Vorgelegt von

Harald Brunner
harald.brunner@campus.lmu.de

Sebastian Straub
s.straub@campus.lmu.de

Bachelor Medieninformatik

WS 2011/12

Seminar: *Effiziente Funktionale Datenstrukturen*

Veranstalter: Prof. Martin Hofmann
Betreuer: Ulrich Schöpp

LFE Theoretische Informatik
LMU München

**Hausarbeit**

# Lazy Rebuilding

Harald Brunner
Sebastian Straub

14.03.2012

# Inhaltsverzeichnis

# Abstract

Rebuilding stellt, in einer dies unterstützenden Datenstruktur, einen Zustand der Balance wieder her. Dies kann, abhängig von der Datenstruktur, die Performanz von Operationen drastisch erhöhen.

Im Folgenden sollten drei aufeinander aufbauende Methodiken des Rebuildings erläutert werden. Konkrete Implementierungsbeispiele sind in Standard ML gegeben.

Zunächst das Batched Rebuilding, welches den Rebuildingvorgang verzögert und nach einer gewissen Zahl an Operationen die perfekte Balance wiederherstellt. Anschließend das Global Rebuilding, welches das Rebuilding nebenläufig auf einer Kopie der Datenstruktur durchführt. Schließlich folgt das Lazy Rebuilding in zwei Varianten, die jeweils mit dem Batched und Global Rebuilding verwandt sind, aber die Vorteile der Lazy Evaluation nutzen.

Die Ausarbeitung basiert auf Kapitel 8 von Chris Okasakis Buch „*Purely Functional Data Structures*“.

# 1 Batched Rebuilding

Da in den meisten Fällen das Herstellen von perfekter Balance nach jedem Update zu teuer ist, müssen, um gute Laufzeiten zu erzielen, Kompromisse eingegangen werden. Es lässt sich hier die Qualität oder die Quantität des Rebuildings verringern.

Ein Beispiel für eine Qualitätsverringerung wäre das lokale Rebuilding in AVL- und Red-Black-Trees. Demgegenüber steht die Quantitätsverringerung, welche beim Batched Rebuilding verwendet wird, hierbei wird das Rebuilding nur nach einer abgeteilten Menge ("batch") von Updates durchgeführt.

## 1.1 Einschränkungen

Um gute amortisierte Laufzeitschranken zu erreichen, sind zwei Bedingungen einzuhalten:

1. Rebuilding passiert nicht zu oft.
2. Einzelne Updates degenerieren die Performanz der folgenden Operationen nicht zu stark.

#### Bedingung 1 - Häufigkeit

Bei angestrebter amortisierter Schranke $O(f(n))$ und Rebuilding-Zeitkomplexität $O(g(n))$ darf das Rebuilding für eine beliebige Konstante $c$ nicht öfter als alle $c \cdot g(n) \div f(n)$ Operationen stattfinden. Hier werden die Kosten der teuren Rebuilding-Operation auf die billigen Operationen dazwischen verteilt.

Zum Beispiel wird in binären Suchbäumen für gängige Operationen gewöhnlich eine Zeitkomplexität von $O(\log n)$ angestrebt, wobei das Rebuilding eine Zeitkomplexität von $O(n)$ hat, damit kann der Baum alle $c \cdot n \div \log n$ Operationen überholt werden.

#### Bedingung 2 - Performanzdegeneration

Nach bis zu $c \cdot g(n) \div f(n)$ Updates müssen einzelne Operationen immer noch eine Zeitkomplexität von $O(f(n))$ aufweisen. Dies erlaubt nur eine Perfor-

manzdegeneration um einen konstanten Faktor (da für eine Konstante $c$ gilt: $O(c \cdot n) = O(n)$).

Updates, welche diese Bedingung erfüllen, werden Weak Updates genannt.

## 1.2 Beispiele

Zur Illustration dieser Bedingungen hier nun zwei Beispiele.

### Einfügen in Suchbäume

Hier ist die Krux die Performanzdegeneration, welche im schlechtesten Fall stattfindem kann. Werden geordnete Elemente eingefügt, entsteht ohne Balancing eine verkettete Liste; die Einfügeoperationen haben eine Zeitkomplexität von $O(n)$, wobei sie eine Zeitkomplexität von $O(\log n)$ halten müssten, um Bedingung 2 zu erfüllen.

### Löschen aus Suchbäumen

Um Batched Rebuilding beim Löschen anzuwenden, können gelöschte Knoten zuerst als gelöscht markiert werden, ohne diese wirklich zu entfernen. Nachdem die Hälfte der Knoten auf diese Weise gelöscht wurde, wird der Baum ohne die gelöschten Knoten neu aufgebaut. Das Löschen sollte eine amortisierte Zeitkomplexität von $O(\log n)$ erreichen, das Rebuilding hat Zeitkomplexität $O(n)$.

Das Rebuilding findet immer nach $\frac{1}{2}n$ Löschungen statt, wenn $n$ die Anzahl der Knoten im Baum ist. Nach Bedingung 1 darf das Rebuilding alle $c \cdot n \div \log n$ Operationen stattfinden, also noch öfter als $\frac{1}{2}n$.

Die Pfadlänge der noch existierenden Knoten ist durchschnittlich nur um eins länger ist als die, welche vorliegen würde, wenn die gelöschten Knoten tatsächlich entfernt worden wären. Dies entspricht also einer Performanzdegeneration um einen konstanten additiven Faktor, erlaubt wäre sogar ein multiplikativer Faktor.

Beide Bedingungen sind damit erfüllt, also könnte Batched Rebuilding gut für dieses Löschverfahren angewendet werden.

## 1.3 Persistenz

Bei persistenter Nutzung von Datenstrukturen, welche Batched Rebuilding verwenden, steigt die amortisierte Zeitkomplexität zur Zeitkomplexität der Rebuilding-

Operation an, da sich die Struktur in einem Zustand vor dem Rebuilding befinden kann, und dieses aufgrund der Persistenz beliebig oft ausgelöst werden kann.

# 2 Global Rebuilding

Das Global Rebuilding eliminiert die Amortisation des Batched Rebuildings, indem nicht die Kosten des Rebuildings auf die anderen Operationen aufgeteilt werden, sondern die Arbeit selbst. Das Rebuilding wird inkrementell zusammen mit anderen Operationen ausgeführt.

