Freie Universität Berlin

Institut für Informatik

AG Theoretische Informatik



Bachelorarbeit

**Was ist der beste Trade-off für die Anzahl der Finger in Fingerbäumen und wie kann man in der Praxis Informationen über die Struktur ausnutzen?**

Anna Schapiro

Gutachter: Prof. Dr. Wolfgang Mulzer

Verfasser: Anna Hannah Schapiro

Matrikel-Nr.: 5039650

E-Mail: anna@schapiro.berlin

Telefon: 0152 5706 6134

Abgabetermin: 24.08.2021 (Campus Management)

**Abstract**

In dieser Arbeit geht es darum externe Pointer – Finger – einzuführen und den Suchprozess in Datenstruckturen durch sie zu beschläunigen. Die Laufzeiten der Such-Funktion werden durch extra Finger in einer Datenstruktur von O(log n) auf O(log d) reduziert [2]. Diese Art der Suche heißt Distanzsuche, da von einer Distanz zur Wurzel der Suchauftrag startet.

Ein gängiges Beispiel einer Distanzsuche ist der Lazy-Finger. Der Finger merkt sich nur seine zuletzt verwendete Position in der Datenstrucktur und startet neue Suchanfragen von dieser Position. Unbrauchbar wird der Lazy-Finger, wenn abwechselnd immer ein maximales und ein mininimales Node in der Datenstruktur gesucht wird. Denn bei so einer Nutzung der Datenstruktur kommt der Lazy-Finger nicht mehr hinterher und verhält sich sogar schlechter als die herkönnliche Wurzelsuche.

Deshalb ist das Ziel dieser Arbeit, eine Simulation der Fingersuche zuprogramieren, denn ich denke die Position und Anzahl der Finger ist eine implizite Vorhersage, wie eine Datenstruktur verwendet wird. Es könnte performanter sein, wenn die Fingersuche sich an den Such-Anfragen der Datenstrucktur orientiert, als wenn man diese struckturelle Information nicht berücksichtigt wird und mit der Wurzelsuche gesucht wird.

**Abstract - english**

Inhaltsverzeichnis

[Abbildungsverzeichnis ix](#_Toc75205126)

[Tabellenverzeichnis xi](#_Toc75205127)

[1 Einleitung 1](#_Toc75205128)

[1.1 Problemumfeld 1](#_Toc75205129)

[1.2 Zielsetzung 2](#_Toc75205130)

[2 Aktueller Forschungsstand: Übersicht über Finger 3](#_Toc75205131)

[2.1 Ein Finger 3](#_Toc75205132)

[2.2 Min und Max Finger 4](#_Toc75205133)

[2.3 Fingersuche mit Direktion 5](#_Toc75205134)

[2.4 Finger verwalten oder mit Fingern suchen 5](#_Toc75205135)

[2.5 Lazy Finger in Binär-Suchbäumen 6](#_Toc75205136)

[2.6 Splaytrees und Fingersuche 7](#_Toc75205137)

[2.7 Splaytrees, Zufall und Optimalität 7](#_Toc75205138)

[2.8 Fingersuche und weitere Datenstrukturen 8](#_Toc75205139)

[2.8.1 K-server Problem und multiple Finger 8](#_Toc75205140)

[2.8.2 SplayNets – Netzwerk Kommunikation 9](#_Toc75205141)

[2.9 Prediktion auf Input Streams für Lokalitätsprinzip 10](#_Toc75205142)

[2.10 Zusammenfassung des Forschungsstandes 10](#_Toc75205143)

[3 Modellierung des Problems 11](#_Toc75205144)

[3.1 Das Model 11](#_Toc75205145)

[3.1.1 Arten des Finger-Management 12](#_Toc75205146)

[3.1.2 Arten der Hauptdatenstrucktur 13](#_Toc75205147)

[3.2 Einwände zum Model 13](#_Toc75205148)

[4 Theoretische Betrachtung 15](#_Toc75205149)

[4.1 Anzahl der Finger 15](#_Toc75205150)

[4.2 Laufzeit betrachtung 17](#_Toc75205151)

[4.3 Erkenntnise für Finger-Arten 17](#_Toc75205152)

[4.4 Bedeutung für SplayTrees 18](#_Toc75205153)

[5 Programmierung der Simulation 20](#_Toc75205154)

[5.1 Vereinfachung des Models 20](#_Toc75205155)

[5.2 Das Vorgehen 20](#_Toc75205156)

[5.2.1 Wiedrichkeiten auf dem Weg 21](#_Toc75205157)

[6 Evaluation der Ergebnisse 23](#_Toc75205158)

[7 Einordnung der Ergebnisse anhand des Forschungsstandes 24](#_Toc75205159)

[8 Offene Fragen 25](#_Toc75205160)

[9 Zusammenfassung 26](#_Toc75205161)

[10 Fazit und Ausblick 27](#_Toc75205162)

[Begriffe und Anhang 28](#_Toc75205163)

[10.1 Begriffe 28](#_Toc75205164)

[10.1.1 Lokalitätsprinzip 28](#_Toc75205165)

[10.1.2 Daten/Values 28](#_Toc75205166)

[10.1.3 Datenstrucktur 29](#_Toc75205167)

[10.1.4 Baum 29](#_Toc75205168)

[10.1.5 Skipliste 30](#_Toc75205169)

[10.1.6 Finger 31](#_Toc75205170)

[10.1.7 Fingersearch und Fingersuche 31](#_Toc75205171)

[10.1.8 Splay tree 31](#_Toc75205172)

[Literaturverzeichnis 33](#_Toc75205173)

Abbildungsverzeichnis

[Abbildung 9.1: Beispiel 1 zum Einfügen einer Grafik 27](#_Toc75205174)

Tabellenverzeichnis

**Es konnten keine Einträge für ein Abbildungsverzeichnis gefunden werden.**

# Einleitung

Alle Datenstrukturen haben im Wesentlichen immer die gleichen Funktionen Search, Insert und Delete. In dieser Arbeit geht es darum externe Pointer, sogenannte Finger, einzuführen und den Suchprozess in Datenstruckturen durch sie zu beschläunigen.

Die Laufzeiten der Search-Funktion werden durch extra Finger in einer Datenstruktur von O(log n) auf O(log d) reduziert [2]. Viele theoretischen Ansätze versuchen dieses d möglichst klein zu halten und nennen diese Art der Suche Distanzsuche.

Ein gängiges Beispiel einer Distanzsuche ist der Lazy-Finger. Der Finger merkt sich nur seine zuletzt verwendete Position in der Datenstrucktur und startet neue Suchanfragen von dieser Position aus. Er ist besonders praktisch, falls die Ergebnisse der Search-Funktion dem lokalitäts Prinzip folgen, also struckturell nahe beisammen in der Datenstrucktur anzutreffen sind.

## Problemumfeld

Unbrauchbar wird der Lazy-Finger, wenn abwechselnd immer ein maximales und ein mininimales Node in der Datenstruktur gesucht wird. Denn bei so einer Nutzung der Datenstruktur kommt der Lazy-Finger nicht mehr hinterher und verhält sich sogar schlechter als die Wurzelsuche.

