

**BITS3 WS2021**

# **Ausarbeitung einer Holdback und Delivery Queue**

*Algorithmen und Datenstrukturen - gelesen von  
Prof. Dr. Christoph Klauck*

KRISTOFFER SCHAAF (2588265)

FAKULTÄT TECHNIK UND INFORMATIK

Department Informatik

Hochschule für angewandte Wissenschaften Hamburg

## Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>2</b>
1.1	Aufgabenstellung . . . . .	2
1.2	Funktion der Holdback Queue . . . . .	3
1.3	Funktion der Delivery Queue . . . . .	3
1.4	Aufbau der Nachrichten . . . . .	3
1.5	Aufgabenerarbeitung . . . . .	3
1.6	Sortierung der Nachrichten . . . . .	4
1.7	Erwartungen . . . . .	5
<b>2</b>	<b>Implementierung der Holdback Queue</b>	<b>6</b>
2.1	Operationen . . . . .	7
2.1.1	initHBQ . . . . .	7
2.1.2	checkHBQ . . . . .	7
2.1.3	pushHBQ . . . . .	9
2.1.4	deliverMSG . . . . .	10
2.1.5	listADT . . . . .	11
2.1.6	dellHBQ . . . . .	11
<b>3</b>	<b>Implementierung der Delivery Queue</b>	<b>12</b>
3.1	Operationen . . . . .	13
3.1.1	initDLQ . . . . .	13
3.1.2	delDLQ . . . . .	13
3.1.3	expectedNr . . . . .	13
3.1.4	push2DLQ . . . . .	14
<b>4</b>	<b>Analyse</b>	<b>16</b>
4.1	Messaufbau . . . . .	16
4.2	Messergebnisse . . . . .	16
<b>5</b>	<b>Fazit</b>	<b>21</b>
5.1	Heap oder List . . . . .	21
5.2	Limit der Delivery Queue . . . . .	22
5.3	Pattern Matching . . . . .	22
5.4	Alternative Anwendung . . . . .	25
	<b>Literaturverzeichnis</b>	<b>26</b>
	<b>Abbildungsverzeichnis</b>	<b>26</b>
<b>A</b>	<b>Anhang</b>	<b>27</b>

# 1 Einleitung

## 1.1 Aufgabenstellung

In folgenden wird die Implementierung einer abstrakten Datenstruktur für eine Client/-Server Anwendung (siehe 1) behandelt.

Die Aufgabe dieser Anwendung ist es Tagesnachrichten von verschiedenen Redakteuren zu verwalten und in korrekter Reihenfolge an den Kunden auszuliefern. Da die Reihenfolge der Nachrichten nicht von den Redakteuren abgestimmt wird, würde diese ohne einen Zwischenserver nicht korrekt sein. Die Nachrichten werden also von dem Redakteur-Client-Programm mit einer eindeutigen Nummerierung zuerst an einen Server gesendet. Dieser wird über das abstrakte Datenstruktur-Konzept einer Delivery und Holdback Queue verwaltet. Das heißt, dass zuerst alle Nachrichten in der Holdback Queue gehalten und nur bei korrekter Nummer an die Delivery Queue weitergegeben werden.

Von der Delivery Queue aus werden die Nachrichten auf Anfrage des Lesers dann in korrekter Reihenfolge an das entsprechende Client-Programm geschickt.

Da es auch hier wieder verschiedene Leser gibt hat der Server die Aufgabe sich für jeden Leser, insofern dieser sich nicht zu lange nicht mehr gemeldet hat, zu merken, welche Nachrichten er schon an diesen verschickt hat. Um den korrekten Ablauf der Anwendung kontrollieren zu können und um beim Implementieren Fehler möglichst einfach eliminieren zu können, werden alle Ausgaben in einer Datei HB-DLQ<Node>.log geschrieben. Der Node auf welchem das System gerade läuft kann über den Erlang Befehl `inet:gethostname()` bestimmt werden.

Da die beiden Client Programme für die Redakteure und Leser und der Server zur Verfügung gestellt wurden, wird es im folgenden um die Implementierungen der Holdback und der Delivery Queue gehen.

Diese wird komplett in der funktionalen Programmiersprache Erlang umgesetzt und die beiden Dateien `hbq.erl` und `dlq.erl` enthalten den Code für die beiden Queues.

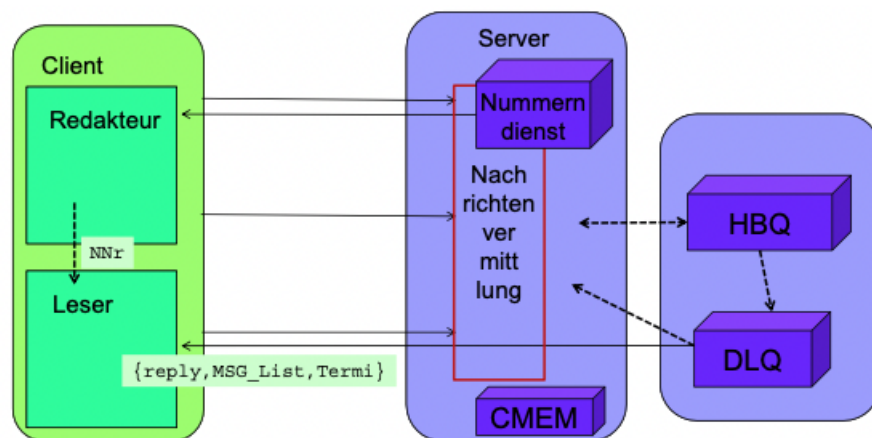


Abbildung 1: Nachrichtendienst [Kla21a]

## 1.2 Funktion der Holdback Queue

Die Holdback Queue enthält alle Nachrichten, die nicht ausgeliefert werden dürfen. Das heißt, dass die enthaltenen Nachrichten nicht die richtigen Nummern für die Delivery Queue haben. Durch regelmäßiges Prüfen wird entschieden, ob inzwischen eine geeignete Nachricht für die Delivery Queue vom Server empfangen wurde.

Da die Nachrichten nach aufsteigender Nummerierung an die Delivery Queue weitergeleitet werden, bietet es sich an diese in der Holdback Queue bereits zu sortieren. Um diese Sortierung möglichst effizient zu gestalten werden verschiedene Algorithmen getestet (siehe Kapitel 1.6.).

## 1.3 Funktion der Delivery Queue

Die Delivery Queue ist der abstrakte Speicher für alle Nachrichten, welche an den Client ausgeliefert werden können. Die Nachrichten sind innerhalb der Queue aufsteigend sortiert. Die eigentliche Schnittstelle zum Server ist aber trotzdem die Holdback Queue, da über diese die Nachrichten wieder versendet werden. Die Delivery Queue ist somit also lokal implementiert und wird von der Holdback Queue aufgerufen.

## 1.4 Aufbau der Nachrichten

Nachrichten werden von den Redakteuren entwickelt und von den Lesern konsumiert. Bis sie beim Leser ankommen, können sie aber durch mögliche Software Fehler, wie z.B. asynchrone Nebenläufigkeiten oder ähnliches, verloren gehen. Die Nachrichten sind Tupel mit mindestens drei Elementen. Zum einen enthalten sie die Nachrichten-Nummer, nach welcher die Nachrichten sortiert werden. Dann enthalten sie eine Textzeile in welcher die eigentliche Nachricht geschrieben steht. Und abhängig davon, welche Queues sie schon durchlaufen haben, enthalten sie Zeitstempel mit den entsprechenden Eintritts- und Austrittszeiten.

## 1.5 Aufgabenerarbeitung

Im Folgenden werden zuerst Entwürfe für die verschiedenen Queues mit ihren Funktionen erstellt, dabei wird vorerst nur die Implementierung der Holdback Queue als Heap und die Delivery Queue als Liste beschrieben. Die Entwürfe sollen nahe am zu implementierenden Code liegen, damit Fehler schnell eliminiert werden können. Um den Code zu testen werden Eunit Tests aus der Library 'eunit/include/eunit.hrl' geschrieben. Diese werden für spezifische Fälle angewendet, welche aus den Diagrammen der Entwürfe entschlossen werden können. So zum Beispiel der Fall, dass alle Elemente bei passender Nummerierung sofort von der Holdback zu der Delivery Queue weitergeleitet werden. Nachdem die erste Version mit der Heapstruktur innerhalb der Holdback Queue funktioniert, wird eine zweite mit einer Listenstruktur implementiert. Diese beiden zu analysieren und zu vergleichen

wird der Hauptbestandteil dieser Ausarbeitung sein. Zum Analysieren werden unter anderem die Benchmarks von Matz Heitmüller genutzt. Des Weiteren wird auch der Einfluss von Implementierungen mit Pattern Matching zu welchen mit if-else Statements verglichen. Nach diesem Teil wird ein Fazit erstellt in welchem die Messergebnisse ausgewertet werden.

## 1.6 Sortierung der Nachrichten

Problemstellung dieser Hausarbeit wird das Sortieren der Nachrichten innerhalb der Holdback Queue sein. Hierfür werden verschiedene Sortieralgorithmen getestet. Für verschiedene Algorithmen werden dann aber auch dementsprechend verschiedene abstrakte Datenstrukturen benötigt. So würde z.B. bei Insertion Sort nur das Konzept der Liste, also das 'Aneinander-pipen' von Elementen, reichen. Bei einem Heap Sort Algorithmus würde aber, wie der Name es schon andeutet, eine Heapstruktur verwendet werden müssen. Dieses Konzept basiert auf der Baumstruktur, es gibt also einen Wurzelknoten welcher einen linken und einen rechten Teilbaum haben kann.

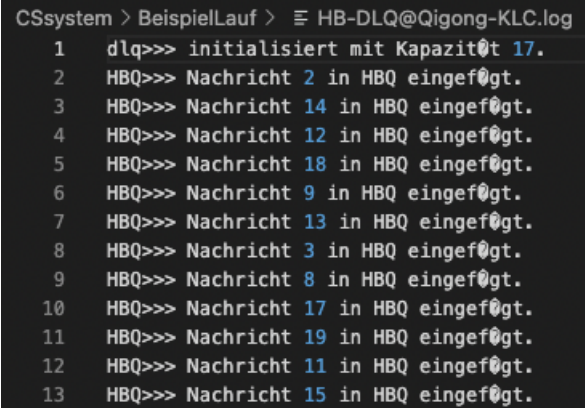
Im vergangenen Praktikum 2 haben wir uns ausführlich mit Sortieralgorithmen befasst und drei bestimmte ausgewertet (siehe Abbildung 2). Aufsteigend, absteigend und random, bezieht sich hierbei auf die Liste welche sortiert wurde.

	aufsteigend aufgebaut $O(n)$	absteigend aufgebaut $O(n^2)$	random aufgebaut $O(n^2)$
<b>insertionSort()</b>	Jedes Element aus list wird nur einmal verglichen und ist dann schon an der richtigen Position.	Jedes Element aus list wird nur einmal verglichen. Danach muss aber nochmal die richtige Position gefunden werden.	Der Aufwand ist ähnlich zu dem absteigenden. Es sollte aber etwas schneller gehen, da die richtige Position im Schnitt schneller gefunden wird.
<b>quickSort()</b> erstes Element als Pivotelement	$O(n^2)$ Da das Pivotelement immer das kleinste (oder größte) in der Liste ist, ist die eine Teilliste fast leer und die andere fast voll.	$O(n^2)$ Der Aufwand ist identisch zu dem aufsteigend sortiertem.	$O(n * \log(n))$ Die Elemente werden in immer kleiner werdende fast gleich große Teillisten aufgeteilt.
<b>heapSort()</b>	$O(n * \log(n))$	$O(n * \log(n))$	$O(n * \log(n))$