## 2.1 Vorgang

Die Datenstruktur enthält zwei Kopien der Daten, eine Working Copy, diese wird für Anfragen während des Rebuildings verwendet, und eine Secondary Copy, in welcher das Rebuilding stattfindet. Es kann auch, wenn gerade nichts zu tun ist, keine Secondary Copy geben. Am Ende eines Rebuilding-Zykluses löst die neu aufgebaute Struktur aus der Secondary Copy die Working Copy ab.

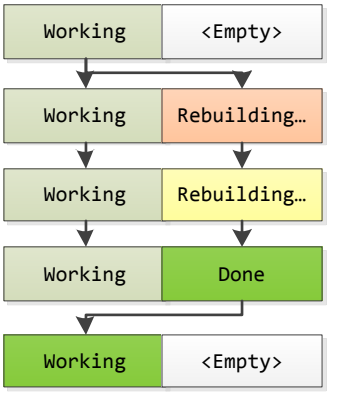


Abbildung 2.1: Schematische Darstellung des Prozesses

## 2.2 Folgen

Dadurch, dass keine Operation besonders lange dauert, liefert Global Rebuilding gute Worst-Case-Laufzeitschranken, und die persistente Nutzung wird ermöglicht.

Dieses System macht das Rebuilding jedoch komplizierter. Der Prozess muss so geplant werden, dass er abgeschlossen ist, bevor die neu aufgebaute Struktur benötigt wird, oder ein neues Rebuilding beginnen sollte. Zudem muss der Prozess als Zustandsfolge repräsentierbar gemacht werden, damit dieser nach und nach ausgeführt werden kann.

Updates werden auch problematisch, weil die Working Copy am Ende entfernt wird und natürlich Updates, welche während des Rebuildings stattgefunden haben, nicht verloren gehen sollten. Deshalb müssen Updates auch auf die Secondary Copy angewendet werden, was oft schwierig ist, da sich diese gerade verändert. Eine Lösung dieses Problems besteht darin, Updates für die Secondary Copy zu puffern und am Ende des Rebuildings anzuwenden, bevor die Working Copy abgelöst wird.

## 2.3 Beispiel: Hood-Melville Real-Time Queues

Bei diesen Schlangen können hinten Elemente angehängt und vorne weggenommen werden. Um dies zu ermöglichen wird die Schlange in zwei Listen aufgeteilt: eine repräsentiert den Anfang, eine das Ende, wobei das letzte Element der Schlange das erste dieser Liste ist. Die Struktur ist genau dann ausgewogen, wenn alle Elemente am Anfang sind, wo sie einfach weggenommen werden können, das Anhängen ist immer möglich.

Sobald die hintere Liste länger ist als die vordere, soll nun das Rebuilding durchgeführt werden, in welchem alle Elemente von hinten nach vorne rotiert werden.

### 2.3.1 Rebuilding

Um diese Rotation durchzuführen, müssen zuerst beide Listen umgedreht werden, damit die aneinander liegenden Elemente in der Mitte der Liste erreichbar werden. Dann wird die vordere Liste umgedreht an die hintere angehängt, wodurch die ganze Schlange dann geordnet in einer Liste vorliegt.

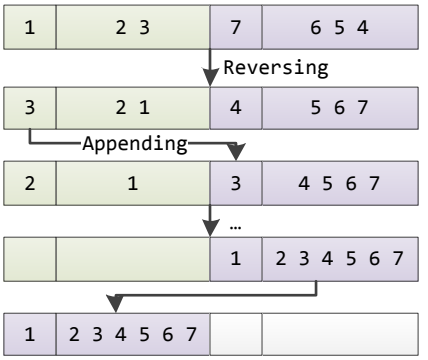


Abbildung 2.2: Beispiel für das Rebuilding von Hood-Melville Real-Time Queues

Um eine Liste umzudrehen, nimmt man eine leere Liste hinzu und hängt alle Elemente schrittweise an diese, bis die ursprüngliche Liste leer ist. Dies lässt sich als Zustand ausdrücken, damit die Listen inkrementell umgedreht werden können.

```
datatype 'a ReverseState =
    Working of 'a list * 'a list
  | Done    of 'a list
fun startReverse xs = Working(xs,[])
fun exec (Working(x::xs,xs')) = Working(xs,x::xs')
  | exec (Working([],xs'))     = Done xs'
```

Die `exec`-Funktion führt immer einen Schritt der Umkehrung aus, und geht am Ende in den `Done`-Zustand über. Dieser Zustand dient der Rückgabe des Ergebnisses und der Signalisierung, dass der Vorgang beendet ist.

Da bei diesen Schlangen zwei Listen gleichzeitig umgedreht werden müssen, erweitert man den Datentyp um weitere Felder. Zudem wird der Zustand `Appending` benötigt, welcher nach dem `Reversing` Elemente von der einen Liste zur anderen rotiert.

```
datatype 'a RotationState =
    Reversing of 'a list * 'a list * 'a list * 'a list
  | Appending of 'a list * 'a list
  | Done      of 'a list
fun startRotation (f,r) = Reversing(f, [], r, [])
fun exec (Reversing(x::f,f',y::r,r')) = Reversing(f,x::f',r,y::r')
  | exec (Reversing([],r,[y],r')) = Appending(f',y::r')
  | exec (Appending(x::f',r')) = Appending(f',x::r')
  | exec (Appending([],r')) = Done r'
```

Das zweite `exec`-Pattern ergibt sich aus der Rebuilding-Bedingung: Wenn das

Rebuilding stattfindet, ist in der hinteren Liste ein Element mehr als in der vorderen.

### 2.3.2 Zeitplanung

Da das Rebuilding zusammen mit den anderen Operation durchgeführt wird, und dieses rechtzeitig fertig sein muss, gilt es, sich zu überlegen, wann und wie oft `exec` aufgerufen werden muss.

Wenn $m$ die Länge der vorderen Liste ist, dann ist die Working Copy nach $m$ Aufrufen von `tail` leer, das heißt nach höchstens $m$ Schritten muss das Rebuilding abgeschlossen sein, damit wieder Elemente verfügbar sind.