Man könnte mit der Min-Max Fingersuche diesem Problem begegen. Die Min-Max-Fingersuche besteht aus einem festen Minimalen Finger und einem festen Maximalen Finger in der Datenstrucktur. Die Fingersuche würde selbst entscheiden, ob die Startposition der Suche vom Maximalen oder vom Minimalsten Node in der Datenstrucktur beginnen sollte.

Diese Fingersuche würde zwar bei den Suchanfragen welche immer das Maximum und das Minimum suchen oder zumindest in deren Nähe suchen deutsch besser performieren als der Lazy-Finger. Jedoch kann man sich auch bei dieser Fingersuche einen ungünstigen Fall konstuieren, wo die Wurzelsuche besser performiert. Z.B. Könnte man annenehmen, dass die Datenstrucktur ein Binärer Suchbaum ist. Die am meisten gesuchten Nodes würden immer vom linken Teilbaum ganz rechts sein und anschließend der ganz linke Node vom rechten Teilbaum würde gesucht werden.

Auch hier würde sich die Wurzelsuche im Vergleich zu der Distanzsuche schneller verhalten, denn die Distanzsuche müsste immer bis zur Wurzel hochgehen, nur um dann als herkömmliche Suche weiterzusuchen.

## Zielsetzung

Man könnte aus den oberen Beispielen entnehmen, dass die Suche am besten alle Finger in alle hochfriquentierten Bereiche der Datenstrucktur stellen sollte. Am Sinnvollsten wären das Bereiche, von denen man aus der Vergangenheit weiß, dass in diesen Bereichen besonders oft gesucht wird.

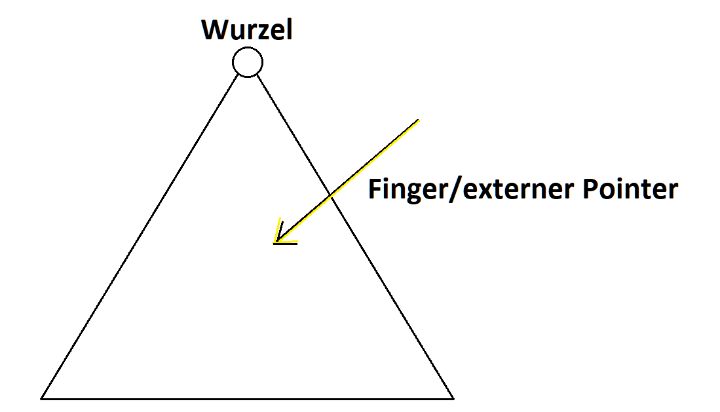
Im Prinzip wäre somit die Aufgabe der Fingersuche eine extra Datenstrucktur zusein, welche das Wissen über die besonders oft nachgefragten Startpositionen der Distanzsuche in sich vereint.

Ich möchte gerne aus den Laufzeiten ableiten, ob verscheidenen Fingersuchen sich je nach Verteilung der Suchanfragen besonders nützlich oder träge verhalten. Zu diesem Zweck erscheint mir sowohl eine theoretische Betrachtung als auch eine Simulation sinnvoll.

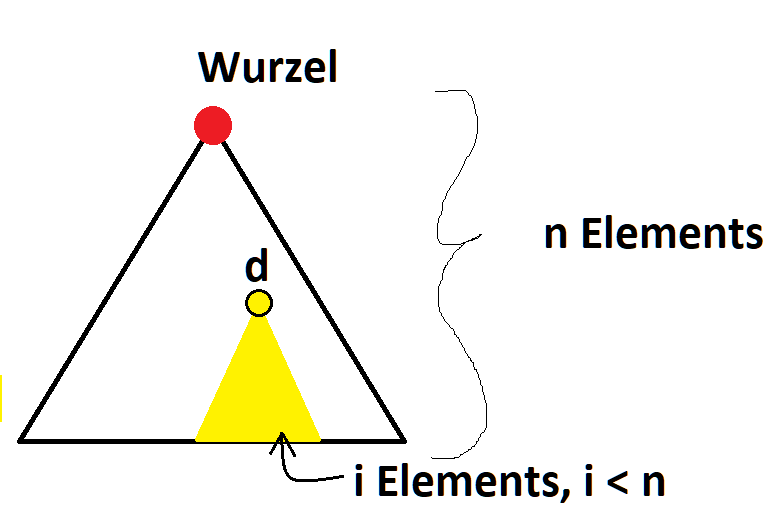
# Aktueller Forschungsstand: Übersicht über Finger

Dieses Kapitel soll ein Überblick über Konzepte für Finger, Fingersuchen und Datenstruckturen, wo sie angewendet werden, geben. In dieser Arbeit gehe ich der übersichtshalber davon aus, dass eine Datenstrucktur beliebig, aber feste Anzahl n von Elementen in sich gespeichernt hat.

## Ein Finger

Die Fingersuche ist beonders anschaulich in Binären-Suchbäumen. In diesem Paper (<https://www.cs.au.dk/~gerth/papers/finger05.pdf>) wird gleich im ersten Kapitel darauf hingewieden, dass Finger externe Pointer sind, welche genutzt werden, um von der Startposition d, statt von der Wurzel aus, zu suchen.

Hier wird vorrausgesetzt, dass der Finger schon so positioniert ist, dass die Finger-Position nächer zum Ergebniss liegt, als die Wurzelsuche und somit das Lokalitätsprinzip für das Suchergebniss und den Finger gilt.

Das führt zur Überlegung, dass die Wurzelsuche-Startposition (rot) auch genannt Root-Finger (S. 181 <https://link.springer.com/content/pdf/10.1007%2F978-3-319-13075-0.pdf>)

in einem Binärbaum 2^n-1 Elemente durchsuchen kann um das Ergebniss zuliefern. Während der Ausschnitt (gelb), welcher durch den Finger-Startposition begrenzt wird, kann nach schon 2^i-1 Elementen das Ergebniss finden. Da i kleiner ist als n, kann man von einem Vorteil sprechen. Man könnte sogar sagen, je näher der Finger am Such-Ergebniss ist, desto exponenziell stärker ist dieser Effekt - veranschaulicht bedeutet das, dass die Fläche von dem gelben Bereich kleiner werden würde.

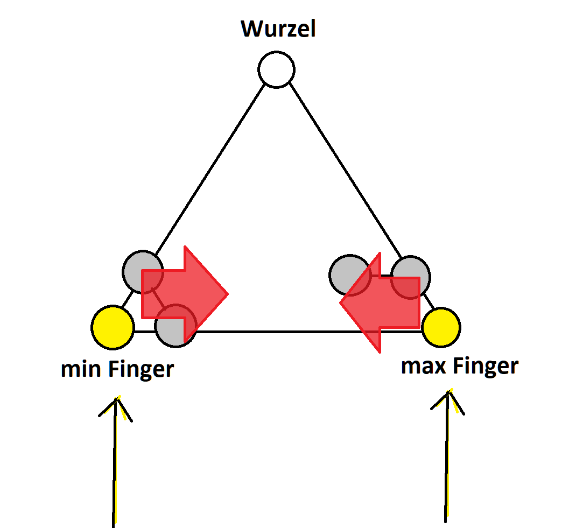
Mathematisch-Formal würde man sagen, die Wurzelsuche ist in der Laufzeitklasse O(log n), während die Distanz Suche auf i-Elementen arbeitet:

n > i = > O(log n) > O(log i).