Abbildung 2: Auswertung Sortieralgorithmen [Sch21]

Die bisherigen Algorithmen wurden aber bisher nur getestet, wenn mit statischen Listen gearbeitet wurde. Es also eine Liste übergeben wurde, diese fertig sortiert wird und dann mit dem nächsten Prozess begonnen wird. In dieser Anwendung wird allerdings dynamisch gearbeitet. Da der Holdback Queue ständig neue Nachrichten zugesendet werden, wäre ein Algorithmus geeignet, welcher im Online-Verfahren arbeitet. Er kennt also nicht alle Elemente welche zu sortieren sind, wenn er mit dem Sortieren beginnt. Interessant hierfür wäre also der InsertionSort Algorithmus. Wie es der bereitgestellten logging Datei "HB-DLQ@Qigong-KLC.log" (siehe Abb. 3) zu entnehmen ist, werden die Nachrichten

von den Client-Servern der Redakteure mit sehr unbeständiger Nummerierung gesendet. Für den InsertionSort Algorithmus wäre dies aber ein Aufwand von  $O(n^2)$ . Im Folgenden wird ein dem InsertionSort ähnlicher Algorithmus verwendet. Im InsertionSort sucht ein Laufindex das nächste unsortierte Element und fügt es an der richtigen Stelle in der Liste ein. In diesem Anwendungsfall wäre der Laufindex immer an der Position null, da hier die neue Nachricht eingefügt wird. Dann wird der Index des Elements so lange erhöht, bis dieses die richtige Position erreicht hat. Wenn davon ausgegangen wird, dass die Elemente in zufälliger Reihenfolge eingefügt werden, entsteht eine Komplexität von  $O(n/2)$ . Beim Entfernen der Nachrichten gibt es in diesem Szenario den Vorteil, dass die Listen Erlang-intern mit dem ersten Element also Listenkopf (Head) beginnen und an diese die restliche Liste (Tail) angehängt wird. Somit kann beim Entfernen des kleinsten Elements mit dem Tail weitergearbeitet werden, was auf einen Aufwand von  $O(1)$  schließen lässt.



```

CSsystem > Beispielllauf > HB-DLQ@Qigong-KLC.log
1  d!q>>> initialisiert mit Kapazität 17.
2  HBQ>>> Nachricht 2 in HBQ eingefügt.
3  HBQ>>> Nachricht 14 in HBQ eingefügt.
4  HBQ>>> Nachricht 12 in HBQ eingefügt.
5  HBQ>>> Nachricht 18 in HBQ eingefügt.
6  HBQ>>> Nachricht 9 in HBQ eingefügt.
7  HBQ>>> Nachricht 13 in HBQ eingefügt.
8  HBQ>>> Nachricht 3 in HBQ eingefügt.
9  HBQ>>> Nachricht 8 in HBQ eingefügt.
10 HBQ>>> Nachricht 17 in HBQ eingefügt.
11 HBQ>>> Nachricht 19 in HBQ eingefügt.
12 HBQ>>> Nachricht 11 in HBQ eingefügt.
13 HBQ>>> Nachricht 15 in HBQ eingefügt.

```

Abbildung 3: Nachrichtendienst [Kla21c]

Der HeapSort Algorithmus an sich setzt eine Komplexität von  $O(n * \log(n))$  voraus. Das liegt daran, dass die zu sortierende Liste zuerst zu einem Heap umstrukturiert werden muss. Da dieser Schritt entfällt, angenommen man strukturiert diesen Heap von Anfang an, dann bleibt eine Komplexität von  $O(\log(n))$ . Beachtet werden muss, dass der Aufwand beim Einfügen und beim Entfernen auch beim Heap nicht identisch ist. Beim Einfügen gilt  $O(\log(n))$ , da das Element eventuell nur den Heap nach oben wandert. Beim Entfernen allerdings gilt  $O(2 * \log(n))$ , da hier das Wurzelement nach dem Entfernen ersetzt wird und der Heap somit auch neu strukturiert werden muss. Der Heap Sortieralgorithmus ist also bei aufsteigend sortierter Liste mit  $O(3 * \log(n))$  leicht effizienter als der InsertionSort Algorithmus, allerdings mit gleichbleibender Effizienz deutlich besser bei random sortierter Liste.

## 1.7 Erwartungen

Das Ziel dieser Ausarbeitung ist es eine möglichst Effiziente Verarbeitung der Nachrichten zu erreichen. Der Fokus wird also auf der Sortierung der Elemente innerhalb der Holdback Queue, aber auch auf der Optimierung des allgemeinen Codes liegen.

Nach dem Vergleichen der beiden Implementierungen der Holdback Queue sollte die Sortierung über einen Heap als effizienter hervorgehen.

Außerdem wird vermutet, dass auch das Auslagern von Funktionen eine höhere Laufzeit zur Folge hat. Deswegen wird in einer Implementierung mit Hilfsfunktionen gearbeitet und in einer anderen auf diese verzichtet, was aber in einer teilweise sehr tiefen Codestruktur und einer dementsprechend schlechten Lesbarkeit resultieren wird.

## 2 Implementierung der Holdback Queue

Wie bereits in Kapitel 1.6 erläutert, wird die Holdback Queue mit zwei verschiedenen Strukturen implementiert.

Für die Struktur des HeapSorts gilt, dass jedes Element die Form  $\{[Nnr, Msg, TScientout, TShbqin], Höhe, linkerTeilbaum, rechterTeilbaum\}$  hat. Außerdem ist die Nnr des Wurzelements stets kleiner als die der Kinderelemente. Die beiden Kinderelemente werden nicht miteinander verglichen. Diese Form des Heaps nennt sich Min Heap (siehe Abb. 4).

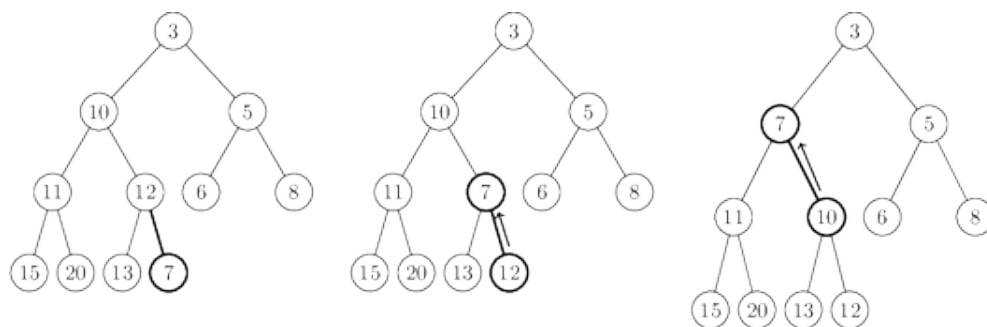


Abbildung 4: Binary Min Heap Insert<sup>1</sup>

Für das Entfernen eines neuen Elements entsteht dadurch der Vorteil, dass das Element im Normalfall die Wurzel des Heaps ist, da es die kleinste Nachrichtennummer haben sollte und somit ganz oben als Wurzel gespeichert ist. Auch beim Einfügen des Elements ist der Heap von Vorteil, da die Elemente ohne erkennbare Sortierung (siehe Abbildung 5) vom Server in den Heap eingefügt werden. Die Elemente werden am höchsten freien Index eingefügt und wandern dann im Heap nach oben, bis sie ihre Position erreicht haben. Im Beispiel ist zu sehen, wie das Element mit der Nummer 7 eingefügt wird und solange mit dem Elternknoten getauscht wird, bis dieses größer ist.

Da das Konzept des HeapSort Algorithmus darauf aufbaut, dass die neuen Elemente als Blatt eingefügt und die sortierten Elemente als Wurzel entfernt werden, kann der implementierte Heap aus dem Praktikum 2 nur teilweise wiederverwendet werden. Da dieser ein Max Heap ist, das heißt das größte Element ist die Wurzel, jetzt aber ein Min Heap benötigt wird, wird der Code größtenteils wiederverwendet, aber in Teilen angepasst. Der Max Heap wurde zusammen mit Leon Schwarzenberger entwickelt und implementiert. Für

<sup>1</sup><https://rosalind.info/glossary/algo-binary-heap/>

weitere Informationen und Entwürfe über die einzelnen Heap Funktionen siehe Anhang A oder [Sch21].

```
CSsystem > Beispielllauf > ≡ HB-DLQ@Qigong-KLC.log
1  dlq>>> initialisiert mit Kapazität 17.
2  HBQ>>> Nachricht 2 in HBQ eingefügt.
3  HBQ>>> Nachricht 14 in HBQ eingefügt.
4  HBQ>>> Nachricht 12 in HBQ eingefügt.
5  HBQ>>> Nachricht 18 in HBQ eingefügt.
6  HBQ>>> Nachricht 9 in HBQ eingefügt.
7  HBQ>>> Nachricht 13 in HBQ eingefügt.
8  HBQ>>> Nachricht 3 in HBQ eingefügt.
9  HBQ>>> Nachricht 8 in HBQ eingefügt.
10 HBQ>>> Nachricht 17 in HBQ eingefügt.
11 HBQ>>> Nachricht 19 in HBQ eingefügt.
12 HBQ>>> Nachricht 11 in HBQ eingefügt.
13 HBQ>>> Nachricht 15 in HBQ eingefügt.
```

Abbildung 5: Nachrichtendienst [Kla21c]

## 2.1 Operationen

### 2.1.1 initHBQ

Beim Initialisieren der Holdback Queue liegt das größte Problem darin, die Queue auf einem Node zu speichern. Das ermöglicht die Erlang Funktion `spawn/1`, welcher als Parameter die Funktion übergeben wird, welche gestartet werden soll. Hierbei wird ein neuer Prozess erzeugt und initialisiert. Die ProzessID dieses neu erzeugten Prozesses wird als Rückgabewert zurückgegeben und in einer Variable gespeichert. Die globale Registrierung des Prozesses findet über den Aufruf `register/2` statt. Als Parameter werden hier einmal die ProzessID des zu registrierenden Prozesses übergeben und das Atom unter welchem dieser Prozess gespeichert werden soll. Außerdem wird die Initialisierung der Delivery Queue über die Holdback Queue aufgerufen. Diesem Aufruf wird der Parameter DLQ-Limit, also die maximale Größe der Delivery Queue, mit übergeben.

Durch Rekursion kann der Status des Prozesses, also in diesem Falle unter anderem die Informationen über die Holdback und Delivery Queue, vollständig in den Parametern der Funktion gehalten werden (frei nach [Heb13]). Der Prozess der Holdback Queue wird hierfür über die Funktion `loop` gestartet. Dieser Funktion wird als Parameter die Holdback Queue, dessen nächster freier Index oder dessen Größe, die Delivery Queue, dessen maximale Größe und die logging Datei übergeben. Somit können diese über jeden `loop` Aufruf beschrieben und gelesen werden. Ob der nächste freie Index oder die aktuelle Größe übergeben wird, hängt von der Struktur der Holdback Queue ab.