Im obigen Code wird das erste Pattern von `exec` genau $m$ mal aufgerufen, da dann die erste Liste leer ist. Das zweite Pattern wird ein mal aufgerufen, das dritte Pattern wieder $m$ mal, und das letzte wieder einmal. Insgesamt sind also $2m + 2$ `exec`-Aufrufe nötig.

Um den rechtzeitigen Abschluss des Rebuildings zu gewährleiten, kann man beim Beginn des Rebuildings zwei `exec`-Aufrufe tätigen und zudem jeweils zwei bei jedem weiteren `snoc` oder `tail`-Aufruf.

### 2.3.3 Updates

Neben der Zeitplanung müssen auch Updates der Secondary Copy beachtet werden. Die folgenden Operationen sind möglich:

- `snoc`: Fügt ein Element am Ende an. Da das Rebuilding eine neue Anfangsliste aufbaut, kann man zu Beginn die hintere Liste leeren und dann einfach nach dem Rebuilding übernehmen.
- `head`: Gibt das erste Element der Schlange zurück. Diese Operation verändert die Struktur nicht, ist also nicht relevant.
- `tail`: Entfernt das erste Element. Dies betrifft die erste Liste, also muss das Element auch aus der Secondary Copy entfernt werden.

Es muss nur `tail` für die Secondary Copy implementiert werden. Man könnte diese Updates puffern, indem man mitzählt wie viele Elemente entfernt wurden, um diese dann nach dem Rebuilding zu entfernen. Es lässt sich hier aber auch einfacher lösen:

Da die beiden Listen umgedreht werden, sind die Anfangselemente am Ende der einen Liste, bevor sie zur anderen rotiert werden. Deshalb kann man einfach aufhören, Elemente zu rotieren, wenn diese entfernt wurden. Man zählt also stattdessen, wie viele Element noch gültig sind, und bricht ab, sobald dieser Zähler Null erreicht.

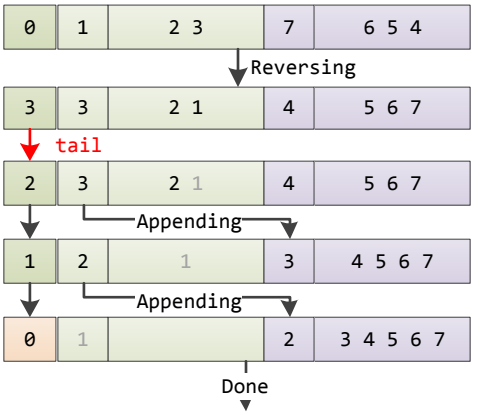


Abbildung 2.3: Beispiel für eine Entfernung

Der um dieses Update erweiterte Code sieht dann so aus:

```
datatype 'a RotationState =
    Idle
  | Reversing of int * 'a list * 'a list * 'a list * 'a list
  | Appending of int * 'a list * 'a list
  | Done of 'a list
fun exec (Reversing(ok,x::f,f',y::r,r')) = Reversing(ok+1,f,x::f',r,y::r')
  | exec (Reversing(ok,[],f',[y],r')) = Appending(ok,f',y::r')
  | exec (Appending(0,f',r')) = Done r'
  | exec (Appending(ok,x::f',r')) = Appending(ok-1,f',x::r')
  | exec state = state

fun invalidate (Reversing(ok,f,f',r,r')) = Reversing(ok-1,f,f',r,r')
  | invalidate (Appending(0,f',x::r')) = Done r'
  | invalidate (Appending(ok,f',r')) = Appending(ok-1,f',r')
  | invalidate state = state
```

Die `invalidate`-Funktion entfernt ein Element, indem entweder der Zähler verringert wird, oder, falls der Zähler schon Null erreicht hat, die bisherige Liste ohne das erste Element zum Ergebnis macht.

### 2.3.4 Vollständige Implementierung

Die bisher behandelten Punkte decken die wichtigsten Aspekte des Rebuildings ab. Die komplette Implementierung, mit Datentyp der gesamten Struktur und allen Details der Funktionsaufrufe, liegt im Anhang vor.

# 3 Lazy Rebuilding

Lazy Rebuilding ist eine Technik zum Balancieren von Datenstrukturen, welche - je nach Ausprägung - auf den eben vorgestellten Rebuildingstrategien Batched oder Global Rebuilding basiert, sich jedoch die in vielen funktionalen Sprachen verfügbare Technik der Lazy Evaluation zu Nutze macht.

## 3.1 Grundlagen

Der Prozess des Rebuilding wird weiterhin durch das Erreichen eines für die Datenstruktur ungünstigen Zustands ausgelöst. Dieser wird z.B. festgestellt durch das Überprüfen der Gültigkeit einer Invariante, sobald eine Operation ausgeführt wird, welche die Datenstruktur verändert.

Die Ausführung des Rebuildings erfolgt im Gegensatz zu den zuvor vorgestellten Methoden jedoch nicht sofort bzw. pseudoparallel, sondern die ganze Operation wird zunächst vollständig suspendiert. Sobald vom Rebuilding betroffenen Daten benötigt werden, wird der Vorgang in einem Zug ausgeführt. Eine Alternative dazu ist das Umwandeln des eigentlich monolithischen Rebuildingprozesses in eine inkrementelle Befehlsfolge, die - ähnlich zum Global Rebuilding - pseudoparallel evaluiert werden kann.

| | Sequenziell | Nebenläufig |
|---|---|---|
| verwandt mit | Batched Rebuilding | Global Rebuilding |
| Amortisierte Laufzeit | $O(1)$ | $O(1)$ |
| Worst-Case Laufzeit | $O(n)$ | $O(1)$ |
| Implementierungen | Banker's Queue | Real-Time Deque |

Tabelle 3.1: Varianten des Lazy Rebuilding

## 3.2 Minimalbeispiel

Im Folgenden soll das Lazy Rebuilding anhand der *Banker's Queue* vorgestellt werden, eine übersichtliche Implementierung einer Warteschleife (Queue), welche Lazy Rebuilding einsetzt.

### 3.2.1 Banker's Queue

Die Datenstruktur besteht aus zwei Streams ($f$ront und $r$ear) mit den Operationen *head*, *tail* und *cons*, welche jeweils in konstanter Zeit ausgeführt werden können. Da die hintere Liste $r$ logisch als umgedreht betrachtet wird, können wir mit der *snoc*-Operation (welche praktisch nur ein *cons* auf der hinteren Liste ist) Elemente am Ende der Liste einfügen.