Das ist der Vorteil von einem einzigen Finger.

## Min und Max Finger

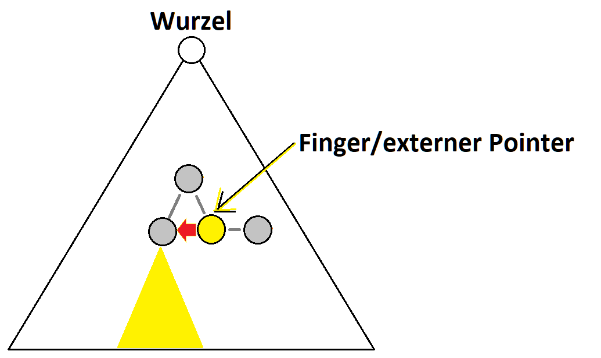
In diesem Buch (S 483, kap. 5.5 https://link.springer.com/content/pdf/10.1007/BF01940876.pdf ) wird die Idee unterbreitet der Fast Finger Search. Das Konzept beinhaltet, dass einfach von allen externen Fingern die Suche gleichzeitig gestartet werden soll. Wenn einer der Finger ein Ergebniss findet, soll die Suche stoppen.

Als Spezialfall wird der Min-Finger und der Max-Finger genannt. Der Min-Finger steht immer auf dem minimalen Node im Tree und der Max-Finger steht immer auf dem größten Node im Tree. Die Fingersuche beginnt zeitgleich. Diese Laufzeit ergibt deshlab:

O(log min{d, n - d})

In diesem Buch wird nähergelegt, dass versteckte Suchen wie in der Join, oder teile der Split-Operation auch von der Min-Max Fingersuche profitieren können, da sie schon auf Teilbäume zugreifen und man deshlab keine temporären Wurzeln von Teilbäumen zusetzen braucht. (S 484 kap 5.7 und 5.9).

## Fingersuche mit Direktion

In diesem Paper aus dem Jahre 1977 (<http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=5A00C23223BF564B3107486113A2D1BE?doi=10.1.1.527.7294&rep=rep1&type=pdf> ) wird explizit eine Fingersuche vorgestellt, welche eine Suchrichtung besitzt. Wenn der Finger auf einer Position steht, kann es passieren, dass er seine Nachbar-Nodes durchsuchen muss, bevor eine runtergerichtete Suche - wie bei der Wurzelsuche - im Baum passiert.

Es wird auch der Fall betrachtet, dass Finger verschoben, also ein Update verlangen; am Beispiel von einer Insert Operation wird festgestellt, dass Finger unter umständen zu nahe beieinander liegen.

Finger werden in diesem Paper vorallem dazu genutzt um auf einzelne Nodes in Binären Suchbäumen, welche Listen enthalten, zuzugreifen.

In dieser Binärbaum-Finger Implementierung kann man also auf einer Ebene die Nodes durchsuchen, falls man einen Finger nicht sofort zur Verfügung hat.

## Finger verwalten oder mit Fingern suchen

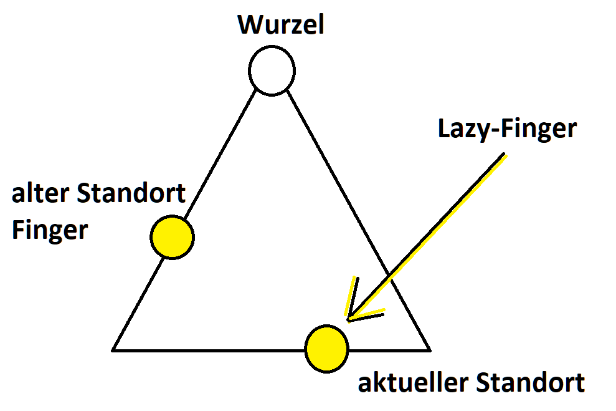
In diesem Buch seite 158, 3. Abs (http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.217.5131&rep=rep1&type=pdf ) werden auf Level-Linked-Binärbäumen Finger immer nur auf Leafs gesetzt. Es wird der Schluss gezogen, dass in der ammortisierten Laufzeit die Logarithmische Suche deutlich stärker zutragen kommt als die constante Laufzeit für die Erzeugung und Änderung der Finger.





Damit könnte man sagen, dass der wichtige zubetrachtende Aspekt in der Laufzeit der Fingernutzung, die Fingersuche an sich ist, und nicht die Verwaltung der Finger. Deshalb fokussiert sich diese Bachleorarbeit nicht auf die Verwaltung der Finger.

## Lazy Finger in Binär-Suchbäumen

In einem etwas etwas aktuellerem Conference Paper aus dem Jahr 2014, S 181 (https://link.springer.com/content/pdf/10.1007%2F978-3-319-13075-0.pdf) werden die Laufzeiten von Lazy Fingern vorgestellt. Ein Lazy Finger ist ein Finger, der sich immer auf das zuletzt gesuchte Element im Baum stellt. Genau von diesem zuletzt gesuchten Element beginnt die neue Suchanfrage. Es wird sich mit der Fragestellung der Abstände von Suchergebniss und Lazyfinger beschäftigt und das im Zusammenhang von Entropien (Informationstheorie).

“Given the pairwise frequencies fX finding the tree that minimizes the execution time of search sequence X using lazy finger takes time O(n3).” Das wäre also die Laufzeit um eine Optimale Lazy-Finger Position in einem Binär-Baum zuberechnen. (3. Abs, S 190)

## Splaytrees und Fingersuche

In diesem Paper aus dem Jahr 2018 (Absatz 3 https://drops.dagstuhl.de/opus/volltexte/2018/10003/pdf/LIPIcs-ISAAC-2018-55.pdf) steht in der Einleitung von mir übersetzt: “Im Jahr 2000 haben Cole [16, 15] gezeigt, dass Splaytrees (asymptotisch) mit der Effizienz der Fingersuche übereinstimmt, die in diesem Zusammenhang als dynamische Fingereigenschaft bezeichnet wird. Dies ist bemerkenswert, da Splay keine expliziten Finger verwendet“

## Splaytrees, Zufall und Optimalität

In dieser Arbeit geht es auch um die Muster in den Suchanfragen an eine Datenstrucktur. Für (03.06.2021 13:50 Uhr 1. Abs, https://en.wikipedia.org/wiki/Splay\_tree ) viele Sequenzen von nicht-zufälligen Operationen arbeiten Splaytrees besser als andere Searchtrees, sogar besser als O(log n) für ausreichend nicht-zufällige Muster. Und das ohne das vorherige kennen des Suchmusters.

