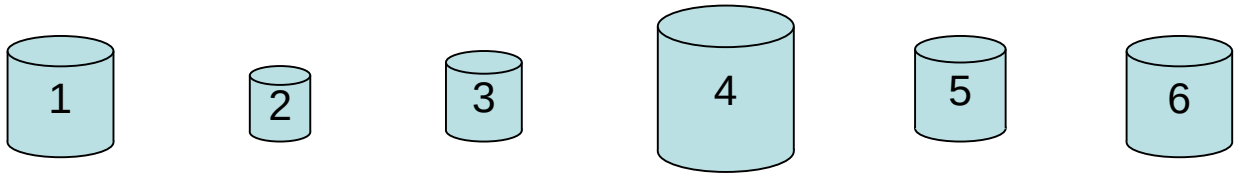
Lösung: SHARE

SHARE

Datenabbildung: wie bei konsistentem Hashing



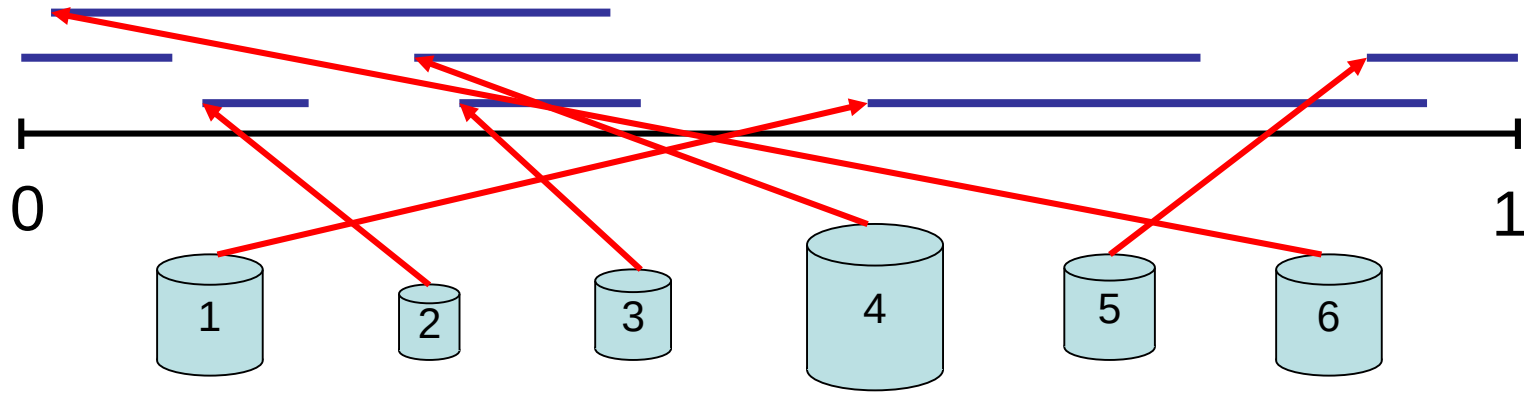
Zuordnung der Speicher: andere Strategie



SHARE

Zuordnung zu Speichern: zweistufiges Verfahren.

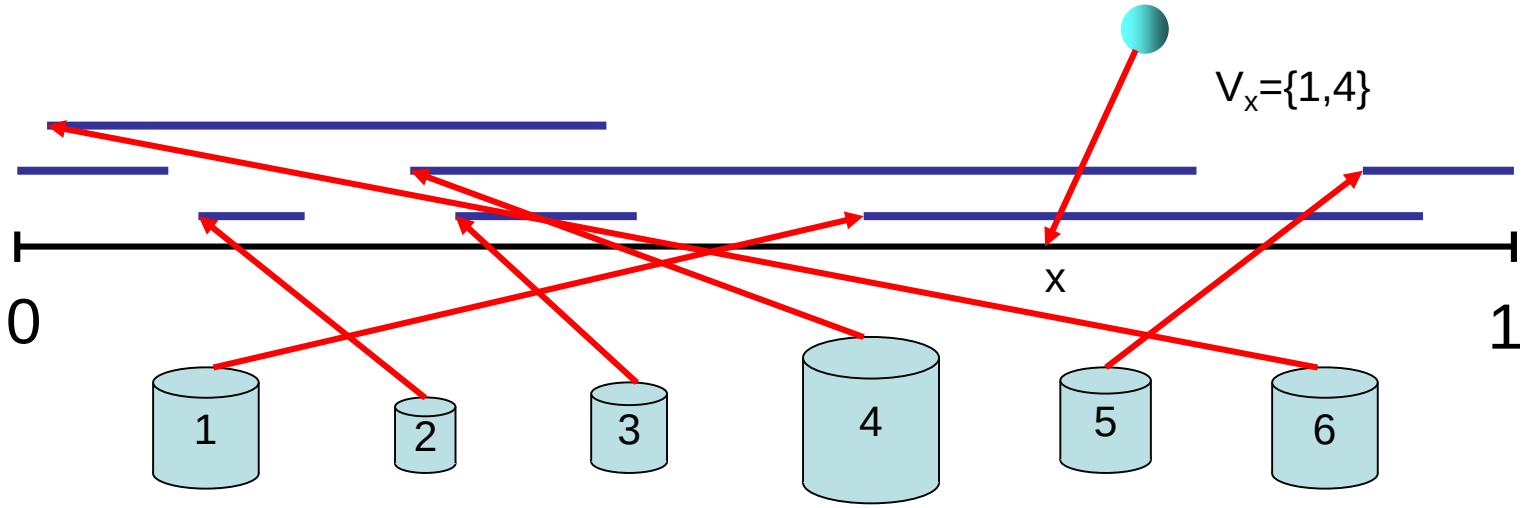
1. Stufe: Jedem Knoten v wird ein Intervall $I(v) \subseteq [0,1)$ der Länge $s \cdot c_v$ zugeordnet, wobei $s = \Theta(\log n)$ ein fester Stretch-Faktor ist. Die Startpunkte der Intervalle sind durch eine Hashfunktion $h: V \rightarrow [0,1)$ gegeben.



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).

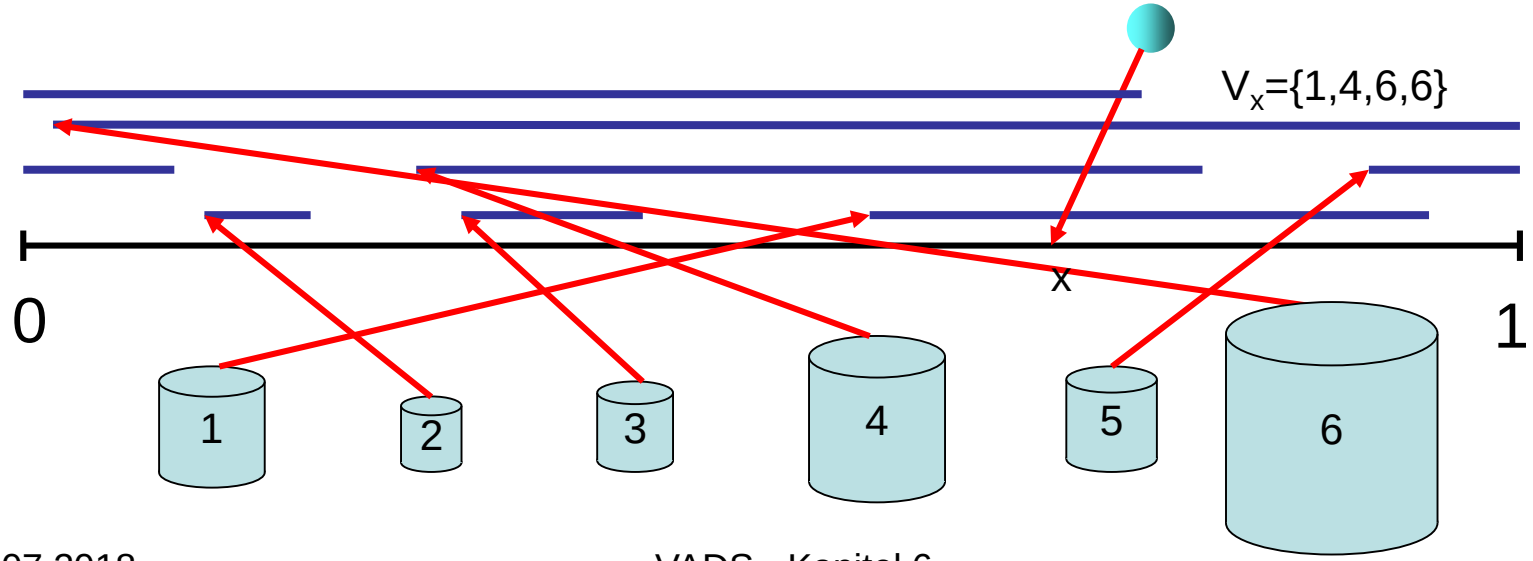


SHARE

Zuordnung zu Speichern: zweistufiges Verfahren.

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 $I(v)$ kommt v sooft in V_x vor wie $I(v)$ x enthält).

Multimenge: v kann mehrfach in V_x vorkommen, falls $c_v>1/s$.

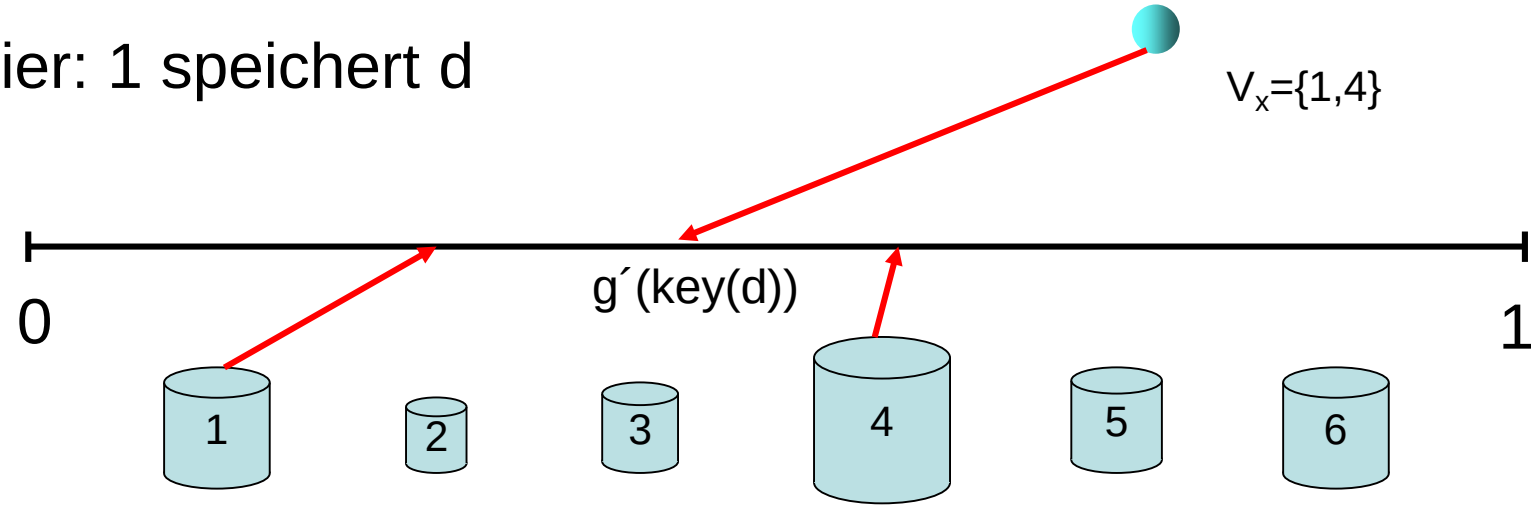


SHARE

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

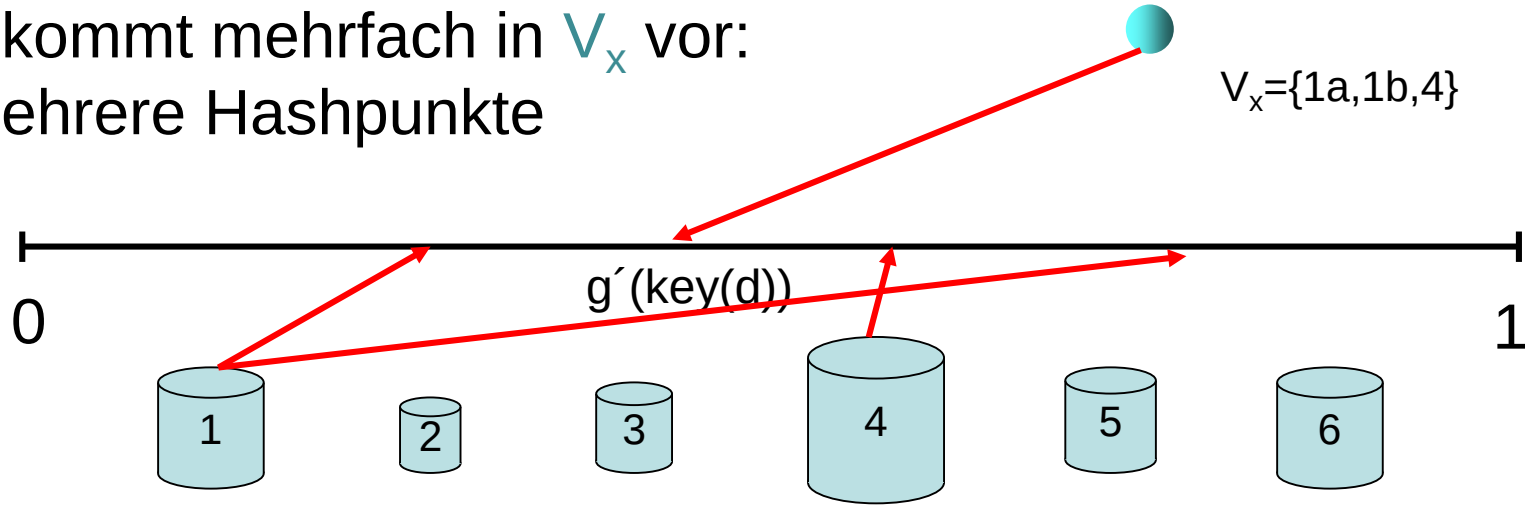


SHARE

Zuordnung zu Speichern: zweistufiges Verfahren.

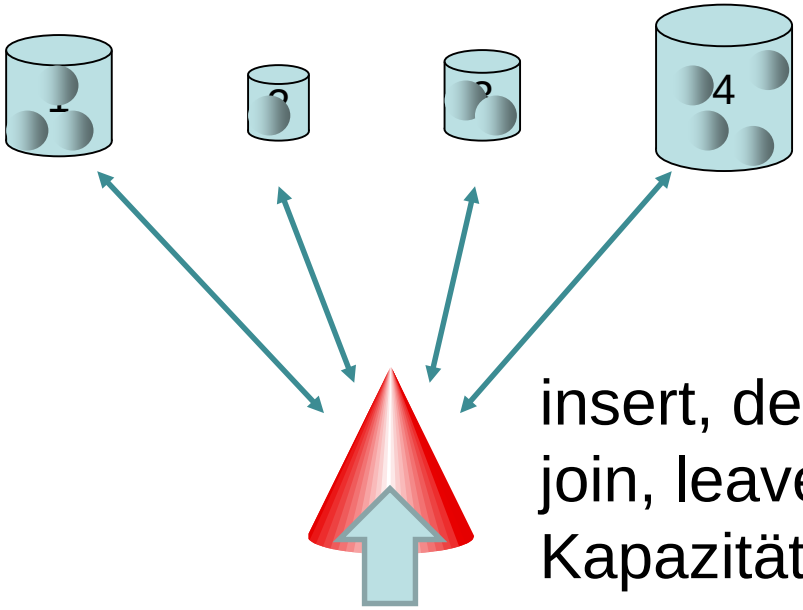
2. Stufe: Für Datum d wird mittels konsistentem Hashing mit Hashfunktionen h' und g' (die für alle Multimengen gleich sind) ermittelt, welcher Knoten in V_x Datum d speichert.

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)

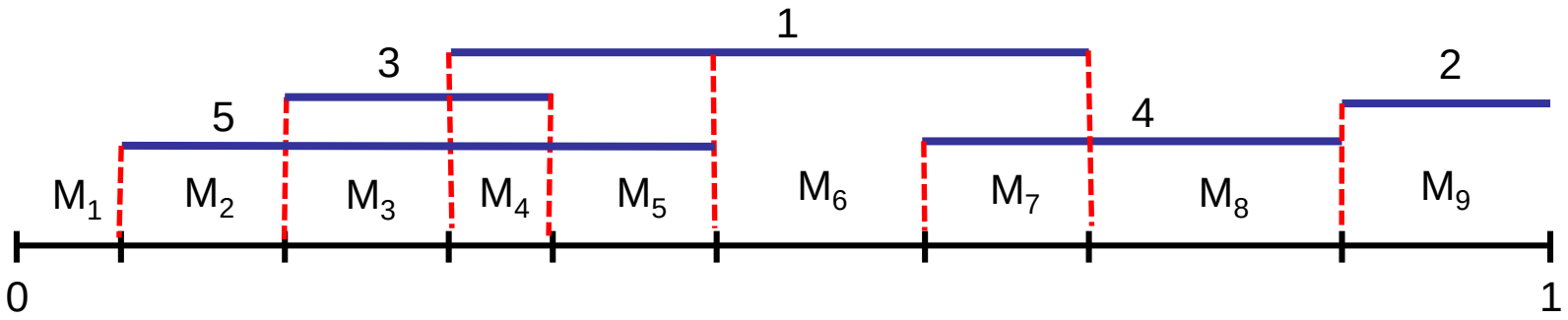
Laufzeit:

- 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

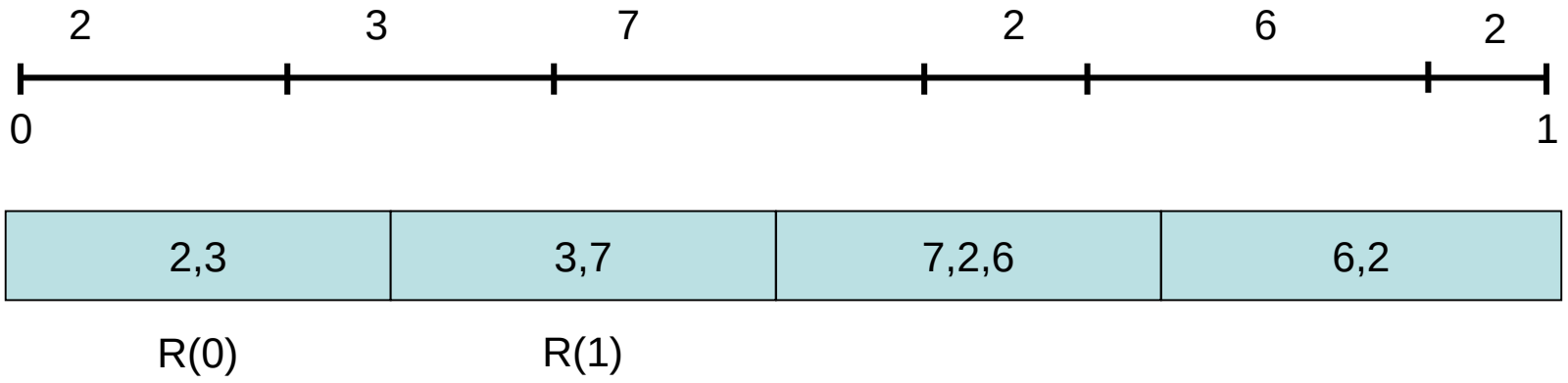
R(0) R(1)

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, 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 $s=c \log n$ ist für genügend großes c.
- Knoten i hat Intervall der Länge $s \cdot c_i$
- Erwarteter Anteil Daten in Knoten i:

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

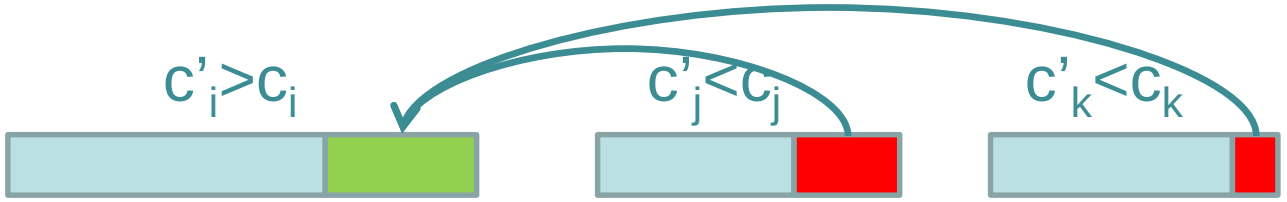
↑ ↑

Phase 1 Phase 2

SHARE

Punkt 3:

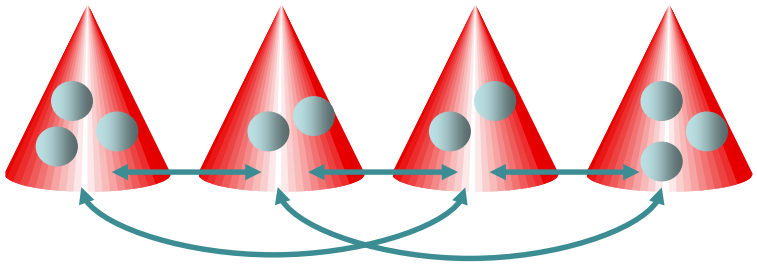
- Betrachte eine Veränderung der Kapazitäten von $(c_1,\ldots,c_n)$ nach $(c'_1,\ldots,c'_n)$
- Unterschied: $c = \sum_i |c_i - c'_i|$
- Optimale Strategie, um Fairness zu bewahren: replaziere einen $c/2$-Anteil der Daten



- 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.

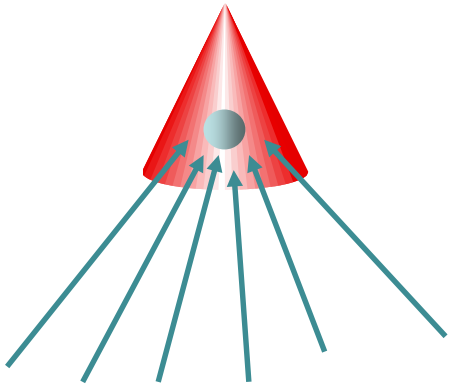
Nichtuniforme Speichersysteme: Kapazitäten können beliebig unterschiedlich sein

Vorgestellte Strategien:

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

Verteilte Hashtabelle

Probleme bei vielen Anfragen auf dasselbe Datum:



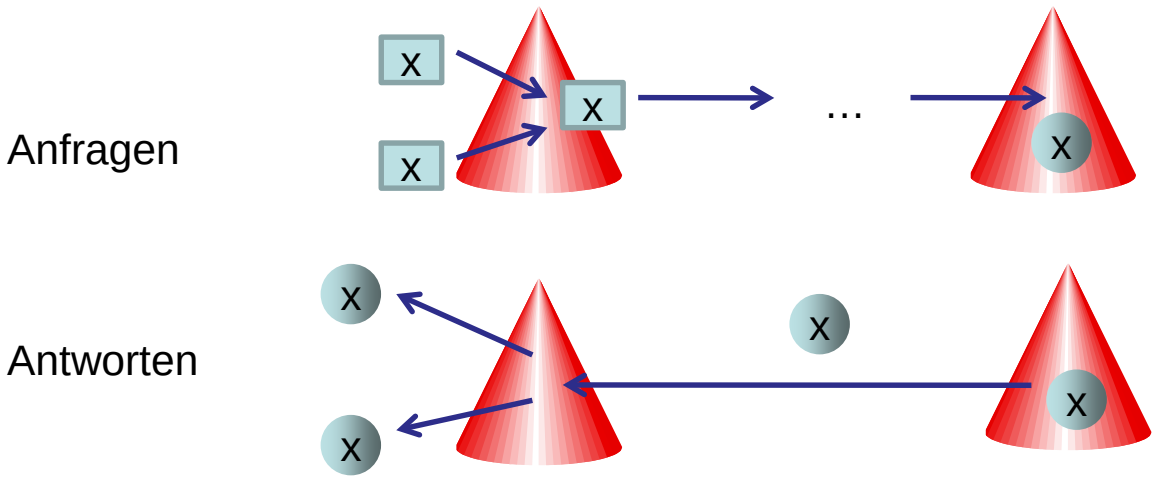
Prozess, der Datum speichert, wird überlastet.

Lösung: Combine & Split

Verteilte Hashtabelle

Combine & Split:

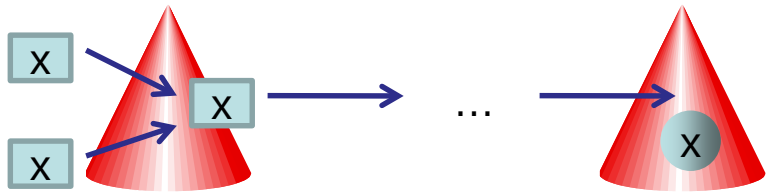
- Jeder Prozess v merkt sich alle Suchanfragen, die bei ihm eintreffen. Hat er für einen Schlüssel x bereits eine Suchanfrage weitergeleitet, dann hält er alle weiteren eingehenden Suchanfragen für x zurück (combine), beantwortet diese (split) sobald er eine Antwort zu x erhält und löscht diese dann 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 (siehe das Routing mit Combining in Kap. 2).

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.

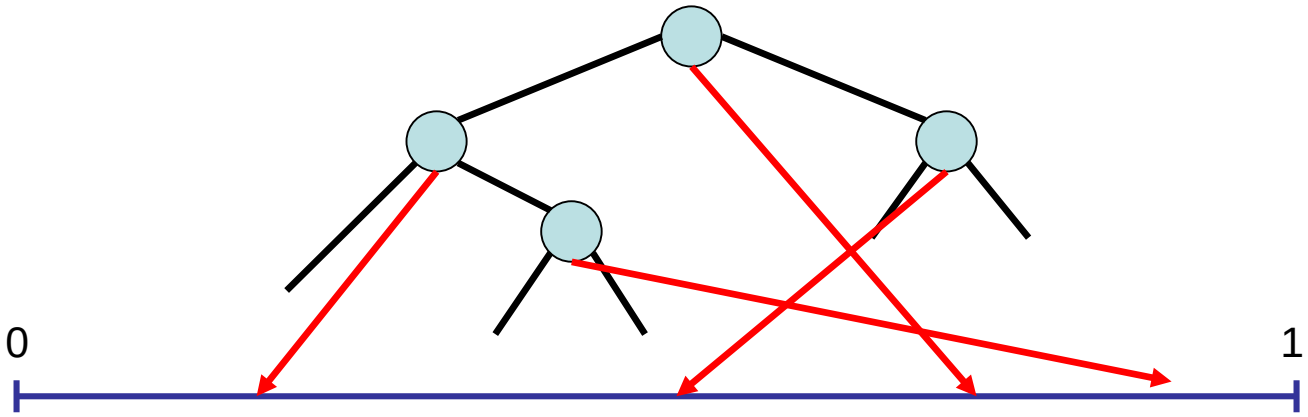
Verteilte Hashtabelle

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 Suchstruktur
- Verteilte Queue
- Verteilter Stack
- Verteilter Heap

Präfix Suche

Gegeben: Schlüsselmenge S.