Damit die Elemente aus $r$ auch wieder gelesen werden können, muss $r$ umgedreht und an das Ende von $f$ angehängt werden, spätestens wenn das erste Element aus $r$ angefordert wird. Die Queue befindet sich also in einem idealen Zustand, wenn die hintere Liste leer ist, also komplett an die vordere Liste angeschlossen wurde.

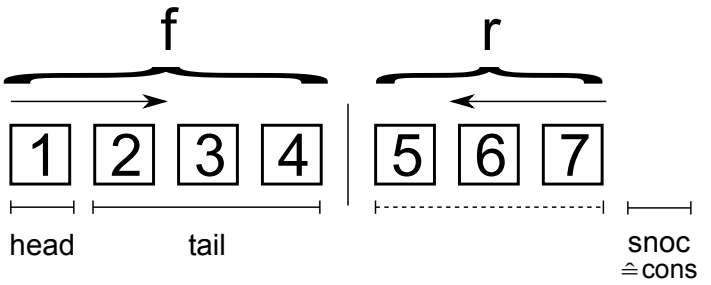


Abbildung 3.1: Aufbau Banker's Queue

### 3.2.2 Rebuilding

Für die Banker's Queue gilt die Invariante

$$|f| \geq |r|$$

Die hintere Liste soll also umgedreht werden, sobald sie die Länge der vorderen Liste erreicht hat. Da wir aber mit Streams arbeiten, wird das Rebuilding nicht

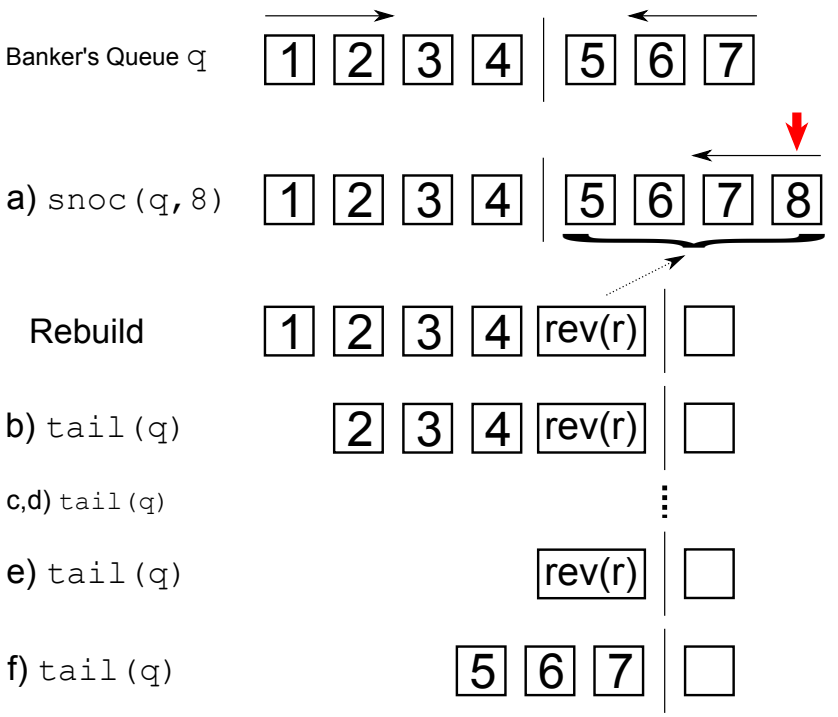


Abbildung 3.2: Rebuilding der Banker's Queue - Worst Case

sofort ausgeführt, sondern vollständig suspendiert. Die Evaluierung erfolgt tatsächlich erst, wenn in den Bereich von $r$ hineingelesen wird. Dies kann frühestens nach $|f|$ tail-Operationen der Fall sein.

Im Folgenden soll ein Rebuildingprozess beispielhaft nachverfolgt werden. Abbildung 3.2 zeigt das Worst-Case-Szenario (nur Ausführung von *tail* nach der Suspendierung des Rebuilding), die Implementierung der Datenstruktur in SML wird in Algorithmus 3.1 gezeigt.

Sobald `snoc(q,8)` ausgeführt wurde, wird die Invariante überprüft (siehe *check*-Funktion) und `f ++ reverse r` suspendiert. Nachfolgend werden vier *tail*-Operationen in konstanter Zeit ausgeführt, die *reverse*-Funktion bleibt weiterhin suspendiert. Beim fünften *tail* wird die suspendierte Funktion schließlich evaluiert und die monolithische Funktion *reverse* wird vollständig ausgeführt, die Laufzeit für diese Operation liegt bei $O(n)$.

Wenden wir die Banker-Methode an, lässt sich leicht feststellen, dass die amortisierte Laufzeit $O(1)$ beträgt, obwohl beim Rebuilding einmalig $O(n)$ erforderlich ist.

**Algorithmus 3.1** Implementierung Banker's Queue

```
structure BankersQueue : QUEUE =
struct
  type 'a Queue = int * 'a Stream * int * 'a Stream

  val empty = (0, $Nil, 0, $Nil)
  fun isEmpty (lenf, _, _, _) = (lenf = 0)

  fun check (q as (lenf, f, lenr, r)) =
        if lenr <= lenf
           then q
           else (lenf+lenr, f ++ reverse r, 0, $Nil)

  fun snoc ((lenf, f, lenr, r), x) =
        check (lenf, f, lenr+1, $(Cons (x, r)))

  fun head (lenf, $Nil, lenr, r) = raise Empty
    | head (lenf, $(Cons (x, f')), lenr, r) = x
  fun tail (lenf, $Nil, lenr, r) = raise Empty
    | tail (lenf, $(Cons (x, f')), lenr, r) =
        check (lenf-1, f', lenr, r)
end
```

Wird das Rebuilding ausgelöst, können weiterhin noch mindestens $|f|$ Schritte ausgeführt werden, bevor die teure Rotation, bestehend aus $|r|$ Schritten, fällig ist. Da aufgrund der Invariante $|f| \geq |r|$ garantiert ist, genügt es, für jede strukturverändernde Operation zwei Guthaben anzusparen, wobei eins sofort wieder ausgegeben wird. Nach $|f|$ Schritten haben also wir bereits genug Guthaben, um `reverse r` vollständig zu bezahlen.