### 2.1.2 checkHBQ

Da die Nachrichten der Holdback Queue regelmäßig auf Auslieferbarkeit geprüft werden sollen, wurde hierfür eine weitere Funktion implementiert. In dieser wird das derzeitige



erste Element der Holdback Queue (im Folgenden 'SmallestElem') mit der von der Delivery Queue erwarteten Nummer ('ExpNrDLQ') verglichen. Somit kann erkannt werden, ob Elemente aus der Holdback Queue verworfen oder an die Delivery Queue ausgeliefert werden sollen. Wenn zum Beispiel  $\text{SmallestElem} == \text{ExpNrDLQ}$  gilt, dann wird SmallestElem an die Delivery Queue ausgeliefert. Im Falle  $\text{SmallestElem} < \text{ExpNrDLQ}$  wird das SmallestElem verworfen, da es nicht mehr benötigt wird und ansonsten die Holdback Queue nur blockieren würde. Benötigt wird es nicht mehr, da die Delivery Queue als aufsteigende Liste ohne Duplikate und ohne fehlende Nachrichtennummern definiert ist. Alle Nachrichten, welche eine kleinere Nachrichtennummer als das aktuell kleinste Element der Delivery Queue haben, werden von dieser nicht mehr angefragt und können somit aus der Holdback Queue verworfen werden. Für den Fall, dass die Holdback Queue keine Elemente mehr enthält wird eine entsprechende Ausgabe geloggt.

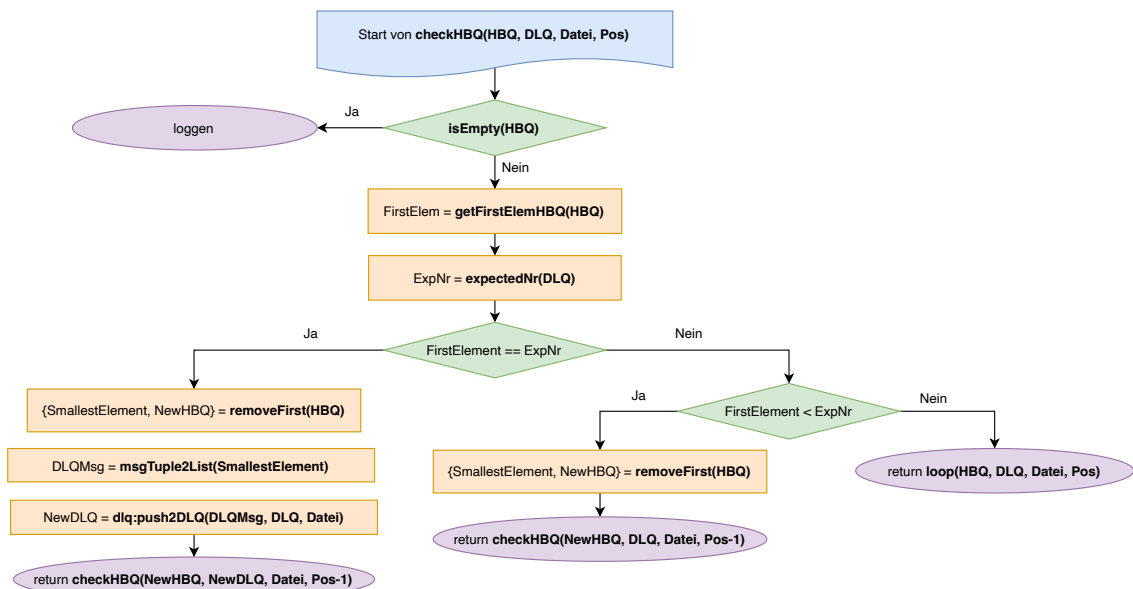


Abbildung 6: checkHBQ

Wenn also zum Beispiel am Anfang nur Elemente  $> 3$  in die Holdback Queue eingefügt wurden, dann fehlen die Elemente 1, 2 und 3 zum Einfügen in die Delivery Queue. In der Fehlermeldung steht folglich "Fehlernachricht für Nachrichten 1 bis 3 generiert.". Die 3 wird als Nachricht zusätzlich in die Delivery Queue eingefügt.

Die Funktion wird nach jeder Ausführung der pushHBQ Funktion aufgerufen. Somit ist sichergestellt, dass nach jeder Veränderung der Queue einmal geprüft wird, ob diese noch synchronisiert ist und keine für die Delivery Queue geeignete Nachricht vom Server geschickt wurde.

### 2.1.3 pushHBQ

In dieser Funktion wird eine Nachricht in die Queue geschrieben. Die Nachricht enthält die entsprechende Nachrichtennummer, den Inhalt der Nachricht und einen Timestamp, wann der Client die Nachricht abgeschickt, bzw. die Holdback Queue diese empfangen hat. Wie bereits beschrieben wird bedingt durch die fehlende Sortierung der Clients jedes Element als Blatt eingefügt.

Zum Bestimmen des Pfades der nächsten freien Blatts wird wie in der letzten Praktikumsaufgabe die `sortv.erl` Datei [Kla] verwendet. Hier wird als Parameter die Position, also der höchste freie Index mit übergeben. Dieser muss also bei jedem Einfügen eines Elements erhöht und beim Löschen eines Elements um eins verkleinert werden.

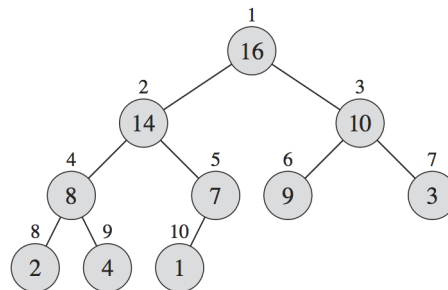


Abbildung 7: Binary Heap Index<sup>2</sup>

---

```

% sortv.erl
% @author: Prof Dr Christoph Klauck, HAW Hamburg
calcPath(Number) -> calcPath(Number, []).
% aktuelle Position ist Wurzel
calcPath(1, Accu) -> Accu;
% aktuelle Position ist gerade
calcPath(Number, Accu) when Number rem 2 == 0 -> calcPath(Number div 2, [l|Accu]);
% aktuelle Position ist ungerade
calcPath(Number, Accu) when Number rem 2 /= 0 -> calcPath((Number-1) div
    2, [r|Accu]).
  
```

---

Anhand dieses Indexes kann der Pfad durch modulo Rechnungen bestimmt werden. Wird wie im Beispiel in der Abbildung 7 eine ungerade Position, in diesem Fall die 11, übergeben, wird der Pfad `[l|r|r]` zurückgegeben. Es müssen also zuerst der linke Teilbaum und dann zweimal der rechte Teilbaum gewählt werden, um in das nächste freie Blatt zu gelangen. Die Position des höchsten Index wird als Integer zusammen mit der Holdback Queue als Parameter in der `loop` Funktion mit übergeben und somit rekursiv gespeichert.

Da in der Delivery Queue nur eine bestimmte Anzahl an Nachrichten stehen kann, wird bei der Holdback Queue ab einer erreichten Größe von  $\frac{2}{3} * \text{sizeof}(DLQ)$  der Inhalt der Queue reduziert. Dafür wird geprüft, wie groß die Lücke zwischen den beiden Queue

<sup>2</sup><https://weibeld.net/algorithms/data-structures.html>

ist. Ein Beispiel dafür ist in Abbildung 8 zu sehen. Hier hat die Holdback Queue diese bestimmte Größe erreicht und die Lücke in der Delivery Queue wird dementsprechend aufgefüllt. Aufgefüllt werden hierbei allerdings nicht alle fehlenden Nachrichten, sondern nur die von der Delivery Queue zuletzt erwartete. Eine ähnliche Fehlermeldung wie in der Abbildung wird in Textform in diese Nachricht als Msg eingetragen.

```
dlq>>> initialisiert mit Kapazität 17.
HBQ>>> Nachricht 2 in HBQ eingefügt.
HBQ>>> Nachricht 14 in HBQ eingefügt.
HBQ>>> Nachricht 12 in HBQ eingefügt.
HBQ>>> Nachricht 18 in HBQ eingefügt.
HBQ>>> Nachricht 9 in HBQ eingefügt.
HBQ>>> Nachricht 13 in HBQ eingefügt.
HBQ>>> Nachricht 3 in HBQ eingefügt.
HBQ>>> Nachricht 8 in HBQ eingefügt.
HBQ>>> Nachricht 17 in HBQ eingefügt.
HBQ>>> Nachricht 19 in HBQ eingefügt.
HBQ>>> Nachricht 11 in HBQ eingefügt.
HBQ>>> Nachricht 15 in HBQ eingefügt.
HBQ>>> Fehlermeldung fuer Nachrichten 1 bis 1 generiert.
dlq>>> Nachricht 1 in DLQ eingefügt.
```

Abbildung 8: Fehlermeldung Beispiel [Kla21b]

Die Elemente welche aus der Holdback Queue an die Delivery Queue gesendet wurden, müssen aus der Holdback Queue gelöscht werden, da ansonsten die Größe der Queue nicht aktualisiert werden kann. Dies geschieht durch regelmäßige Kontrolle nach dem Einfügen eines Elements, ob das kleinste Element der Holdback Queue in die Delivery Queue eingefügt werden kann (siehe 2.1.2). Das kleinste Element der Holdback Queue ist dessen Wurzelement. Dieses muss größer gleich der erwarteten Nummer der Delivery Queue sein. Ansonsten kommt es zu einem Ausnahmefall, welcher aber bedingt durch das Verwerfen von Nachrichten, welche kleiner als die erwartete Nummer sind, eigentlich nicht vorkommen dürfte.

Ein weiteres Problem entsteht, wenn Elemente eingefügt werden, welche eine kleinere Nachrichtennummer haben als das zu erwartende nächste Element der Delivery Queue. In diesem Falle wird die einzufügenden Nachricht verworfen und es wird `NNr|expNrDLQ` an den Prozess gesendet, welcher die Holdback Queue beauftragt hat.

#### 2.1.4 deliverMSG

In dieser Funktion wird die Delivery Queue über die Holdback Queue beauftragt die zu der übergebenen Nachrichtennummer zugehörige Nachricht zu senden. Der Client welcher die Nachricht empfangen soll wird als ProzessID mit im Parameter der Funktion übergeben. Über den Funktionsaufruf `dlq:deliverMSG(MSGNr, ClientPID, Queue, Datei)` wird also die entsprechend zu sendende Nachrichtennummer, die ProzessID, die Delivery Queue und die in die zu loggende Datei übergeben. Wenn die übergebene Nachricht nicht verfügbar ist, dann wird die Nachricht mit der nächst größeren Nummer gesendet. Als Antwort von

dem Prozess der Holdback Queue wird dann die gesendete Nachrichtennummer gesendet.

### 2.1.5 listADT

Diese Funktion ist aufgeteilt in zwei verschiedene Funktionen: listHBQ und listDLQ. Es wird in beiden der Inhalt der jeweiligen Queue ausgegeben. Dabei wird die Reihenfolge der Queue beibehalten. Ausgegeben werden alle enthaltenden Nachrichtennummern in Form einer Liste.