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

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

Lösung: Trie Hashing

Trie

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

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

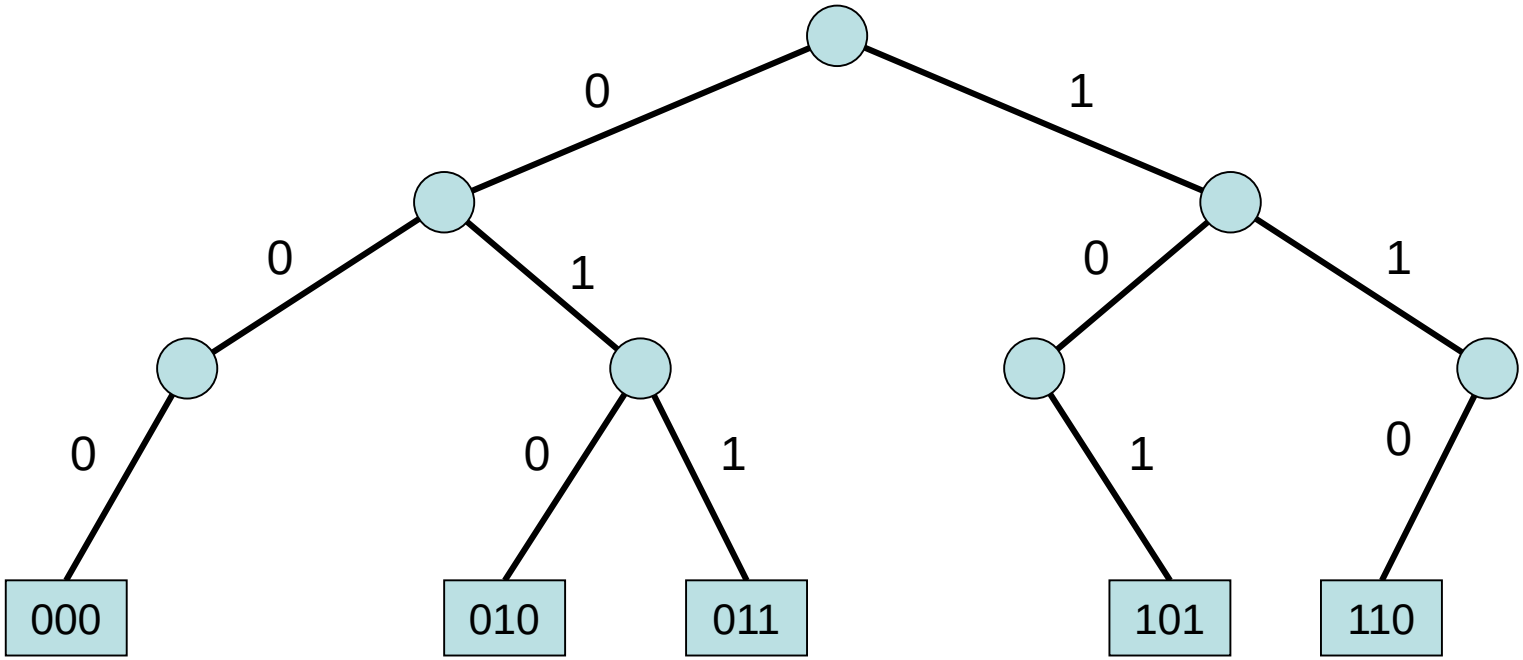
Hier: alle Schlüssel aus $\{0,1\}^W$

Beispiel:

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

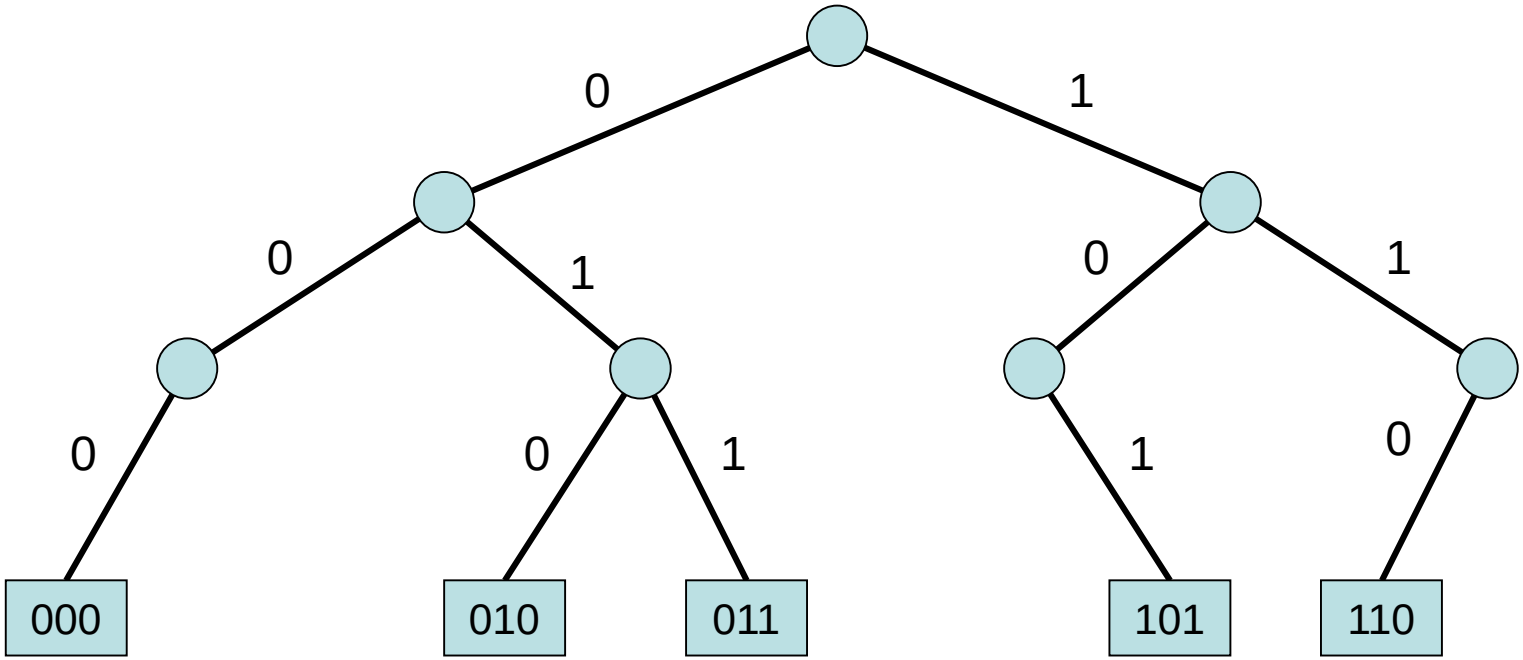
Trie

Trie aus Beispielzahlen:



Trie

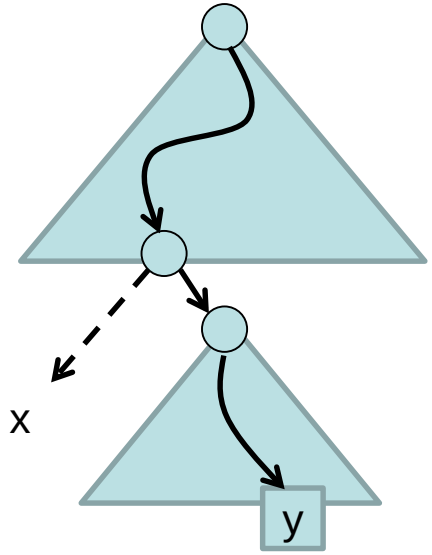
search(4) (4 entspricht 100):



Ausgabe: 5 (größter gem. Präfix)

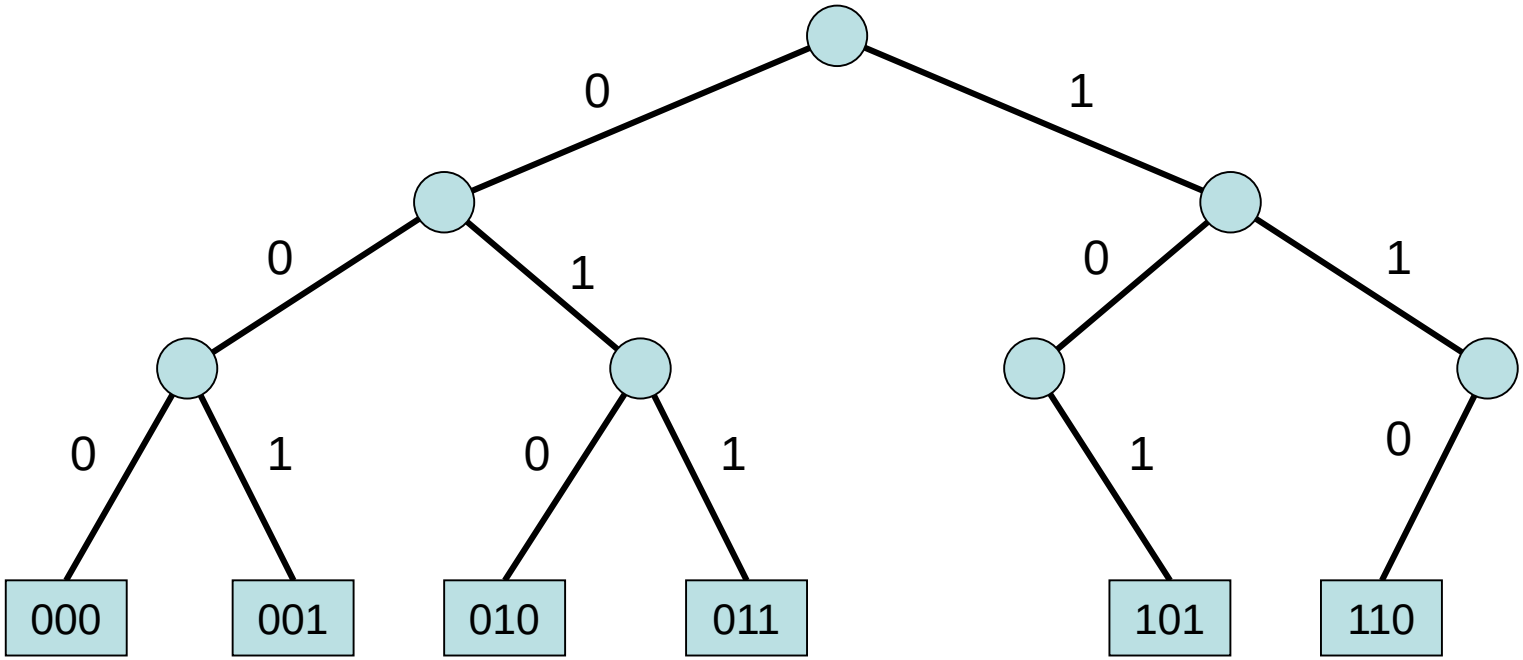
Trie

Im Allgemeinen: eine search(x) Anfrage folgt den Kanten im Trie solange deren Label einen Präfix von x bilden. Wenn keine Kante mehr zur Verfügung steht um dem Präfix von x zu folgen, wird die Anfrage zu einem beliebigen Blatt y weitergeleitet, da alle dieselbe Präfixübereinstimmung mit x haben.



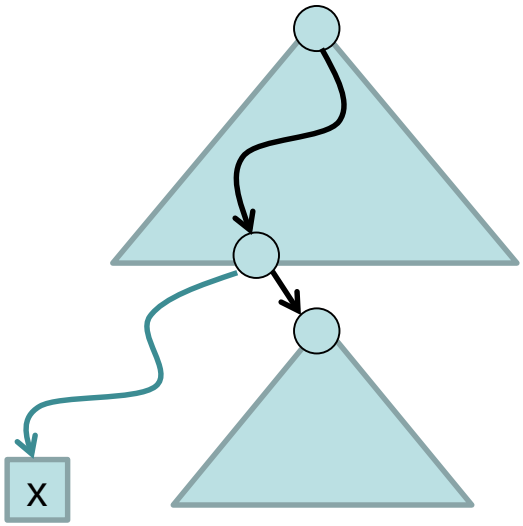
Trie

insert(1) (1 entspricht 001):



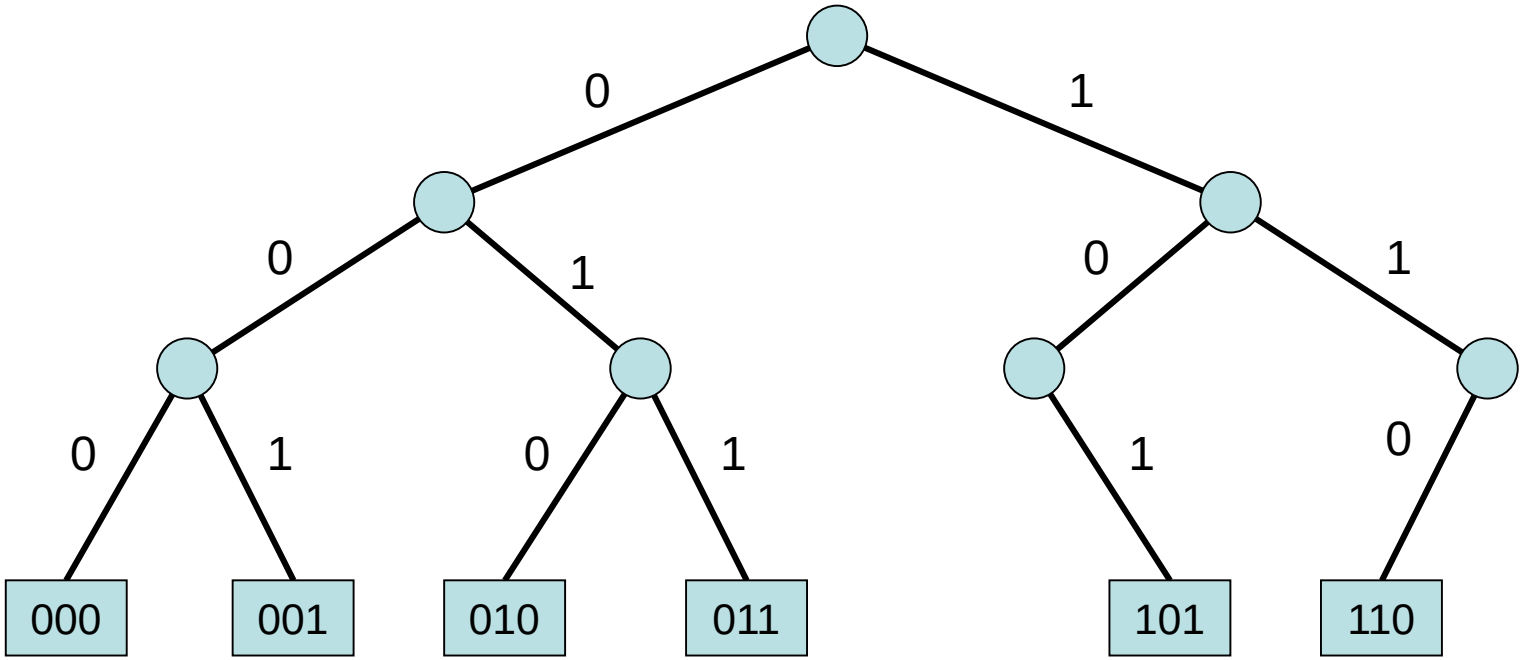
Trie

Im Allgemeinen: eine insert(x) Anfrage folgt den Kanten des Tries solange deren Label einen Präfix von x bilden. Wenn keine Kante mehr zur Verfügung steht um dem Präfix von x zu folgen, wird ein neuer Pfad (der Länge der übrig gebliebenen Bits in x) erzeugt, der zu einem neuen Blatt x führt.



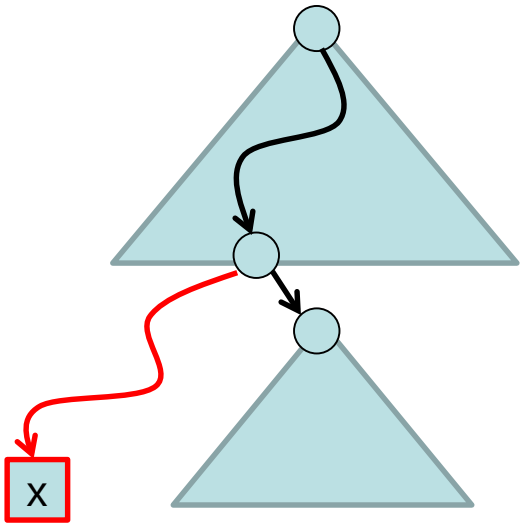
Trie

delete(5):



Trie

Im Allgemeinen: eine delete(x) Anfrage folgt den Kanten des Tries bis zum Blatt x. Falls x nicht (als Blatt) existiert, terminiert die delete Operation. Sonst wird x und die Kette der Knoten von x bis zum ersten Knoten mit mindestens zwei Kindern gelöscht.



Patricia Trie

Probleme:

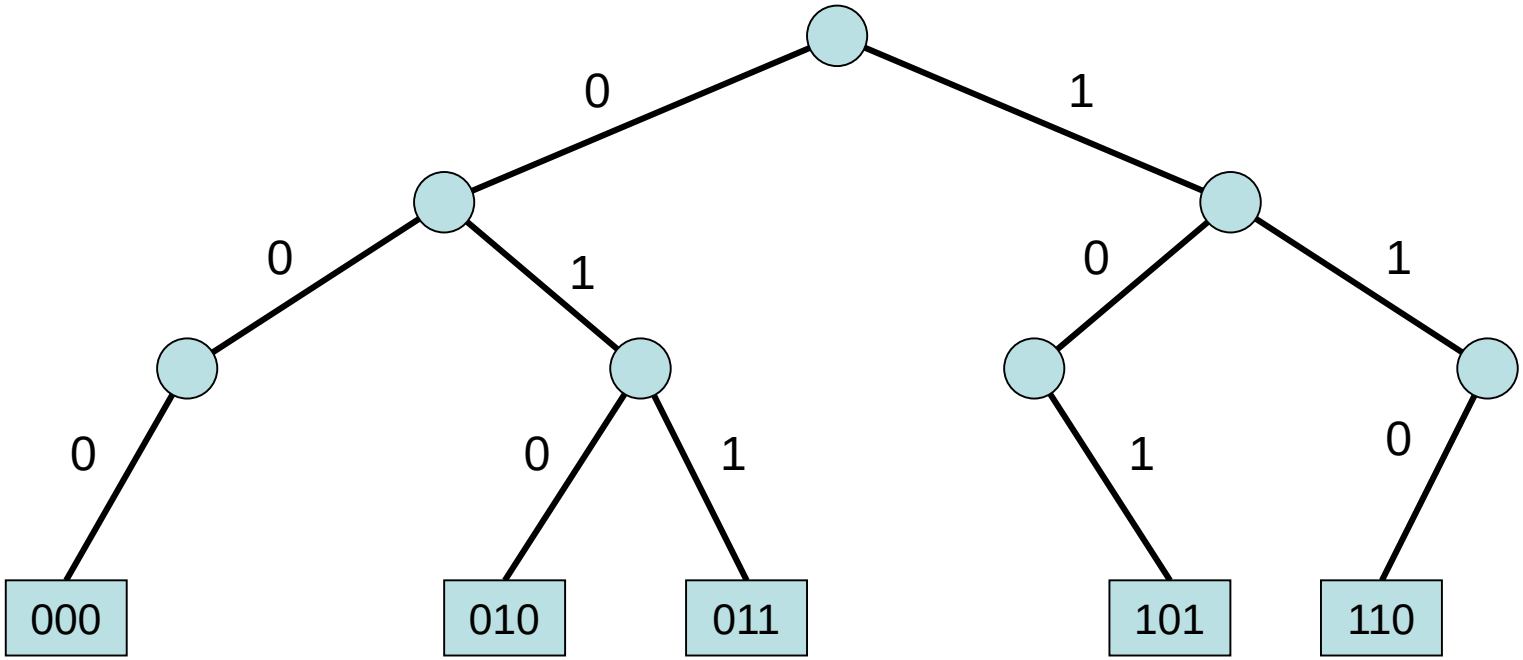
- Präfixsuche im Trie mit Schlüsseln aus $x \in \{0,1\}^W$ kann $\Theta(W)$ Zeit dauern.
- Insert und delete benötigen bis zu $\Theta(W)$ Restrukturierungen

Verbesserung: verwende Patricia Tries

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

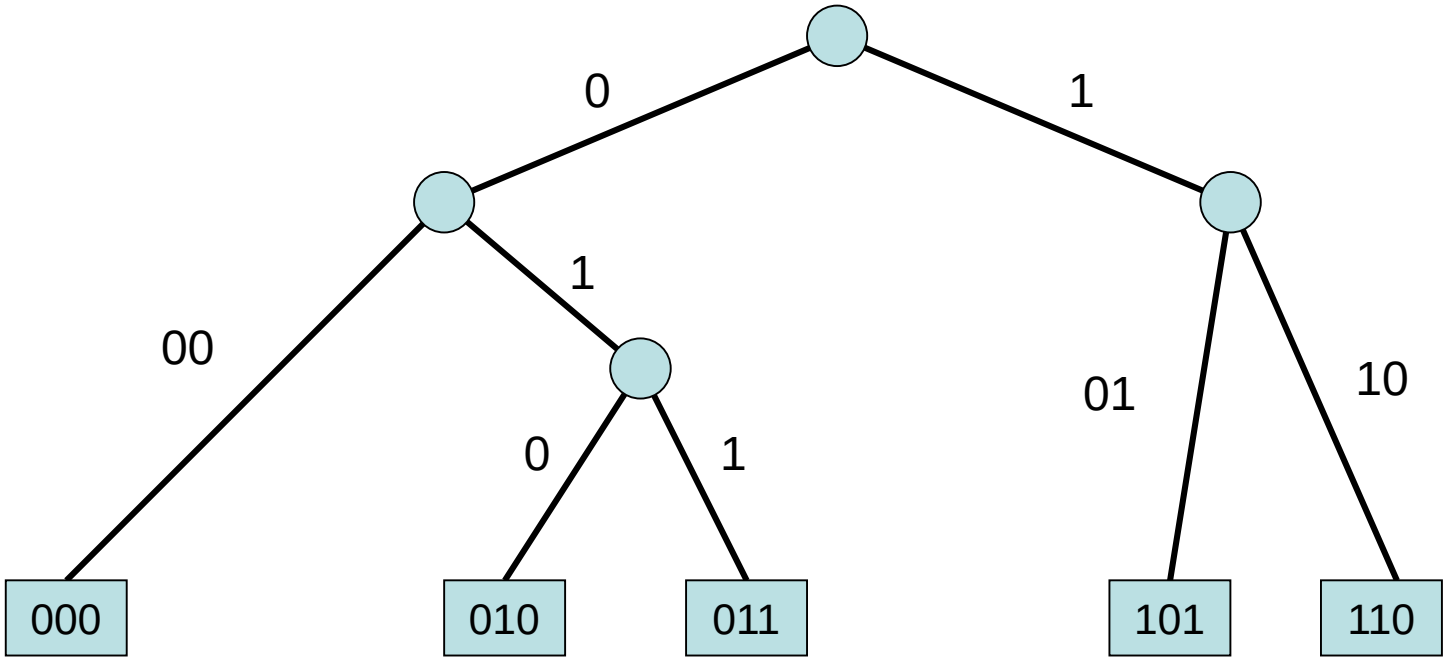
Trie

Beispiel 1:



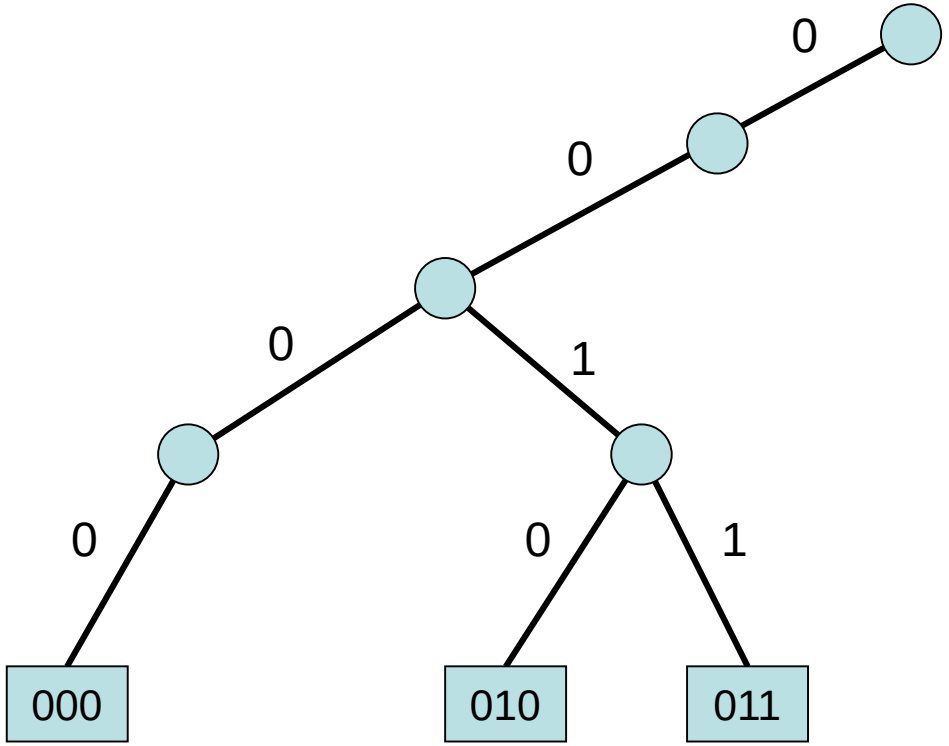
Patricia Trie

Beispiel 1:



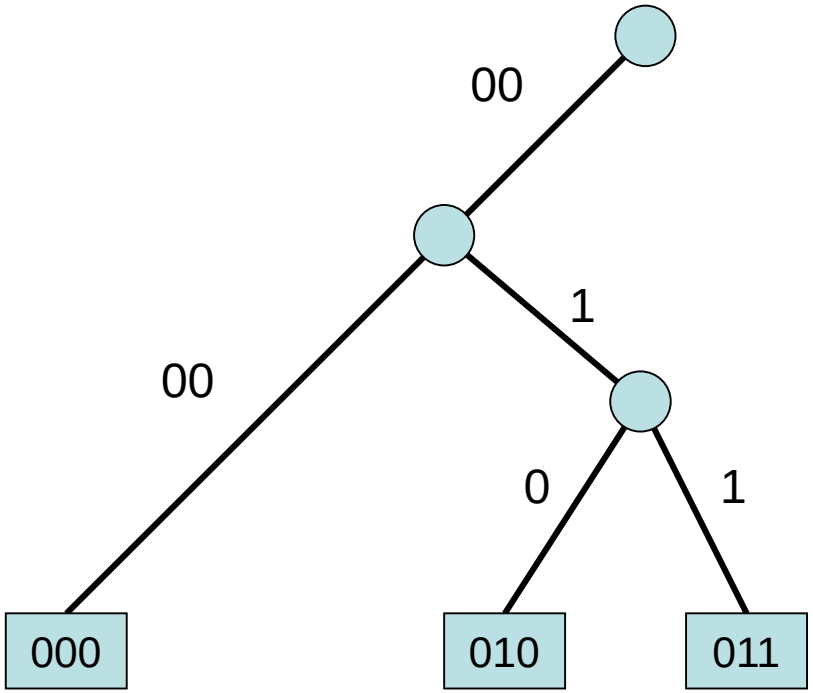
Trie

Beispiel 2:



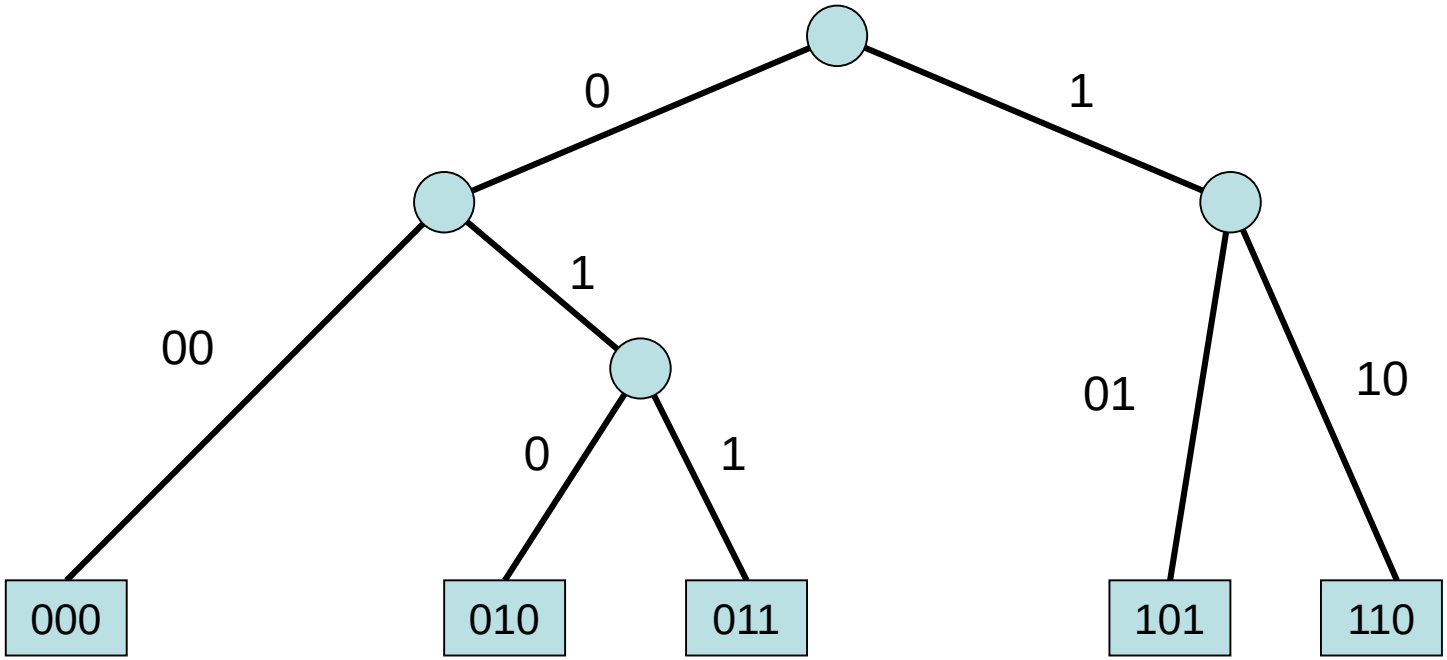
Patricia Trie

Beispiel 2:



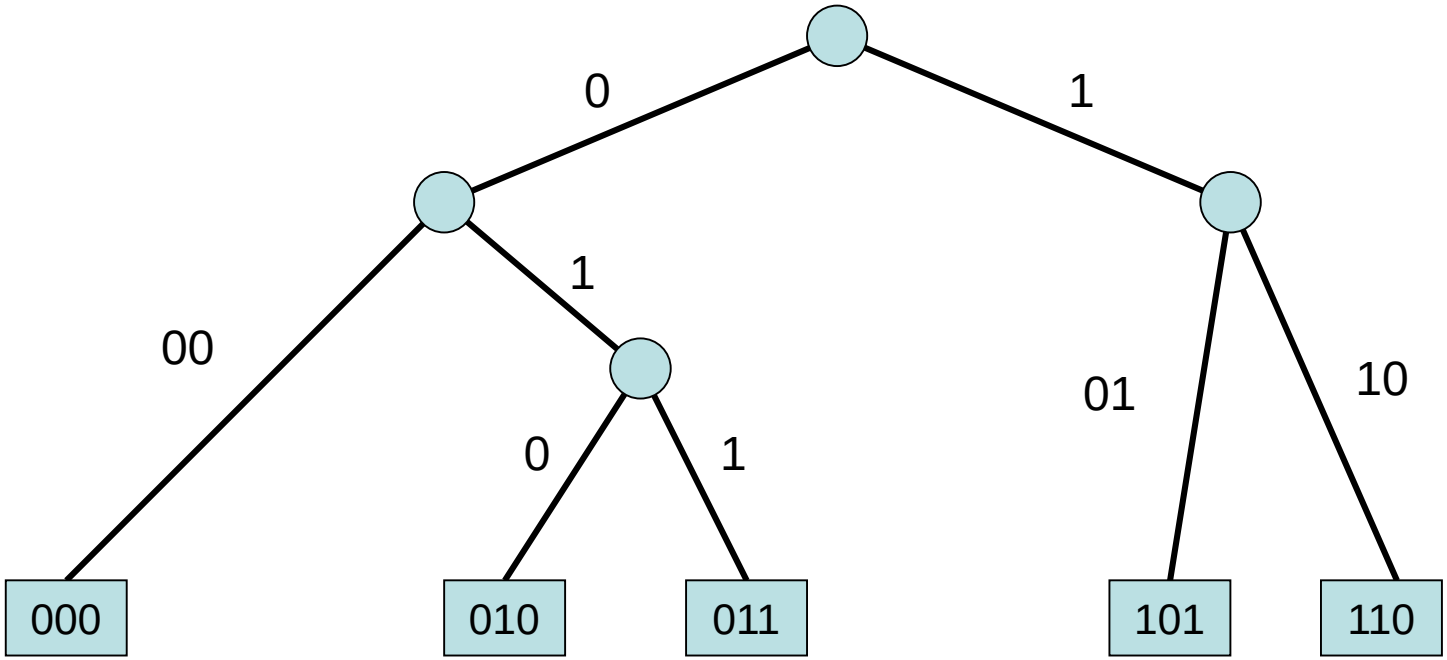
Patricia Trie

Jeder Baumknoten (bis auf Wurzel) hat Grad 2.