Wenn man annimmt, dass die Suchanfragen an eine Datenstrucktur nicht chaotisch sind, spricht also viel dafür Splaytrees statt Binäre-Suchbäume einzusetzen.

Die Eigenschaft von Splaytrees, dass sie sich wie in Kapitel „ 2.7 Splaytrees und Fingersuche“ beschrieben, wie eine Statische Datenstrucktur (Binärbaum) verhalten kann, bezeichnet man als „statische Optimalität“.

Eine ähnliche Vermutung gibt es für dynamische Datenstruckturen – also Datenstruckturen, welche sie zur Laufzeit des Programms ändern. „Diese Vermutung ist als „dynamische Optimalität“ bekannt und gilt als eines der bekanntesten offenen Probleme auf dem Gebiet der Datenstrukturen.“ ([https://de.wikipedia.org/wiki/Splay-Baum](https://de.wikipedia.org/wiki/Splay-Baum%2003.06.2021) , 14:00 Uhr 03.06.2021)

**KAPITEL 2.6 und 2.7 sind ein Wiederspruch**

## Fingersuche und weitere Datenstrukturen

Das Konzept der Finger, als externe Pointer, hat keine Einschränkung für spezielle Datenstruckturen. Daher möchte ich in diesem Kapitel aufzählen, welche weitere Datenstruckturen mir bei der Recherche aufgefallen im Zusammenhang mit der Fingersuche:

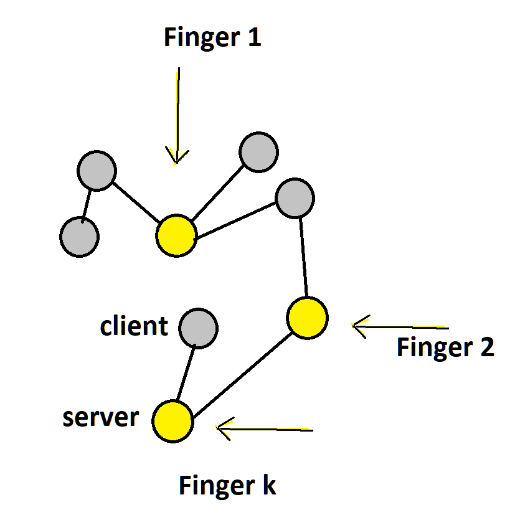
* Lists (2 Finger auf den Extremwerten, um schneller Listoperationen auszuführen, u.a. auch merge-Operation)

<http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=5A00C23223BF564B3107486113A2D1BE?doi=10.1.1.527.7294&rep=rep1&type=pdf>

* Treaps (<https://www.cs.au.dk/~gerth/papers/finger05.pdf> )
* Skip-List (nur mit 1 Finger, <https://www.cs.au.dk/~gerth/papers/finger05.pdf> )
* (2,4) Level Linked Tree (<https://www.cs.au.dk/~gerth/papers/finger05.pdf> )

Die (2,4) Level Linked Trees sind besonders hervorzuheben da sie maximal ausgebaut im Hinblick auf die Erreichbarkeit aller Nodes ist. Damit wird eine besonders flexible und auch effiziente Fingersuche ermöglicht.

### K-server Problem und multiple Finger

In diesem Paper aus dem Jahr 2018 (<https://drops.dagstuhl.de/opus/volltexte/2018/10003/pdf/LIPIcs-ISAAC-2018-55.pdf>) wird versucht dem k-server Problem mit k Fingern zubegegnen. Es wird davon ausgegangen, dass alle Server und alle Clients eine „Karte aller Nodes“, also die komplette Information zum Entscheiden haben.

Die Grundsätzliche Idee ist, dass jedem Server ein Finger zugewiesen wird und Anhand von diesen Fingern dann die nächstmöglichen Clients gefunden oder bedient werden. Der Algorithmus gibt also die Information zurück welcher Finger für welchen Client benutzt wurde. Das ist eine alternative Anwendung der Fingersuche.

### SplayNets – Netzwerk Kommunikation

In dieser Doktorarneit (https://arxiv.org/pdf/1705.09555.pdf) wird das Konzept der SplayNets präsentiert. SplayNets arbeiten laut der Arbeit auf P2P Netzwerken und versuchen die optimalen Kommunikationswege mit Hilfe von Splaytrees als Fingersuchen zu sogenannten SplayNets zuvereinfachen. Jeder Node erhält also einen Finger und Pointer zwischen den Fingern werden gewichtet und mit Splaytrees optimiert aufgespannt.

Es werden Anwendungsbereiche eines Overlay-Konzepts für Tor und weitere P2P Networks, sowie für intranet Kommunikation in (Facebooks) Datencentren vorgestellt, da 60% des Traffics interne Kommunikation ist.

## Prediktion auf Input Streams für Lokalitätsprinzip

In diesem Paper 2003 (<https://dl.acm.org/doi/pdf/10.1145/781131.781159> ) wird der Sachverhalt der Vorhersage von Inputdaten auf verschiedene Systeme analysiert. Es wird explizit auf den Nutzen des Vorhersagens im Sinne des Lokalitätsprinzips erwähnt, und es wird ein Splaytree in diesem Zusammenhag benutzt, da er am schnellsten ist. Es gibt auch abgewandelte Variationen mit gewichteten Kanten oder geclusterten Nodes in Splaytree.

In diesem Paper wird nicht explizit von Fingern gesprochen, jedoch werden die Ergebnisse der Vorhersage für den Splaytree wie Finger verwendet.

## Zusammenfassung des Forschungsstandes

Die Finger und die Fingersuche mit Splaytrees sind in verschiedenen Fachgebieten präsent.

So sind sie in der technischen Mathematik, im Bereich von Netzwerkkommunikation und auch im Bereich der Datenstruckturen, als auch Informationstheorie anzutreffen.

Der Begriff des Fingers ist nicht in allen Gebieten bekannt und wird z.B. in der Netzwerkkommunikation nicht als besonders erwähnenswert gehalten - und trotzdem wird dieses Konzept verwendet.