### 3.2.3 Bewertung

Die Banker's Queue ist eine einfache Realisierung einer Warteschlange, welche Lazy Rebuilding verwendet.

Die amortisierte Laufzeit liegt bei $O(1)$, jedoch wird das Rebuilding weiterhin zu einem bestimmten Zeitpunkt in einem Zug durchgeführt, da es monolithische Operationen beinhaltet. Wegen der Abhängigkeit dieser Operationen von der Länge der Queue ergibt sich eine Worst-Case-Laufzeit von $O(n)$.

Aufgrund der Memoisierung ist diese gute amortisierte Laufzeit selbst bei persistenter Nutzung der Datenstruktur zu erreichen, ein wesentlicher Vorteil gegenüber dem Batched Rebuilding.

## 3.3 Erweiterte Implementierungen

Nachdem nun die grundlegenden Prinzipien des Lazy Rebuilding vorgestellt wurden, werden zwei weitere Datenstrukturen betrachtet: Zunächst eine Double-Ended Queue (*Deque*), welche wie schon die Banker's Queue die sequenzielle Variante des Lazy Rebuilding verwendet.

Anschließend folgt mit der Real-Time Queue ein Vertreter der nebenläufigen Variante des Lazy Rebuilding.

### 3.3.1 Banker's Deque

Bei der Banker's Deque handelt es sich um eine Variante der Queue, bei der die Operationen *head*, *cons* und *tail* an beiden Enden verfügbar sind, am hinteren Ende mit *last*, *snoc*, *init* bezeichnet.

Die Implementierung erfolgt wieder durch zwei Streams, wobei die Elemente des hinteren Streams in umgekehrter Reihenfolge angeordnet sind. Eine perfekte Balance wie bei einer gewöhnlichen Queue gibt es nun nicht mehr, statt dessen befindet sich die Datenstruktur im Allgemeinen im bestmöglichen Zustand, wenn beide Streams ungefähr gleich lang sind.

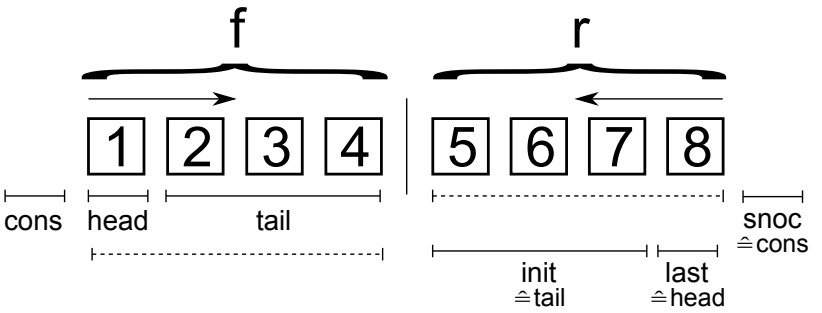


Abbildung 3.3: Aufbau Banker's Deque

#### Rebuilding

Das Rebuilding wird bei der Banker's Deque ausgelöst, sobald eine der Listen um einen konstanten Faktor länger ist als die andere. Die Invariante lautet

$$|f| \leq c \cdot |r| + 1 \quad \wedge \quad |r| \leq c \cdot |f| + 1$$

für ein $c > 1$. Die längere Liste wird auf die Durschnittslänge beider Listen gekürzt, der Rest davon umgedreht und an die kürzere Liste hinten angehängt. Auch hier werden, da wir auf Streams arbeiten, sämtliche Operationen zunächst suspendiert.

Sollte ein Umsortieren von der hinteren längeren Liste $r$ zur vorderen Liste $f$ erfolgen, sieht die Implementierung so aus, dass

```
val r' = take (j, r)
val f' = f ++ reverse(drop(j, r))
```

wobei $j = \frac{|f|+|r|}{2}$, also die mittlere Länge beider Streams.

Von $r$ werden also mit *take* die ersten $j$ Elemente entnommen. Diese Operation ist inkrementell, verursacht bei der Evaluierung also keine hohen Kosten.

An das hintere Ende von $f$ werden die verbleibenden Elemente von $r$ (erreicht mit *drop*) in umgekehrter Reihenfolge angehängt. Da sowohl drop als auch reverse monolithische Operationen sind, muss für beide in ausreichender Weise vor der Evaluierung gezahlt werden, um eine gute amortisierte Laufzeit zu erreichen.

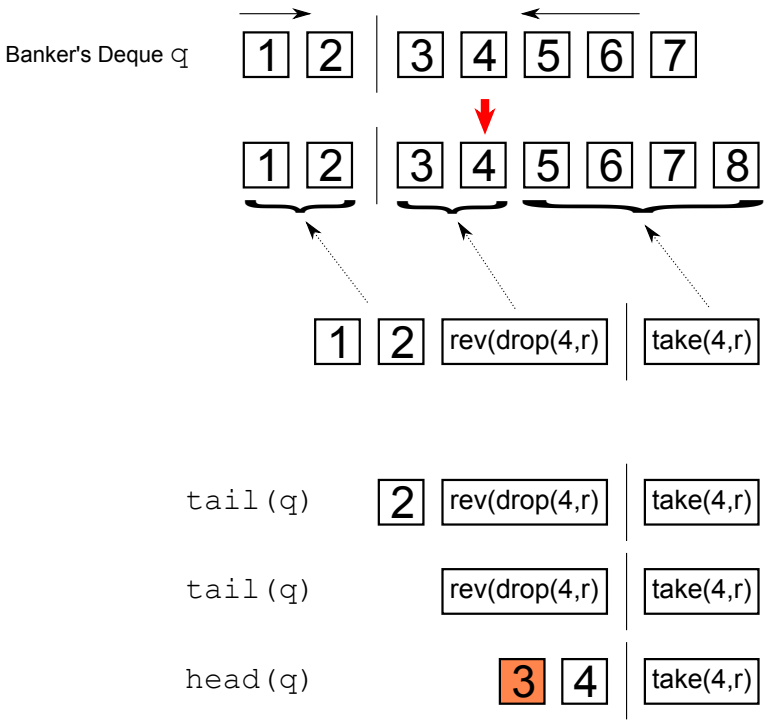


Abbildung 3.4: Rebuilding der Banker's Deque - Worst Case

**Anwendungsbeispiel** (siehe Abbildung 3.4)

Die Deque wird mit $c = 2$ initialisiert, ein Rebuilding erfolgt also z.B. wenn $|r| \leq 2 \cdot |f| + 1$. Hat $f$ Länge 2, muss $r$ also auf Länge 6 anwachsen, um ein Rebuilding auszulösen.