**listHBQ** Für diese Funktion wird über alle Elemente der Holdback Queue iteriert und jeweils die Nachrichtennummer in eine separate Liste geschrieben werden. Da die Holdback Queue hier in einer Heap Struktur umgesetzt ist, gibt es zwei mögliche Sortierungen. Zum einen könnte nach Index sortiert werden. Dementsprechend würde das Wurzelement als erstes ausgegeben werden, dann alle Elemente mit der nächst kleinsten Höhe von links nach rechts gelesen usw.. Die andere Möglichkeit wäre es den Heap bei der Ausgabe zu sortieren. Es würde also das Wurzelement als erstes ausgegeben werden, vor der nächsten Ausgabe wird dann aber erst der Heap neu strukturiert. Das Element mit dem höchsten Index wird die neue Wurzel. Dann versickert dieses so lange nach unten, bis es keinen kleineren Teilbaum mehr hat. Erst danach wird das nächste Element (also wieder die Wurzel) ausgegeben. Der Vorteil der Index-Sortierung ist die kleinere Laufzeit, da im Vergleich zur Nachrichtennummer-Sortierung nicht nach jeder Ausgabe neu strukturiert werden muss. Allerdings sind bei der Index-Sortierung die Nachrichtennummern in der Ausgabe zufällig, was z.B. das händische Finden eines Elements aufwändiger machen würde. Da die Funktionalität in diesem Falle im Vordergrund steht, wird bei listHBQ eine Liste in aufsteigend sortierter Reihenfolge ausgegeben.

Die Ergebnisse werden in eine log Datei geschrieben und bei Erfolg wird als Rückgabewert ein ok als Antwort gesendet.

**listDLQ** Da es schon eine Funktion listDLQ für die Delivery Queue gibt, wird diese hier verwendet. Als Queue wird einfach die in der Holdback Queue gespeicherte DLQ übergeben und die zurückgegebene Liste wird dann in die Logging Datei geschrieben.

### 2.1.6 dellHBQ

Um die HBQ zu löschen muss durch die rekursive Implementierung die Funktion loop/4 beendet werden. Dadurch, dass diese Funktion als Schleife implementiert ist, wird durch den dellHBQ die Abbruchbedingung simuliert. Nach diesem Aufruf sind dann weder die Elemente innerhalb der Queue noch vorhanden, noch ist die PID der Queue weiterhin referenziert. Außerdem wird das Löschen der Delivery Queue von dieser Funktion initialisiert.

### 3 Implementierung der Delivery Queue

Die Delivery Queue enthält alle Nachrichten, die an den Leser ausgeliefert werden dürfen. Sie hat einen begrenzten Speicher zur Verfügung, welcher nicht überschritten werden darf. Dieser wird bei Initialisierung der Queue übergeben und innerhalb der Queue gespeichert. Die eigentliche Delivery Queue wird als First In - First Out Liste implementiert. Die Nachricht die als erstes eingefügt wurde, verlässt also als erstes auch wieder die Liste (siehe Abbildung 9).

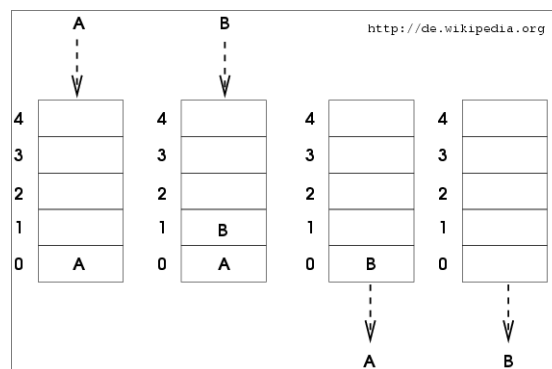


Abbildung 9: FIFO<sup>3</sup>

Dafür werden nun also die neuen Elemente immer an die Liste angehängt, so dass die Nachricht mit der kleinsten Nummer immer am letzter Stelle eingefügt wird. Hierfür bietet sich in Erlang das Anfügen an.

---

```
NewDLQ = OldDLQ ++ [NewMessage]
```

---

Da in dieser Funktion zwei Listen zu einer kombiniert werden, muss das zweite Element als Liste übergeben werden. Es ist als Form `NewMessage = [NNr, Msg, TScilentout, TShbqin, TSdlqin]` zwar schon eine Liste, aber da die Nachricht als ein Element eingefügt werden soll, muss es als Liste mit nur einem Element angefügt werden. Die Liste wird aufsteigend sortiert, da theoretisch genauso viele Elemente in die Liste eingefügt, wie auch wieder aus der Queue gelöscht werden.

Ein Problem bei der Delivery Queue stellt die vorgegebene maximale Größe dar. Da diese nicht im Speicherplatz reserviert werden kann (keine In-Place Lösungen in Erlang), wie zum Beispiel über ein `malloc` in C oder über ein Attribut wie in Java, muss die Größe hier im Prozess der Queue oder direkt in der Queue gespeichert werden. Die erste Möglichkeit zur Speicherung der Größe bietet aber viele Fehlerquellen. Die Implementierung würde wie folgt aussehen:

---

```
spawn(fun() -> loop(Size, Datei, 0) end).
```

---

```
loop(MaxSize, Datei, ActSize) ->
```

---

<sup>3</sup>[https://de.wikipedia.org/wiki/First\\_In\\_First\\_Out](https://de.wikipedia.org/wiki/First_In_First_Out)

---

```

receive
  {From, getVariables} ->
    From ! {reply, {MaxSize, Datei, ActSize}}
    loop(MaxSize, Datei, ActSize);
  {From, setVariables, {NewMaxSize, NewDatei, NewActSize}}
  From ! {reply, variablesSet}
  loop(NewMaxSize, NewDatei, NewActSize)
end.

```

---

Über die Schnittstelle  $\{self(), getVariables\}$  und  $\{self(), setVariables, \{NewMaxSize, NewDatei, NewActSize\}\}$  können nun sowohl von der Holdback Queue, als auch von der Delivery Queue aus das Limit der Delivery Queue, die aktuelle Größe und die logging-Datei gelesen und geschrieben werden. Die Delivery Queue wäre somit eine zum Teil entfernte abstrakte Datenstruktur. Ein Problem, welches nach der Implementierung entstanden ist, war das sehr aufwändige Debuggen von Fehlern oder Aufrufen innerhalb der Delivery Queue.

Eine einfachere Lösung wäre das Speichern der Variablen innerhalb der Queue. Diese neue Queue hat die Struktur eines Tupels mit drei Elementen, welche zum einen die maximale Größe und die aktuelle Größe sind und zum anderen die eigentliche Queue in Form einer Liste - also  $DLQ = \{MaxSize, ActSize, [Msg1, Msg2, \dots]\}$ . So werden mögliche Fehler, welche durch Nebenläufigkeiten entstehen können, eliminiert.

Im Folgenden wird mit der zweiten Lösung, also der Speicherung der Elemente innerhalb der Delivery Queue als Tupel, gearbeitet.

### 3.1 Operationen

#### 3.1.1 initDLQ

In der Initialisierungsfunktion der Delivery Queue wird die Queue erzeugt. Die Funktion hat den Aufruf `initHBQ` und bekommt die maximale Größe und die logging-Datei mit übergeben. Nach erfolgreicher Initialisierung wird die Initialisierungsgröße der Delivery Queue geloggt und ein Tupel mit den Elementen `MaxSize`, `ActSize` und der eigentlichen Queue zurückgegeben.

#### 3.1.2 delDLQ

Da die Delivery Queue in dieser Implementierung nicht als entfernte abstrakte Datenstruktur umgesetzt wurde, muss auch dementsprechend kein Prozess beendet werden. Die Funktion gibt also beim Aufruf nur ein 'ok' zurück.

#### 3.1.3 expectedNr

Diese Funktion liefert die Nachrichten Nummer die als nächstes in der Delivery Queue gespeichert werden kann. Da die kleinste Nummer von dem Leser Client benötigt wird

und außerdem keine Duplikate vorkommen sollten, wird in der Delivery Queue die größte Nummer gesucht und diese um eins erhöht. Die größte Nummer ist stets das letzte Element der Delivery Queue. Es wird also rekursiv eine Teilliste der Queue aufgerufen und das erste Element dieser gespeichert, bis die Teilliste leer ist (siehe getLastElem/1). Auf das zuletzt gespeicherte Element wird eine 1 addiert und das Ergebnis ist die erwartete Nummer.

---

```
getLastElem([_NNr, _Msg, _TSclientout, _TShbqin, _TSdlqin] | []) -> NNr;
getLastElem([_Head|Tail]) -> getLastElem(Tail).
```

---

### 3.1.4 push2DLQ

Diese Funktion wird von der Holdback Queue aufgerufen, wenn diese eine bestimmte Größe erreicht hat und Elemente an die Delivery Queue übergibt. Die zugehörige Schnittstelle der Holdback Queue ist {self(), {request, pushHBQ, Msg}}.

Die Funktion push2DLQ speichert die übergebene Nachricht in der Delivery Queue. Da die Queue bereits sortiert ist und das letzte Element in der Queue das Größte ist, kann das neue Element einfach an die Liste angefügt werden. Zusätzlich wird ein Zeitstempel angefügt, welcher über eine von Erlang bereitgestellte Funktion erfasst wird. Bei jedem Funktionsaufruf wird außerdem die maximale Größe mit der aktuellen Größe verglichen. Wenn die Delivery Queue die maximale Größe erreicht hat, dann wird beim Einfügen eines neuen Elements das Älteste gelöscht. Dies kann in Erlang sehr effizient umgesetzt werden. Durch die Aufteilung der Liste in Startelement und Restliste kann zum Löschen des ersten Elements einfach mit der Restliste weitergearbeitet werden.

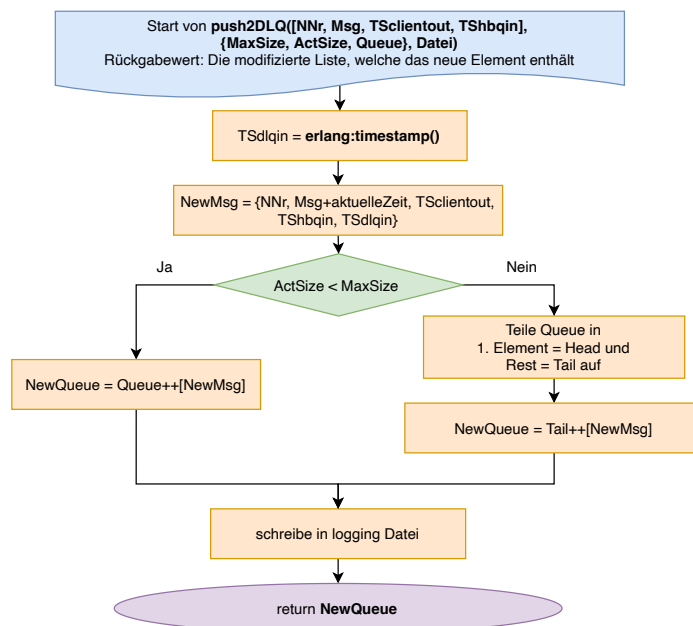


Abbildung 10: pushDLQ

### 3.1.5 deliverMSG

Diese Funktion sendet die im Parameter übergebene Nachrichtennummer an die übergebene ProzessID des Clients. Dafür wird durch die Elemente der Delivery Queue gelaufen, bis das Element entweder gefunden wurde oder das übergebene Element größer ist.