# Modellierung des Problems

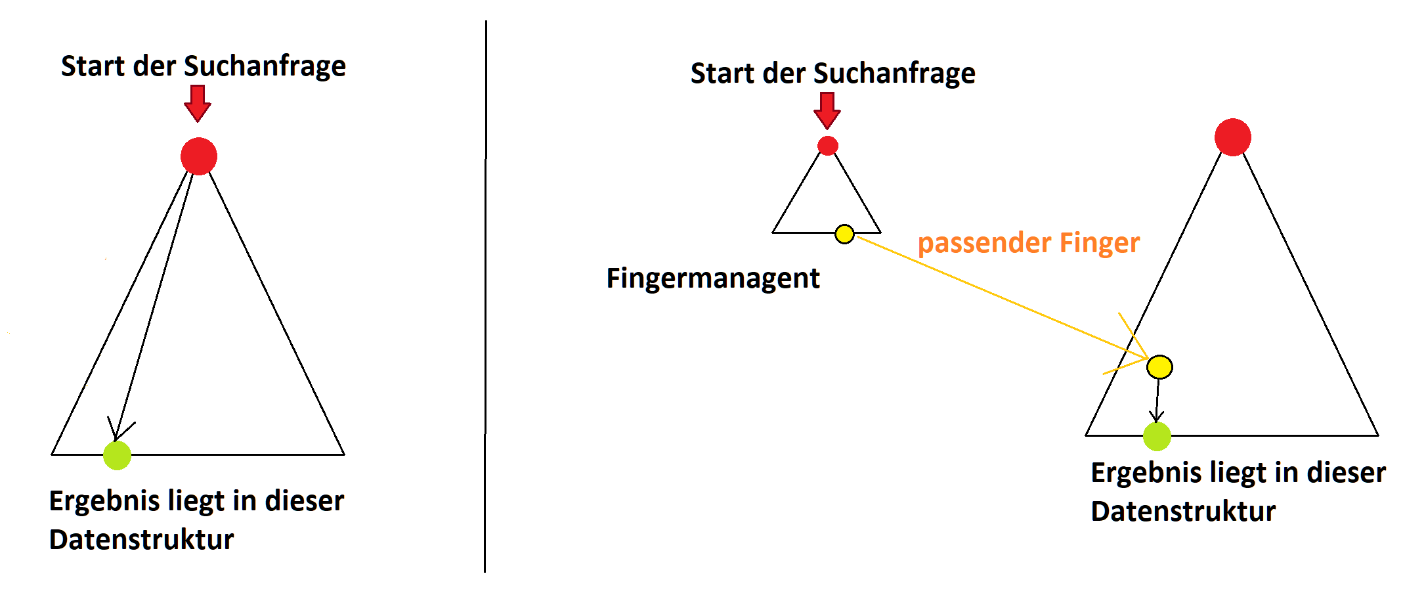
## Das Model

Das Problem, welches ich angehen möchte, ist, dass je nachdem welches Konzept (vorheriges Kapitel) für die Fingersuche implementiert wird, es zu nachteiligem Verhalten führen kann. Meines erachtens ist allein die Nutzung eines konkreten Finger-Konzepts eine Vorhersage wie eine Datenstrucktur verwendet wird.

* Z.B. Beim Lazyfinger, wird indirekt davon ausgegangen, dass nur 1 Bereich im Binären-Suchbaum gefragt ist. Jeder Wert ausserhalb dieses Clusters ist ein ungünstiger Fall. Was passiert, wenn die gesammten Suchaufträge ungünstige Fälle sind? Bei einer Zick-Zack Verteilung der Keys wäre das tatsächlich eine ungünstige Nutzung, weil 2 Cluster vermischt abgefragt werden.
* Bei dem Min-Max Finger sind schonmal 2 Cluster abgedeckt, jedoch bei einem dritten Cluster in der mitte der Datenstrucktur, würden diese Finger auch nicht besser als die Wurzelsuche performieren.

Es gibt theoretische Arbeiten, welche sich mit optimalen Fingerpositionen (vorheriges Kapitel) befassen, diese Lösungsvorschläge sind nicht trivial nachzuvollziehen.

Mein Ansatz ist daher die Simulation, man kann durch leichte programmierung eine Simulation und damit eine Verbesserung durch Finger zeigen.

Um sich dieser Frage zu nähern, würde ich folgenden Versuchsaufbau starten, statt üblicher Suchen ohne die Finger (Abb links) würde ich ein Finger Management vorschalten (Abb rechts). Dieses Finger-Management ist allgemeiner gefgasst und speichert in sich selbst die Vorhersage, wie die große Datenstrucktur verwendet wird. 

1. Man hat eine Separate Datenstrucktur, wo alle Finger/externe Pointer verwaltet werden, das Fingermanagement
2. In dieser Fingermanagement Datenstrucktur werden bei Suchanfragen die passenden Finger herrausgesucht
3. Diese Finger werden übergeben und als Distanzsuche in der eigentlichen Datenstrucktur verwendet
4. Das Suchergebnis wird aus der großen Datenstrucktur zurückgegeben

Je nach Versuch würde die Fingermanagent Datenstruktur entweder ein Lazy-Finger, Min-Max-Finger oder ein Splaytree sein.

Das Kernstück dieser Arbeit ist also das Fingermanagent.

Im Unterschied zur weitverbreiteten Wurzel-Suche (Abb. Links), wird in der Fingersuche (Abb rechts) die Startposition der Suchanfragen anders sein.

### Arten des Finger-Management

Das Fingermanagement ist nur eine Datenstruktur in der Finger verwaltet werden. Da diese Datenstruktur als integraler Bestandteil der Fingersuche angesehen werden kann, ist es vorteilhaft sich Gedanken zumachen, welche Datenstruktur an dieser Stelle sinnvoll ist.

* Am Beispiel des Lazy-Fingers wäre das Fingermanagemtsystem also einfach ein Pointer mit Speicherplatz für einen Node.
* Im Falle der Min-Max Fingersuche wäre z.B. das Fingermanagementsystem zwei Pointer, welche in einer Liste verwaltet werden könnten.

Man könnte auch weitere Datenstruckturen inbetracht ziehen, wie binäre Suchbäume, dann stellt man sich logischerweise die Frage welche Pointer genau so einen Baum aufgenommen werden. Man könnte sich an dieser Stelle ein Rating-Mechanismus für die beliebtesten Finger überlegen und diese dann berücksichtigen. Man könnte auch einfach den Zufall entscheiden lassen welche Finger in das Fingermanagement-System aufgenommen werden sollen.

* Oder man entscheidet sich für einen Splaytree. Der Splaytree hat einen bubble-up Effekt für die am häufigesten verwendeten Finger – und einen sink-to-bottom Effekt für unbenutzte Finger. Mit einem Splaytree müsste man sich keine größeren Gedanken machen, wie man Finger einem wartungs-aufwändigen Rating unterzieht, sortiert und verwaltet. Der Splaytree optimiert sich dahingehend selbst.

### Arten der Hauptdatenstrucktur

Mein Model beschränkt sich nicht nur – wie durch die Skizze angedeutet – auf Bäume. Die Hauptdatenstrucktur kann auch eine Liste, Skipliste oder eine historisch seltsam gewachsene Datenbank sein.

Fingersuchen können auch auf Graphen angewendet werden, z.B. eine Rolle in verteillten P2P-Netzen spielen, hier gibt es eine Doktorarbeit aus dem Jahr 2017 zu SplayNets (<https://arxiv.org/pdf/1705.09555.pdf> ).

## Einwände bezüglich des Models

Es wurde der Kritikpunkt von Kommilitonen eingracht, dass man auch Balancierungs- und weitere Baum-Typische Operationen betracheten muss. In dieser Arbeit gehe ich der Einfachheit halber davon aus, dass in solchen Operationen eine versteckte Suche enthalten ist, welche einen eigenen Finger oder eigenes Fingerkonzept haben. Somit sind auch diese Operationen in diesem Versuch mit inbegriffen und die Erkenntnisse könnten eventuell auch auf diese Operationen übertragen werden.