Ist dies der Fall, werden die involvierten Operationen `take(4,r)` und `reverse(drop(4,r))` suspendiert. Für den worst case von $|f|$ *tail*-Operationen muss nun genug Guthaben angesammelt werden, um für die monolithischen Operationen *take* und *reverse* zahlen zu können.

Da der Stream $r$ nur um den konstanten Faktor $c$ länger ist als $f$, wird jedoch vor der Evaluierung zu genüge für die anstehende Operation gezahlt, um die Laufzeit von der Länge der Liste abzukoppeln. Es ergibt sich also eine amortisierte Laufzeit von $O(1)$.

#### Bewertung

Die Banker's Deque ist wie die Banker's Queue ein Vertreter des sequenziellen Variante des Lazy Rebuilding und dementsprechend analog fällt das Fazit aus: Die Rebuilding-Operationen amortisieren sich ausreichend, bevor die Suspendierung evaluiert wird, um eine amortisierte Laufzeit von $O(1)$ zu erreichen, die Worst-Case-Laufzeit beläuft sich jedoch weiterhin auf $O(n)$, da das Rebuilding in einem Schritt ausgeführt werden muss.

### 3.3.2 Real-Time Deque

Dem Makel der Worst-Case-Laufzeit $O(n)$ möchte die Real-Time Deque begegnen, dabei aber dennoch nicht auf das Prinzip des Lazy Rebuilding verzichten. Das Ziel ist eine Datenstruktur, die jederzeit bereit ist, Echtzeitdaten aufzunehmen, ohne einen zwischengeschalteten Puffer zu benötigen, der aktiviert werden müsste, wenn ein Rebuildingprozess die Datenstruktur auslastet und für kurze Zeit nicht ansprechbar macht. Der Preis dafür ist ein (pseudo-)nebenläufiges Rebuilding, das die Kosten für jede strukturverändernde Operation für die Dauer des Rebuildings um einen konstanten Faktor erhöht.

#### Rebuilding

Um eine schrittweise Abarbeitung des Rebuildings zu ermöglichen, müssen zunächst alle monolithischen Operationen in inkrementelle Operationen umgewandelt werden, in diesem Fall *drop* und *reverse*. Diesen Vorgang beschreibt Chris Okasaki in Kapitel 7 seinen *Purely Functional Data Structures* und soll nicht das wesentliche Augenmerk dieses Abschnitts sein.

Die Strategie für das Auslösen des Rebuildings ist die gleiche wie bei der Banker's Deque: Die Invariante wird stets durch die *check*-Funktion gewahrt, ein übermäßiges Anwachsen eines der beiden Streams löst ein Rebuilding aus:

```
val r' = take (j, r)
val f' = rotateDrop (f, j, r)
```

wobei $j$ wiederum $\frac{|f|+|r|}{2}$.

`rotateDrop` ist die inkrementelle Variante der *drop*-Funktion, sie ruft sich selbst so lange rekursiv auf, bis alle $j$ Elemente vom Anfang der Liste entfernt wurden, forciert diesen Vorgang aber nicht. Das heißt wiederum, dass sich die Datenstruktur in einem unbrauchbaren Zustand befindet, solange `rotateDrop` nicht abgeschlossen ist, es gilt also diese Operation abzuschließen, bevor die betroffenen Daten angefordert werden.

Analog dazu folgt die Implementierung von `rotateRev`, dem inkrementellen Bruder von *reverse*. `rotateRev` wird von `rotateDrop` aufgerufen, sobald die Funktion abgeschlossen wurde. An dieser Stelle sei auf die komplette Implementierung der Real-Time Deque in SML im Anhang verwiesen.

Entscheidend ist an dieser Stelle der Datentyp der Real-Time Deque:

$$len_f \times f_{working} \times f_{secondary} \times len_r \times r_{working} \times r_{secondary}$$

Beide Streams besitzen anscheinend wieder eine working- und eine secondary Copy. Betrachtet man aber die Implementierung der check-Funktion, wird deutlich, dass beide Kopien auf ein- und derselben persistenten Datenstruktur arbeiten:

```
1 fun check (q as (lenf, f, sf, lenr, r, sr)) =
2   (* ... *)
3   let val i = (lenf+lenr) div 2
4       val j = lenf+lenr-i
5       val r' = take (j, r)
6       val f' = rotateDrop (f, j, r)
7   in (i, f', f', j, r', r') end
8   (* ... *)
```

In Zeile 7 werden Referenzen auf die jeweils gleiche (persistente) Datenstruktur gelegt. Dies hat auch zur Folge, dass Rebuildingprozesse memoisiert werden und somit auf beiden Streams verfügbar sind.

Diesen Effekt macht sich nun die Real-Time Deque zu Nutze: Auf der working copy werden weiterhin gewöhnliche Operationen durchgeführt, während die secondary copy lediglich dazu dient, in den vom Rebuilding betroffenen Bereich hineinzulesen, und so eine Evaluation der nun inkrementellen Funktionen zu erzwingen. Mit jeder Operation wird die secondary copy um ein paar Elemente verkürzt und das Rebuilding einen kleinen Schritt weitergeführt.

Wurde die secondary copy schließlich zu Ende gelesen, ist das Rebuilding abgeschlossen und die Datenstruktur befindet sich wieder in einem sicheren Zustand. Der Endzustand der neu aufgebauten Datenstruktur wurde memoisiert und die working copy kann direkt darauf zugreifen.

### Wie oft?

Entscheidend bei der Verwendung dieser Methode ist, dass das Rebuilding komplett abgeschlossen wurde, bevor der Anwender tatsächlich Daten aus dem betroffenen Bereich lesen kann. Der worst case ist hier wiederum das sture Aufrufen von *tail*, bis man in diesen Bereich gelangt.