---

```
getMSGatMSGNr(_MSGNr, []) -> [-1,nokb,0,0,0];
getMSGatMSGNr(MSGNr, [[NNr, Msg, TSclientout, TShbqin, TSdlqin]|_Tail]) when
    MSGNr =< NNr->
    [NNr, Msg, TSclientout, TShbqin, TSdlqin];
getMSGatMSGNr(MSGNr, [_Head|Tail]) -> getMSGatMSGNr(MSGNr, Tail).
```

---

In obiger Funktion ist noch einmal das rekursive Durchlaufen der Liste gezeigt. Der Funktion wird eine Liste übergeben, in diesem Falle die Delivery Queue und dann wird von dieser Liste so oft das erste Element abgeschnitten, bis eine der beiden Abbruchbedingungen eintreffen. Anhand von Pattern Matching kann dann effizient verglichen werden, ob die Liste leer ist oder ob die erste Nachricht der übergebenen Liste größer gleich der gesuchten Nachrichtennummer ist. Gelöscht werden die Elemente aus der Delivery Queue allerdings erst, sobald diese ihre maximale Größe erreicht hat und somit durch die Funktion push2HBQ verkleinert wird. Das eigentliche Senden der Nachricht an den Clienten findet innerhalb der Delivery Queue statt.

Der Aufruf hat den Aufbau: ClientPID ! {reply,SendMessage,Terminated}

Um den gesamten Prozess terminieren zu können, wird der Delivery Queue von der Hold-back Queue mitgeteilt ob diese noch Elemente enthält. Wenn das nicht mehr der Fall ist wird die Nachricht [-1,nkob,0,0,0] mit einem weiteren Parameter true übergeben, welcher signalisiert, dass der Prozess beendet werden soll.

### 3.1.6 listDLQ

Die Funktion gibt eine Liste mit den Nummern der in der Delivery Queue enthaltenen Nachrichten zurück. Die Nummern werden nach aufsteigender Größe sortiert sein, da am Kopf der Queue angefangen wird. Somit ist also auch die Reihenfolge der Liste eingehalten.

### 3.1.7 lengthDLQ

Diese Funktion gibt die Anzahl der in der Delivery Queue enthaltenen Nachrichten zurück. Dafür kann die im Tupel der Delivery Queue enthaltende Variable ActSize zurückgegeben werden.



## 4 Analyse

### 4.1 Messaufbau

Um die verschiedenen Implementationen zu testen, wird ein Benchmark genutzt, welches von Matz Heitmüller (ITS3) freundlicherweise zur Verfügung gestellt wurde. In diesem Benchmark werden über eine Funktion eine bestimmte Menge an Elementen, in diesem Falle Nachrichten mit drei Elementen - der Nachrichtennummer, dem Text 'Dummy' und einem Timestamp - gesendet und empfangen. Der Ablauf des Client-Server-Systems kann unter verschiedenen Bedingungen simuliert werden. So kann die Liste mit welchem die Holdback Queue gefüllt wird unterschiedliche Vorsortierungen haben. Damit können die Heap und List Struktur verglichen werden. Der Ablauf des Benchmarks ist ähnlich zu dem der Laufzeitmessungen aus den vorherigen Praktika. Die Zeit zum Senden und Empfangen der Eingabelisten wird in einer csv-Datei für verschiedene Listengrößen gespeichert und nach Abschluss der Messung über die Python matplotlib Library geplottet. Die Schrittgröße zum Vergrößern und die Sortierung der Eingabelisten, die Anzahl der Startelemente, die der Schritte und die übergebenen Holdback Queues sind parametrisiert. So können in einem Plot auch mehrere Implementierungen miteinander verglichen werden. An der y-Achse ist die Zeit des Sende-Empfangs-Prozesses in Millisekunden pro Element und an der x-Achse die Größe der Eingabeliste dargestellt. Eine weitere Funktion des Benchmarks ist das Erzeugen einer Eingabeliste, anhand welcher die aus der Aufgabenstellung hervorgehende reale Bedingung simuliert werden kann. Das Limit der Delivery Queue wird dynamisch erhöht und ergibt sich aus  $DLQLimit = \text{ceil}(InputVal * \text{length}(Eingabeliste) / 100)$ . Die Variable InputVal ist als Default 100, kann im Benchmark aber auch verändert werden. Bei Default Einstellungen ist das Limit der Delivery Queue also immer gleich der Größe der Eingabeliste. Analysiert werden die zwei Holdback Queue Strukturen unter verschiedenen Bedingungen und außerdem die Implementierungen mit und ohne Pattern Matching. Die Startgröße der Liste wird im Folgenden 100 sein, die Schrittgröße liegt bei 100 und die Anzahl der Schritte ist entweder 100 oder 250.

### 4.2 Messergebnisse

Der Plot aus Abbildung 11 zeigt die Messergebnisse des Benchmarks der Holdback Queue mit und ohne Pattern Matching im Vergleich. Die orange Trendlinie und die blauen Punkte gehören zur Holdback Queue mit Pattern Matching. Gemessen wurde mit 100 Schritten, die zuletzt übergebene Liste enthielt also 10100 Elemente. Die Nachrichten in der Liste waren aufsteigend sortiert. Auffällig ist, dass sich die Messungen kaum unterscheiden. Die Trendlinien sind beide linear und scheinen nahezu identisch zu sein. Auch die Streuung der Messwerte um diese Trendlinien ist sehr gering. Die Dauer des Senden und Empfangens von 10000 Elementen beträgt 0,24ms pro Element.



Abbildung 11: aufsteigend - vgl. Heap (mit und ohne Pattern-Matching)

In dem Plot aus Abbildung 12 ist der Vergleich der Holdback Queue implementiert mit interner Liste und mit internem Heap gezeigt. Gemessen wurde mit 100 Schritten, die Sortierung der Elemente ist wieder aufsteigend. Die orange Linie und die blauen Punkte gehören wieder zu dem Heap. Dessen Messwerte sind sehr ähnlich zu den Messwerten des oberen Plots. Allerdings gilt das auch für die List Messwerte. Die Streuung ist hier etwas größer, allerdings immer noch sehr gering und voraussichtlich eher Hintergrundprozessen des Betriebssystems während der Messdurchführung geschuldet. Die Trendlinien sind somit auch wieder linear. Hier dauert der Prozess bei 10000 Elementen wie bei der letzten Messung 0,24ms pro Element.

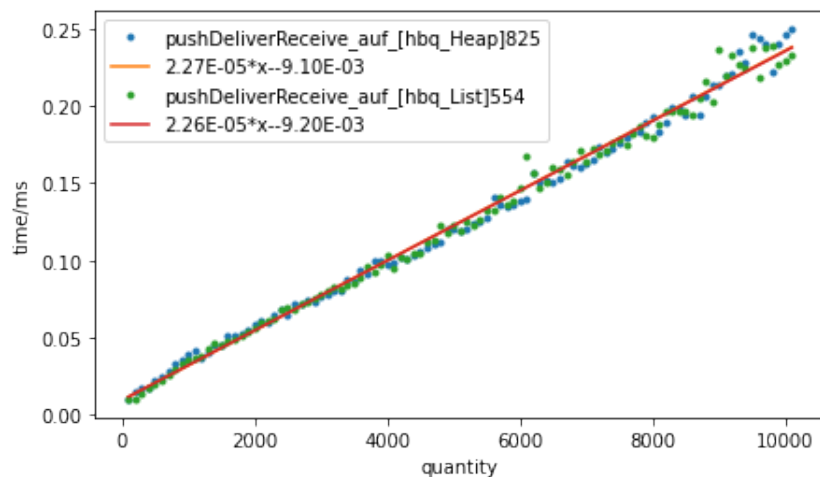


Abbildung 12: aufsteigend - vgl. Heap, List

Dieser Plot (Abbildung 13) zeigt die Messungen der zwei verschiedenen Holdback Queue Strukturen, dieses Mal bei zufällig sortierter Liste. Auch hier wurde die Messung mit 100 Schritten durchgeführt. Im Vergleich zu den letzten beiden Plots ist hier ein deutlicher Unterschied zwischen den beiden Messungen zu erkennen. Der Heap hat nun die rote

Trendlinie und die grünen Punkte. Der Heap ist bei 10000 Elementen in der Eingabeliste bei 0,175ms pro Element, während die Liste hier 0,225ms braucht. Das ergibt eine Differenz von 0,05ms bei 10000 Elementen. Außerdem ist auffällig, dass die Messungen bei zufälliger Eingabeliste um mindestens 0,015ms schneller war. Der Verlauf beider Messungen ist linear, allerdings ist die Streuung der Werte in der Messung für die Liste deutlich höher. Bedingt kann dies aber auch daran liegen, dass die Eingabeliste immer zufällig sortiert ist. Durch die Implementierung der Holdback Queue als Liste ist die Sortierung einer eher aufsteigenden Liste deutlich schneller als die einer eher absteigend sortierten.

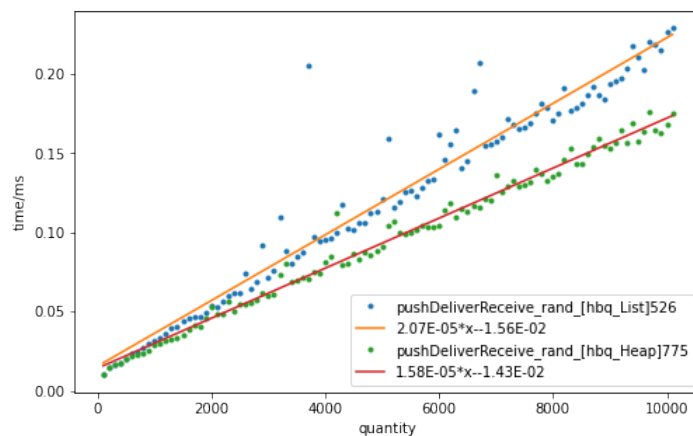


Abbildung 13: random - vgl. Heap, List

Im Plot aus Abbildung 14 werden die Messungen der Holdback Queue mit internem Heap und mit interner Liste bei realen Bedingungen gezeigt. Dies bedeutet, dass die Eingabeliste so sortiert ist, wie die Elemente von dem Client System an den Server, bzw. von dem Server an die Holdback Queue gesendet werden. Die Reihenfolge der Nachrichten ist annähernd aufsteigend. Vorstellen kann man sich das wie eine lineare Trendlinie mit hoher Wertestreuung. Die Trendlinien sind wieder nahezu identisch und die Werte sind gering gestreut. Das Senden und Empfangen von 10000 Elementen dauert 0,27ms pro Element.

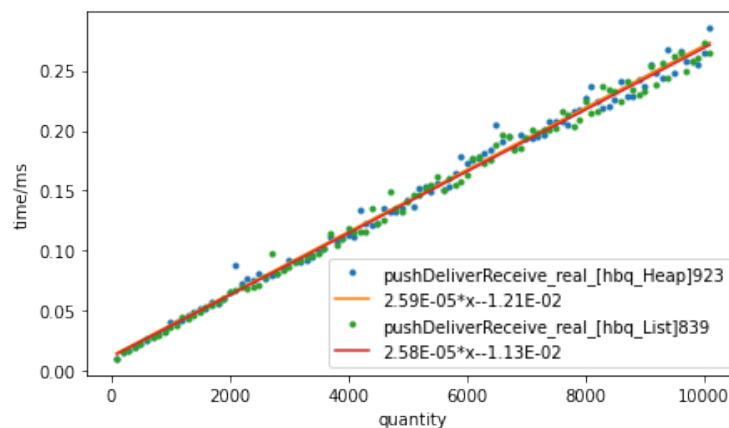


Abbildung 14: real - vgl. Heap, List

Im nächsten Plot (Abbildung 15) sind die Messungen der drei verschiedenen Holdback Queues (interner Heap, interne Liste und kein Pattern-Matching) mit 250 Schritten dargestellt. Im letzten Durchlauf wurden hier also 25100 Elemente in die Holdback Queue eingefügt. Der Benchmark wurde wieder unter realen Bedingungen simuliert. Der Verlauf der Trendlinien ist linear und die, der drei verschiedenen Implementierungen, liegen wieder sehr nah beieinander. Bei 25000 Elementen in der Eingabeliste ist in dieser Messung die Holdback Queue mit interner Liste mit 0,68ms pro Element am schnellsten, danach folgt die Queue mit internem Heap, aber ohne Pattern Matching, mit 0,74ms und danach die Holdback Queue mit internem Heap und Pattern Matching. Diese hat eine benötigte Dauer von 0,76ms. Die Streuung der Werte ist zu Beginn sehr gering und wird bei steigender Größe immer höher.