In einem Paper (vorheriges Kapitel) wird davon gesprochen, dass Splaytrees und Finger-Suchen sich asymptotisch gleich verhalten. Man könnte daher davon ausgehen, dass man einfach die Hauptdatenstrucktur als Splaytree zu implementieren braucht und sich den Overhead mit dem Fingermanagementsystem sparen kann.

Dieser Überlegung würde ich gerne einige Argumente entgegenstellen:

* Splaytrees arbeiten oft im Cache, das bedeutet man versucht diese Datenstrucktur möglichst klein zu halten. Damit wäre dieser Lösungsansatz statt der Fingersuche Splaytrees in der Hauptdatenstrucktur zuverwenden, nur für kleine Datenstruckturen geeignet.
* Wenn man annimmt, dass die Hauptdatenstrucktur auf verteilten Systemen arbeitet, dann macht es keinen Sinn diese Datenstrucktur nur wegen Suchanfragen umzubauen, das wäre unter Umständen sehr aufwändig. Und wenn man bedenkt, dass viele Datenbanken historisch gewachsen sind, ist es vielleicht auch einfacher externe Finger anzubringen als die komplette Datenbank jedesmal umzustruckturieren.
* Ein weiteres Argument wäre theoretischer Natur, man kann genelell Fingersuchen auf unendlich wachsenden Datenstruckturen anwenden, solange sie auf einem Bereich suchen, welcher sich im Moment der Suche nicht ändert. Bei Splaytrees kann es außerdem zu der nachteiligen linearen Formation kommen, das ist bei besonders großen Splaytrees ungünstig.

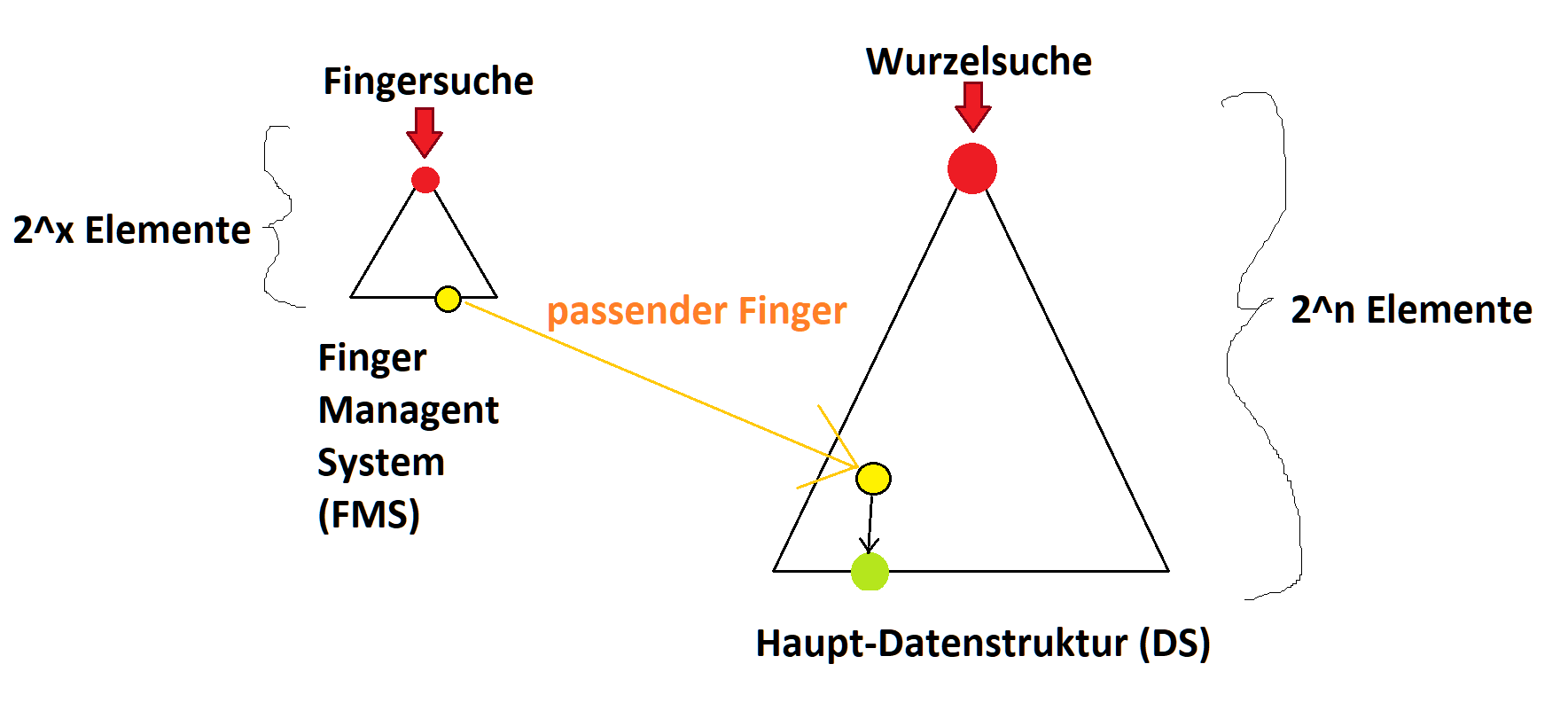
# **Theoretische Betrachtung**

In diesem Kapitel möchte ich mich mit den durchsuchten Nodes in den Datenstrukturen und der Anzahl der Finger bei der Fingersuche beschäftigen, sowie der Laufzeitbetrachtung.

## Anzahl der Finger

Der Übersichthalber würde ich das zuvor vorgestellte Modell um die Anzahl der möglichen durchzusuchenden Elemente ergänzen:

* 2^n ist die Anzahl der Elemente in der Hauptdatenstruktur, wo alle Elemente gespeichert sind
* 2^x ist die Anzahl aller Finger



1. Jetzt kann man das Ziel formulieren.

Die Laufzeit der Wurzelsuche in der Datenstrucktur soll größer sein, als die zukonkurrierende Suche - welche aus zwei Teilsuchen der Fingermanagement-Suche und der Distanzsuche besteht. In dem Moment, wo die zusammengesetzte Suche die Laufzeit der Wurzelsuche überschreiten sollte, verliert das Fingermanagement seine Berechtigung, daher wird dieses Vergleichszeichen als eine Schranke von mir gesehen:

O(Wurzelsuche) > O(Fingersuche)

> O(Fingersuche in FMS) + O(Distanzsuche)

1. Man kann die obere Formel für beliebige Datenstruckturen verwenden. Allerdings würde ich gerne die Annahme treffen, dass das Fingermanagementsystem (FMS) und die Hauptdatenstrucktur beides binäre-Suchbäume sind. Damit kann log\_2 einheitlich zur Berechnung der Anzahl der durchlaufenen Nodes verwendet werden.