Chris Okasakis Implementierung verwendet die Methoden `exec1` und `exec2`, welche jeweils die secondary copy um ein bzw. zwei Elemente verkürzen. Mit jedem *cons/snoc* wird `exec1` und mit jedem *tail/init* wird `exec2` aufgerufen.

```
fun exec1 ($(Cons (x, s))) = s
  | exec1 s = s
fun exec2 s = exec1 (exec1 s)
```

Diese Methode funktioniert nur, da Okasaki den Wert $c$, welcher den maximalen Faktor für den Längenunterschied angibt, auf die Werte 2 und 3 einschränkt.

Angenommen sei der worst case: $c = 3$ und $|r| = 3 \cdot |f| + 1$. Bei der nächsten *snoc*-Operation muss ein Rebuilding ausgelöst werden, um die Invariante zu wahren.

Die Länge von $f'$ beträgt danach $2 \cdot |f| + 1$. Bei der $|f| + 1$ten *tail*-Operation muss das Rebuilding abgeschlossen sein, und dieses Ziel wird auch erreicht, wenn pro normalem *tail*-Aufruf zwei *tail*-Operationen (wegen `exec2`) auf die secondary copy angewandt werden.

Für *cons* ist die Situation wesentlich entspannter, da $f$ auf die dreifache Länge anwachsen müsste, um ein erneutes Rebuilding auszulösen. Da mit jedem *cons* aber ein Element der secondary copy entfernt wird, ist dieses Ziel schon viel früher erreicht.

### Bewertung

Die Real-Time Deque nutzt die Möglichkeiten funktionaler Sprachen auf sehr elegante Weise: Lazy Evaluation, Persistenz und Memoisierung haben klare Vorteile gegenüber strikter Ausführung, manuellem Scheduling und der manuellen Synchronisation zweier Kopien der gleichen Datenstruktur, aber der Reihe nach:

Der Datentyp `RotationState`, wie wir ihn vom Global Rebuilding kennen, entfällt vollständig. Stattdessen erfolgt eine nicht ganz intuitive Umwandlung

monolithischer Funktionen in ihre inkrementellen Pendants. Die Evaluierung ist dafür aber umso leichter, es müssen lediglich suspendierte Methoden forciert werden, eine Fallunterscheidung nach verschiedenen Zuständen ist nicht nötig.

Entscheidender Vorteil ist jedoch das Wegfallen der doppelten Datenstruktur. Die Ausführung des Rebuildings auf einer einzigen persistenten Datenstruktur spart Speicherplatz und verlangt keinerlei Synchronisation zwischen verschiedenen Kopien.

Auch wenn die worst-case Laufzeit $O(1)$ erreicht wurde, die Schattenseite bleibt dennoch: Rebuilding braucht Zeit, auch wenn man es in noch so kleine Stücke aufteilt, und damit steigen auch die Kosten für jede einzelne Operation während des Rebuildingprozesses, wenn auch nur um einen konstanten Faktor.

## 3.4 Fazit Lazy Rebuilding

In diesem Kapitel wurden die zwei grundlegenden Varianten des Lazy Rebuilding beleuchtet:

Zunächst die strikt sequenzielle Variante, mit den Anwendungsbeispielen Banker's Queue und Deque, welche eine gute amortisierte Laufzeit von $O(1)$ erreichen können, aber grundsätzlich aufgrund monolithischer Operationen, die beim Rebuilding nötig sind, eine worst-case Laufzeit von $O(n)$ in Kauf nehmen müssen. Die Implementierung in funktionalen Sprachen, die Lazy Evaluation unterstützen, ist dabei einfach und sämtliche genannten Vorteile sind auch - im Gegensatz zum sonst ähnlichen Batched Rebuilding - bei persistenter Nutzung der Datenstruktur gültig.

Anschließend wurde die nebenläufige Version mit der Real-Time Deque vorgestellt, welche die teuren monolithischen Operationen aufbricht und verteilt, um eine worst-case Laufzeit von $O(1)$ zu erreichen.

Allen Implementierungen gemeinsam ist die effizienter Nutzung der Konzepte Lazy Evaluation und Persistenz, anstatt sich auf eigene Datentypen und Schedulingstrategien zu verlassen. Dies gelingt durchgehend gut und hat keine Nachteile gegenüber den zuvor besprochenen Methoden Batched Rebuilding und Global Rebuilding. Im Gegenteil, Vorteile ergeben sich teilweise im Speicherbedarf (keine „echte“ secondary copy erforderlich) und vor allem bei der Persistenz.

Das Rebuilding wird nicht mehr stur ausgeführt, wenn die Datenstruktur sich in einem ungünstigen Zustand befindet, sondern nur wenn auch Daten aus diesem Bereich angefordert werden. Dieses Kriterium disqualifiziert das Konzept des Lazy Rebuilding leider für die meisten baumartigen Datenstrukturen, aber bei der Verwendung listenartiger Strukturen kann Lazy Rebuilding die Performanz und die Wiederverwendbarkeit der Datenstruktur bereichern.

# 4 Anhang

## Zuordnung

Harald Brunner: Batched Rebuilding und Global Rebuilding

Sebastian Straub: Lazy Rebuilding

## Sourcecodes

**Algorithmus 4.1** Implementierung Banker's Queue

```
structure BankersQueue : QUEUE =
struct
  type 'a Queue = int * 'a Stream * int * 'a Stream

  val empty = (0, $Nil, 0, $Nil)
  fun isEmpty (lenf, _, _, _) = (lenf = 0)

  fun check (q as (lenf, f, lenr, r)) =
        if lenr <= lenf
           then q
           else (lenf+lenr, f ++ reverse r, 0, $Nil)

  fun snoc ((lenf, f, lenr, r), x) =
        check (lenf, f, lenr+1, $(Cons (x, r)))

  fun head (lenf, $Nil, lenr, r) = raise Empty
    | head (lenf, $(Cons (x, f')), lenr, r) = x
  fun tail (lenf, $Nil, lenr, r) = raise Empty
    | tail (lenf, $(Cons (x, f')), lenr, r) =
        check (lenf-1, f', lenr, r)
end
```