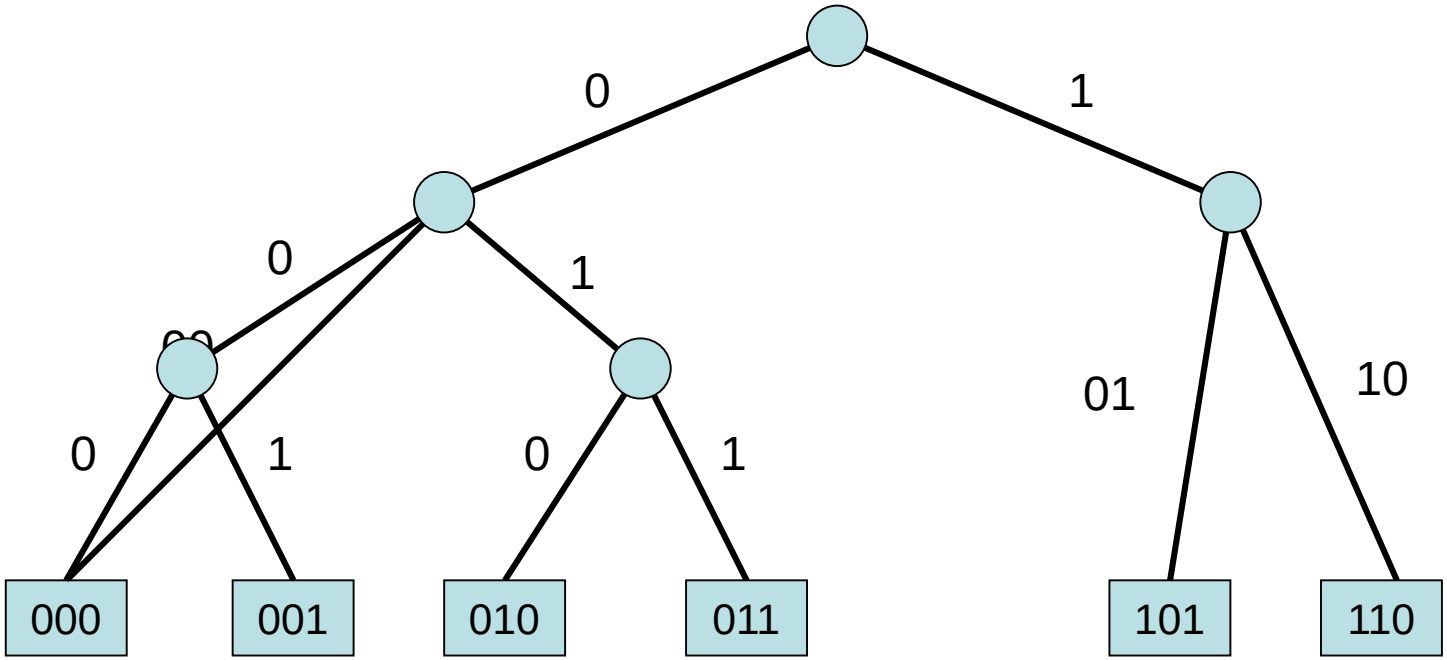
Patricia Trie

Search(4): wie im Trie



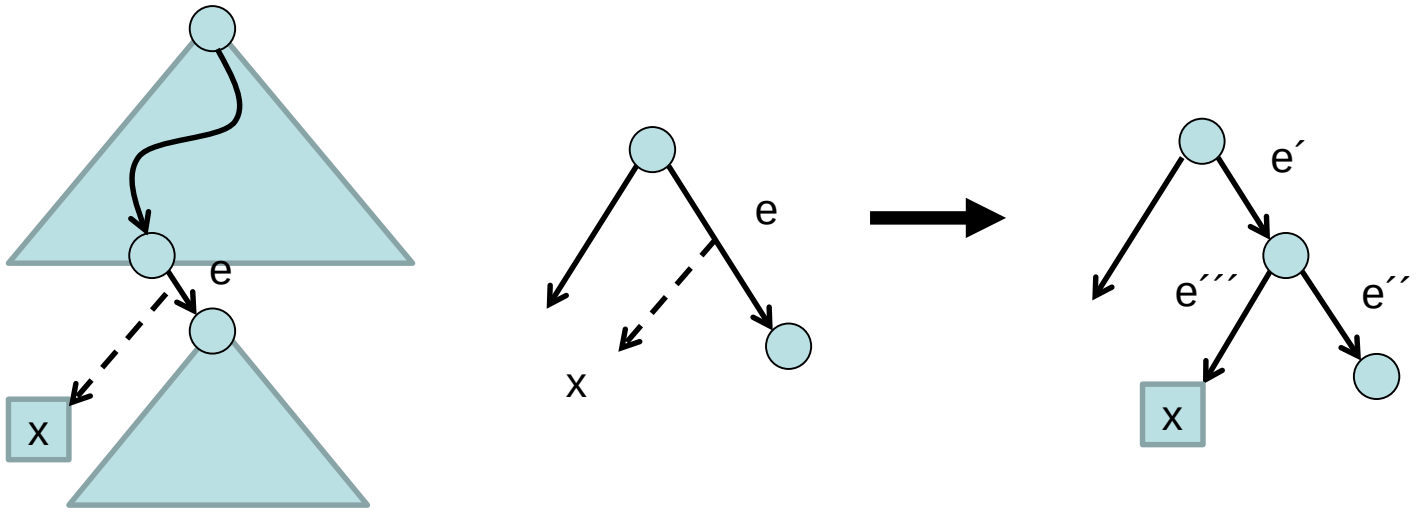
Patricia Trie

Insert(1): wie im Trie, nur komprimiert



Patricia Trie

Allgemein:

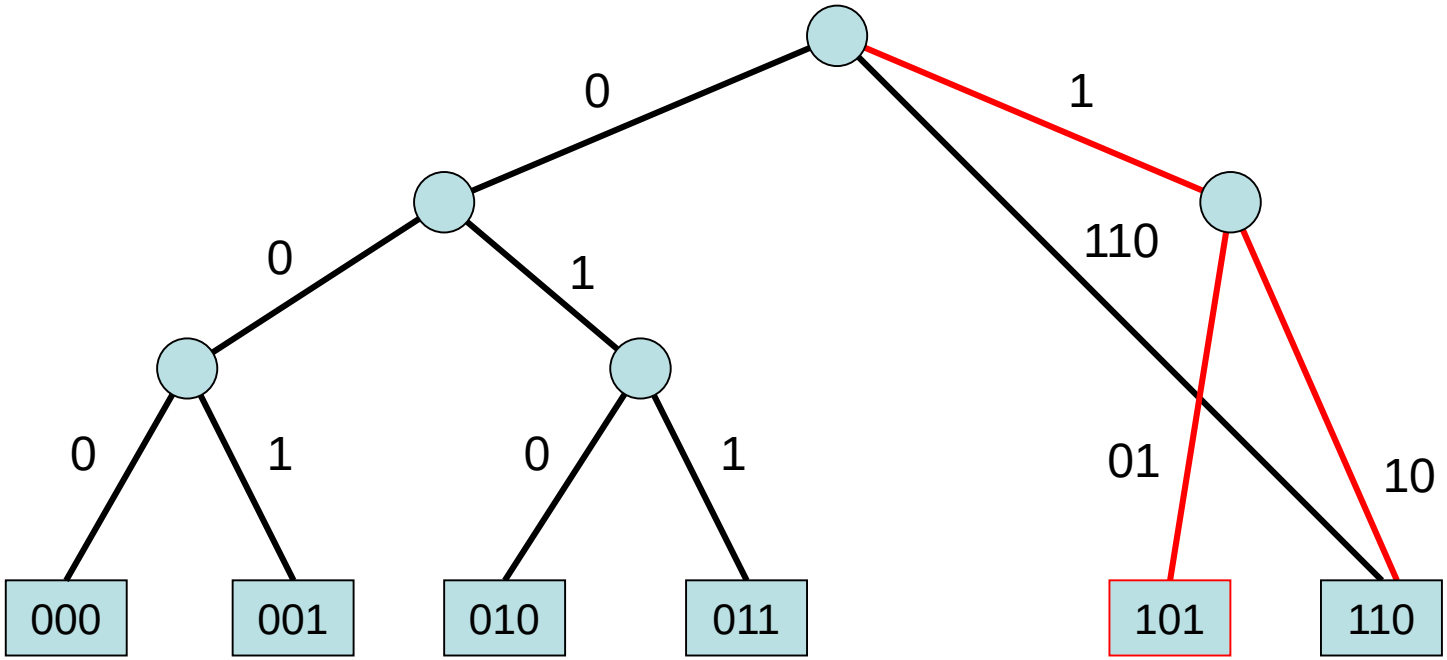


Beispiel: l(e)=10010, x=…10110100

Dann sind l(e´)=10, l(e´´)=010 und l(e´´´)=110100.

Patricia Trie

Delete(5): wie im Trie, nur komprimiert



Patricia Trie

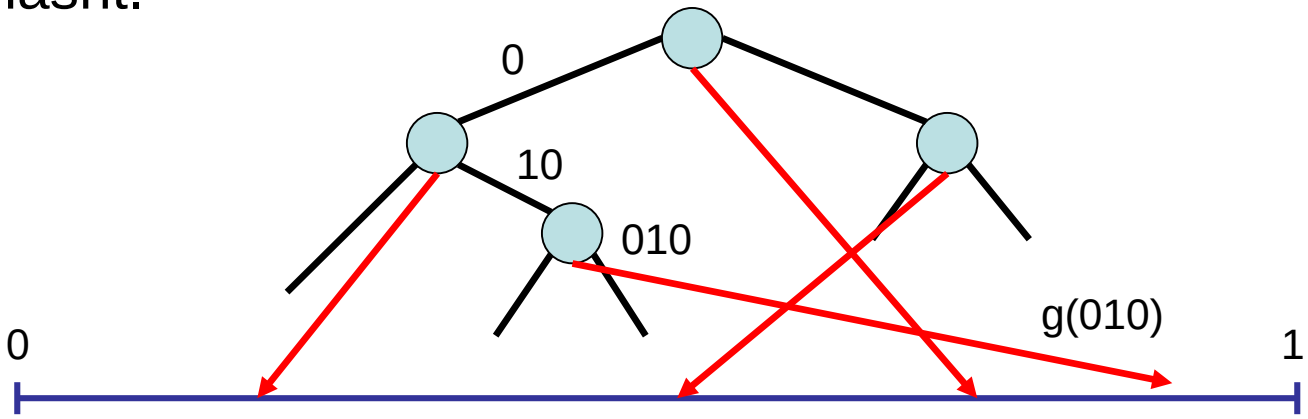
- Search, insert und delete wie im einfachen binären Suchbaum, nur dass wir Labels an den Kanten haben.
- Suchzeit immer noch O(W), aber Restrukturierungsaufwand nur noch O(1)

Idee: Um Suchzeit zu verringern, hashen wir Patricia Trie auf [0,1) (damit konsistentes Hashing oder SHARE anwendbar ist).

Patricia Trie

Hashing auf [0,1):

- Knotenlabel: Konkatenation der Kantenlabel von Wurzel
- Jeder Knoten wird gemäß seines Knotenlabels in [0,1) gehasht.

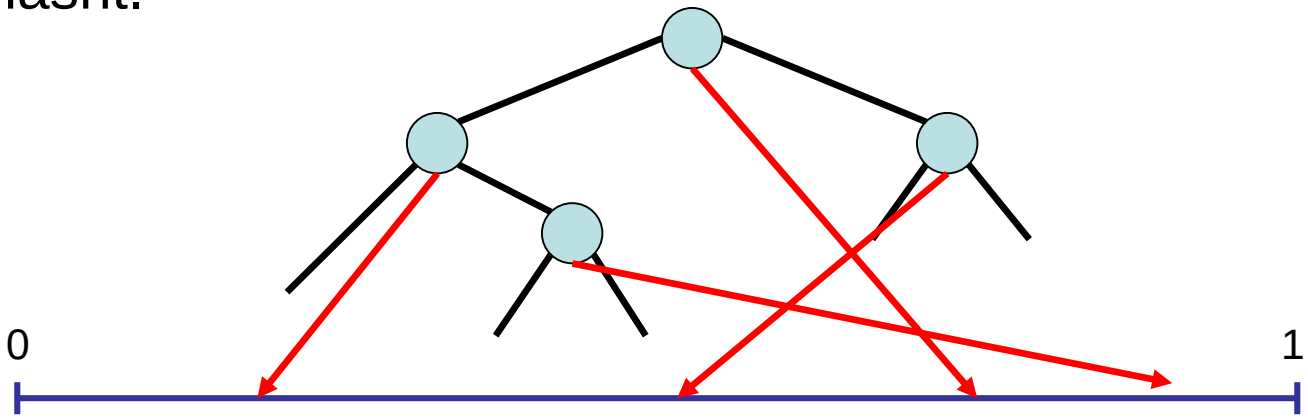


- Dann kann auf jeden Patricia Knoten mit einer DHT-Anfrage zugegriffen werden, falls sein Label bekannt.

Patricia Trie

Hashing auf [0,1):

- Knotenlabel: Konkatenation der Kantenlabel von Wurzel
- Jeder Knoten wird gemäß seines Knotenlabels in [0,1) gehasht.

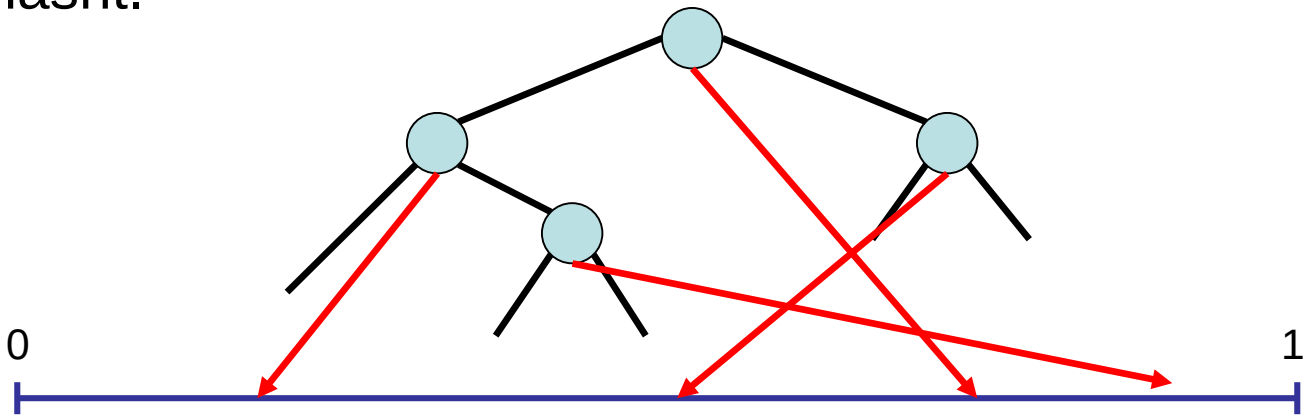


Problem: Binäre Suche über Knotenlabel wäre dann denkbar für Zeitreduktion O(W) → O(log W), aber noch nicht machbar.

Patricia Trie

Hashing auf [0,1):

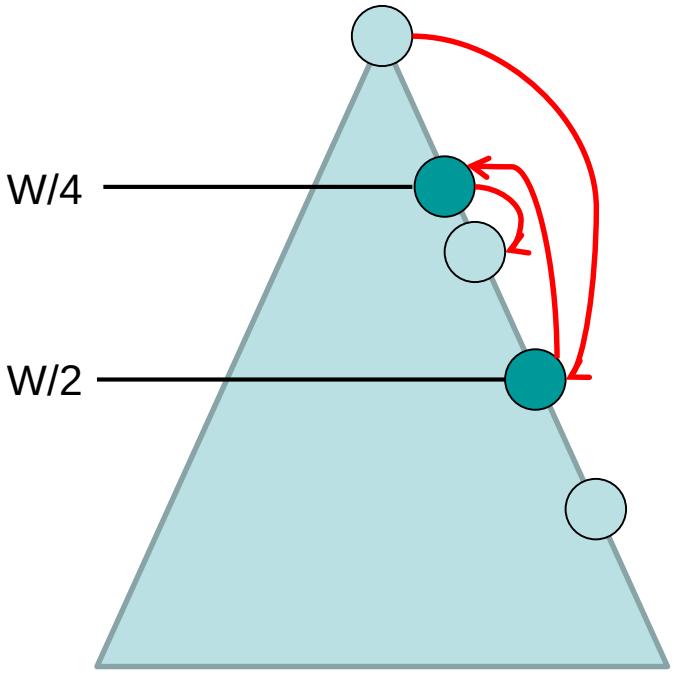
- Knotenlabel: Konkatenation der Kantenlabel von Wurzel
- Jeder Knoten wird gemäß seines Knotenlabels in [0,1) gehasht.



Lösung: Füge geeignete Stützknoten ein (msd-Knoten).

Trie Hashing

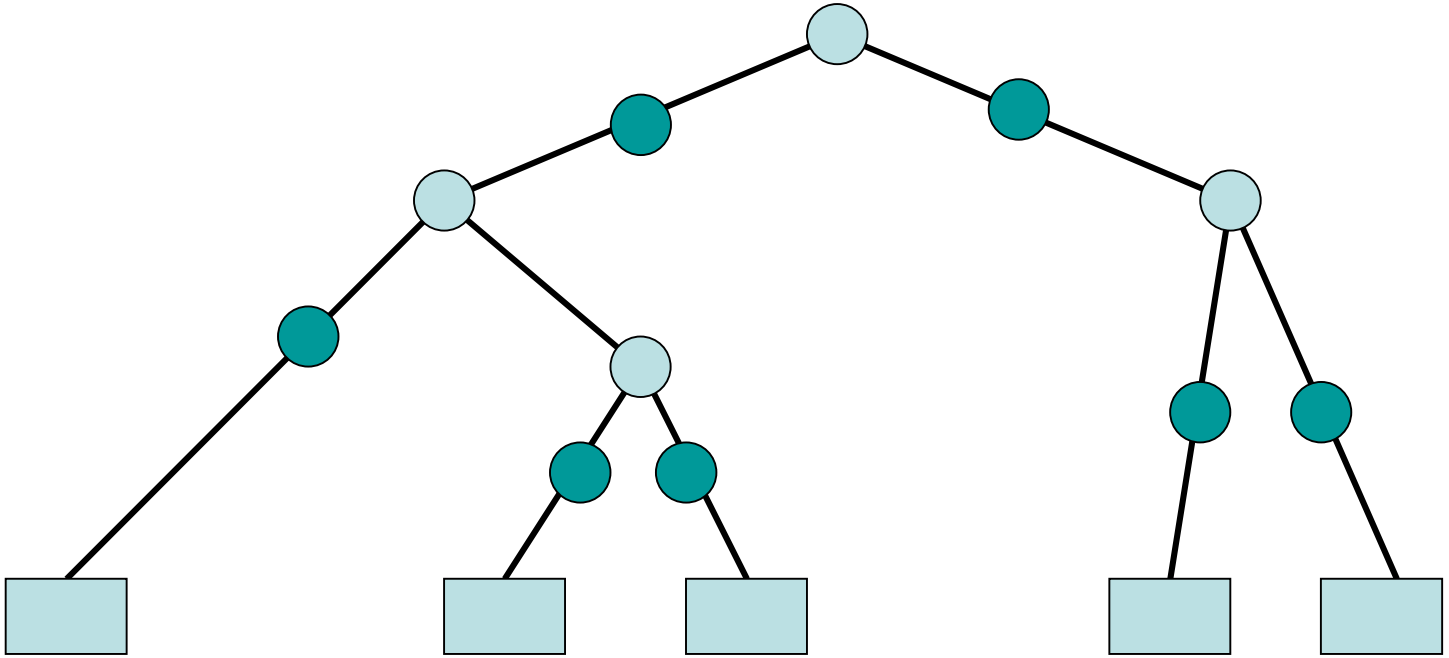
Lösung: Füge geeignete Stützknoten ein (msd-Knoten).



Idee: Labellänge des msd-Knotens ist gleich der Länge, die zuerst zwischen zwei Patricia Trie Knoten bei binärer Suche besucht würde.

Trie Hashing

msd-Knoten:



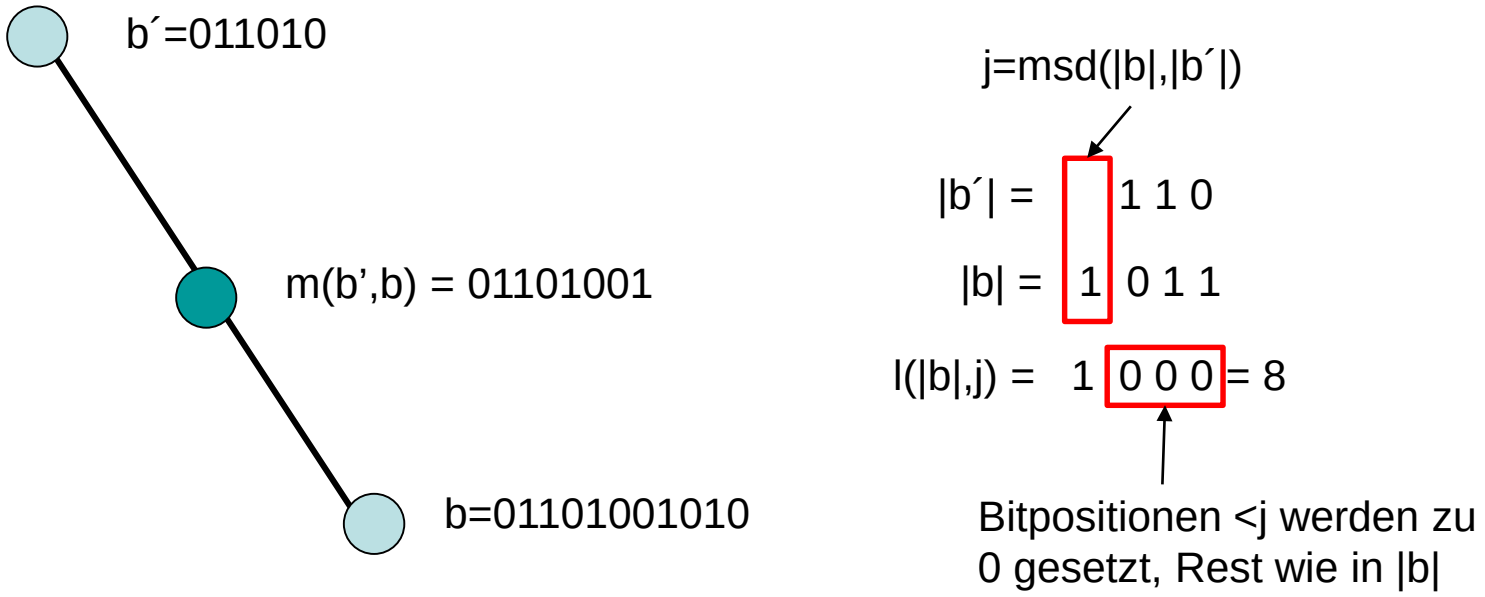
Trie Hashing

- Für eine Bitfolge x sei $|x|$ die Länge von x.
- Für einen Baumknoten v sei $b(v)$ die konkatenierte Bitfolge der Kanten des eindeutigen Weges von der Wurzel zu v (d.h. der gem. Präfix aller Elemente unter v).
- $msd(f,f')$ für zwei Bitfolgen $0^* \circ f$ und $0^* \circ f'$: höchstes Bit, in dem sich $0^* \circ f$ und $0^* \circ f'$ unterscheiden ($0^*=0000\ldots$).
- Betrachte eine Bitfolge b mit Binärdarstellung von $|b|$ gleich $(x_k,\ldots,x_0)$. Sei b' ein Präfix von b.
  Die msd-Folge $m(b',b)$ von b' und b ist das Präfix von b der Länge $\sum_{i=j}^{k} x_i\, 2^i$ mit $j=msd(|b|,|b'|)$.

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

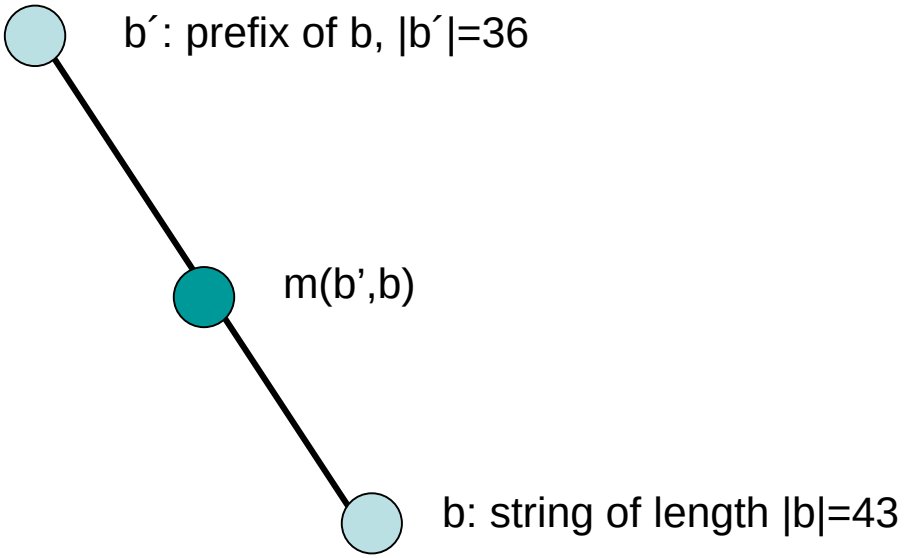
Trie Hashing

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



Trie Hashing

Weiteres Beispiel:



j=msd(|b|,|b´|)

|b´| = 1 0 0 1 0 0

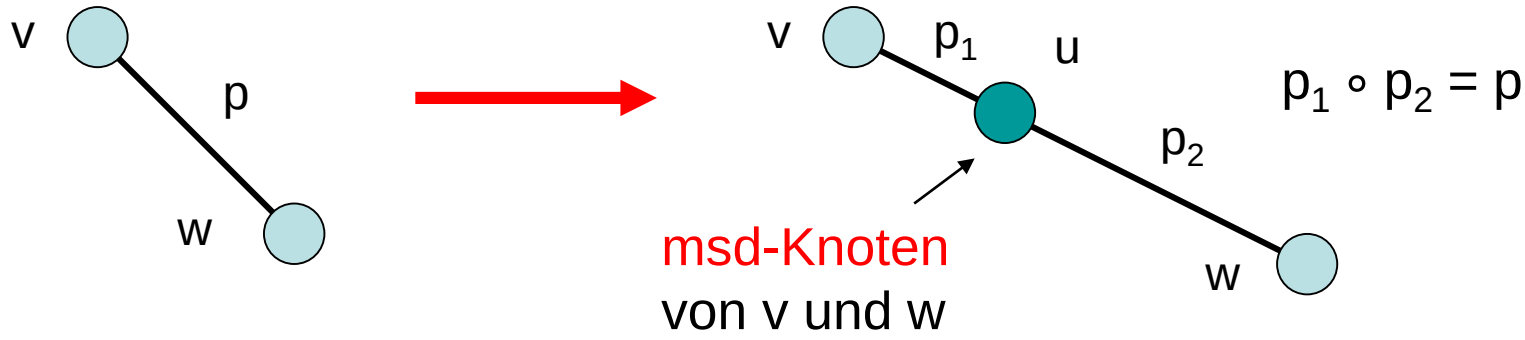
|b| = 1 0 1 0 1 1

l(|b|,j) = 1 0 1 0 0 0 = 40

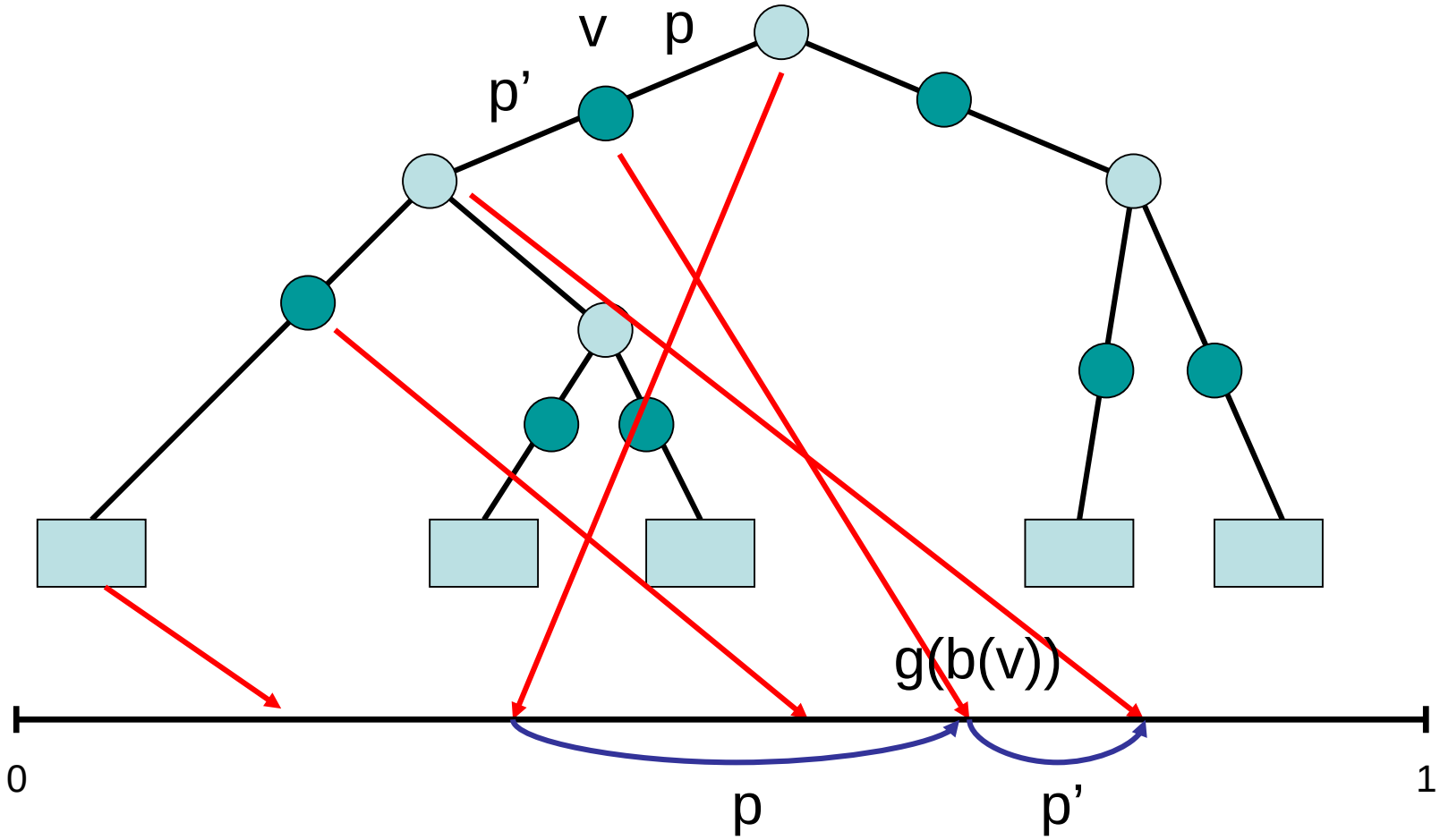
Bitpositionen <j werden zu 0 gesetzt, Rest wie in |b|

Trie Hashing

Strategie: Wir ersetzen jede Kante e={v,w} im Patricia Trie durch zwei Kanten {v,u} und {u,w} mit b(u)=m(b(v),b(w)) und bilden resultierenden Patricia Trie auf [0,1) ab.