Der gesuchte Wert ist x – das ist die Anzahl der Finger:

O(Wurzelsuche) > O(Fingersuche in FMS) + O(Distanzsuche)

log(n) > log(x) + log(d)

log(n) > log(x\*d)

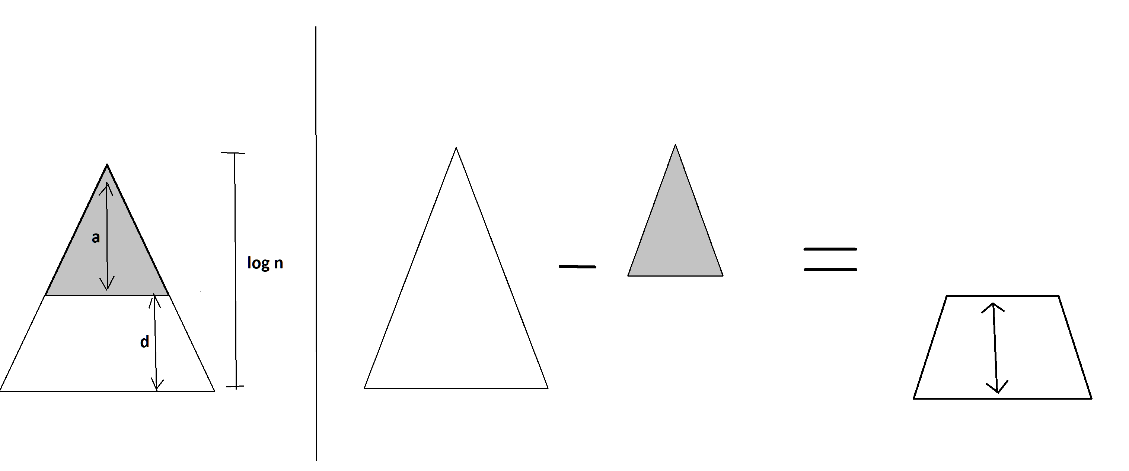
n > x\*d

n/d > x

1. Nun könnte man eine Abschätzung tätigen.

a sei der Unterschied zwischern der Wurzel in der Hauptsdatenstrucktur und der Distansz-Suche-Startposition d.

Graphisch gesprochen benutzt meine Abschätzung folgendes:



log n – log a = log d (Abschätzung)

log (n/a) = log d

n/a = d

n = d\*a

1. Nun kann man 1) und 3) kombinieren:

x < n/d (1)

x < (d\*a)/d (3)

x < a

Ausformuliert bedeutet diese Aussage, dass die Höhe bis zum Startpunt der Distanzsuche vorgibt, was die maximale Anzahl der Finger ist.

Aus dieser Formel kann man also ableiten, dass das Finger-Management-System (FMS) nach der Höhe a „absgeschnitten“ oder gekürzt werden sollte, um einen absoltuen Vorteil zu bieten.

## Laufzeit betrachtung

Im vorherigen Kapitel habe ich ermittelt das die Anzahl der Finger vom Finger-Management-System maximal a-viele sind und dementsprechend dauert es maximal log\_2(a) Nodes um den richtigen Finger zufinden. Und dann nochmal d-viele Nodes um das richtige Endergebnis zufinden.

Um auf die Ausgangsformel zurückzukehren:

O\_worst(Fingersuche) = O\_worst(Fingersuche FMS) + O\_worst(Distanzsuche)

= log\_2(a) + log\_2(d)

= log\_2(a\*d)

## Erkenntnise für Finger-Arten

Man kann an dieser Laufzeit-Formel gut erkennen, dass wenige (a) und tief gelegte Finger (d) in der Hauptdatenstruktur tendenziell besonders gut performieren:

O\_worst(Fingersuche) = log\_2(a\*d)

Das würden also Min- und Max-Finger als auch der Lazy-Finger vorteilhaft erfüllen.

Außerdem kann man ablesen, wie viele Cluster in der Hauptdatenstrucktur mit der Fingersuche maximal abdecken kann - nämlich a Stück. Und je tiefer die Distanzsuche anfängt, desto mehr Cluster können durch Finger angesteuert werden.

## Bedeutung für SplayTrees

Wie in der Recherche (1.1 Splaytrees und Fingersuche) herausgefunden, verhält sich der Splaytree bei genug chaotischen Anfragen laufzeittechnisch, wie ein statischer binärer Suchbaum - unter weniger chaotischen Anfragesequenzen - sogar deutlich besser als dieser.

Da in der Laufzeitbetrachtung ein binärer Suchbaum als Annahme für das Fingermanagement getroffen wurde, kann man Schlussfolgern, dass diese Laufzeitvermutung auch für Splaytrees als untere Schranke, in einer asymptotischen Laufzeit, gilt. Mathematisch formuliert:

1. Die Laufzeitformel für Suche mit Fingern:

O\_worst(Fingersuche) = O\_worst(Fingersuche FMS) + O\_worst(Distanzsuche)

1. Ausführung mit Splaytrees:

O\_worst(Fingersuche) = O\_worst(Splaytree) + O\_worst(Distanzsuche)

1. Worst-Case für Splaytree - Suchanfragen sind chaotisch genug:

O\_worst(Splaytree) = O\_worst(Binärsuchbaum)

= O(log\_2(a))

* O\_worst(Fingersuche) = O\_worst(Splaytree) + O\_worst(Distanzsuche)
* O\_worst(Fingersuche) = O\_worst(log\_2(a)) + O\_worst(log\_2(d))
* O\_worst(Fingersuche) = log\_2(a\*d)

Man könnte Schlussfolgern, dass es nicht schadet einen Splaytree für die Fingersuche zu verwenden.

# Programmierung der Simulation

In diesem Kapitel möchte ich auf die Simulation eingehen. Meine Simulation wurde in Python programiert. Ich habe auf möglichst viele Libarys zu verzichten versucht, da ich sowieso in die Datenstruckturen selbst eingreifen möchte und die maximale Kontrolle über das Vorgehen in der Simulation haben wollte.

## Vereinfachung des Models

Im vorherigen Kapitel habe ich ein allgemeines Model für die Fingersuche vorgestellt, welches die Verwaltung der Finger als eigenständige Datenstrucktur versteht. Alle Aspekte und Besonderheiten kann ich in dieser Bachelorarbeit deshalb nicht simulieren, dafür ist das Thema zu groß. Ich werde mich daher bei der Hauptdatenstrucktur und Fingersuche auf folgende vereinfachte Sachverhalte festlegen:

* Meine Hauptdatenstrucktur wir ein balancierter Baum sein
* Mein Fingermanagent wird mit einem Lazy-Finger, Min-Max-Finger und mit einem Splaytree implementiert

Ich möchte nochmals erwähnen, dass das Model sich nicht auf Bäume, Listen, Heaps, Graphen usw. beschränkt.

## Das Vorgehen

Folgendes Vorgehen für den Versuchsaufbau habe ich gehabt.