**Algorithmus 4.2** Implementierung Banker's Deque

```
functor BankersDeque (val c : int) : DEQUE =  (* c > 1 *)
struct
  type 'a Queue = int * 'a Stream * int * 'a Stream

  val empty = (0, $Nil, 0, $Nil)
  fun isEmpty (lenf, f, lenr, r) = (lenf+lenr = 0)

  fun check (q as (lenf, f, lenr, r)) =
        if lenf > c*lenr+1 then
          let val i = (lenf+lenr) div 2
              val j = lenf + lenr - i
              val f' = take (i, f)
              val r' = r ++ reverse (drop (i, f))
          in (i, f', j, r') end
        else if lenr > c*lenf + 1 then
        let val j = (lenf+lenr) div 2
              val i = lenf + lenr - j
              val r' = take (j, r)
              val f' = f ++ reverse (drop (j, r))
          in (i, f', j, r') end
        else q

  fun cons (x, (lenf, f, lenr, r)) = check (lenf+1, $(Cons (x, f)), lenr, r)
  fun head (lenf, $Nil, lenr, $Nil) = raise Empty
    | head (lenf, $Nil, lenr, $(Cons (x, _))) = x
    | head (lenf, $(Cons (x, f')), lenr, r) = x
  fun tail (lenf, $Nil, lenr, $Nil) = raise Empty
    | tail (lenf, $Nil, lenr, $(Cons (x, _))) = empty
    | tail (lenf, $(Cons (x, f')), lenr, r) = check (lenf-1, f', lenr, r)

  val snoc = stub
  val last = stub
  val init = stub
end
```

**Algorithmus 4.3** Implementierung Real-Time Deque

```
functor RealTimeDeque (val c : int) : DEQUE =  (* c = 2 or c = 3 *)
struct
  type 'a Queue =
            int * 'a Stream * 'a Stream * int * 'a Stream * 'a Stream

  val empty = (0, $Nil, $Nil, 0, $Nil, $Nil)
  fun isEmpty (lenf, f, sf, lenr, r, sr) = (lenf+lenr = 0)

  fun exec1 ($(Cons (x, s))) = s
    | exec1 s = s
  fun exec2 s = exec1 (exec1 s)
  fun rotateRev ($Nil, r, a) = reverse r ++ a
    | rotateRev ($(Cons (x, f)), r, a) =
        $(Cons (x, rotateRev (f, drop (c, r), reverse (take (c, r)) ++ a)))
  fun rotateDrop (f, j, r) =
        if j < c then rotateRev (f, drop (j, r), $Nil)
  else let val ($(Cons (x, f'))) = f
       in $(Cons (x, rotateDrop (f', j-c, drop (c,r)))) end

  fun check (q as (lenf, f, sf, lenr, r, sr)) =
        if lenf > c*lenr+1 then
          let val i = (lenf+lenr) div 2   val j = lenf+lenr-i
              val f' = take (i, f)        val r' = rotateDrop (r, i, f)
          in (i, f', f', j, r', r') end
        else if lenr > c*lenf+1 then
          let val j = (lenf+lenr) div 2   val i = lenf+lenr-j
              val r' = take (j, r)        val f' = rotateDrop (f, j, r)
          in (i, f', f', j, r', r') end
        else q

  fun cons (x, (lenf, f, sf, lenr, r, sr)) =
        check (lenf+1, $(Cons (x, f)), exec1 sf, lenr, r, exec1 sr)
  fun head (lenf, $Nil, sf, lenr, $Nil, sr) = raise Empty
    | head (lenf, $Nil, sf, lenr, $(Cons (x, _)), sr) = x
    | head (lenf, $(Cons (x, f')), sf, lenr, r, sr) = x
  fun tail (lenf, $Nil, sf, lenr, $Nil, sr) = raise Empty
    | tail (lenf, $Nil, sf, lenr, $(Cons (x, _)), sr) = empty
    | tail (lenf, $(Cons (x, f')), sf, lenr, r, sr) =
        check (lenf-1, f', exec2 sf, lenr, r, exec2 sr)

  val snoc = stub
  val last = stub
  val init = stub
end
```

**Algorithmus 4.4** Implementierung Hood-Melville Real-Time Queue

```
structure HoodMelvilleQueue : QUEUE =
struct
  datatype 'a RotationState =
      Idle
    | Reversing of int * 'a list * 'a list * 'a list * 'a list
    | Appending of int * 'a list * 'a list
    | Done of 'a list

  type 'a Queue = int * 'a list * 'a RotationState * int * 'a list

  fun exec (Reversing (ok, x :: f, f', y :: r, r')) =
        Reversing (ok+1, f, x :: f', r, y :: r')
    | exec (Reversing (ok, [], f', [y], r')) = Appending (ok, f', y :: r')
    | exec (Appending (0, f', r')) = Done r'
    | exec (Appending (ok, x :: f', r')) = Appending (ok-1, f', x :: r')
    | exec state = state

  fun invalidate (Reversing (ok, f, f', r, r')) =
        Reversing (ok-1, f, f', r, r')
    | invalidate (Appending (0, f', x :: r')) = Done r'
    | invalidate (Appending (ok, f', r')) = Appending (ok-1, f', r')
    | invalidate state = state

  fun exec2 (lenf, f, state, lenr, r) =
        case exec (exec state) of
            Done newf => (lenf, newf, Idle, lenr, r)
    | newstate => (lenf, f, newstate, lenr, r)

  fun check (q as (lenf, f, state, lenr, r)) =
        if lenr <= lenf then exec2 q
        else let val newstate = Reversing (0, f, [], r, [])
             in exec2 (lenf+lenr, f, newstate, 0, []) end

  val empty = (0, [], Idle, 0, [])
  fun isEmpty (lenf, f, state, lenr, r) = (lenf = 0)

  fun snoc ((lenf, f, state, lenr, r), x) =
        check (lenf,f,state,lenr+1,x::r)
  fun head (lenf, [], state, lenr, r) = raise Empty
    | head (lenf, x :: f, state, lenr, r) = x
  fun tail (lenf, [], state, lenr, r) = raise Empty
    | tail (lenf, x :: f, state, lenr, r) =
        check (lenf-1, f, invalidate state, lenr, r)
end
```