Trie Hashing



Trie Hashing

Datenstruktur:

Jeder Hasheintrag zu einem Baumknoten v speichert:

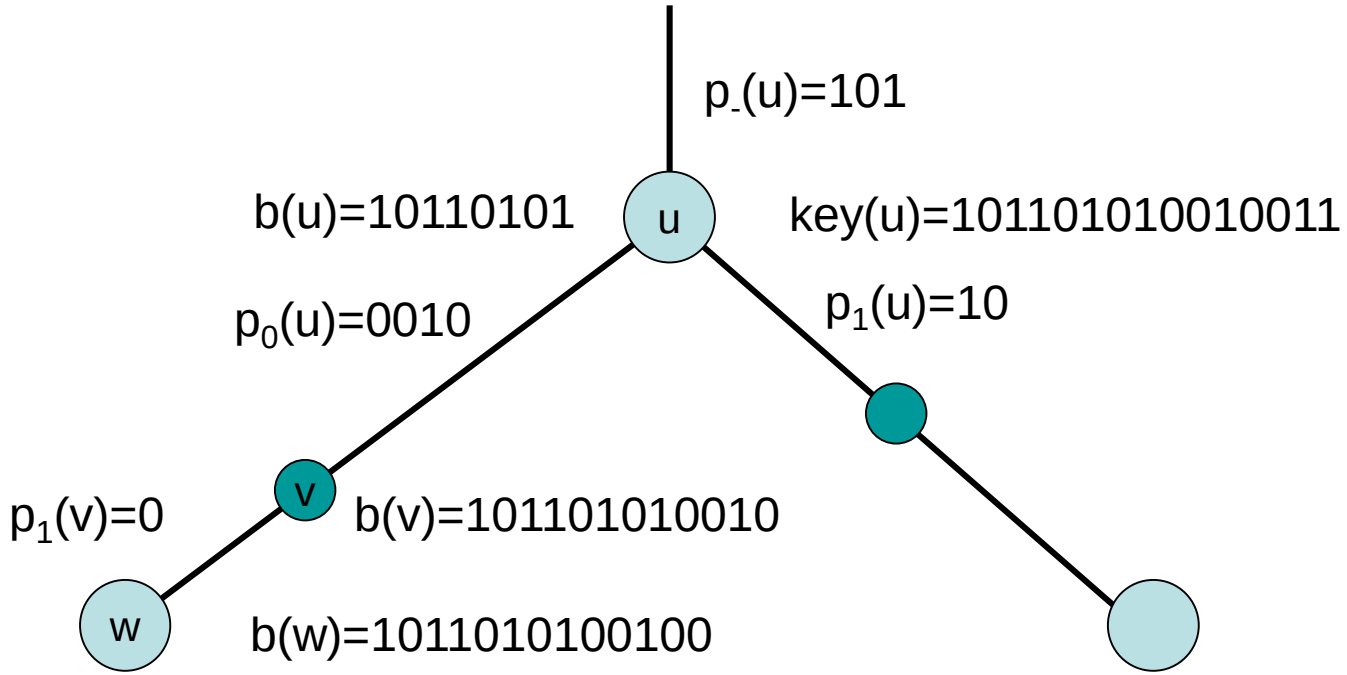
1. Bitfolge $b(v)$ zu v
2. Schlüssel $key(v)$ eines Elements e, falls v Knoten im Original Patricia Trie ist (für Suchergebnis)
3. Bitfolgen $p_x(v)$ der Kanten bzgl. nächstem Bit $x \in \{0,1\}$ zu Kindern von v (für Suche)
4. Bitfolge $p_{-}(v)$ zum Vaterknoten (für Restrukturierung)

Jeder Hasheintrag zu einem Element e speichert:

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

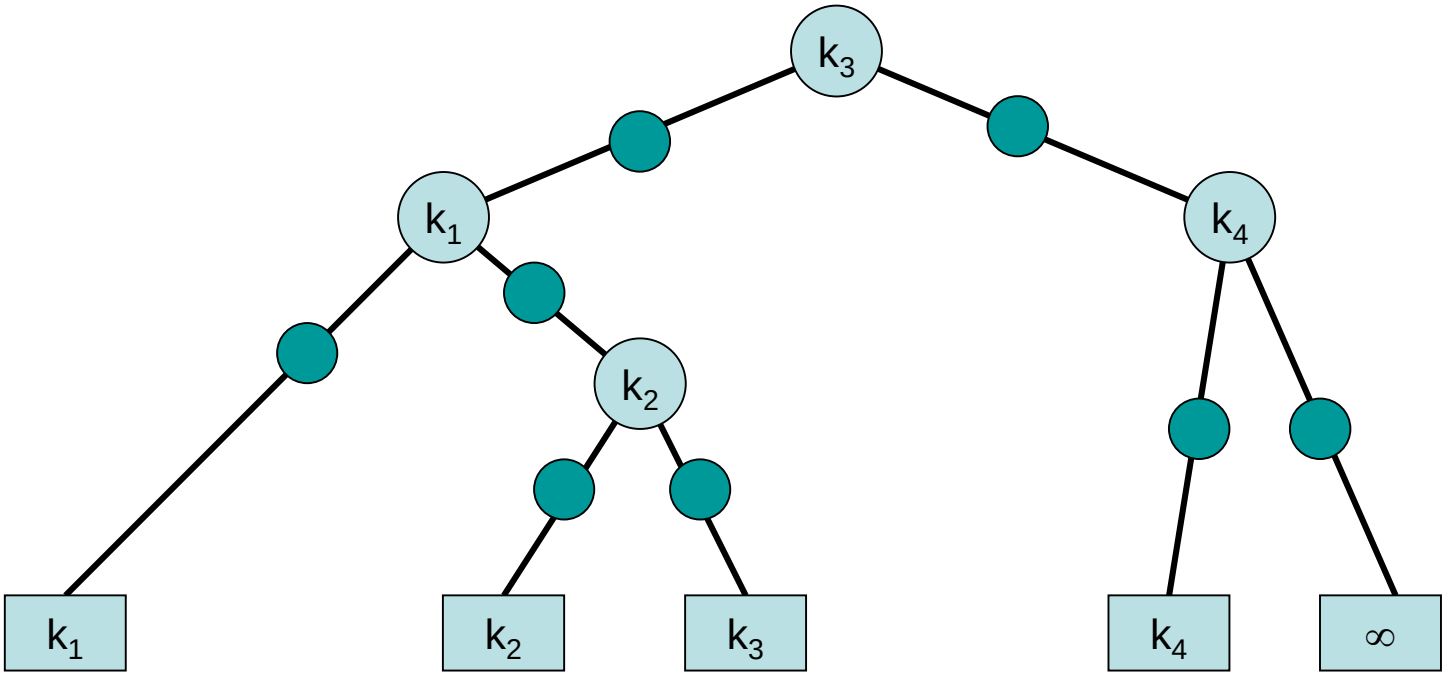
Trie Hashing

Beispiel:



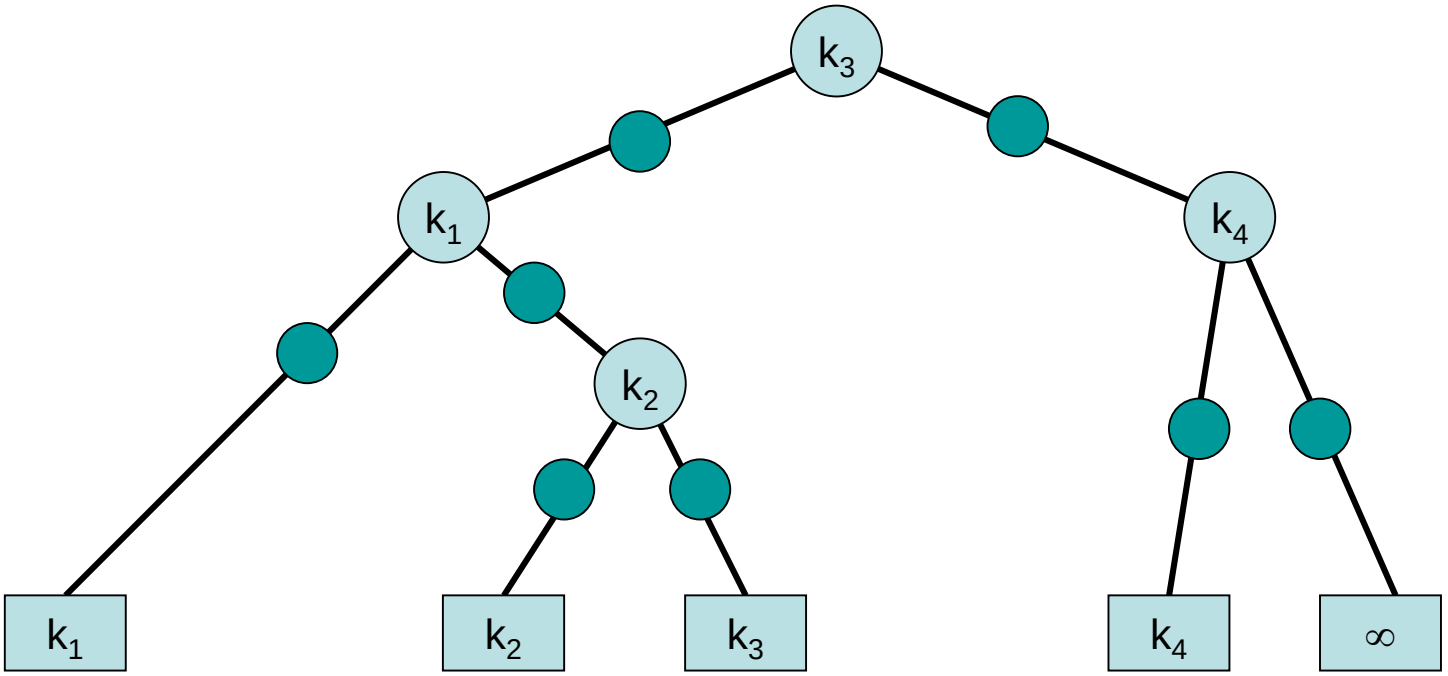
Trie Hashing

Forderung: jeder Schlüssel ist in exakt einem Baumknoten gespeichert (möglich durch ∞).



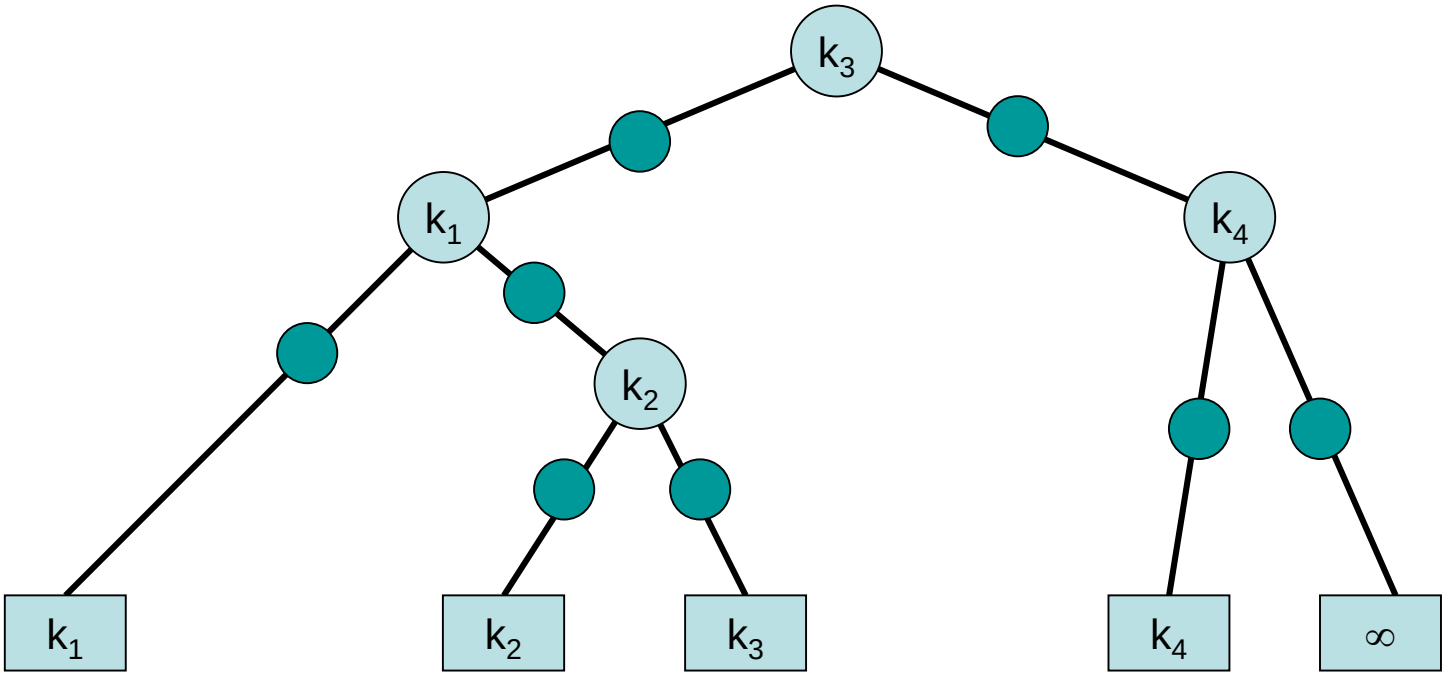
Trie Hashing

Invariante: b(v) ist ein Präfix des Schlüssels, der in Knoten v gespeichert ist.



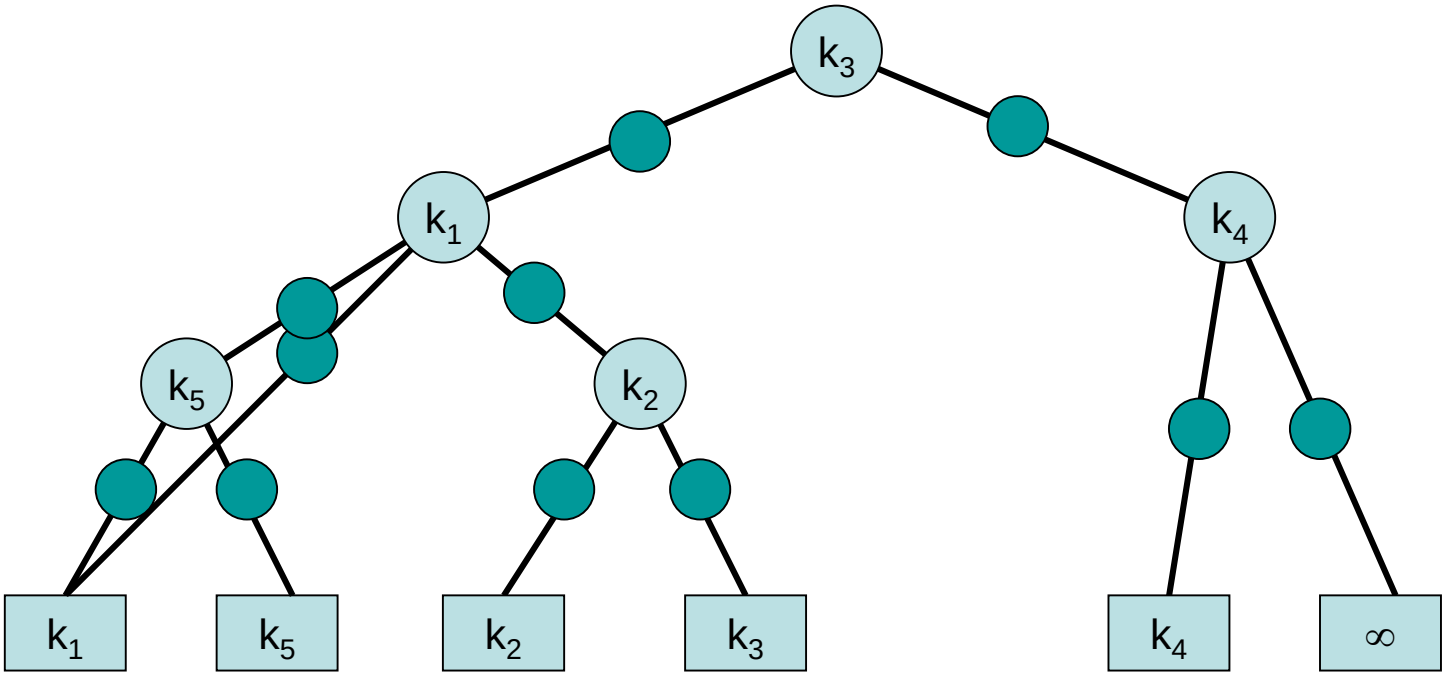
Trie Hashing

Wir vergessen zunächst die Suchproblematik und illustrieren insert und delete.



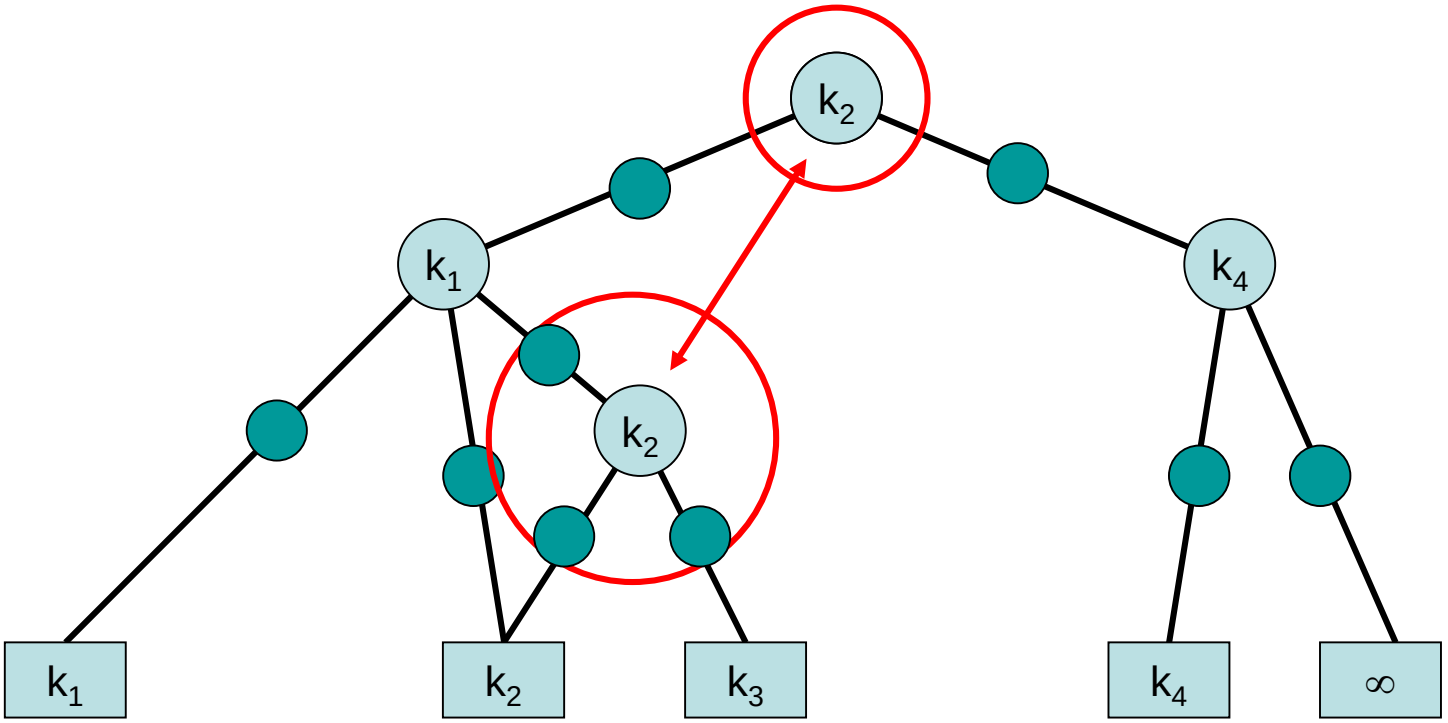
Trie Hashing

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



Trie Hashing

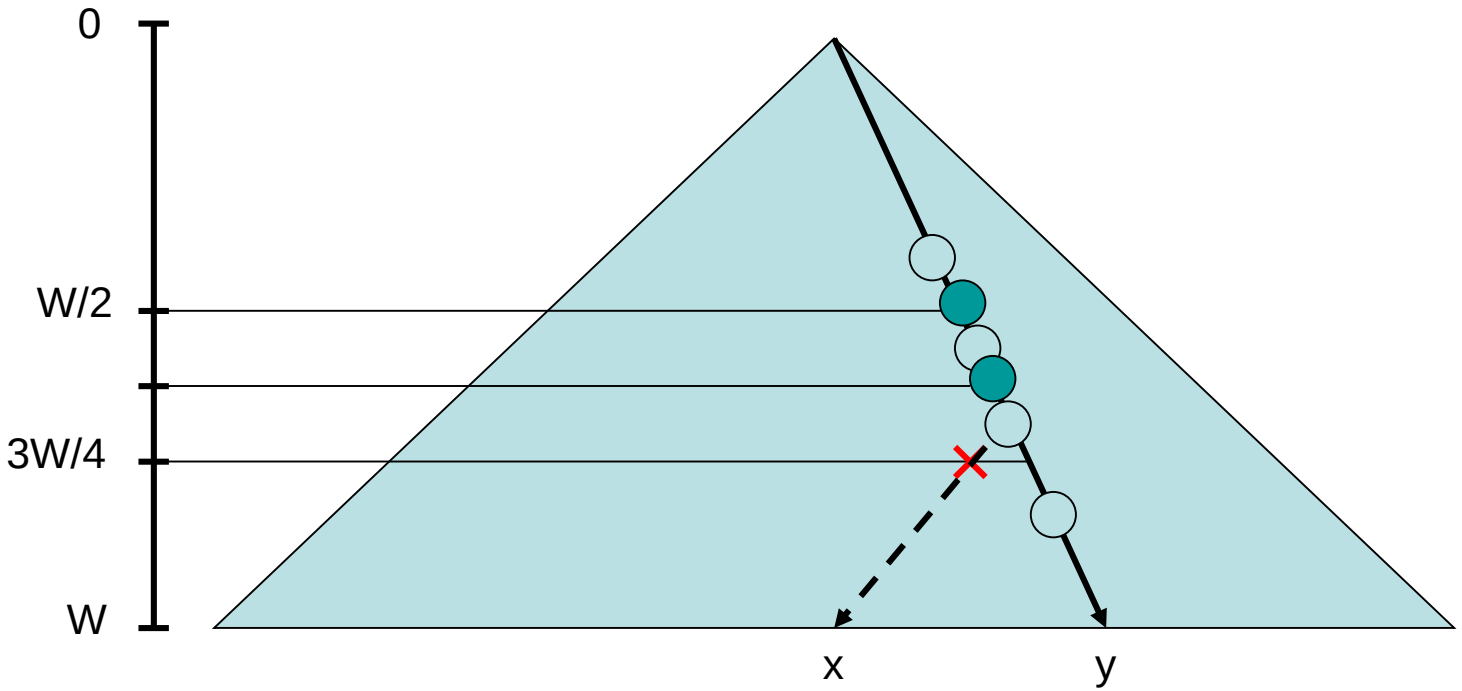
Delete(k_3): wie im binären Suchbaum



Trie Hashing

Search(x): (W Zweierpotenz)

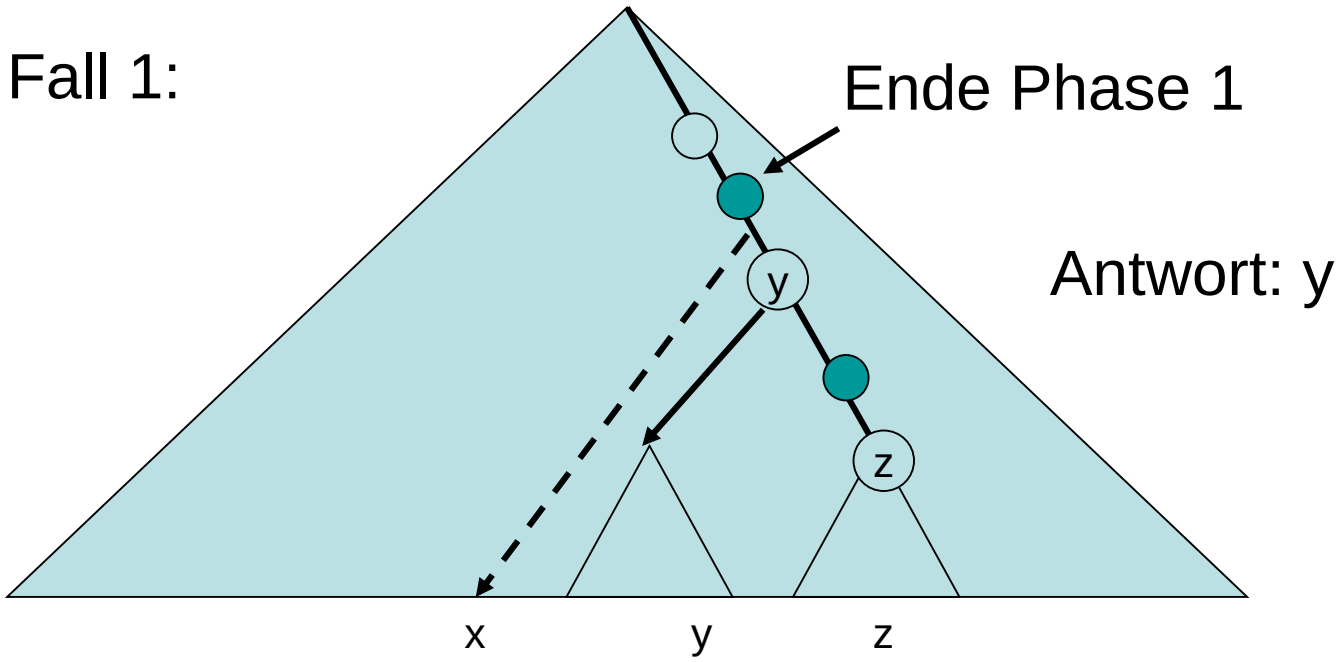
Phase 1: binäre Suche über msd-Knoten



Trie Hashing

Search(x): (W Zweierpotenz)

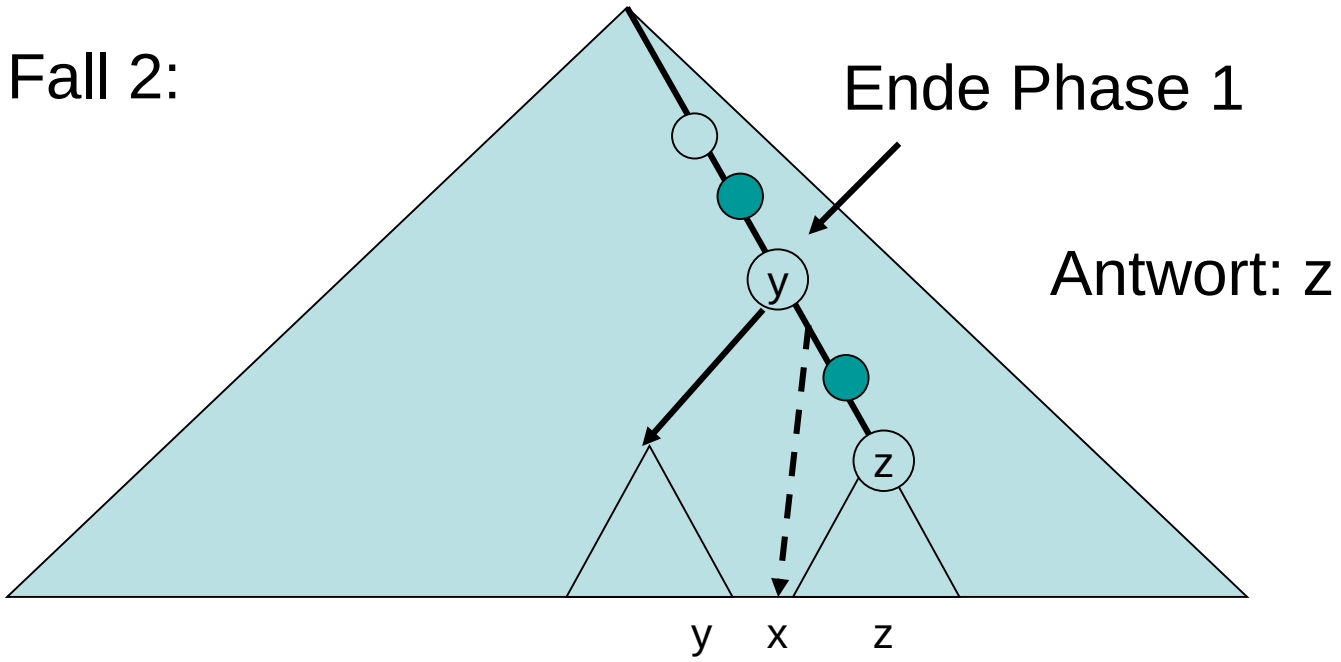
Phase 2: lies Schlüssel aus Baumknoten



Trie Hashing

Search(x): (W Zweierpotenz)

Phase 2: lies Schlüssel aus Baumknoten



Trie Hashing

- $x \in \{0,1\}^W$ habe Darstellung $(x_1,\ldots,x_W)$
- Hashfunktion: $h:U \rightarrow [0,1)$

T[h(x)] in DHT: lookup(x) Operation

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

Trie Hashing

search(x): (Fortsetzung)

// Phase 2: Laufe zu Knoten mit größtem Präfix

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

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

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

else

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

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

if v ist msd-Knoten then

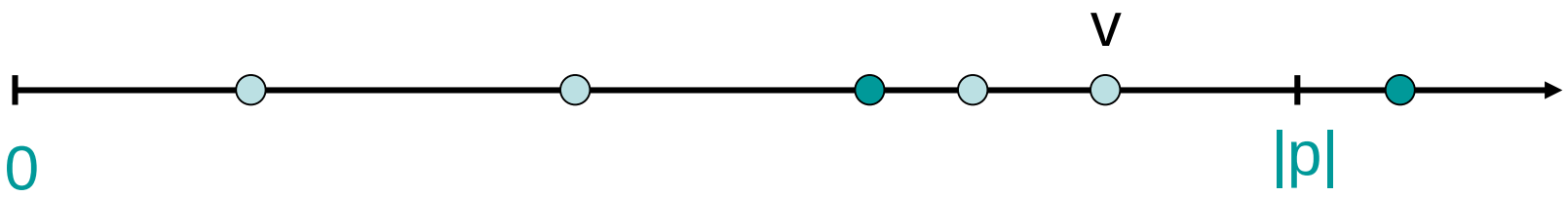
$v := T[h(b(v) \circ p)]$ für Bitfolge p aus v

return key(v)

Trie Hashing

Korrektheit von Phase 1:

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

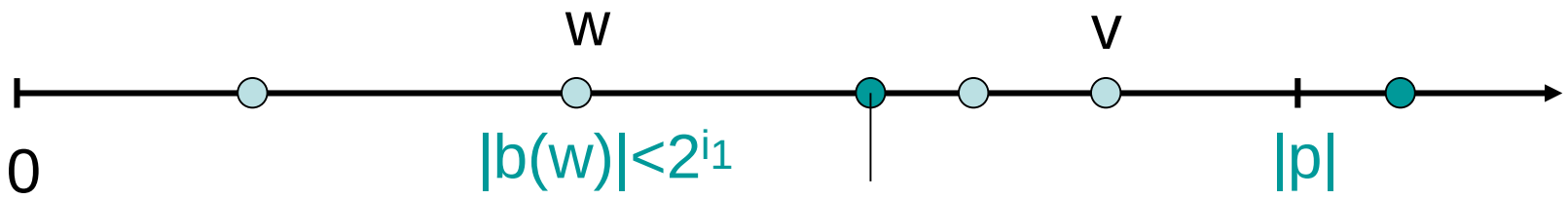


Binärdarstellung von $|b(v)|$ habe Einsen an den Positionen $i_1,i_2,\ldots$ (i_1 maximal)