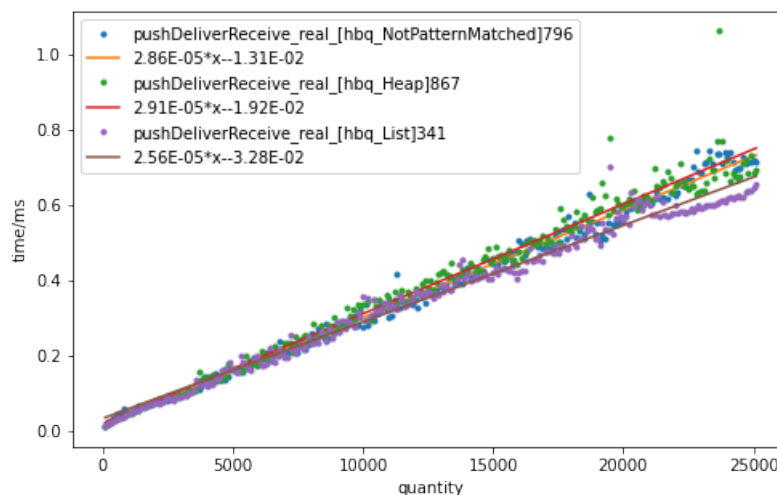


Abbildung 15: real - vgl. List, Heap (mit und ohne Pattern-Matching)

Im Plot aus Abbildung 16 werden die Holdback Queues mit interner Liste und Heap jeweils mit verschiedenen Delivery Queue Limits initialisiert. Diese Delivery Queue Limits entsprechen 1% und 10% der übergebenen Eingabeliste. Zu Beobachten ist, dass die Holdback Queues mit kleineren Delivery Queue Limits fast um das 5-fache schneller sind. Die Trendlinien der beiden Queues mit 1% sind fast identisch, die Holdback Queue mit internem Heap (blaue Werte) beendet den Prozess bei 10% etwas schneller als die mit interner Liste. Allerdings sind die blauen Werte vereinzelt stark gestreut.

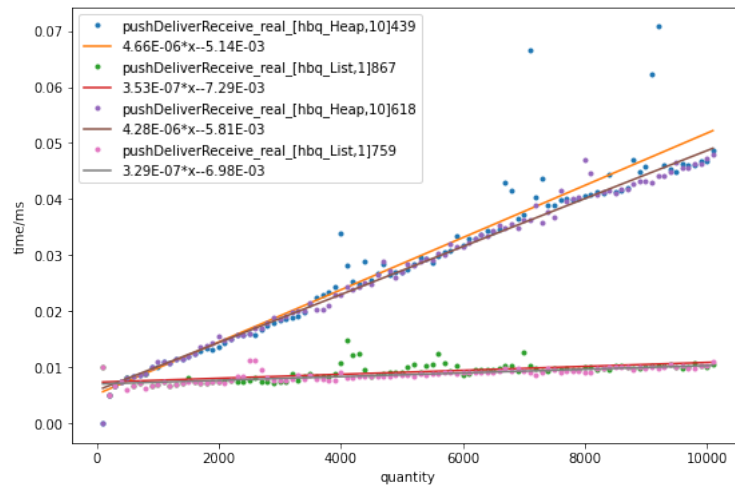


Abbildung 16: real - vgl. Heap, List (1%,10%)

Im letzten Plot (Abbildung 17) wurden wieder die Delivery Queue Limits parametrisiert. Statt 1% und 10% hier auf jeweils 10% und 100%. Gemessen wurden die zwei verschiedenen Holdback Queue Strukturen mit random sortierten Eingabelisten. Die Messungen bei 10% sind wieder fast identisch und haben einen sehr flachen Verlauf. Bei 10000 Elementen dauert der Prozess 0,038ms pro Element. Die Messungen bei 100% ergeben hingegen 0,175ms pro Element für den internen Heap und 0,225ms für die interne Liste. Hier ist der interne 0,05ms

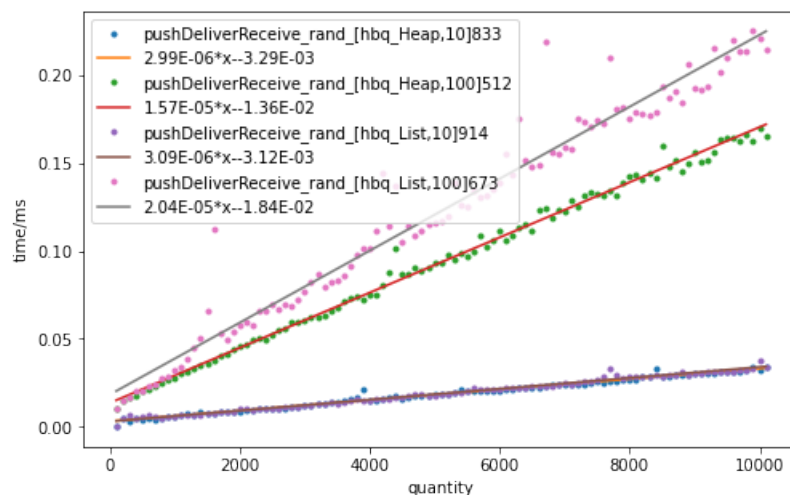


Abbildung 17: rand - vgl. Heap, List (10%,100%)

## 5 Fazit

### 5.1 Heap oder List

Die Holdback Queue mit internem Heap und mit interner Liste wurden in den obigen Plots in verschiedenen Szenarien getestet und verglichen. In Abbildung 12 ist zu erkennen, dass bei aufsteigender Eingabeliste kein Unterschied zwischen den beiden Implementierungen zu erkennen ist. Wie in Kapitel 1.6 beschrieben, werden die Elemente bei der Liste jeweils hinten angefügt, während hingegeben beim Heap alle Elemente an den nächsten leeren freien Index angefügt werden. Da beim Anfügen an die Liste durch die ganze Liste iteriert wird, beim Heap aber hingegen nur vom Wurzelement aus das letzte freie Blatt gesucht werden muss, entsteht eine Komplexität von  $O(n)$  für die Liste und eine von  $O(\log(n))$  für den Heap. Beim Entfernen eines Elements hingegen, wird das erste Element der Liste abgeschnitten und mit dem Rest der Liste weitergearbeitet was eine Komplexität von  $O(1)$  zur Folge hat. Beim Heap hingegen muss das Wurzelement entfernt und dann das neue Wurzelement bestimmt werden was in  $O(2 \cdot \log(n))$  resultiert. Die Gesamtkomplexität ist also  $O(n)$  für die Liste und  $O(3 \cdot \log(n))$  für den Heap. Folglich sollte der Heap also schneller sein als die Liste.

Nun wurde im Benchmark allerdings immer die Delivery Queue mit eingebunden. Es lässt sich also schließen, dass diese so ineffizient ist, dass die Optimierungen der Holdback Queue wenig Einfluss auf den Gesamtprozess hat. Auch hier gilt 'Die Kette ist nur so stark wie das schwächste Glied', denn egal wie schnell ein Teilprozess Elemente verarbeitet, wenn der Nachfolgeprozess nicht mindestens genauso schnell ist, bildet sich im Nachfolgeprozess ein Stau und die Optimierung des ersten Teilprozesses wird hinfällig. Zur Bestätigung dieser Vermutung wurde eine Messung mit den einzelnen Holdback Queues durchgeführt (siehe Abbildung 18). Gemessen wurde hierbei nur der Prozess vom Eintritt der Nachrichten in die Holdback Queue, bis zum Verlassen mit aufsteigender und realer Eingabeliste.

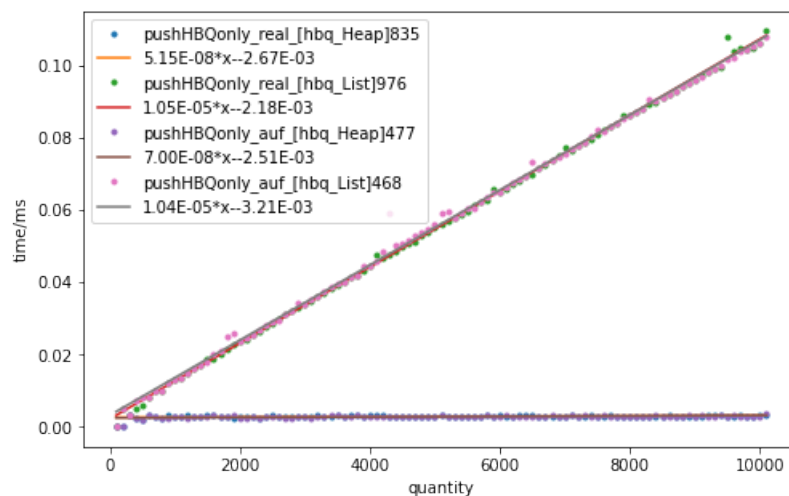


Abbildung 18: real - vgl. List, Heap ohne Delivery Queue

Hier ist der Unterschied zwischen den Implementierungen mit interner Liste und internem Heap klar zu erkennen. Die Trendlinien der aufsteigenden Eingabeliste sind nahezu identisch und die der real sortierten auch. Allerdings ist die Steigung der Holdback Queue mit interner Liste ca. 30° steiler.

Die Umsetzung der Holdback Queue mit einem internen Heap, um die Elemente besser zu sortieren, erweist sich also als durchaus effizient. Bei einer neuen Implementierung dieser gesamten Aufgabe sollte also auch die Delivery Queue und nicht nur die Holdback Queue optimiert werden. Eine mögliche Herangehensweise wäre es, die Sortierung der Delivery Queue zu verändern, sie also zum Beispiel absteigend zu sortieren. Dafür müsste analysiert werden, ob wirklich genauso viele Elemente in die Queue eingefügt, wie auch wieder gelöscht werden, wie es in Kapitel 3 angenommen wurde. Ein Ausgang dieser Analyse könnte sein, dass nach Terminierung der Holdback Queue keine weiteren Elemente mehr in die Delivery Queue eingefügt werden müssen und somit auch keine mehr aus dieser gelöscht werden. Somit wäre also eine Delivery Queue mit absteigender Liste effizienter, da das von Erlang angebotene 'aneinander-pipen' von Elementen schneller ist, als der '++' Operator.

## 5.2 Limit der Delivery Queue

Die Plots aus Abbildung 16 und Abbildung 17 zeigen, wie groß der Einfluss des Delivery Queue Limits auf die Gesamtlaufzeit ist. Je kleiner dieses Limit hier ist, desto schneller ist der Prozess. Da die Elemente ab Erreichen einer Holdback Queue Größe von 2/3 des Delivery Queue Limits von der Holdback an die Delivery Queue weitergegeben werden, wird bei einem kleineren Delivery Queue Limit mit kleineren Holdback Queues gearbeitet. Ein Element in eine kleinere Liste oder einen kleineren Heap einzufügen dauert dementsprechend weniger, da weniger Iterationen durchgeführt, bzw. Teilbäume gesucht werden müssen.