Trie Hashing

Korrektheit von Phase 1:

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

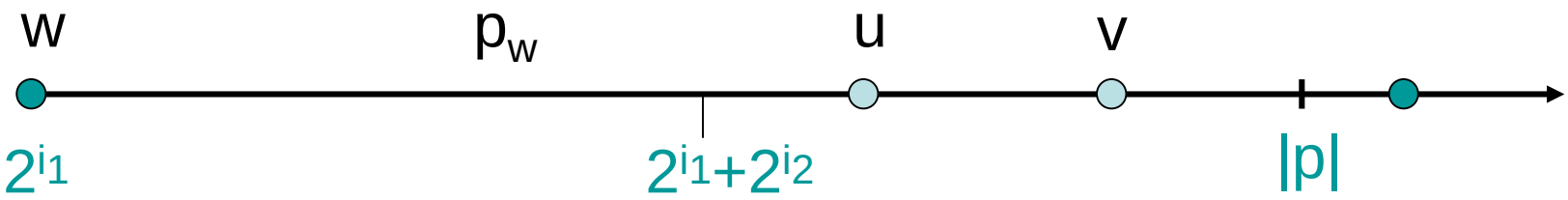


msd-Knoten muss bei 2^{i1} existieren,
wird bei binärer Suche gefunden

Trie Hashing

Korrektheit von Phase 1:

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

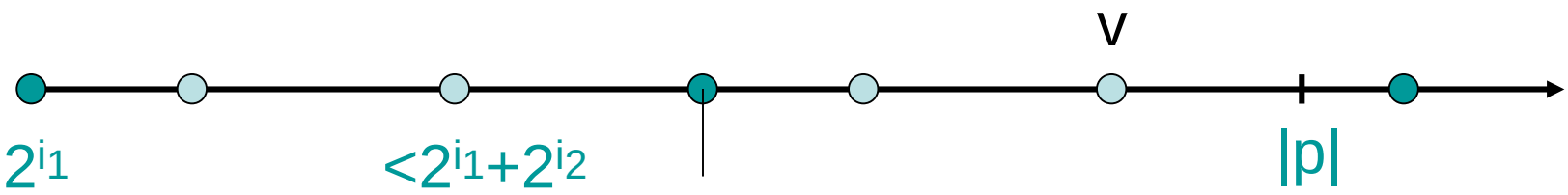


a) kein msd-Knoten bei $2^{i_1}+2^{i_2}$: nur dann, wenn kein Patricia-Knoten u mit $2^{i_1}<|b(u)|\leq 2^{i_1}+2^{i_2}$, aber das wird durch p_w erkannt und abgedeckt

Trie Hashing

Korrektheit von Phase 1:

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

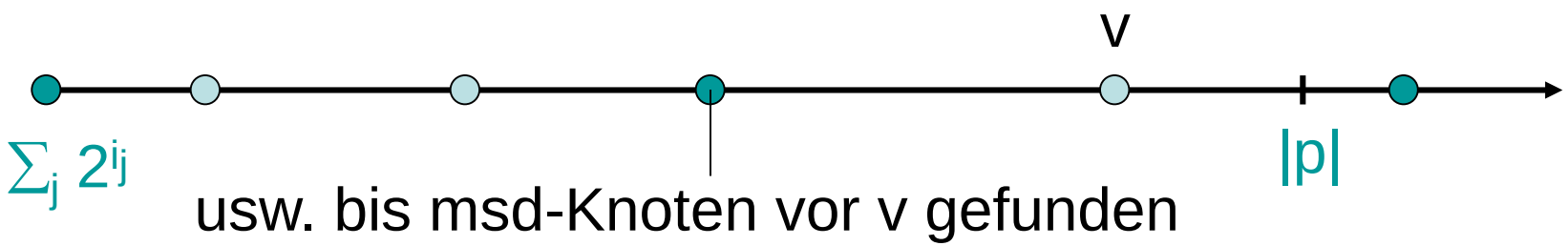


b) msd-Knoten bei $2^{i_1}+2^{i_2}$: wird durch binäre Suche gefunden

Trie Hashing

Korrektheit von Phase 1:

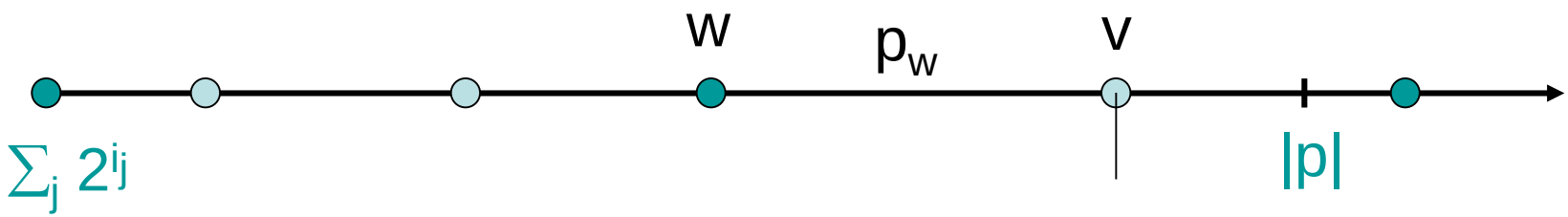
- Sei p der größte gemeinsame Präfix von x und einem Element $y \in S$ und sei $|p|=(z_k,\ldots,z_0)$.
- Patricia Trie enthält Route für Präfix p
- Sei v letzter Knoten auf Route bis p
- Fall 1: v ist Patricia Knoten



Trie Hashing

Korrektheit von Phase 1:

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

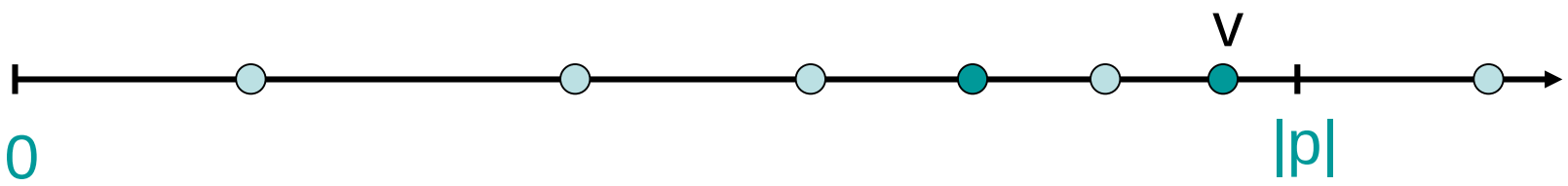


Durch Beachtung von p_w im Algorithmus wird dann auch v gefunden.

Trie Hashing

Korrektheit von Phase 1:

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



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

Trie Hashing

Aufwand für search:

- Phase 1 (binäre Suche): O(log W) Lesezugriffe auf DHT
- Phase 2 (entlang Patricia Trie): O(1) Lesezugriffe auf DHT

Insgesamt O(log W) Lesezugriffe auf DHT.

Aufwand für insert:

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

Also O(log W) Lese- und O(1) Schreibzugriffe auf DHT

Aufwand von delete: O(1) Lese- und Schreibzugriffe auf DHT (nur Restrukturierung)

Trie Hashing

Schlüsseluniversum: U

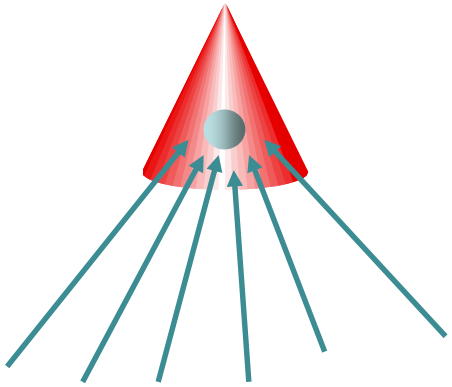
- Wortgröße: $W = \lceil \log |U| \rceil$
- Anzahl Lesezugriffe (search, insert): $O(\log W) = O(\log \log |U|)$
- Anzahl Schreibzugriffe (insert, delete): $O(1)$

Beispiel: $U = \{a, \ldots, z\}^{10}$

Anzahl Zugriffe ~6

Trie Hashing

Probleme bei vielen Anfragen auf dieselben Patricia Trie Knoten:



Prozess, der Knoten speichert, wird überlastet.

Auch hier Lösung: Combine & Split

Nachfolgersuche

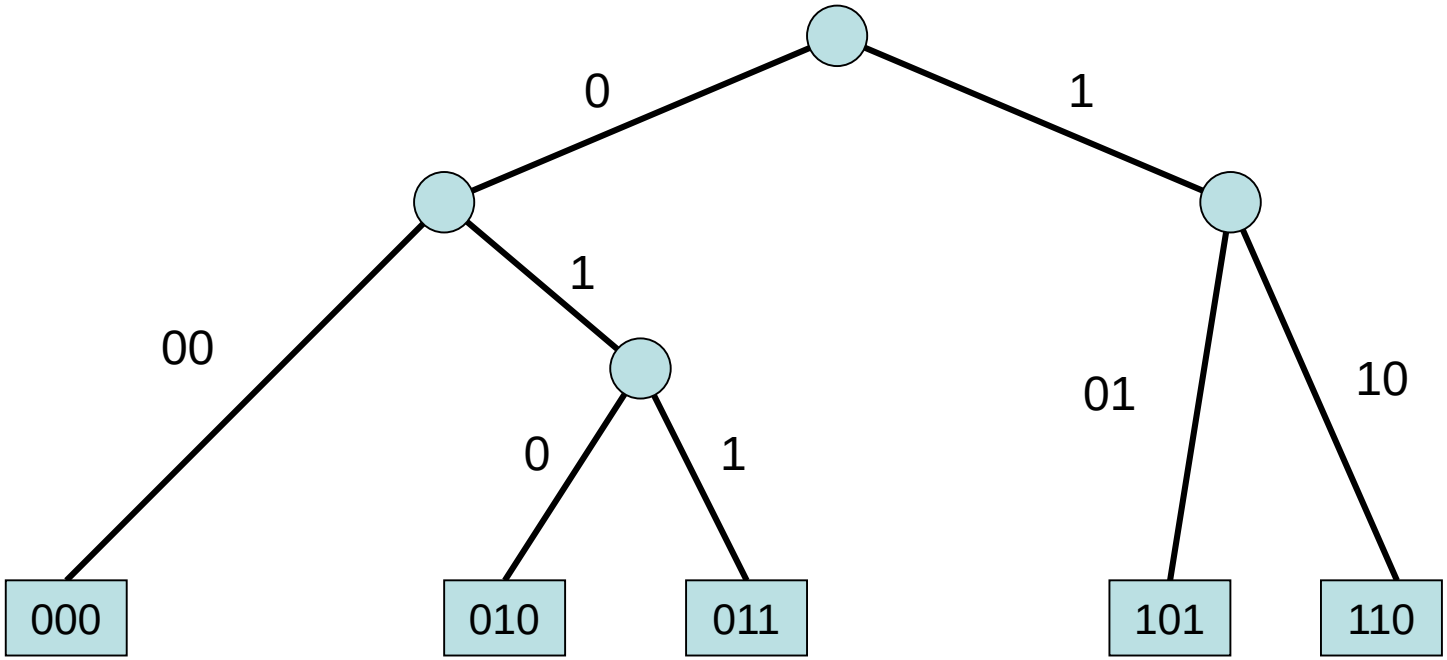
- Alle Schlüssel kodiert als binäre Folgen $\{0,1\}^w$
- S: gegebene Schlüsselmenge
- Nächster Nachfolger eines Schlüssels $x \in \{0,1\}^w$: der lexikographisch nächste Schlüssel von x in S

Problem: finde für einen Schlüssel $x \in \{0,1\}^w$ den nächsten Nachfolger $y \in S$

Einfache Lösung: auch hier Trie Hashing, aber wir verwenden nur normalen Patricia Trie (ohne msd-Knoten)

Nachfolgersuche

Search(4) ergibt 5 (bzw. 101)



Nachfolgersuche

Aufwand für search:

- Suche entlang Patricia Trie: im worst case O(W) Lesezugriffe auf DHT

Aufwand für insert:

- O(W) Lesezugriffe auf DHT für search-Operation
- O(1) Restrukturierungen im Patricia Trie

Also O(W) Lese- und O(1) Schreibzugriffe auf DHT

Aufwand von delete: O(1) Lese- und Schreibzugriffe auf DHT

(nur Restrukturierung)

Kann Suche nach nächstem Nachfolger beschleunigt werden?

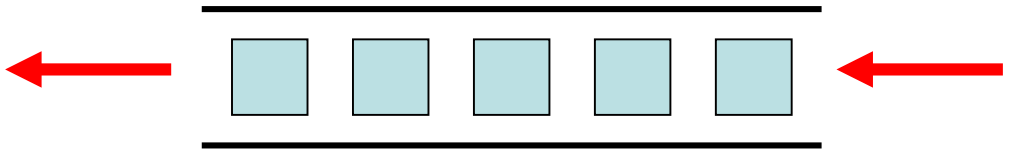
Übersicht

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

Konventionelle Queue

Eine Queue Q unterstützt folgende Operationen:

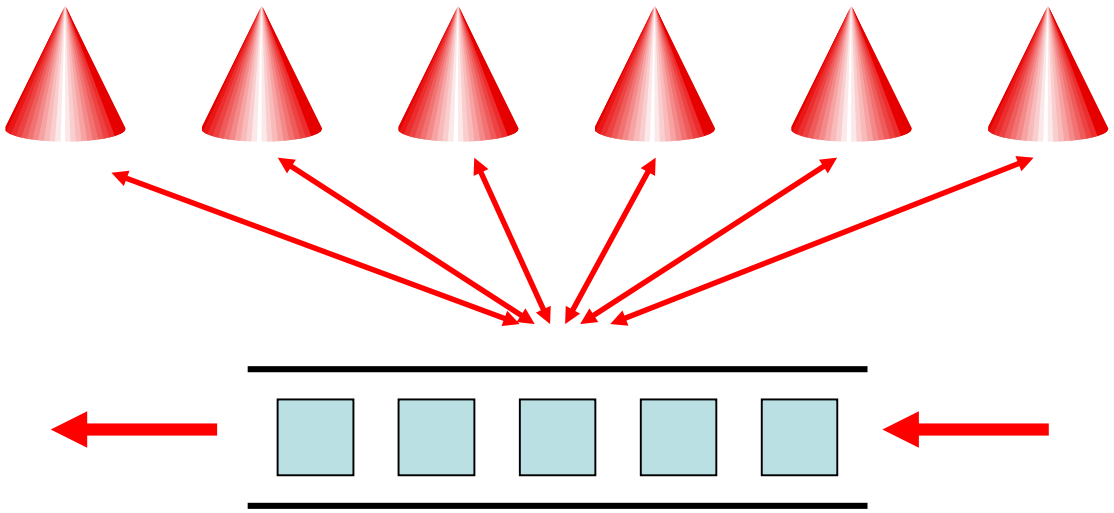
- enqueue(Q,x): fügt Element x hinten an die Queue Q an.
- dequeue(Q): holt das vorderste Element aus der Queue Q heraus und gibt es zurück



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

Verteilte Queue

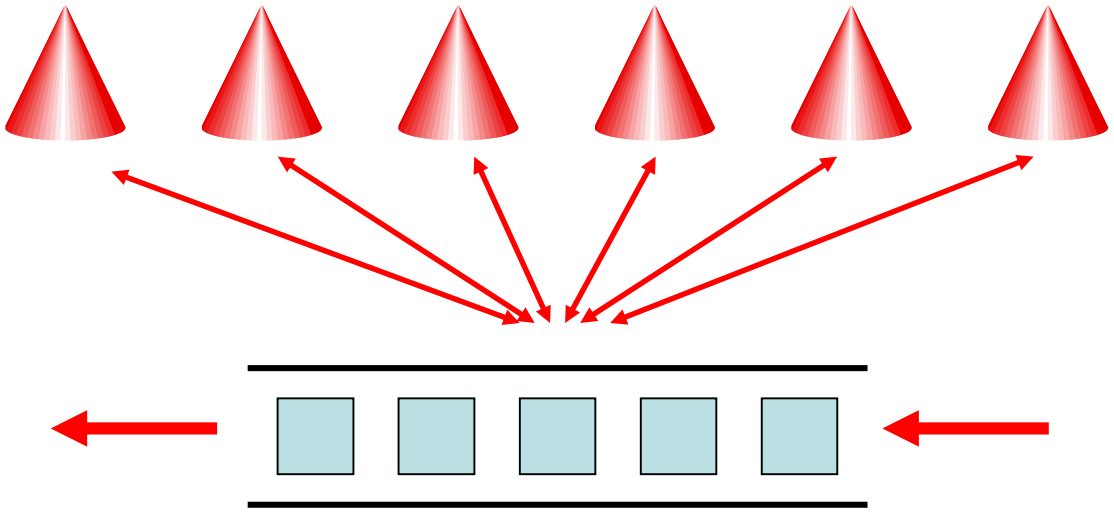
Viele Prozesse agieren auf Queue:



Verteilte Queue

Probleme:

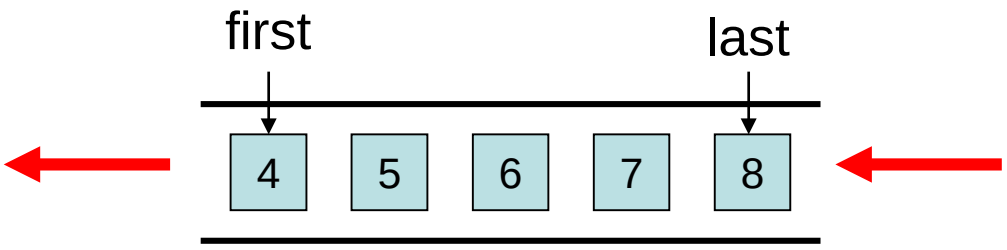
- Speicherung der Queue
- Realisierung von enqueue und dequeue



Verteilte Queue

Speicherung der Queue:

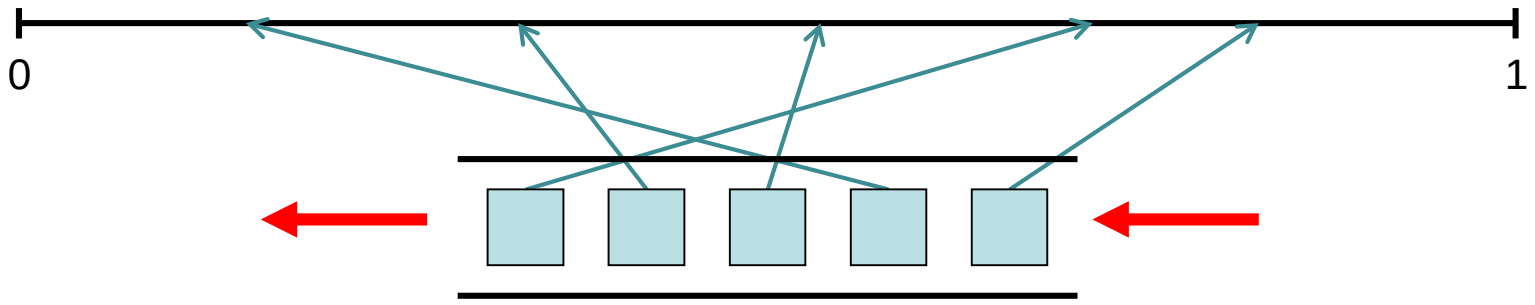
- Jedes Element x besitzt eindeutige Position $pos(x) \geq 1$ in der Queue (das vorderste hat die kleinste und das hinterste die größte Position).



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.

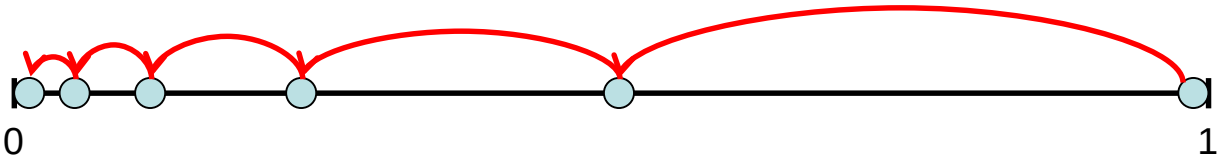
Noch zu klären: Punkt 1 in enqueue und dequeue.

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 enq(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 sequentielle Konsistenz, selbst wenn die verteilte Queue im legalen Zustand ist!

Verteilte Queue

Annahme: verteilte Queue ist im legalen Zustand

Einfache Lösung für sequentielle Konsistenz:

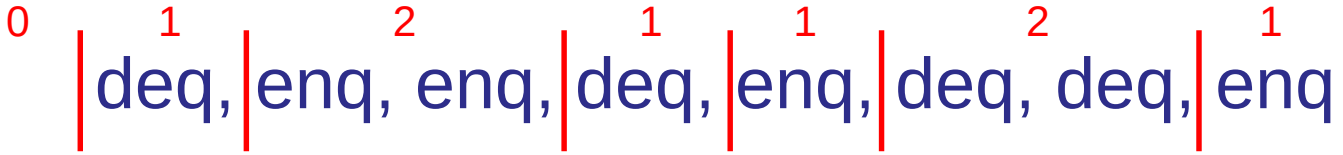
- Jeder Knoten v wartet solange mit der Ausführung einer Operation, bis die vorherige Operation bearbeitet worden ist.

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 (bzw. es reicht, solange zu warten, bis er eine Rückmeldung vom Anker für die vorige Folge bekommen hat).

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 ($[x_1,y_1]$, $[x_2,y_2]$, $[x_3,y_3]$, …,$[x_k,y_k]$) zurück an die Rücksprungadresse von serve(a_1,a_2,…,a_k).
- Von dort aus werden die Intervalle aufgeteilt wie vorher für separate enq(Q,i) und deq(Q,i) 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 put und get Anfragen ausgeschickt werden. Gleichzeitig kann er dann eine neue serve-Anfrage stellen.

Sequentielle Konsistenz

Formale Definition der sequentiellen Konsistenz für eine 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:

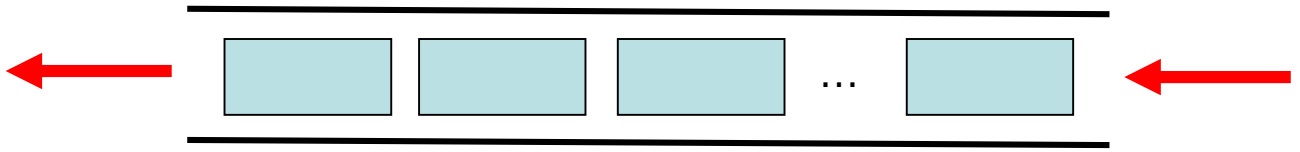
1. Für alle $(Enq_v(i), Deq_w(j)) \in M$ ist $Enq_v(i) < Deq_w(j)$. (Ein Dequeue kann nur ein bereits eingefügtes Element zurückgeben.)
2. Für alle $(Enq_u(i), Deq_v(j)) \in M$ gilt: es gibt kein unzugeordnetes $Deq_w(k)$ mit $Enq_u(i) < Deq_w(k) < Deq_v(j)$ und es gibt kein unzugeordnetes $Enq_w(k)$ 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.)

Sequentielle Konsistenz: zusätzlich zu 1-3

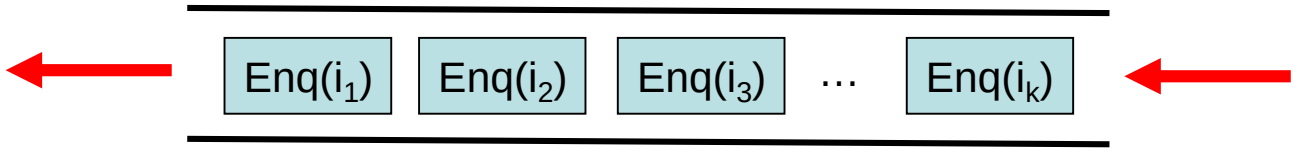
4. Für alle $v \in V$ und $i \in \mathbb{N}$ ist $Op_v(i) < Op_v(i+1)$.

Sequentielle Konsistenz

Angenommen, die Linearisierbarkeit sei bisher erfüllt. Dann gilt für die momentane Queue:



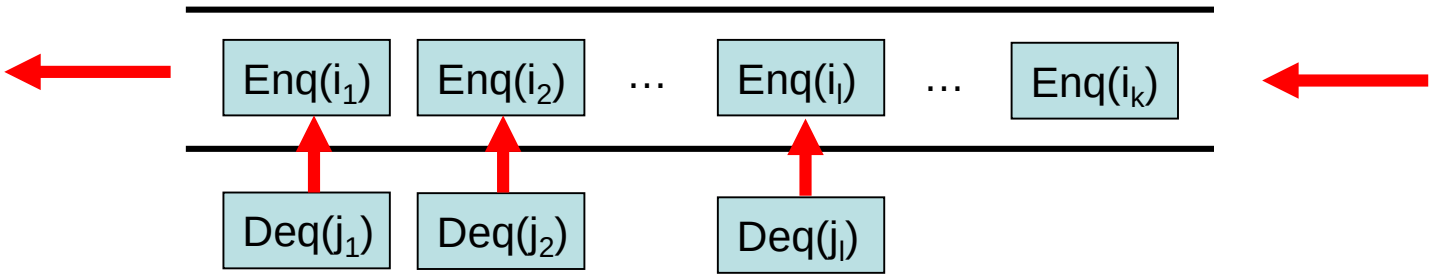
entspricht



mit $Enq(i_1)<Enq(i_2)<Enq(i_3)<\ldots<Enq(i_k)$.

Sequentielle Konsistenz

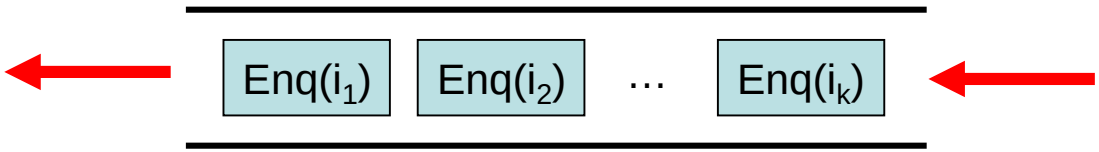
Für $l \leq k$ dequeue Anfragen $Deq(j_1) < Deq(j_2) < Deq(j_3) < \ldots < Deq(j_l)$ wird dann die Linearisierbarkeit bei folgendem Matching bewahrt:



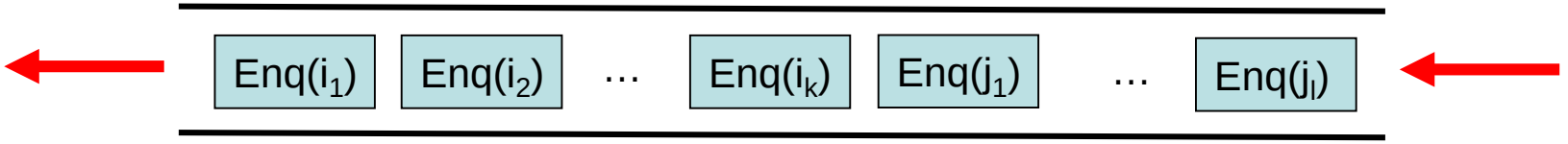
Ähnliches gilt auch für $l > k$ (wobei die letzten $l-k$ Deq Anfragen nicht gematched werden).

Sequentielle Konsistenz

Weiterhin wird für l neue enqueue Anfragen $Enq(j_1)<Enq(j_2)<Enq(j_3)<\ldots<Enq(j_l)$ die Linearisierbarkeit wie folgt bewahrt:



ergibt



Sequentielle Konsistenz

Ordnung „<“: wird induktiv aufgebaut.

- Interpretiere eine serve(a_1,a_2,…,a_k) Anfrage als eine serve(S_1,S_2,…,S_k) Anfrage, in der jedes S_i die Folge von a_i enqueue bzw. dequeue Anfragen darstellt, aus denen das a_i hervorgegangen ist.
- Für eine serve(S_1,S_2,…,S_k) Anfrage mit S_i=($Op_{i,1}$,…,Op_{i,a_i}) definieren wir dann die Ordnung $Op_{1,1}$<…<Op_{1,a_1}<$Op_{2,1}$<…<Op_{k,a_k}, was Bedingung 4 (sequentielle Konsistenz) für eine neu erzeugte serve(a_1,a_2,…,a_k) Anfrage sicherstellen würde.
- Aufbauend auf dieser Regel wird Bedingung 4 bei der Kombination von serve(a_1,a_2,…,a_k), serve(b_1,b_2,…,b_k) serve(c_1,c_2,…,c_k),… Anfragen zu einer serve(z_1,z_2,…,z_k) Anfrage mit z_i=a_i+b_i+c_i+… für alle i bewahrt, sofern die Ordnung der (durch a_i, b_i, c_i,… repräsentierten) kombinierten Teilfolgen bewahrt bleibt (was induktiv gezeigt werden kann).
- Der Anker führt nun jede serve(a_1,a_2,…,a_k) Anfrage so aus, dass zuerst alle Anfragen in S_1, dann alle Anfragen in S_2, usw, und zum Schluss alle Anfragen in S_k bearbeitet werden. Dabei vergibt er Intervalle, so dass die Bedingungen auf den Folien 160-162 erfüllt sind. Damit stellt der Anker sicher, dass neben Bedingung 4 auch die Bedingungen 1-3 erfüllt sind und damit (sofern seine Vorgaben auch so umgesetzt werden) die sequentielle Konsistenz gilt.
- Bei der Aufteilung der Intervalle auf dem Weg vom Anker zurück zu den Knoten wird die sequentielle Konsistenz induktiv bewahrt (was wiederum durch Induktion gezeigt werden kann), so dass am Ende die sequentielle Konsistenz bei den Startknoten gewährleistet ist.

Verteilte Queue

Für die Korrektheit der enqueue und dequeue Anfragen muss zusätzlich sichergestellt sein:

Für jede Position pos wird put(pos,x) vor get(pos) ausgeführt.

Mögliche Lösung:

- Sei S_i der i-te Batch an enqueue und dequeue Anfragen, der vom Anker bearbeitet worden ist.
- Eine get Anfrage in S_i darf erst dann bearbeitet werden, wenn alle put Anfragen in allen S_j mit $j \leq i$ bearbeitet worden sind.
- Um zu wissen, dass alle put Anfragen in allen S_j mit $j \leq i$ bearbeitet worden sind, senden alle in einem S_j beteiligten Knoten ein ACK(l,r) an den Anker sobald alle seine put Anfragen (mit Positionen in [l,r]) in S_j abgeschlossen sind (auch wenn er keine hatte). Die ACKs werden zum Anker hin aggregiert.
- Der Anker schickt seinerseits ein ACK an alle Knoten zurück, sobald er das aggregierte ACK für ein S_j bekommen hat, so dass diese wissen, dass alle put Anfragen für S_j abgeschlossen sind.

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 (über enqueues) neue Positionen zuzuweisen.

Verteilte Queue

Selbststabilisierung:

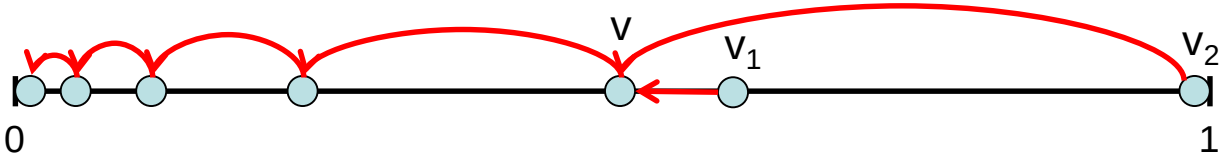
- Verteilte Hashtabelle: bereits vorher betrachtet
- Anker v_0: Knoten ist Anker, solange er keinen linken Vorgänger hat. Sonst gibt er die Ankerfunktion auf (und transferiert gegebenenfalls first und last).

Probleme:

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

Mögliche Lösung:

- Alle Knoten v mit Elementen 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:

- Monotone Suchbarkeit des Ankers:
  Ist einmal eine Anfrage von v zu v_0 gelangt, ist das auch in Zukunft so. Hier können wir statt des de Bruijn Graphen den Brücken Skip+-Graphen verwenden. Der Rückweg ist garantiert, da die Knoten sich die Referenzen für die Rücksprünge merken.
- Monotone Suchbarkeit für put und get Anfragen:
  Hier können wir die monotone Suchstrategie für die verteilte Hashtabelle verwenden.

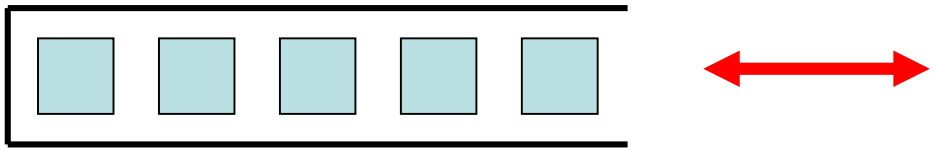
Übersicht

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

Konventioneller Stack

Ein Stack S unterstützt folgende Operationen:

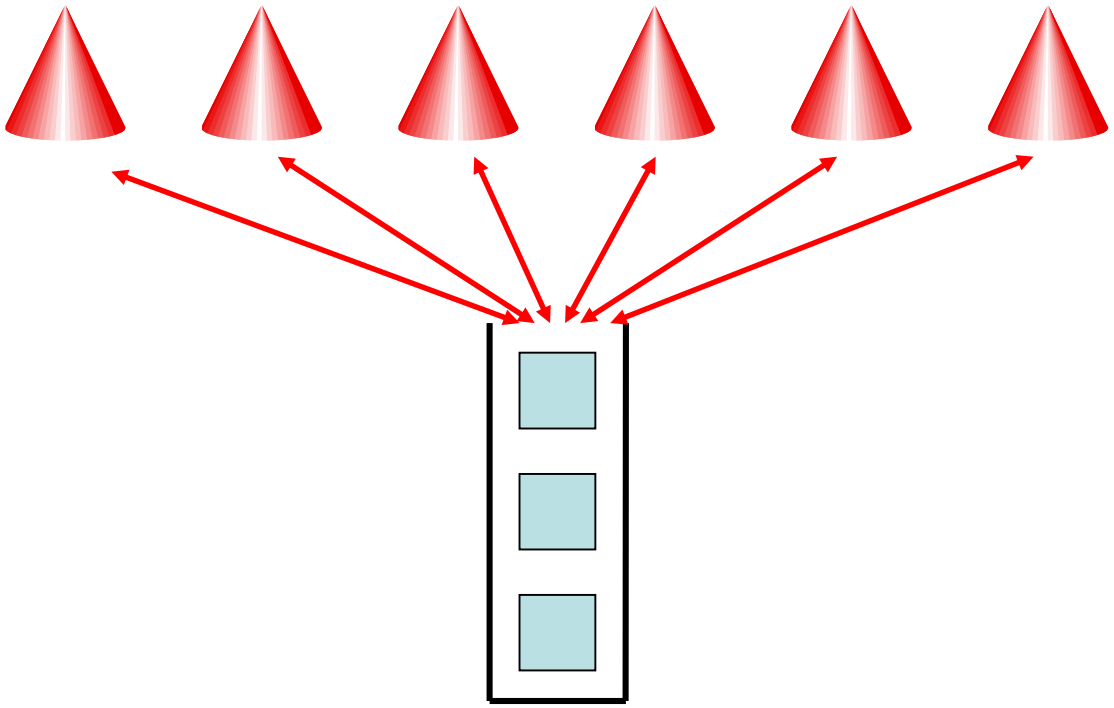
- push(S,x): legt Element x oben auf dem Stack ab.
- pop(S): holt das oberste Element aus dem Stack S heraus und gibt es zurück



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

Verteilter Stack

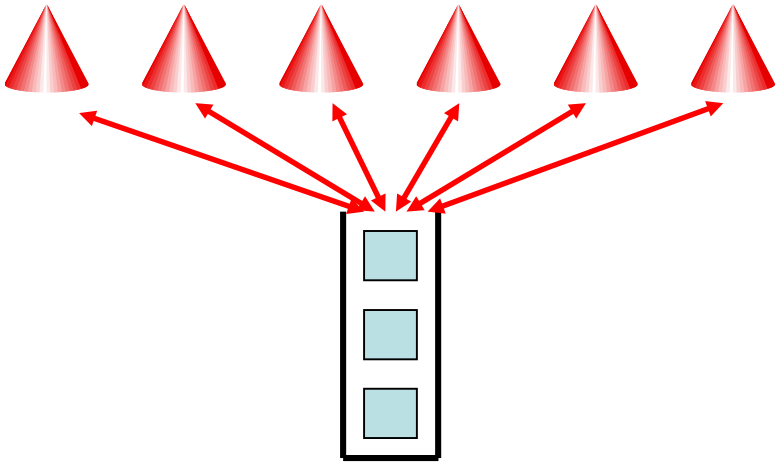
Viele Prozesse agieren auf Stack:



Verteilter Stack

Probleme:

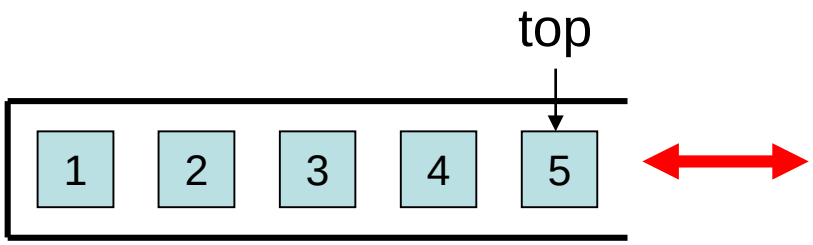
- Speicherung des Stacks
- Realisierung von push und pop



Verteilter Stack

Speicherung des Stacks:

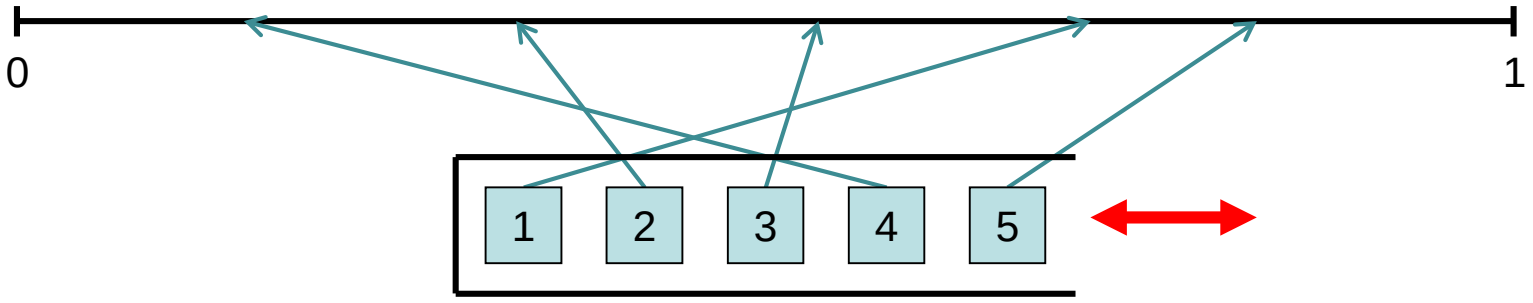
- Jedes Element x besitzt eindeutige Position pos(x)≥1 im Stack (das oberste hat die größte Position).



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 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.

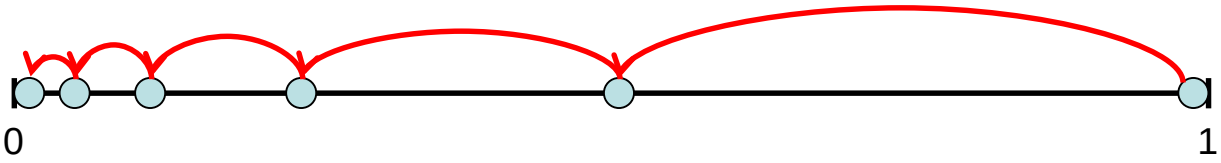
Noch zu klären: Punkt 1.

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 pushall(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

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

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.

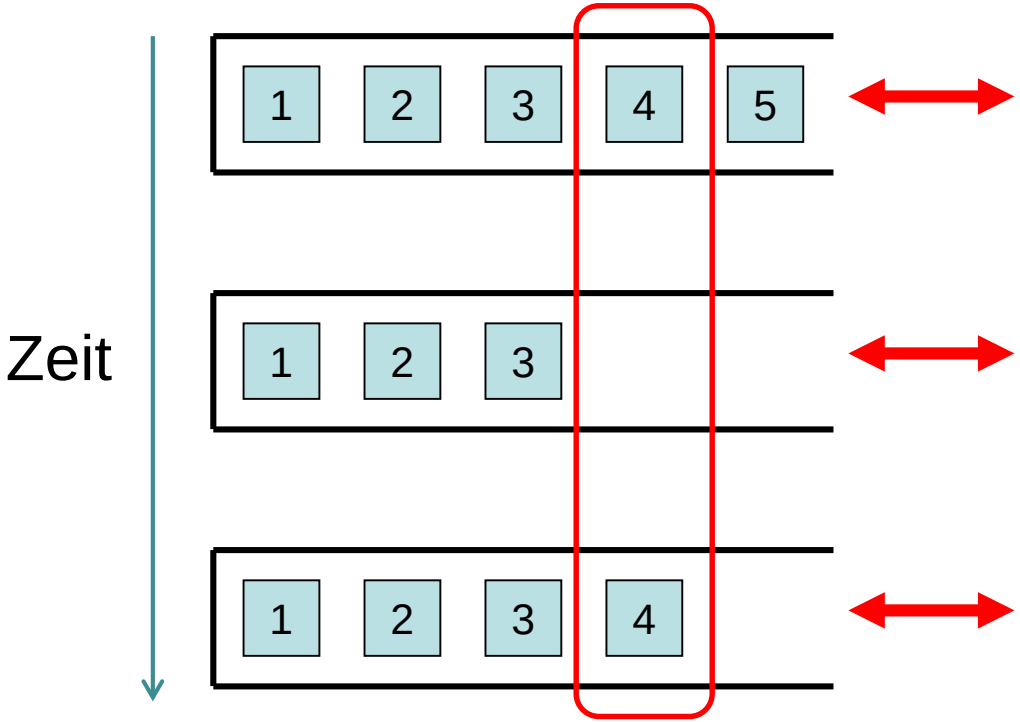
Noch zu klären: Punkt 1.

Hier können wir ähnlich wie für enq und deq vorgehen.

Neues Problem: push Anfragen könnten dieselbe Position zugewiesen bekommen.

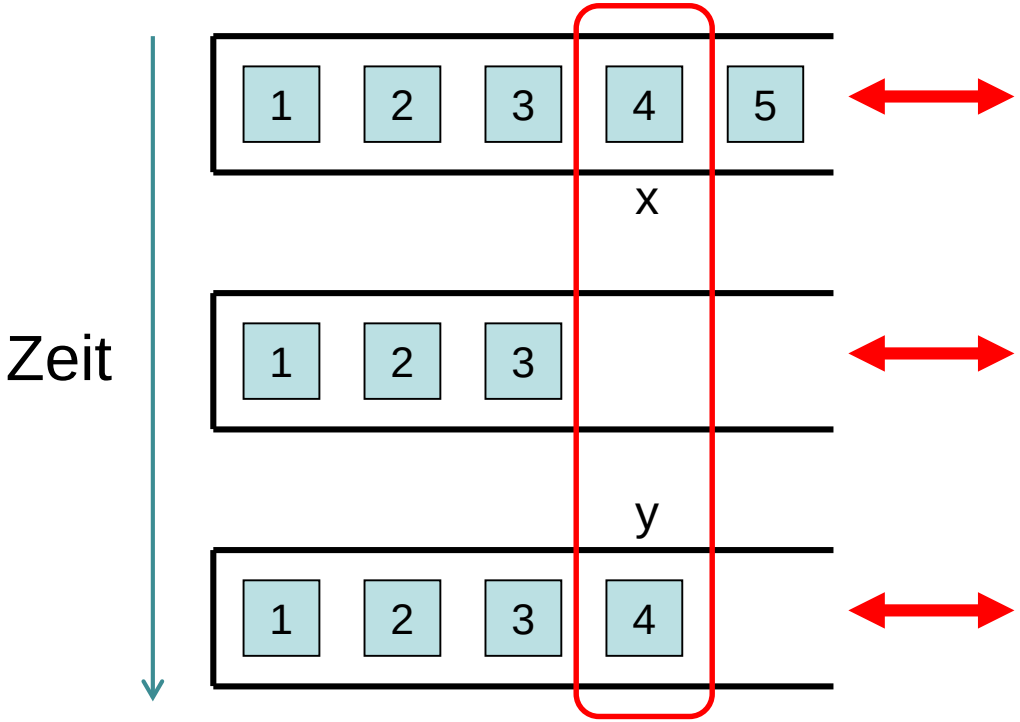
Verteilter Stack

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.



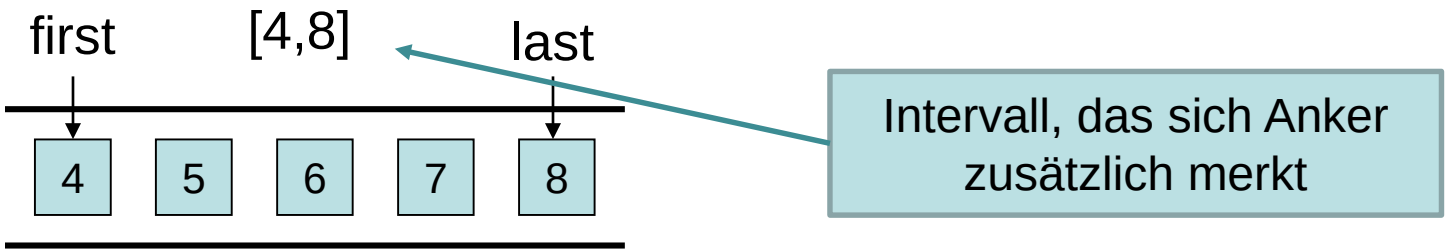
Verteilter Stack

Neues Problem: push Anfragen könnten dieselbe Position zugewiesen bekommen.

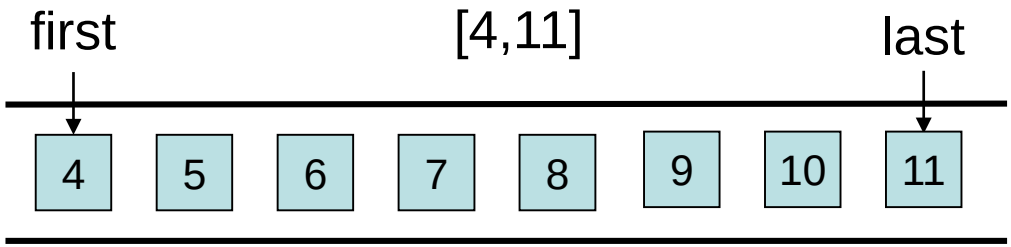
Mögliche Lösung (einfacher Fall):

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

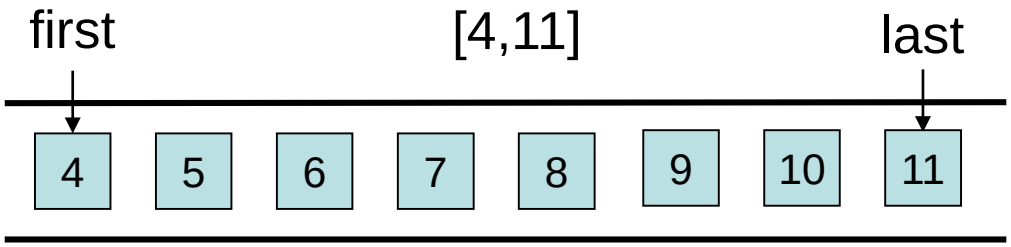
Verteilter Stack



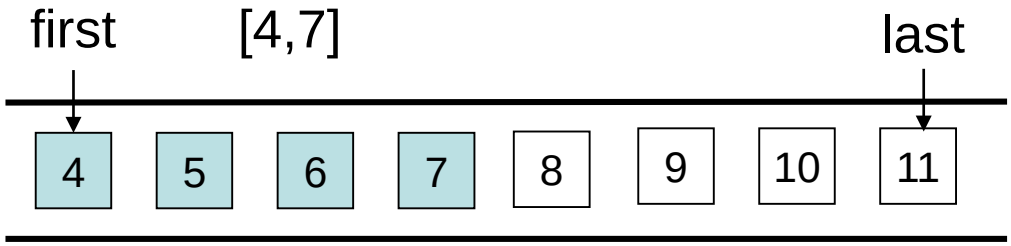
pushall(S,3):



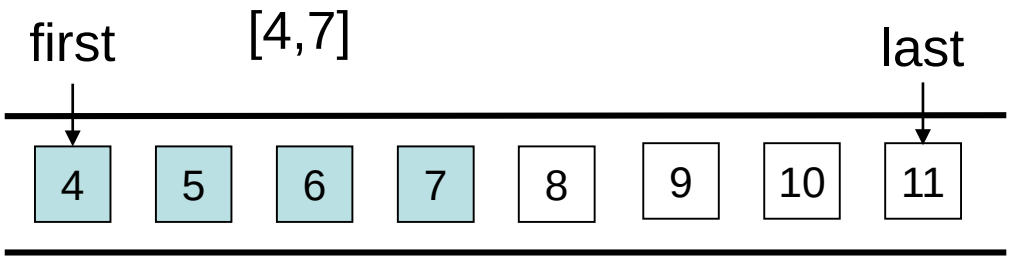
Verteilter Stack



popall(S,4): gibt [8,11] zurück

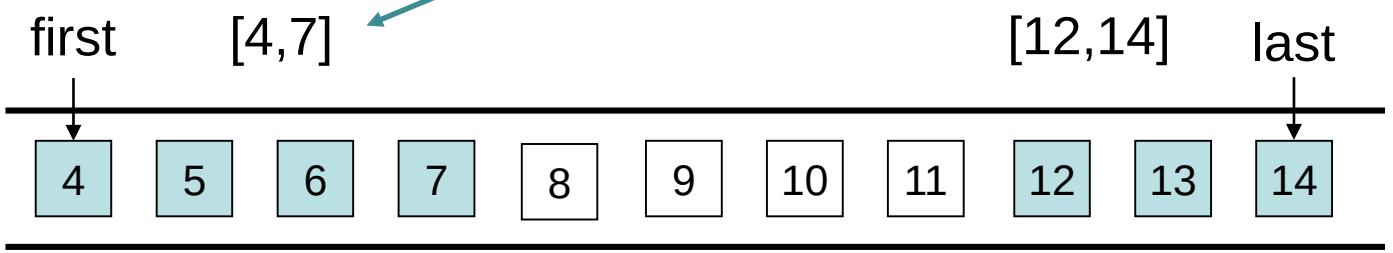


Verteilter Stack

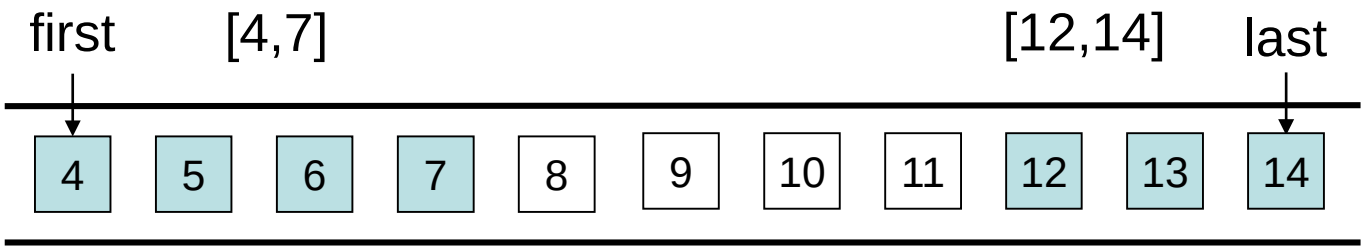


pushall(S,3): [12,14]

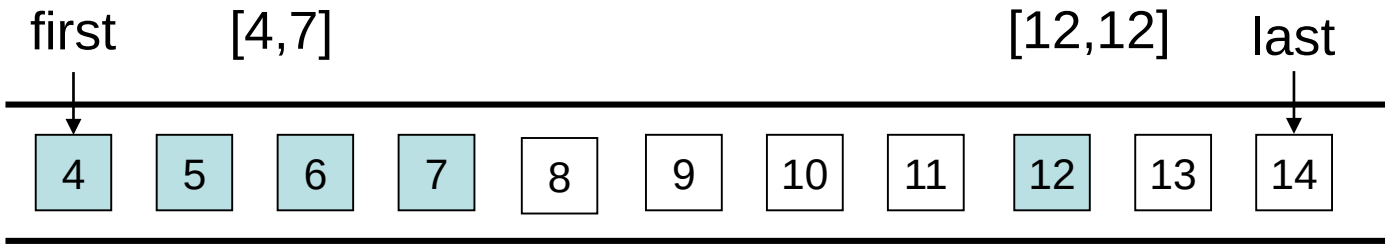
Anker muss sich nun zwei Intervalle merken



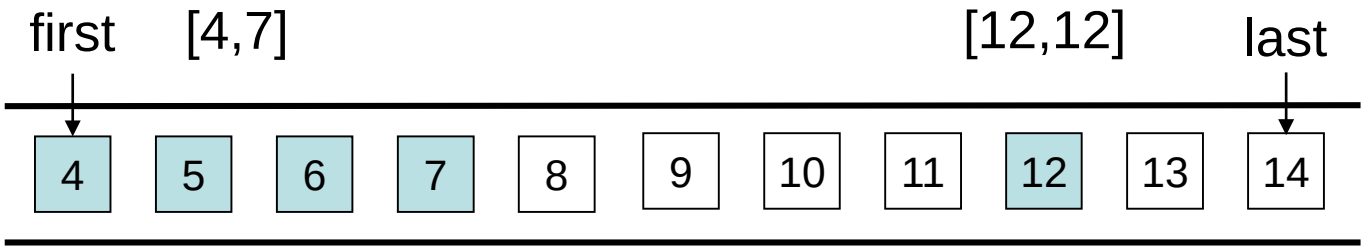
Verteilter Stack



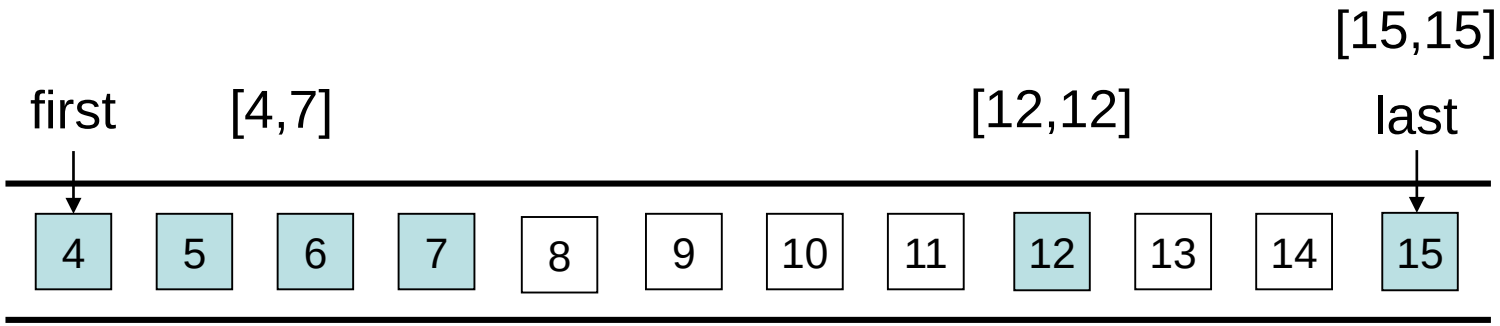
popall(S,2): gibt [13,14] zurück



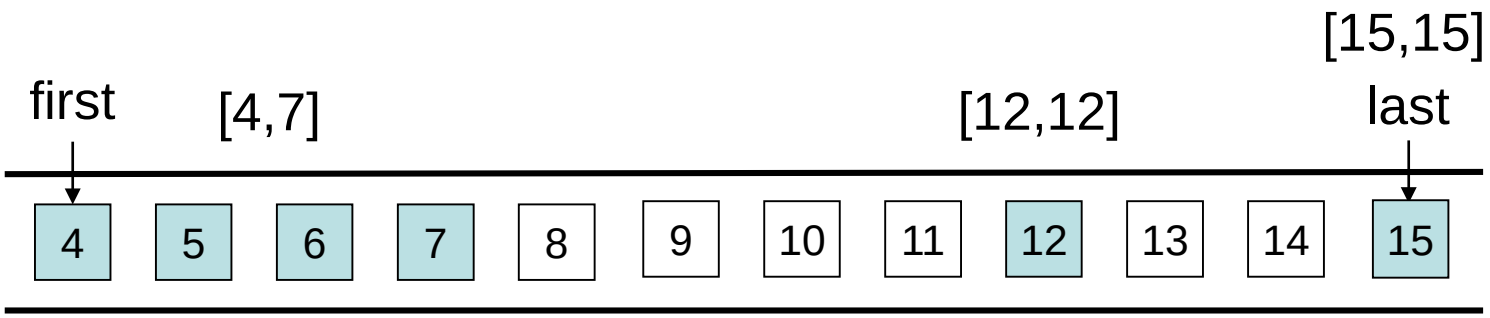
Verteilter Stack



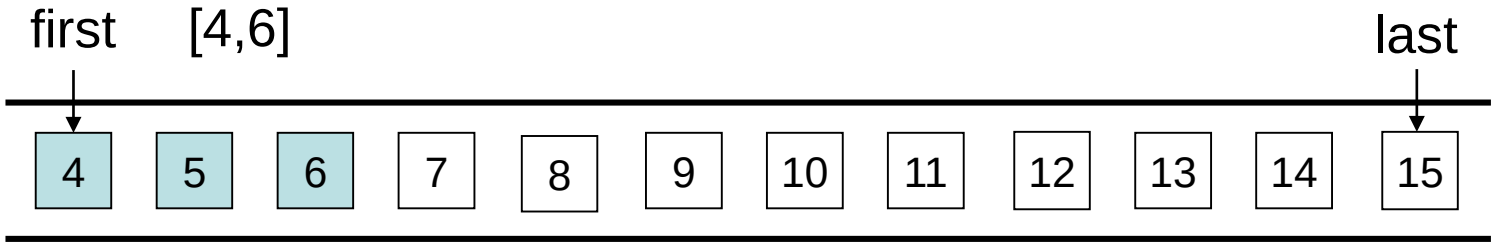
pushall(S,1): gibt [15,15] zurück



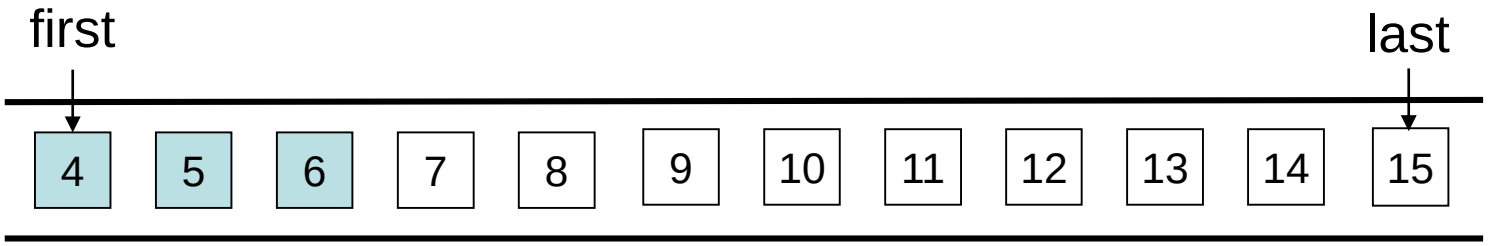
Verteilter Stack



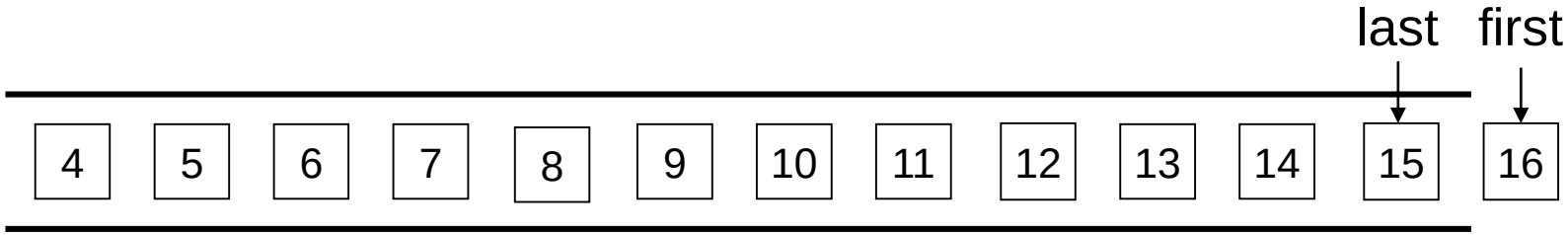
popall(S,3): liefert ([15,15],[12,12],[7,7])



Verteilter Stack



popall(S,3): gibt [4,6] zurück



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.

Problem: höherer Speicheraufwand!

Verteilter Stack

Alternative Lösung:

- Anker merkt sich zwei Zähler: top und round. Anfangs ist top=0 und round=1.
- Verwende Batch-Bearbeitung über serve(a_1,a_2,…,a_k) Anfragen wie für die Queue, aber mit einigen Änderungen.