## 5.3 Pattern Matching

Pattern Matching um die Effizienz der Holdback Queue zu erhöhen, hat sich anhand der Plots aus Abbildung 11 und Abbildung 15, im Widerspruch zur Annahme aus Kapitel 1.7 eher als wenig maßgebend erwiesen.

Da vorherige Auswertungen der Plots bereits erwiesen haben, dass die Delivery Queue einen starken Einfluss auf die Laufzeiten haben, wurde eine weitere Messung durchgeführt (siehe Abbildung 19). In dieser wird mit einer Eingabeliste, welche bis zu 25000 Elementen in aufsteigend sortierter Reihenfolge hat, noch einmal die Holdback Queue mit und ohne Pattern Matching gemessen. Die Queue ohne Pattern Matching ist anfangs noch leicht ineffizienter, dessen Trendlinie verläuft aber leicht flacher als die der Holdback Queue mit Pattern Matching. Da die Zeiten der Messdurchläufe hier sehr gering sind ist die Streuung sehr hoch. Da diese aber sehr willkürlich ist, wird sie nicht weiter ausgewertet.

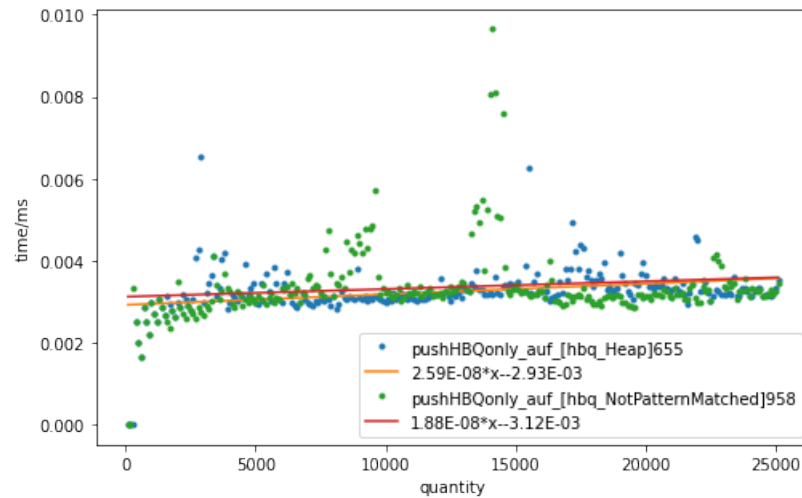


Abbildung 19: auf - vgl. Heap ohne Delivery Queue (mit und ohne PatternMatching)

Zum Vergleich der beiden verschiedenen Implementierungen folgt ein Codeauszug aus der Holdback Queue ohne Pattern Matching. Dieses Beispiel soll die Tiefe des Codes und nicht die Funktionalität veranschaulichen, daher wird die Initialisierung mancher Parameter nicht gezeigt.

---

```

loop(HBQ, DLQ, Datei, Pos, DLQLimit) ->
  receive
    {From, {request, pushHBQ, [NNr, Msg, TSclientout]}} ->
      if
        NNr < ExpNr ->
          true ->
            if
              Pos < (DLQLimit*2/3) ->
                true ->
                  if
                    SNNr == ExpNr ->
                      ...
                    true ->
                      ...
                  end
                end
              end
            end
          end;

```

---

Nun folgt das Beispiel der Holdback Queue mit Pattern Matching. Auch hier wurden wieder große Teile des Codes entfernt, um nur die Tiefe zu verdeutlichen. Statt fast die gesamte Funktion in einem Block zu implementieren, werden hier die zwei Hilfsfunktionen pushHBQ/8 und pushHBQHelp/10 verwendet.

---



```

loop(HBQ, DLQ, Datei, Pos, DLQLimit) ->
    receive
        {From, {request, pushHBQ, [NNr, Msg, TSclientout]}} ->
            pushHBQ([NNr, Msg, TSclientout, erlang:timestamp()], ExpNr, HBQ, DLQ,
                Datei, Pos, DLQLimit, From);
        ...

% -----
pushHBQ([NNr, _Msg, _TSclientout, _TShbqin], ExpNr, HBQ, DLQ, Datei, Pos,
    DLQLimit, From) when NNr < ExpNr -> ...;
pushHBQ([NNr, Msg, TSclientout, TShbqin], _ExpNr, HBQ, DLQ, Datei, Pos,
    DLQLimit, From) when Pos < (DLQLimit*2/3) -> ...;
pushHBQ([NNr, Msg, TSclientout, TShbqin], ExpNr, HBQ, DLQ, Datei, Pos, DLQLimit,
    From) ->
    pushHBQHelp([NNr, Msg, TSclientout, TShbqin], ExpNr, DLQ, Datei, Pos,
        DLQLimit, DLQMsg, SNNr, TempHBQ, From).

pushHBQHelp([NNr, Msg, TSclientout, TShbqin], ExpNr, DLQ, Datei, Pos, DLQLimit,
    DLQMsg, SNNr, TempHBQ, From) when SNNr == ExpNr -> ...;
pushHBQHelp([NNr, Msg, TSclientout, TShbqin], ExpNr, DLQ, Datei, Pos, DLQLimit,
    DLQMsg, SNNr, TempHBQ, From) -> ...;

```

Auffällig ist sofort, dass die erste Variante trotz der vielen Ebenen übersichtlicher wirkt als die zweite. Durch die vielen Parameter (pushHBQHelp/10 hat entsprechend zehn Parameter), welche den Hilfsfunktionen im unteren Auszug übergeben werden, strecken sich die Funktionsköpfe und sind schwer zu lesen. Letztendlich ist der Code aber kompakter und beim Hinzufügen weiterer Ebenen, wird wahrscheinlich die Variante mit dem Pattern Matching wieder übersichtlicher.

Die Zeitmessungen haben wie bereits erwähnt erwiesen, dass keine Variante effizienter als die andere ist. Laut dem von Erlang gegebenen 'Efficiency Guide'<sup>4</sup> wird das Pattern Matching vom Compiler zu einem switch-case generiert.

```

% Pattern Matching
foo(X, []) -> X;
foo(X, [Head|Tail]) -> Tail.
% wird zu Code kompiliert, welcher folgendem Beispiel gleicht:
foo(X,Y) ->
    case Y of
        [] -> X;
        [Head|Tail] -> Tail
    end.

```

Das switch-case Statement im Allgemeinen ist sehr effizient. Der Compiler erstellt eine Art

<sup>4</sup>[https://www.erlang.org/doc/efficiency\\_guide/functions.html#pattern-matching](https://www.erlang.org/doc/efficiency_guide/functions.html#pattern-matching)

Tabelle mit den möglichen Werten, welche alle den gleichen Typen wie die übergebene Variable haben. Statt dann wie beim if-else Statement jede Bedingung zu prüfen, kann beim Aufruf des switch-case Statements der richtige Wert aus der Tabelle gesucht und der zugehörige Pfad ausgewählt werden. Ab fünf Fällen wird der switch-case deutlich effizienter als der if-else, da dann als Tabelle intern ein Lookup-Table oder eine Hash-List implementiert ist und somit alle Elemente innerhalb der gleichen Zeit gefunden werden können (frei nach [Mig17]). Da in dem Code dieser Aufgabe nie mehr als fünf Fälle geprüft wurde, ist das Optimieren durch Pattern Matching hier nicht ausschlaggebend. Außerdem benötigen die Hilfsfunktionen und die vielen Parameter mehr Speicher und Zeit, diesen zu schreiben und zu lesen. Erkennen tut man dies an den Größen der Kompilierten beam-Dateien. Somit hat die kompilierte Datei der Holdback Queue ohne Pattern Matching 1kB (16%) weniger als die andere. Im Plot der Abbildung 15 ist ab einer Eingabelistengröße von ca. 10000 auch zu erkennen, dass die Holdback Queue ohne Pattern Matching (die orange Trendlinie) leicht effizienter als die mit ist.

Beim Verwenden von wenig Parametern und wenig Hilfsfunktionen ist Pattern Matching also die bessere Variante, besonders, wenn mit vielen Fällen innerhalb der Funktion gearbeitet wird.

## 5.4 Alternative Anwendung

Die Holdback und Delivery Queue in dieser Form können in vielen anderen Anwendungsgebieten angewendet werden. Dabei sollten aber Modifizierungen hinsichtlich der Sortialgorithmen innerhalb der Holdback Queue vorgenommen werden. Wenn Werte aus einer großen Menge komplett zufällig an den Server gesendet werden und mehrere Clients nun die Werte in richtiger Reihenfolge erwarten, dann sollte die Größe der Holdback Queue fast so groß, wie die der Wertemenge sein, damit keine Werte durch einen Overflow der Holdback Queue verloren gehen. Die interne Struktur der Holdback Queue wäre dann der Heap. Effizient wäre diese abstrakte Datenstruktur, da die Sortierung der Werte unabhängig von der Ausgabe an die Clients stattfindet. Würde nur die Holdback Queue genutzt werden und diese würde die Elemente über einen Heap sortieren, dann würde das kleinste Element als Wurzel solange dort stehen, bis alle Clients es empfangen haben und erst danach könnte das neue kleinste Element gesucht werden. Angenommen die Elemente werden in aufsteigender Reihenfolge an den Server geschickt, wäre nur eine Delivery Queue am sinnvollsten, die Größe muss nicht der Wertemenge entsprechen, sondern hängt jetzt von der Geschwindigkeit der Clients ab.

## Literaturverzeichnis

- [Heb13] Fred Hebert. *Learn You Some Erlang For Great Good*. Jan. 2013. URL: <https://learnyousomeerlang.com/>.
- [Kla] Christoph Klauck. *sortv.erl*.
- [Kla21a] Christoph Klauck. *AD\_Aufgabe\_HA*. 2021.
- [Kla21b] Christoph Klauck. *Client\_1clientA@Qigong-KLC.log*. 2021.
- [Kla21c] Christoph Klauck. *HB-DLQ@Qigong-KLC.log*. 2021.
- [Mig17] Gaurav Miglani. *switch vs if else*. Jan. 2017. URL: <https://www.geeksforgeeks.org/switch-vs-else/>.
- [Sch21] Leon Schwarzenberger; Kristoffer Schaaf. "Entwurf Praktikum 2, Algorithmen und Datenstrukturen". Dez. 2021.