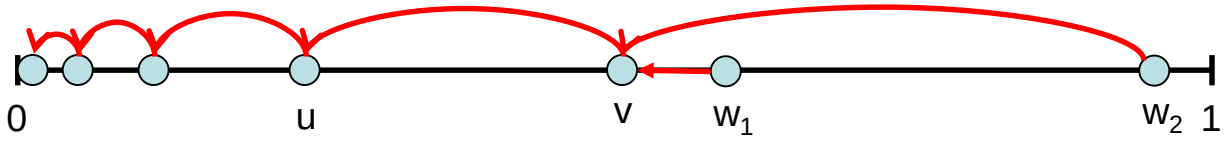
Bearbeitung von serve(a_1,a_2,…,a_k) Anfragen:

- Eine serve(a_1,a_2,…,a_k) Anfrage repräsentiert jetzt aufeinanderfolgende pushall(S, a_1), popall(S,a_2), pushall(S,a_3),… Anfragen.
- Alle Knoten arbeiten in Runden. In jeder Runde generiert jeder Knoten nur eine serve Anfrage. Die serve Anfragen einer Runde werden zum Anker hin zu einer serve(a_1,a_2,…,a_k) Anfrage aggregiert (wie wir noch beschreiben werden).
- Für diese serve(a_1,a_2,…,a_k) Anfrage berechnet der Anker dann Tupel ([x_1,y_1],t_1) , ([x_2,y_2],t_2), ([x_3,y_3],t_2) (wie auf den nächsten Folien erklärt) für die in der serve Anfrage aggregierten pushall(S,a_1), popall(S,a_2), pushall(S,a_3),… Anfragen und schickt diese zusammen mit dem round Wert zurück an die Rücksprungadresse von serve(a_1,a_2,…,a_k). Von dort werden die Intervalle dann weiter aufgeteilt und zusammen mit round zurückgegeben, bis jeder Knoten Intervalle für seine erzeugte serve Anfrage erhält.
- Eine Folge von k push(x_i) Anfragen, die das Intervall ([pos,pos+k-1],t) zugeordnet bekommt, generiert dann put(pos+i,(round,t,x_i)) Anfragen, welche den Eintrag (round,t,x_i) in dem für den Hashwert von pos+i verantwortlichen Knoten in der verteilten Hashtabelle speichert. Bei pop Anfragen ist das ähnlich, aber die Bearbeitung der get(pos+i,(round,t)) Anfragen werden wir später noch näher beschreiben.

Verteilter Stack

Aggregierung der serve Anfragen zu einer einzigen serve Anfrage beim Anker:

- Jeder Knoten v wartet solange mit dem Verschicken einer serve Anfrage, bis er von allen Vorgängern im Aggregationsbaum (hier w_1 und w_2) eine serve Anfrage erhalten hat.



- Sobald v eine serve Anfrage von jedem Vorgänger erhalten hat, z.B. serve(a_1,a_2,…,a_k) und serve(b_1,b_2,…,b_k), aggregiert er diese zusammen mit seiner eigenen serve(c_1,c_2,…,c_k) Anfrage zu einer serve(d_1,d_2,…,d_k) Anfrage (d.h. $d_i=a_i+b_i+c_i$) und ist damit bereit, eine serve Anfrage zu verschicken.
- Jeder Knoten v, der bereit ist, eine serve Anfrage zu verschicken, schickt diese solange zum Nachfolger u (in timeout), bis er eine Bestätigung von u erhalten hat.

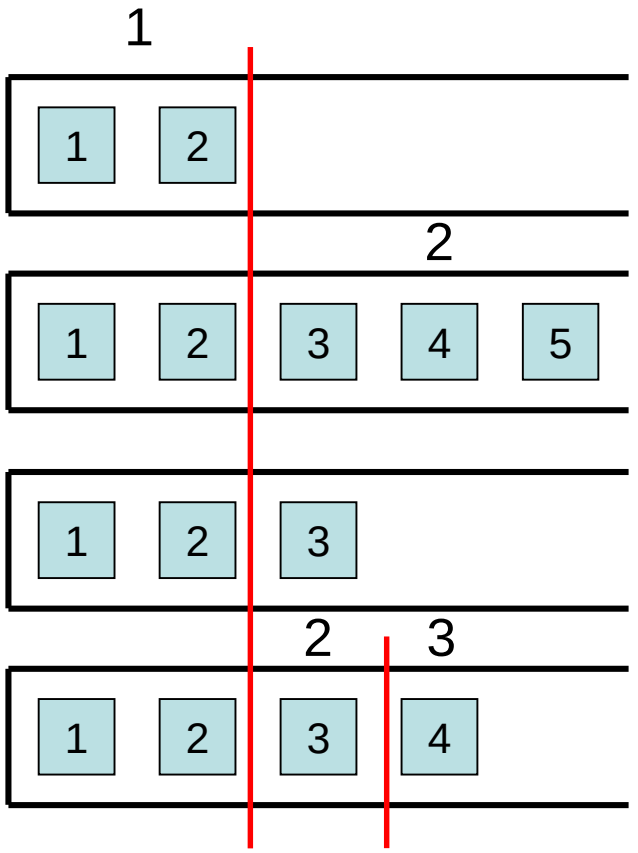
Verteilter Stack

Bei der Bearbeitung der aggregierten serve(a_1,a_2,…,a_k) Anfrage verwendet der Anker einen Zeitstempel t gestartet mit 1.

- Der Anker zerlegt zunächst die serve(a_1,a_2,…,a_k) Anfrage in aufeinanderfolgende pushall(S, a_1), popall(S,a_2), pushall(S,a_3),… Anfragen.
- Bei jeder Bearbeitung einer pushall(S,i) Anfrage wird t um 1 erhöht, das Intervall [top+1,top+i] zusammen mit t zurückgegeben und top danach auf top+i gesetzt.
- Bei jeder Bearbeitung einer popall(S,i) Anfrage wird das Intervall [max{1,top-i+1},top] aufgeteilt in Intervalle mit gleichem Zeitstempel zurückgegeben und top danach auf max{0,top-i} gesetzt.
- Zusätzlich zu den Intervallen schickt der Anker seinen aktuellen Wert von round zurück und setzt am Ende t auf 1 und round auf round+1.

Verteilter Stack

Bei der Bearbeitung der serve(a_1,a_2,…,a_k) Anfrage verwendet der Anker Zeitstempel t gestartet mit 1:



In diesem Beispiel:
(a_1,a_2,…,a_k)=(3,2,1,2,3,4,2)

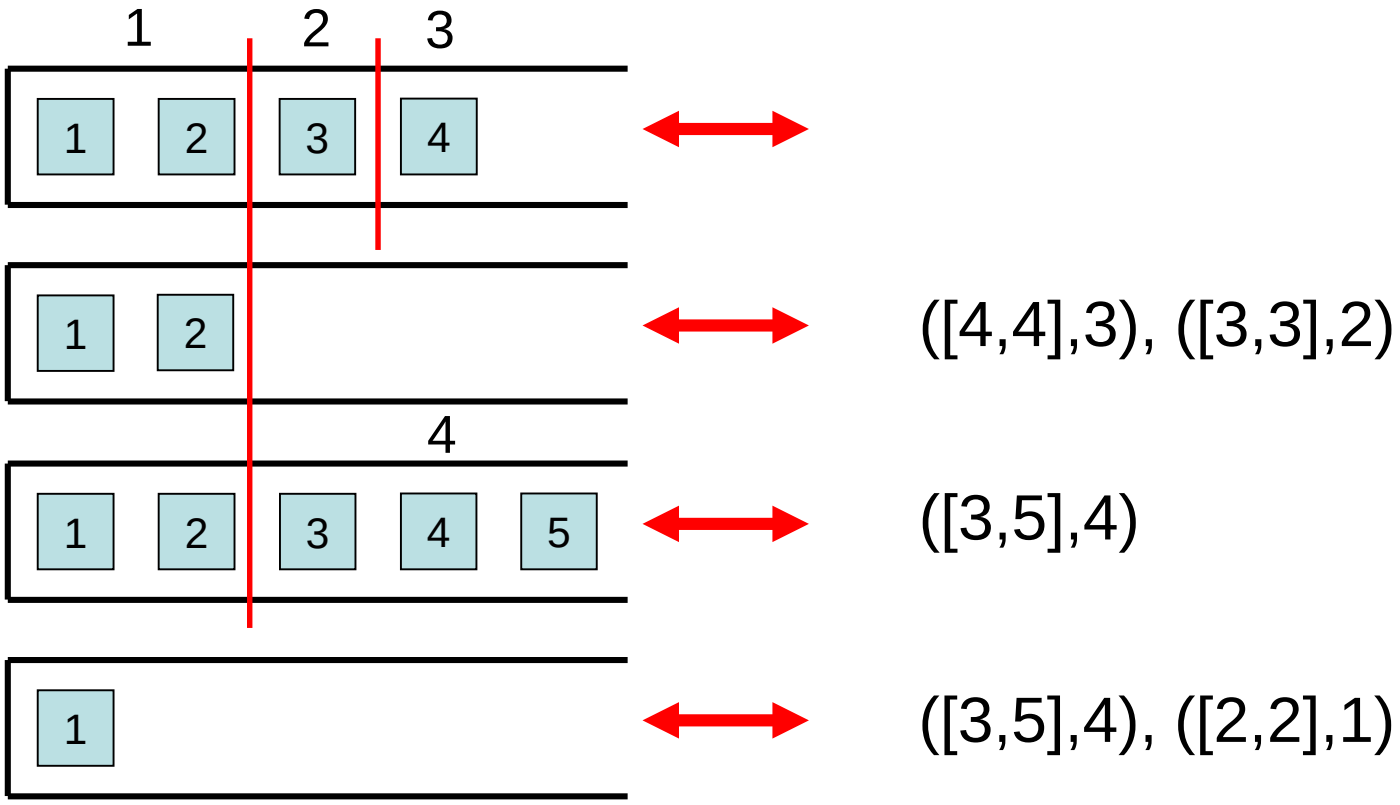
pushall(S,a_1): ([3,5],2)

popall(S,a_2): ([4,5],2)

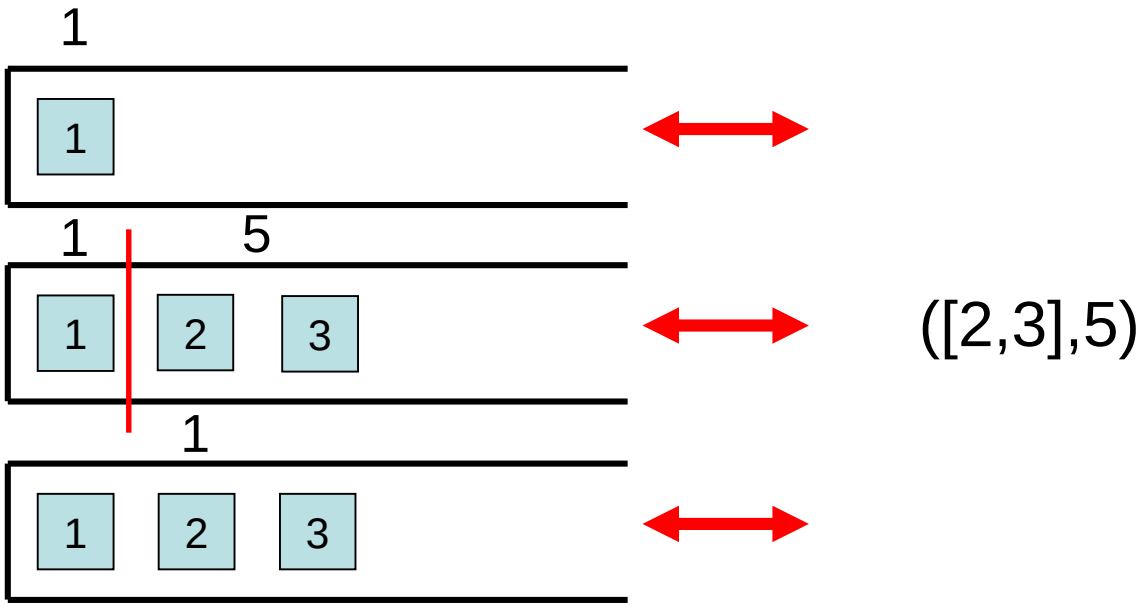
pushall(S,a_3): ([4,4],3)

Verteilter Stack

Auswahl der Zeiten am Beispiel:



Verteilter Stack



Am Ende setzt der Anker die Zeit t wieder auf 1 und erhöht round um 1.

Verteilter Stack

Anzahl Intervalle pro serve Operation:

- Verwende die Potentialfunktion
  $\phi(S)$=Anzahl Zeiten im Stack S
- pushall(S,i): erhöht $\phi(S)$ um 1.
- popall(S,i): erniedrigt $\phi(S)$ bis auf min. 1.

Ergebnis: Bei serve(a_1,a_2,…,a_k) Anfrage maximal $2k$ Intervalle der Form $([l,r],t)$ (statt minimal k wie ohne die Zeitstempel), also geringer Overhead.

Verteilter Stack

put(pos,(round,t,x)) Anfrage:

- Speichert den Eintrag (round,t,x) in dem für den Hashwert von pos verantwortlichen Knoten in der verteilten Hashtabelle ab.

get(pos,(round,t)) Anfrage:

- Schaut in dem Knoten nach, der für den Hashwert von pos zuständig ist, ob dort ein Eintrag der Form (round,t,x) existiert. Falls ja, wird x zurückgegeben und (round,t,x) gelöscht. Falls stattdessen t=1 ist und ein Eintrag der Form (r,t´,x´) für ein r<round und ein beliebiges t´ existiert, wird x´ zurückgegeben und danach (r,t´,x´) gelöscht.

Wie leicht zu überprüfen ist, wird damit jeder pop Anfrage eine eindeutige push Anfrage zugeordnet. (Übung)

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? Warum geht das nicht bei der Queue?

Sequentielle Konsistenz:

- Ähnlich zur Queue: wir brauchen eine formale Definition der sequentiellen Konsistenz und müssen dann zeigen, dass diese korrekt im System umgesetzt wird (Übung)

Verteilter Stack

Selbststabilisierung: ähnliche Probleme wie bei der Queue…

Monotone Korrektheit: auch ähnlich zur Queue.

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

Übersicht

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

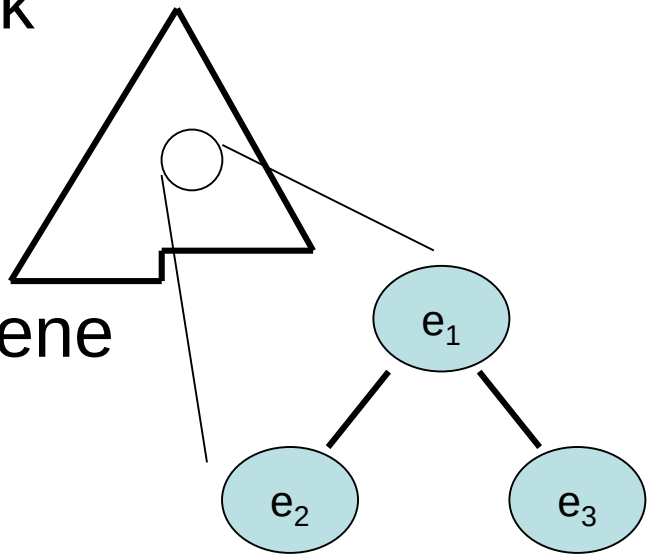
Konventioneller Heap

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

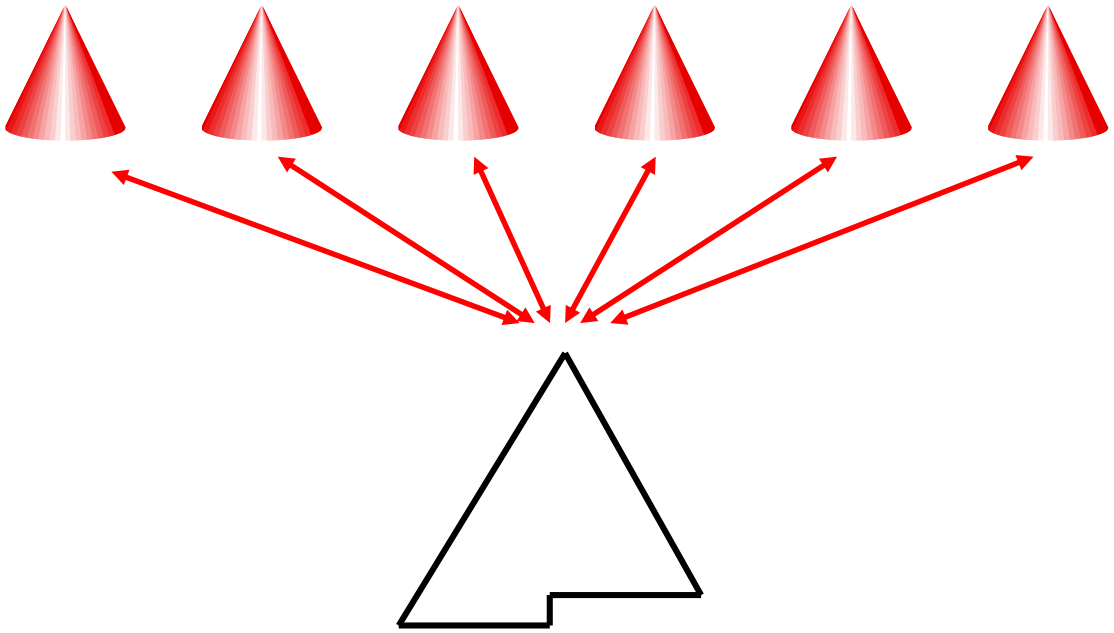
Zwei Invarianten:

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



Verteilter Heap

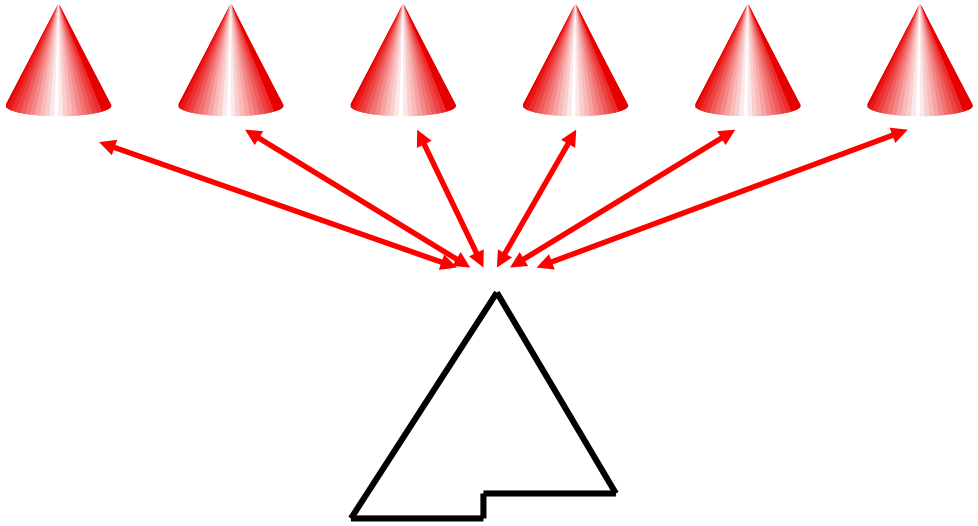
Viele Prozesse agieren auf Heap:



Verteilter Heap

Probleme:

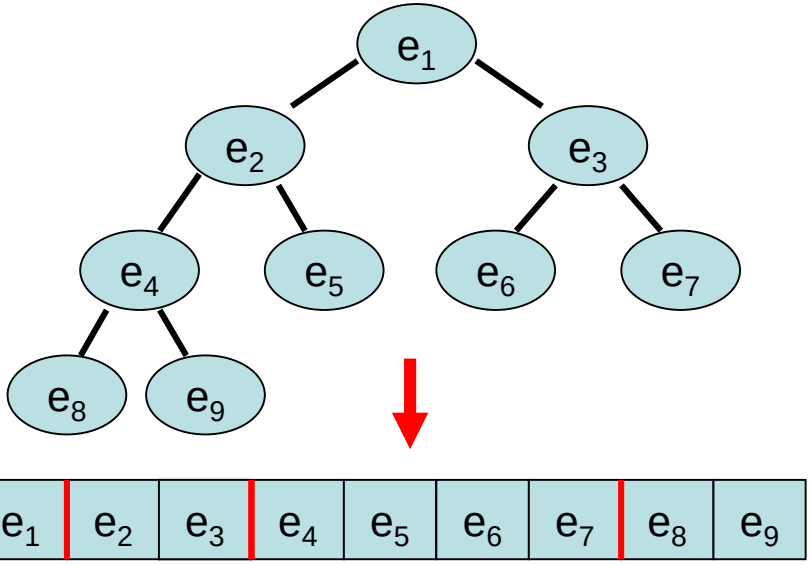
- Speicherung des Heaps
- Realisierung von insert und deleteMin



Verteilter Heap

Speicherung des Heaps:

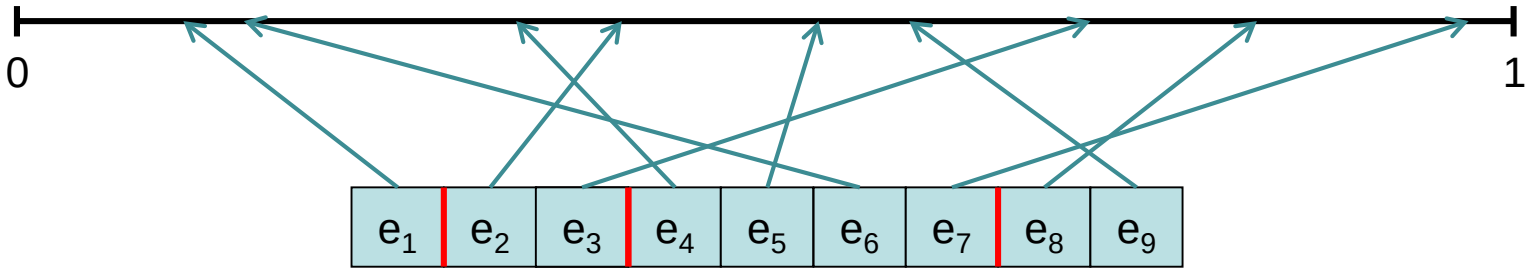
- Jedes Element x besitzt eine eindeutige Position $pos(x) \geq 1$ im Heap wie bei der Feldrealisierung des konventionellen Heaps



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):

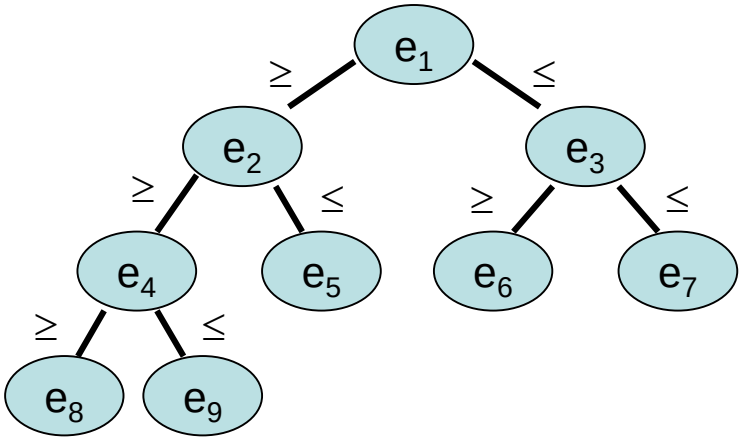
1. Stelle delmin(H,1)-Anfrage, um eine Nummer pos zu erhalten.
2. Führe get(pos) auf der verteilten Hashtabelle aus, um das unter pos gespeicherte Element x zu löschen und zu erhalten.

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.



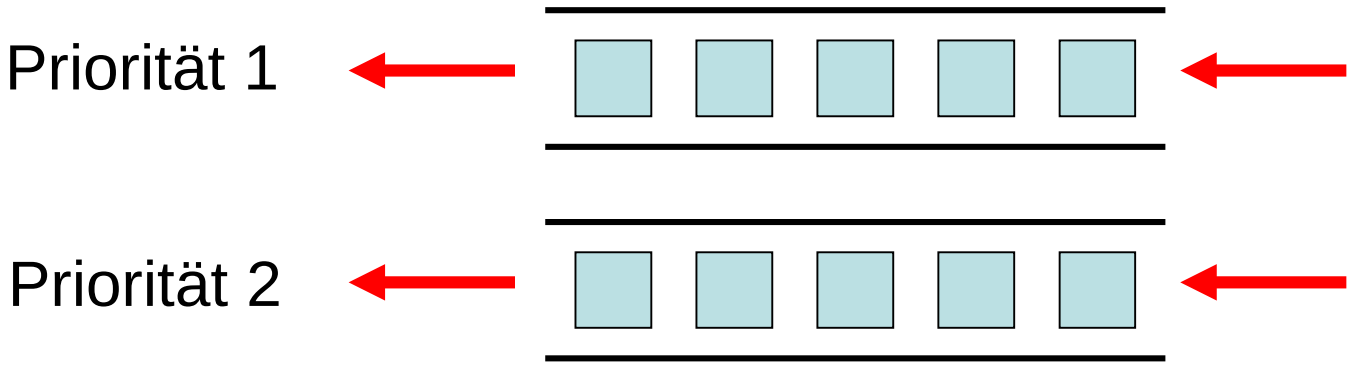
Verteilter Heap

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:

- Jedes Element x hat eine beliebige, eindeutige Priorität $prio(x) \in \{0,1\}^{c \cdot \log n}$ für eine vorgegebene Konstante c. (D.h. es gibt maximal polynomiell viele Elemente im System.)
- Größenbeschränkung der Prioritäten sinnvoll: sonst werden Nachrichten zu groß.
- Eindeutige Prioritäten: kann durch Ergänzung einer eindeutigen Info (Knoten-ID,…) erreicht werden.

- Der Anker merkt sich lediglich in Variable m die Anzahl der Elemente, die zurzeit im System sind.
- Insert und deleteMin Anfragen werden in separaten Phasen bearbeitet.

Verteilter Heap

Insert Phase:

- Jeder Knoten v mit einer Insert(H,x) operation schickt eine ins(H,1) Anfrage in Richtung des Ankers und führt eine put(key(x),x) Anfrage auf der verteilten Hashtabelle aus. Sonst schickt v lediglich eine ins(H,0) Anfrage in Richtung des Ankers.
- Die ins(H,i) Anfragen werden zu einer einzigen ins(H,i_0) Anfrage beim Anker wie für die serve Anfragen für den Stack aggregiert. (D.h. ein Knoten wartet mit der Weiterleitung, bis er eine Anfrage von allen Vorgängern hat.)
- Weiterhin schicken alle Knoten v, die gemäß der verteilten Hashtabelle verantwortlich für ein put(key(x),x) sind, bei Erhalt dieser put Anfrage eine putack(H,1) Anfrage in Richtung des Ankers. Diese werden wie bei der Queue zum Anker hin aggregiert (d.h. es wird keinen Wert auf eine einzige putack(H,j_0) Anfrage beim Anker gelegt).
- Sobald der Anker putack(H,j_1),…,putack(H,j_k) Anfragen erhalten hat mit $i_0=j_1+j_2+\ldots+j_k$, schickt er ein ACK zurück zu allen Knoten (welches durch den Aggregationsbaum vervielfältigt wird).
- Sobald ein Knoten v das ACK erhalten hat, weiß er, dass die Insert Phase erfolgreich beendet wurde.

Verteilter Heap

DeleteMin Phase:

- Jeder Knoten v mit einer deleteMin(H,x) operation schickt eine delmin(H,1) Anfrage in Richtung des Ankers. Sonst schickt v lediglich eine delmin(H,0) Anfrage in Richtung des Ankers.
- Die delmin(H,i) Anfragen werden zu einer einzigen delmin(H,i_0) Anfrage beim Anker wie für die serve Anfragen für den Stack aggregiert.
- Der Anker bestimmt daraufhin die i_0–kleinste Priotität p im System (welches die maximale Priorität ist, falls $i_0 \geq m$ ist) und schickt diese an alle Knoten (mittels des Aggregationsbaums).
- Bei Erhalt von p bestimmt Knoten v die Anzahl der in ihm gespeicherten Elemente $m(v)$ mit Priotität $\leq p$ und schickt pnum(H,m(v)) zurück an den Anker.
- Die pnum(H,i) Anfragen werden zu einer einzigen pnum(H,j_0) Anfrage zum Anker hin aggregiert (wobei $j_0 = \min\{i_0, m\}$ ist).
- Der Anker schickt daraufhin das Intervall $[1,j_0]$ zurück, welches rückwärts über den Aggregationsbaum gemäß der aggregierten $m(v)$-Werte so aufgeteilt wird, dass jeder Knoten v ein Intervall $[l(v),r(v)]$ der Größe $m(v)$ erhält.
- Jeder Knoten v führt daraufhin für jedes $i \in [1,m(v)]$ eine put(l(v)+i-1,x_i) Anfrage auf der verteilten Hashtabelle aus, wobei $\{x_1,\ldots,x_{m(v)}\}$ die Menge seiner Elemente mit Priorität $\leq p$ ist. Damit sind die j_0 Elemente nun für die deleteMin Anfragen abrufbar.

Verteilter Heap

DeleteMin Phase (Fortsetzung):

- Jeder Knoten v, der gemäß der verteilten Hashtabelle verantwortlich für ein put(i,x) ist, schickt bei Erhalt dieser put Anfrage eine putack(H,1) Anfrage in Richtung des Ankers aus. Diese werden wie bei der Queue zum Anker hin aggregiert (d.h. es wird keinen Wert auf einzige putack(H,j_0) Anfrage beim Anker gelegt).
- Sobald der Anker putack(H,j_1),…,putack(H,j_k) Anfragen erhalten hat mit $j_0=j_1+j_2+\ldots+j_k$, schickt er das Intervall $[1,j_0]$ zurück, welches rückwärts über den Aggregationsbaum gemäß der aggregierten delmin(H,i) Anfragen so aufgeteilt wird, dass jeder Knoten v mit einer deleteMin Operation ein eindeutiges Element $i(v)\in[1,j_0]\cup\{\perp\}$ erhält.
- Falls $i(v)\neq\perp$ ist, führt v eine get(i(v)) Anfrage auf der verteilten Hashtabelle aus. (Sonst erhält v kein Element, was passieren kann, wenn $i_0>m$ ist).
- Jeder Knoten v, der gemäß der verteilten Hashtabelle verantwortlich für ein get(i) ist, schickt bei Erhalt dieser get Anfrage eine getack(H,1) Anfrage in Richtung des Ankers aus. Diese werden wie bei der Queue zum Anker hin aggregiert (d.h. es wird keinen Wert auf einzige getack(H,j_0) Anfrage beim Anker gelegt). Weiterhin schickt v das x, das in ihm unter dem Schlüssel i gespeichert ist (siehe die put Anfragen oben) an den Knoten zurück, der die get(i) Anfrage gestellt hat (was damit seine deleteMin Operation abschließt).
- Sobald der Anker getack(H,j_1),…,getack(H,j_k) Anfragen erhalten hat mit $j_0=j_1+j_2+\ldots+j_k$, schickt er ein ACK zurück zu allen Knoten (welches durch den Aggregationsbaum vervielfältigt wird).
- Sobald ein Knoten v das ACK erhalten hat, weiß er, dass die deleteMin Phase erfolgreich beendet wurde.

Verteilter Heap

Noch zu klären: Bestimmung der k-kleinsten Priorität von m Prioritäten, wobei $m=O(poly(n))$ ist. Das machen wir mithilfe der verteilten k-Selektion.

Verteilte k-Selektion:

Der Algorithmus besteht aus 3 Phasen.

- Phase 1: Reduktion der Kandidaten für die k-kleinste Priorität von m auf $O(n^{3/2} \log n)$.
- Phase 2: Reduktion der Kandidaten auf $O(n^{1/2})$.
- Phase 3: Bestimmung der k-kleinsten Priorität aus den verbliebenen $O(n^{1/2})$ Elementen.

Verteilte k-Selektion

Phase 1: Reduktion der Kandidaten für die k-kleinste Priorität von $m=n^q$ auf $O(n^{3/2} \log n)$. Sei $k_1=k$ und $m_1=m$. Am Anfang sind alle Elemente aktiv.

for $i:=1$ to $\log q +1$ do

- Der Anker verschickt $k_i \in \{1,\ldots,m_i\}$ an alle Knoten.
- Jeder Knoten v berechnet die $\lfloor k_i/n \rfloor$-kleinste und die $\lceil k_i/n \rceil$-kleinste Priorität, $p_{min}(v)$ und $p_{max}(v)$, seiner aktiven Elemente.
- Die Knoten aggregieren diese Prioritäten in Richtung des Ankers, so dass der Anker am Ende $p_{min}=\min_{v \in V} p_{min}(v)$ und $p_{max}=\max_{v \in V} p_{max}(v)$ kennt.
- Der Anker verschickt daraufhin p_{min} und p_{max} an alle Knoten.
- Jeder Knoten v entfernt alle Elemente mit Prioritäten $<p_{min}$ oder $>p_{max}$ von der Menge seiner aktiven Elemente. Sei $l(v)$ die Anzahl der deaktivierten Elemente $<p_{min}$ und $r(v)$ die Anzahl der deaktivierten Elemente $>p_{max}$.
- Jeder Knoten v schickt $l(v)$ und $r(v)$ in Richtung des Ankers. Diese werden in Richtung des Ankers zu einer einzigen Nachricht aggregiert, so dass der Anker am Ende die Anzahl $l=\Sigma_{v \in V} l(v)$ und $r=\Sigma_{v \in V} r(v)$ kennt. Daraufhin setzt der Anker $m_{i+1}:=m_i-(l+r)$ und $k_{i+1}:=k_i-l$.

Verteilte k-Selektion

Lemma 6.7: Sei p die k-kleinste Priorität. Dann gilt zu jedem Zeitpunkt in Phase 1, dass $p_{min} \leq p \leq p_{max}$ ist.

Beweis: durch Induktion

- Induktionsanfang: Zu Beginn ist p die k-kleinste Priorität der Menge aller aktiven Elemente.
- Angenommen, $p_{min} > p$. Dann hat jeder Prozess v ein $p_{min}(v)$ gewählt mit $p_{min}(v) > p$.
- Das bedeutet dann aber, dass die Anzahl der aktiven Elemente mit Priorität $\leq p$ höchstens $(\lfloor k/n \rfloor - 1) \cdot n \leq (k/n - 1) \cdot n < k$ ist, was ein Widerspruch ist.
- Angenommen, $p_{max} < p$. Dann hat jeder Prozess v ein $p_{max}(v)$ gewählt mit $p_{max}(v) < p$.
- Das bedeutet dann aber, dass die Anzahl der aktiven Elemente mit Priorität $< p$ mindestens $\lceil k/n \rceil \cdot n \geq (k/n) \cdot n = k$ ist, was auch ein Widerspruch ist.
- Also ist im ersten Schleifendurchlauf $p_{min} \leq p \leq p_{max}$ und wegen $k_2 := k - l$ Priorität p die k_2-kleinste Priorität der noch verbliebenen Elemente.
- Induktionsschluss: Zu Beginn des i-ten Schleifendurchlaufs sei p die k_i-kleinste Priorität der Menge aller aktiven Elemente. Der Beweis ist dann analog zum Induktionsanfang.

Verteilte k-Selektion

Lemma 6.8: Am Ende von Phase 1 sind nur noch $O(n^{3/2} \log n)$ aktive Elemente übrig.

Beweisskizze:

- Im Erwartungswert besitzt jeder Knoten v m/n viele Elemente, und die $\lfloor k/n \rfloor$-kleinste und die $\lceil k/n \rceil$-kleinste Priorität bilden global gesehen ungefähr die k-kleinste Priorität.
- Die Abweichungen von der k-kleinsten Priorität sind in der Tat genügend gering, dass mit hoher Wahrscheinlichkeit p_{min} mindestens die $k-O((nm \log n)^{1/2})$ kleinste Priorität ist und p_{max} höchstens die $k+O((nm \log n)^{1/2})$ kleinste Priorität ist.
- Es verleiben also nach dem ersten Schleifendurchlauf nur $m_2=O((nm \log n)^{1/2})=O(n^{(q+1)/2} (\log n)^{1/2})$ aktive Elemente.
- Induktiv kann dann gezeigt werden, dass $m_{i+1}=O(n^{1+q/2\text{^}i} (\log n)^{1/2})$ ist.
- Damit sind nach $\log q+1$ Durchläufen nur $O(n^{3/2} \log n)$ aktive Elemente übrig.

Verteilte k-Selektion

Sei p die k-kleinste Priorität unter den Prioritäten der verbliebenen $m=O(n^{3/2} \log n)$ (aktiven) Elementen.

Phase 2:

- Der Anker schickt k und m an alle Knoten.
- Jeder Knoten v entscheidet mit Wahrscheinlichkeit $n^{1/2}/m$ für jedes seiner (aktiven) Elemente unabhängig von den anderen Elementen, ob dessen Priorität ein Splitter wird oder nicht.
- Die Knoten aggregieren die Anzahl der Splitter zum Anker, so dass der Anker am Ende die Gesamtzahl s der Splitter kennt. Es kann gezeigt werden, dass mit hoher Wahrscheinlichkeit $s=\Theta(n^{1/2})$ ist.
- Der Anker initiiert die Sortierung der s Splitter (welche wir später beschreiben werden), so dass am Ende jeder Splitter x seinen Rang $r(x)\in\{1,\ldots,s\}$ in der sortierten Splitterfolge kennt.
- Die Splitter x_1 und x_2 mit Rängen $r(x_1)=k\cdot(s/m)-\delta$ und $r(x_2)=k\cdot(s/m)+\delta$ mit $\delta=\Theta(n^{1/4}(\log n)^{1/2})$ informieren den Anker über sich.
- Der Anker schickt dann x_1 und x_2 an alle Knoten, welche alle Elemente deaktivieren, deren Prioritäten $<x_1$ oder $>x_2$ sind.
- Die Anzahl der deaktivierten Elemente für jeden Knoten v, $l(v)$ und $r(v)$, wird zum Anker hin aggregiert, so dass der Anker am Ende $l=\Sigma_{v\in V} l(v)$ und $r=\Sigma_{v\in V} r(v)$ kennt. Daraufhin setzt der Anker k auf $k-l$ und m auf $m-(l+r)$.

Verteilte k-Selektion

Lemma 6.9: In Phase 2 sind x_1 und x_2 mit hoher Wahrscheinlichkeit so gewählt, dass $l<k$ und $r>m-k$ ist (und somit die gesuchte Priorität innerhalb der aktiven Elemente ist).

Beweis: verwendet Chernoff Schranken.

Lemma 6.9: Phase 2 reduziert die Anzahl der aktiven Elemente von m auf $O((m \log n)/n^{1/2+\delta})$.

Beweis: verwendet Chernoff Schranken.

Nach höchstens 3 Anwendungen von Phase 2 haben wir also nur noch $m=O(n^{1/2})$ aktive Elemente.

Phase 3 (Bestimmung der k-kleinsten Priorität aus den verbliebenen $O(n^{1/2})$ Elementen): kann reduziert werden auf verteilte Sortierung von $O(n^{1/2})$ Elementen (was wir bereits in Phase 2 benötigen).

Verteiltes Sortieren

Gegeben: Menge S von $m=n^{1/2}$ Elementen $x_1,\ldots,x_m$, verteilt über die Knoten so dass jeder Knoten maximal ein Element besitzt.

Gesucht: Rang $r(x)$ von jedem Element $x \in S$.

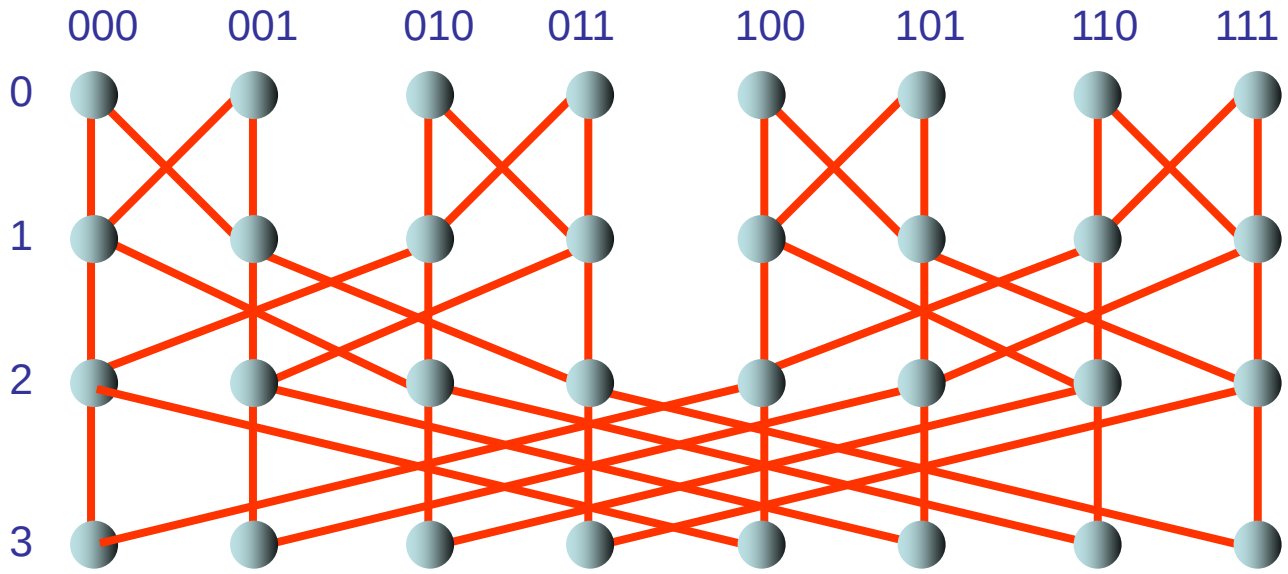
- Phase 0: route jedes x_i zu einem zufällig ausgewählten Knoten (damit sich die x_i's gleichmäßig verteilen).
- Phase 1: erzeuge m Kopien $x_{i,j}$ von jedem Element $x_i \in S$, $1 \leq j \leq m$, so dass jeder Knoten maximal $O(\log n)$ Kopien besitzt (wird noch erläutert).
- Phase 2: Route jede Kopie zum Knoten, der für $h(i,j)$ verantwortlich ist, wobei $h:\{1,\ldots,m\}^2 \rightarrow [0,1)$ eine pseudo-zufällige Hashfunktion ist mit $h(i,j)=h(j,i)$. Dadurch treffen sich die Elemente $x_{i,j}$ und $x_{j,i}$.
- Phase 3: Route die Kopien rückwärts, so dass am Ende $x_{j,i}$ am Startpunkt von $x_{i,j}$ in Phase 2 ist und $x_{i,j}$ am Startpunkt von $x_{j,i}$ in Phase 2.
- Phase 4: Falls $x_{i,j}<x_{j,i}$, dann schicke ein $sum(i,0)$ in Richtung des Startpunkts von x_i in Phase 1 und sonst ein $sum(i,1)$ in Richtung des Startpunkts von x_i in Phase 1. Aggregiere die Werte für jedes i (in timeout), so dass der Startpunkt von x_i in Phase 1 eine einzige aggregierte Nachricht $sum(i,r)$ mit dem Rang r von x_i erhält.

Verteiltes Sortieren

Illustration der Phasen: Bufferfly (=ausgerollter Hypercube).

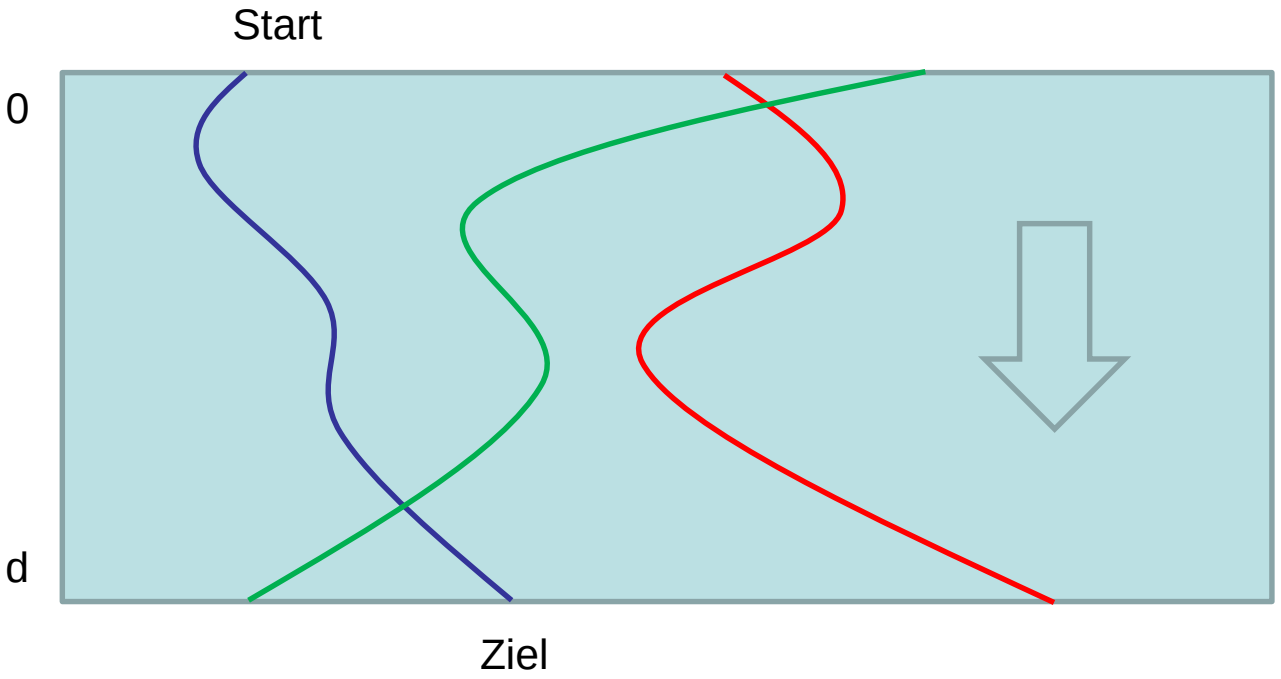
Zur Erinnerung: jeder Knoten in Ebene 0 besitzt Binärbaum zu allen Knoten in Ebene d.

Annahme: Knoten Spalte i werden durch Prozess i emuliert.



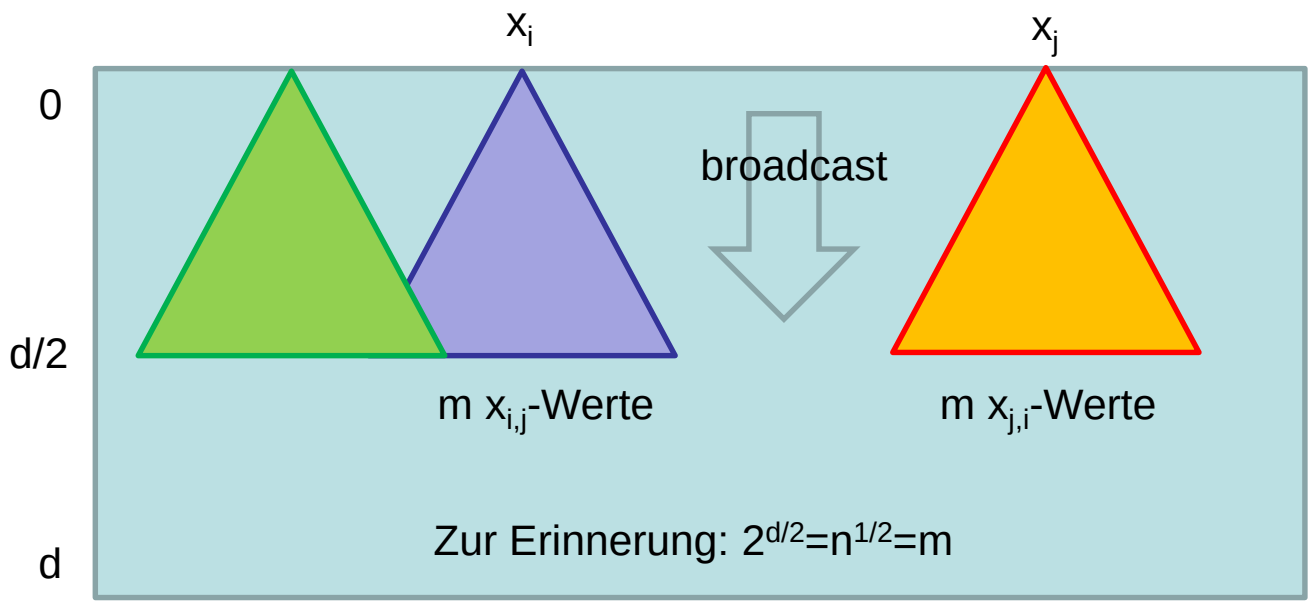
Verteiltes Sortieren

Phase 0: route jedes x_i zu einem zufällig ausgewählten Knoten (damit sich die x_i‘s gleichmäßig verteilen).



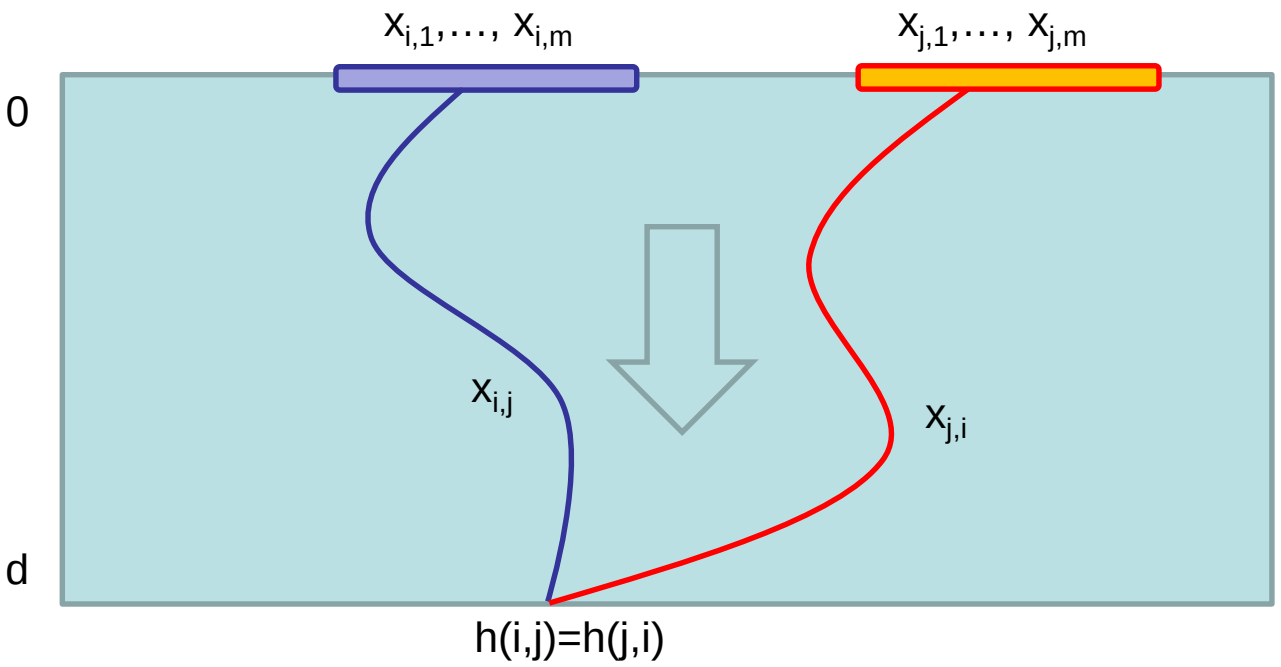
Verteiltes Sortieren

Phase 1: erzeuge m Kopien $x_{i,j}$ von jedem Element $x_i \in S$, $1 \leq j \leq m$, so dass jeder Knoten maximal O(log n) Kopien besitzt.



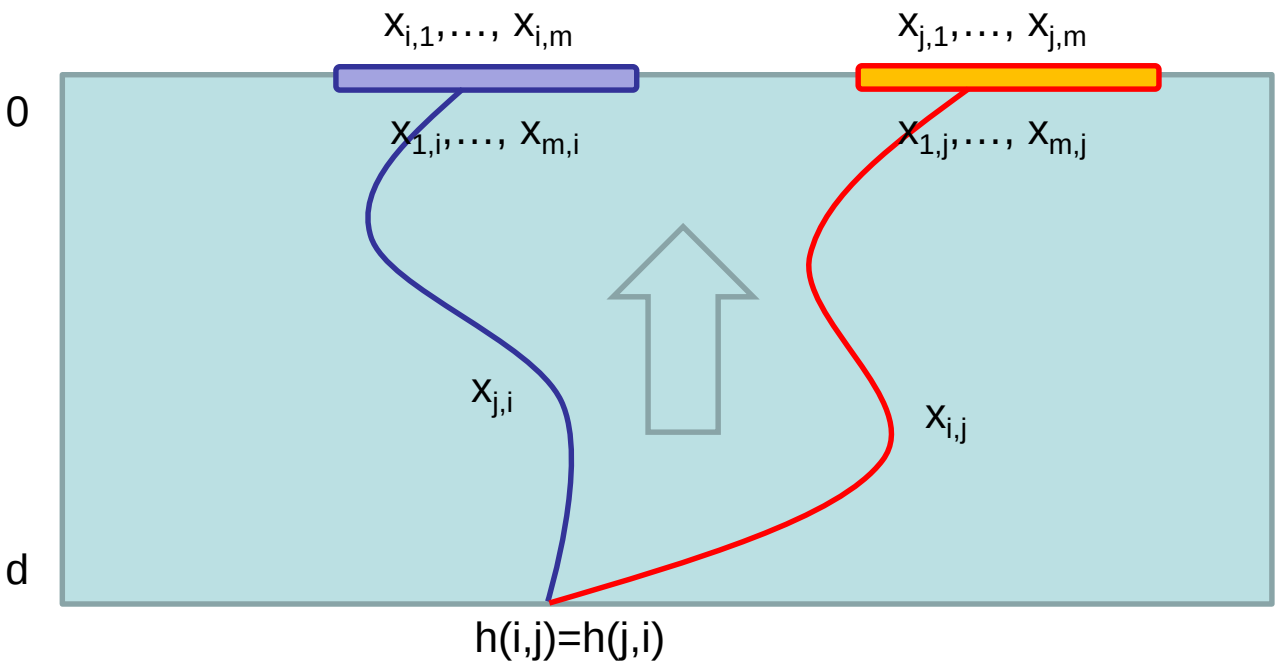
Verteiltes Sortieren

Phase 2: Route jede Kopie zum Knoten, der für $h(i,j)$ verantwortlich ist, wobei $h:\{1,\ldots,m\}^2 \rightarrow [0,1)$ eine pseudo-zufällige Hashfunktion ist mit $h(i,j)=h(j,i)$. Dadurch treffen sich die Elemente $x_{i,j}$ und $x_{j,i}$.



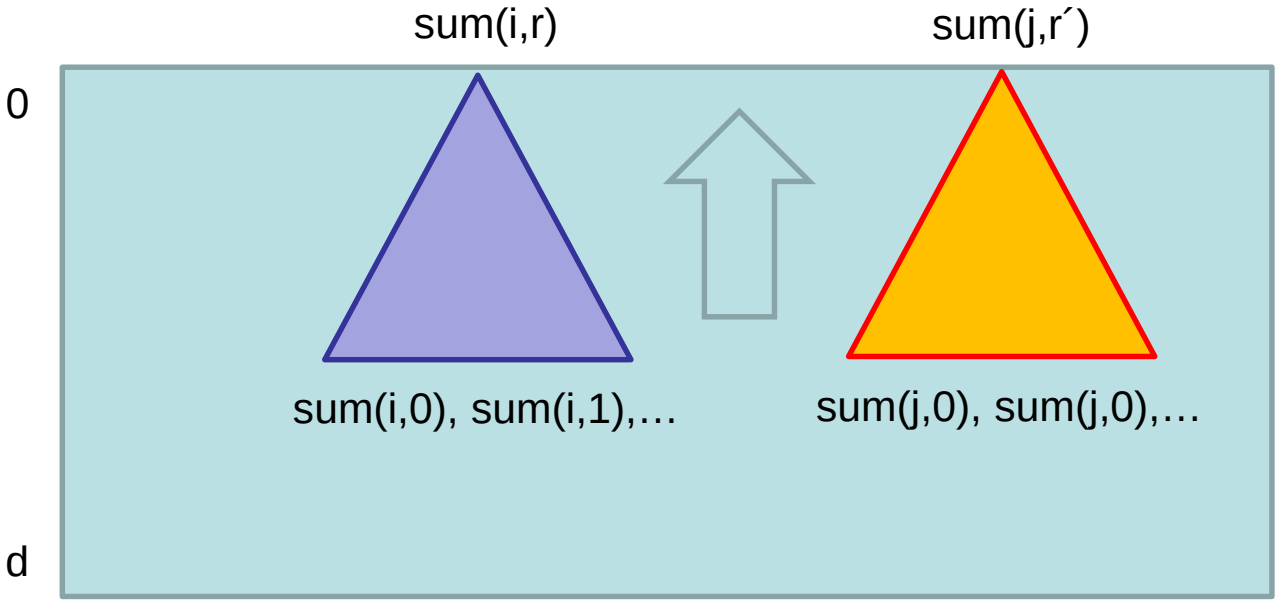
Verteiltes Sortieren

Phase 3: Route die Kopien rückwärts, so dass am Ende $x_{j,i}$ am Startpunkt von $x_{i,j}$ in Phase 2 ist und $x_{i,j}$ am Startpunkt von $x_{j,i}$ in Phase 2.



Verteiltes Sortieren

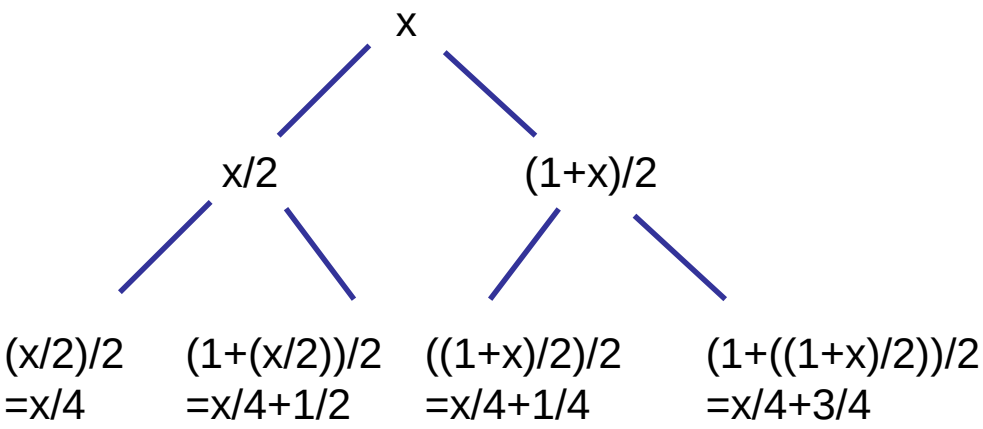
Phase 4: Falls $x_{i,j}<x_{j,i}$, dann schicke ein sum(i,0) in Richtung des Startpunkts von x_i in Phase 1 und sonst ein sum(i,1) in Richtung des Startpunkts von x_i in Phase 1. Aggregiere die Werte für jedes i, so dass der Startpunkt von x_i in Phase 1 eine Nachricht sum(i,r) mit dem Rang r von x_i erhält.



Verteiltes Sortieren

Statt eines Bufferflies kann auch ein de Bruijn Graph verwendet werden (für den wir bereits eine dynamische Version kennen):

- Routing zu zufälligem Knoten: Bitshifting-Strategie
- Broadcasting: verwende folgenden Baum



Tiefe i: Positionen $x/2^i + j/2^i$ für alle $j \in \{0,\ldots,2^i-1\}$.

Verteilter Heap

Zur Erinnerung: wir müssen die Insert und deleteMin Anfragen in separaten Insert- und deleteMin-Phasen bearbeiten.

$\rightarrow$Dadurch ist keine so massive Parallelität erreichbar wie mit den serve Anfragen, es sei denn, wir benötigen keine sequentielle Konsistenz sondern sind zufrieden mit Linearisierbarkeit. (Warum?)

Gibt es eine alternative Lösung, für die eine massive Parallelität möglich ist?

Selbststabilierung: einfacher als bei der Queue oder dem Stack, da der Anker lediglich wissen muss, wieviele Elemente zurzeit im System sind, was er durch ein einfaches Zählen am Anfang der deleteMin-Phase erreichen kann.

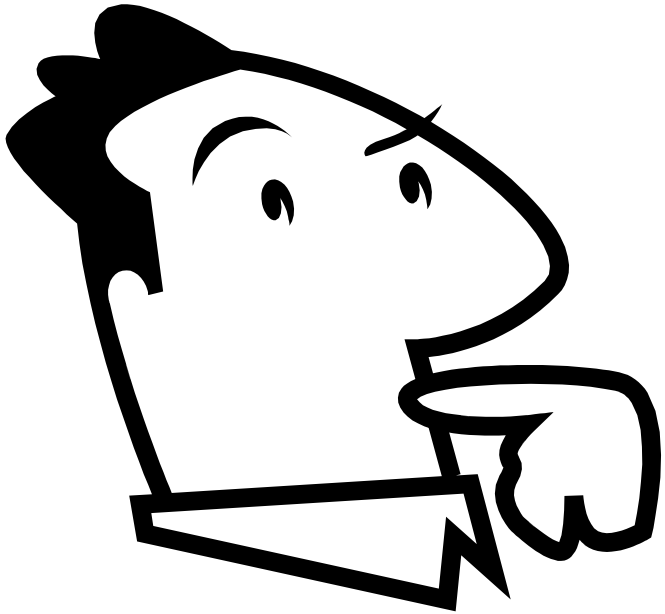
Verteilter Heap

Monotone Korrektheit: hier müssen wir den Skip+-Graphen verwenden, aber dann sollte es (mit einiger Mühe) umsetzbar sein.

Auch hier bieten sich interessante Software-projekte an.

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?