## Abbildungsverzeichnis

1	Nachrichtendienst [Kla21a]	2
2	Auswertung Sortieralgorithmen [Sch21]	4
3	Nachrichtendienst [Kla21c]	5
4	Binary Min Heap Insert	6
5	Nachrichtendienst [Kla21c]	7
6	checkHBQ	8
7	Binary Heap Index	9
8	Fehlermeldung Beispiel [Kla21b]	10
9	First In First Out	12
10	pushDLQ	14
11	aufsteigend - vgl. Heap (mit und ohne Pattern-Matching)	17
12	aufsteigend - vgl. Heap, List	17
13	random - vgl. Heap, List	18
14	real - vgl. Heap, List	18
15	real - vgl. List, Heap (mit und ohne Pattern-Matching)	19
16	real - vgl. Heap, List (1%,10%)	20
17	rand - vgl. Heap, List (10%,100%)	20
18	real - vgl. List, Heap ohne Delivery Queue	21
19	auf - vgl. Heap ohne Delivery Queue (mit und ohne PatternMatching)	23
20	insertHeap of Holdback Queue	27
21	removeFirst/Last Heap of Holdback Queue	28
22	weitere Heap Funktionen	29

## A Anhang

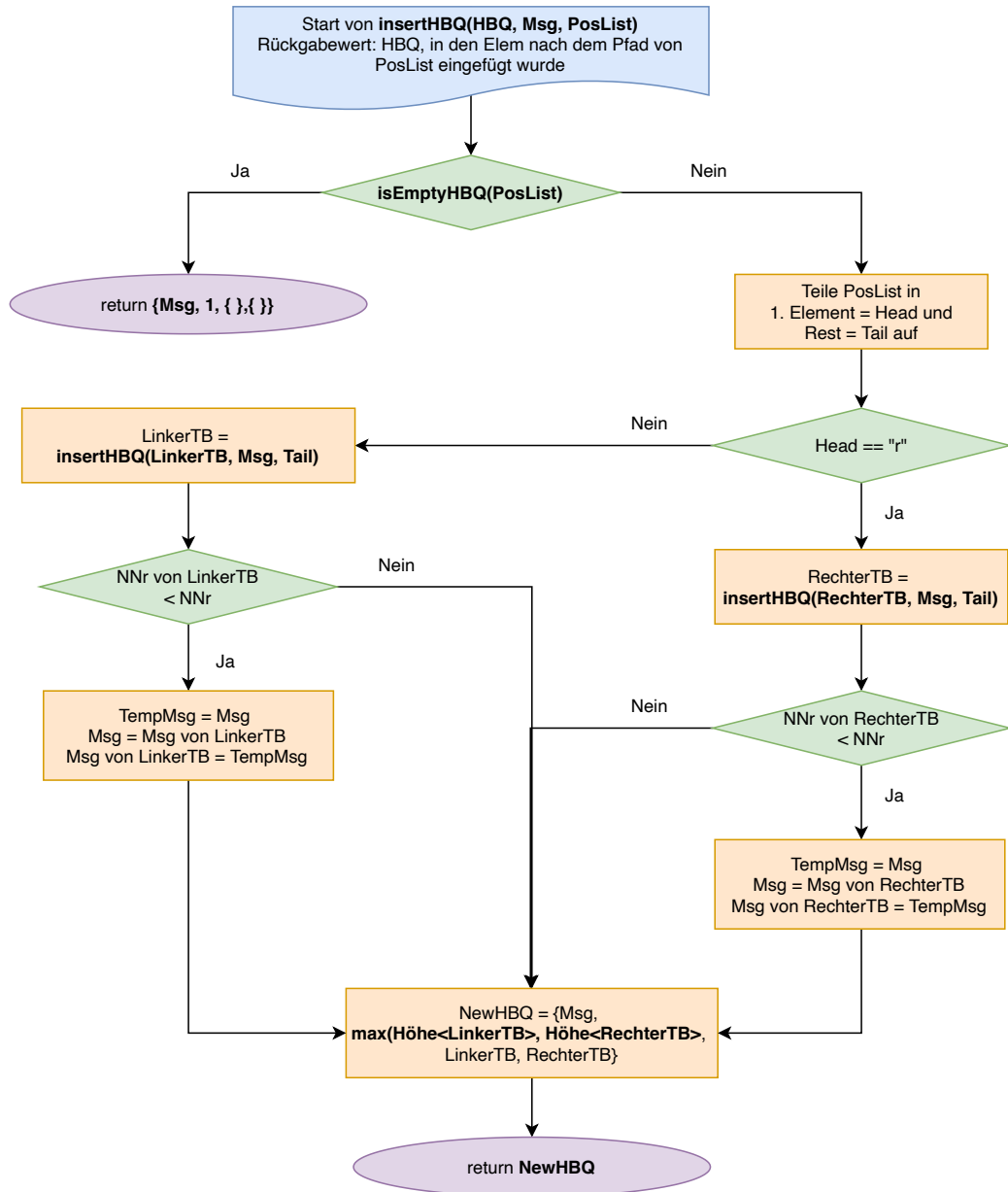


Abbildung 20: insertHeap of Holdback Queue

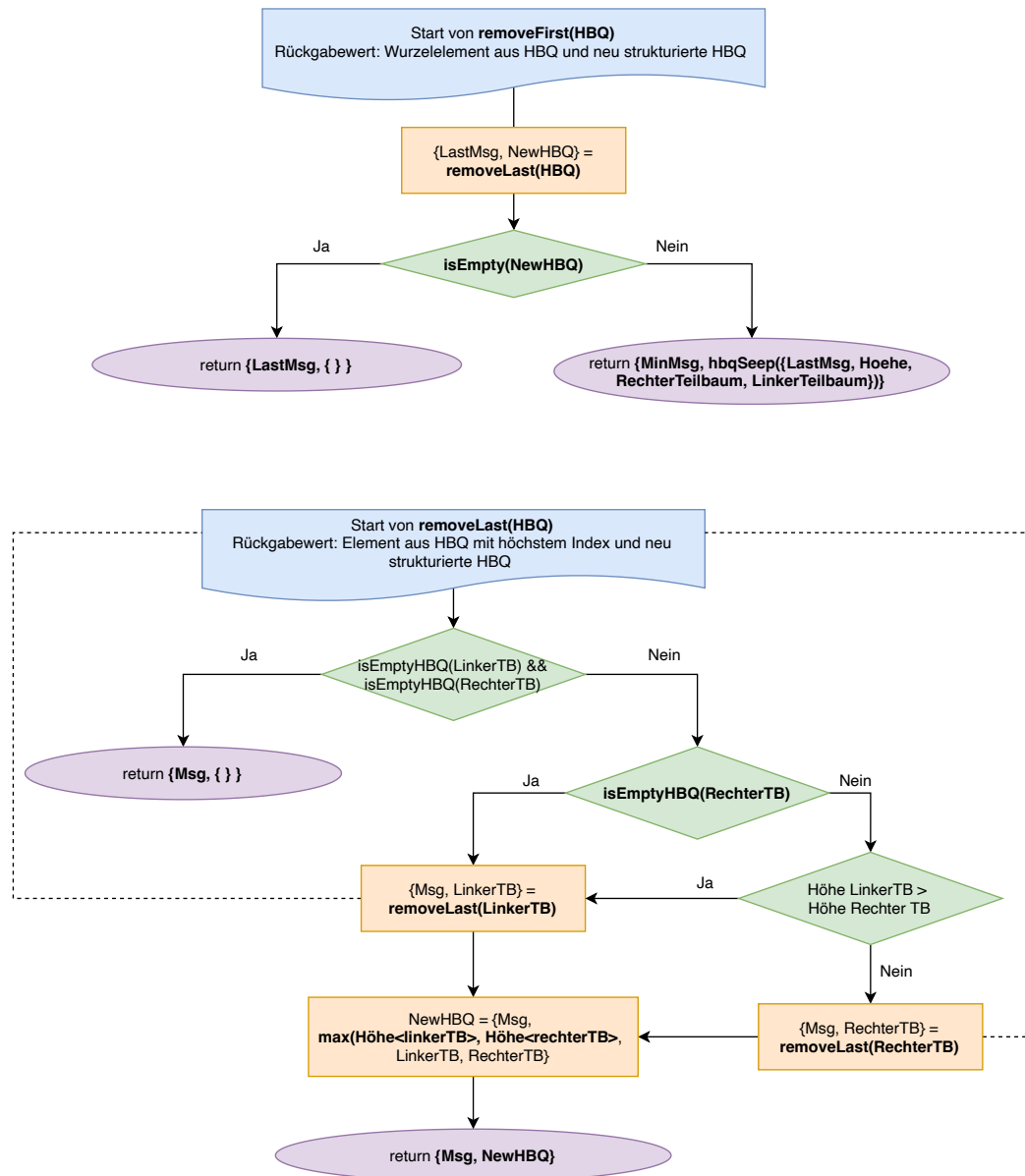


Abbildung 21: removeFirst/Last Heap of Holdback Queue

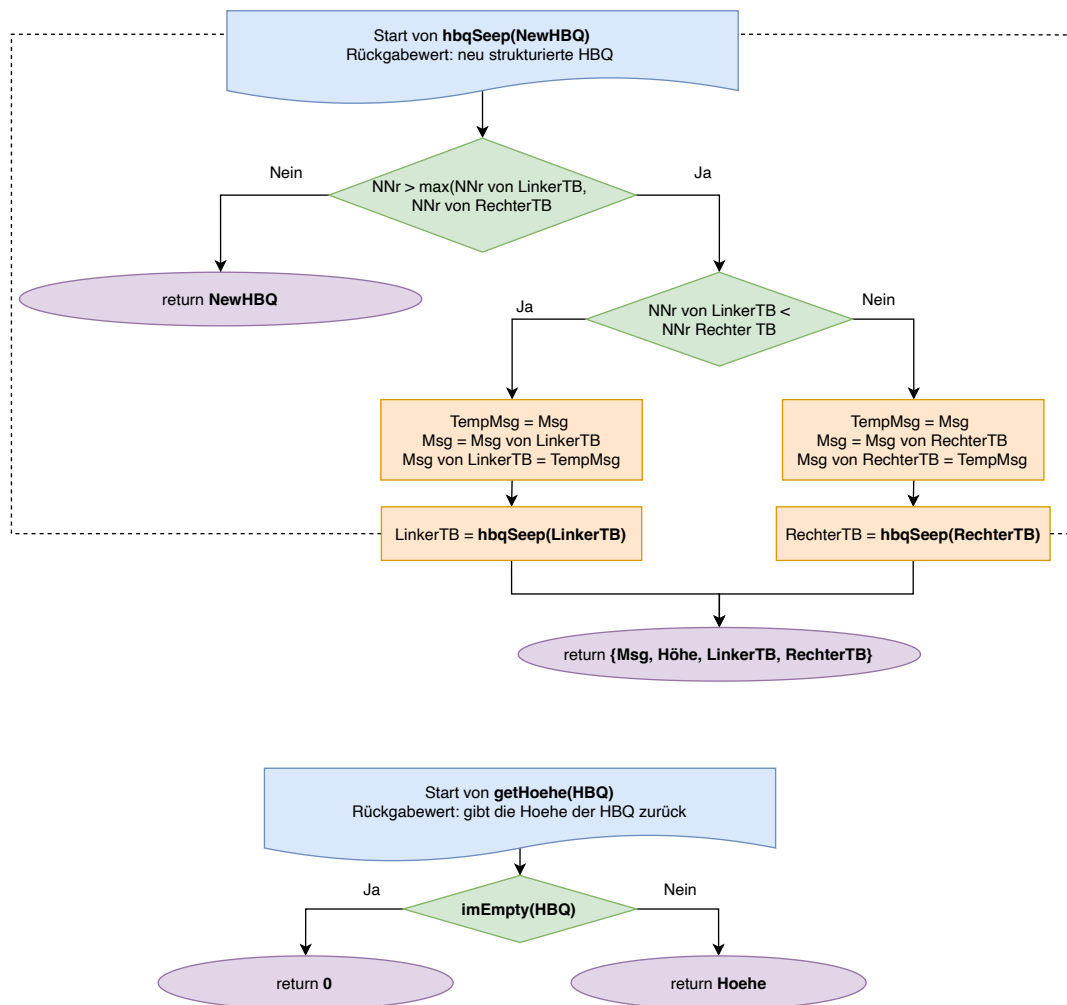


Abbildung 22: weitere Heap Funktionen