Erster Schritt – Hauptdatenstruktur initieren:

* Ein Binärbaum mit dem Funktionieren: insert, delete und down-search sowie two-directional-search erstellen
* Die die Datenstrucktur wird mit Daten befüllt, bis sie voll ist (Hauptdatenstruktur)
* Finger-Managementsystem definieren (Lazy-Finger, Min-Max-Finger, Splaytree)

Zweiter Schritt – Searchverteilung erstellen:

* Listen mit Zahlenreihen erstellen
* Fingersuche auf der Datenstrucktur mit den Listen ausführen

Dritter Schritt – Wissensgewinn

* Analyse welchen Speed-up die Finger gebracht haben (Zeit, Anzahl der Nodes, usw)
* Mögliche Verbesserungen, Fingerpositionen
* Graphisch veranschaulichen

# Wiedrichkeiten auf dem Weg

Zu Anfangs dieser Arbeit habe ich aus github mir einige Implementierungen von verschiedenen Datenstrukturen angesehen. Als ich festgestellt habe, dass irgendwo Nodes welche Pointer besitzen, doppelt geführt wurden, habe ich alles neu geschrieben, weil ich sichergehen wollte, dass es keine Fehler an dieser Stelle gibt. Im Nachhinein hat sich ergeben, dass die Ursache dieses „Fehlers“ eine kleine print-methode zuviel war an der falschen Stelle.

Ich habe auf Windows 10 mit Python Version 3.6 programiert, da ich auch Zeiten in Mili-Secundenbereich messen wollten, um zu sehen, ob z.B. Splaytrees auch im realen Leben einen Speed-Up erreichen, bin ich auf folgende Unwegsamkeit gestoßen.

Das Betriebssystem kann Prozesse paralellisieren und je nach Priorität ausführen. Die Frage, die sich ergibt, ist folgende - welche Zeit wird genau gemessen? Die prozessinterne Zeit oder die Systemzeit, welche der Prozess insgesammt benötigt. Beides sind relevante Informationen.

Aktuell habe ich es im Lock-Down nicht hinbekommmen die prozessinterne Uhrzeit auf Windows mit Python zumessen, lediglich die Systemzeit, welche meine Prozesse insgesammt gebraucht haben, um mit der Fingersuche zu suchen. Es scheint unerwarteterweise recht kompliziert zu sein, die richtigen Funktionen für sowas in Python windowskonform zu finden.

Aber selbst diese gemessene prozessübergreifende Ausführungsdauer ist nicht schlecht, jedoch durch die betriebssystem-interne Paralellisierung und vermutlich auch eine versteckte Optimierung unterliegen diese Messwerte starken Schwankunden.

# Evaluation der Ergebnisse

# Einordnung der Ergebnisse anhand des Forschungsstandes

# Offene Fragen

Ein nützlicher Aspekt für das Finger Management ist, dass es nach einer Meta-Information über die Daten aufgebaut werden kann, hier gibt es vielleicht nützliche Algorithmen:

* Eine Idee wäre es die Daten nach Clustern einzuteilen und in jeden Mittelpunkt eines Clusters immer einen Finger zusetzen
* Wenn man geschickt solche Cluster vorhersagen kann, dann kann man im Vorfeld schon eine Optimierte Datenstruktur anlegen
* http://page.mi.fu-berlin.de/mulzer/pubs/statoptIJCGA.pdf

# Zusammenfassung

# Fazit und Ausblick



Abbildung 10.1: Beispiel 1 zum Einfügen einer Grafik

# Begriffe und Anhang

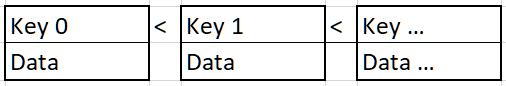
## Begriffe

### Lokalitätsprinzip

### Daten/Values

Laut dem Duden sind Daten durch Beobachtungen, Messungen, statistische Erhebungen u.a. gewonnene Zahlenwerte, welche auf Beobachtungen, Messungen, statistischen Erhebungen oder Angaben formulierbare Befunde sind. (<https://www.duden.de/rechtschreibung/Daten>, 04.05.2021, 14:00 Uhr) Daten und Values werden in dieser Arbeit analog sowohl in der schriftlichen Ausarbeitung als auch im Code für die Simulkation verwendet.

In meiner Arbeit wird vorallem die Ordinale Eigenschaft der Daten/Values vorrausgesetzt. Ordinal bedeutet, dass man die Daten ordnen oder einer Reihenfolge zuweisen kann. In dieser Arbeit haben Datenwerte/Values deshlab jeweils einen Key, dieser Key unterliegt der ordinalen Eigenschaft. Insbesondere kann man über die Key‘s aussagen, dass wenn sie verglichen werden, es möglich ist, dass die Keys entweder im Verhältniss größer, gleich oder kleiner zueinender zustellen. ([https://de.wikipedia.org/wiki/Ordinalzahl 04.05.2021](https://de.wikipedia.org/wiki/Ordinalzahl%2004.05.2021), 04.05.21 14:07 Uhr) ([https://www.unibw.de/hum-bildungswissenschaft/professuren/swm/methodenskripte/deskriptive-statistik.pdf 04.05.2021](https://www.unibw.de/hum-bildungswissenschaft/professuren/swm/methodenskripte/deskriptive-statistik.pdf%2004.05.2021), 14:11 Uhr )



Die Kombination aus Datenwerten/Value mit Key heißen Knoten. Knoten können in einer Liste wie in der oberen Abbilung angeordnet werden.

### Datenstrucktur

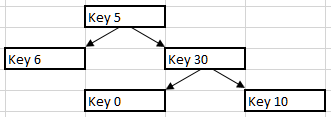
„Wie für viele fundamentale Begriffe der Informatik gibt es auch für […] Algorithmen und Datenstrukturen, nicht eine einzige, scharfe, allgemein akzeptierte Definition.“ ([https://www.fernuni-hagen.de/mi/studium/module/pdf/Leseprobe-komplett\_01662.pdf Seite 1](https://www.fernuni-hagen.de/mi/studium/module/pdf/Leseprobe-komplett_01662.pdf%20Seite%201), erster Satz, 04.05.13:50 Uhr). Daher halte ich mich an die Definition aus Wikipedia:

„In der Informatik und Softwaretechnik ist eine Datenstruktur ein Objekt, welches zur Speicherung und Organisation von Daten dient. Es handelt sich um eine Struktur, [in der] die Daten in einer bestimmten Art und Weise angeordnet und verknüpft werden, um den Zugriff auf sie und ihre Verwaltung effizient zu ermöglichen.“ ([https://de.wikipedia.org/wiki/Datenstruktur Absatz 1](https://de.wikipedia.org/wiki/Datenstruktur%20Absatz%201), 04.05.2021 13:54 Uhr)

In dieser Arbeit untersuchten Datenstruckturen haben drei wesentliche Opetationen oder Zugriffsmöglichkeiten auf die Daten: insert, delete und search.

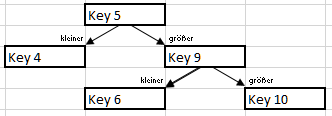
### Baum

Das ist eine Datenstruktur, mit dem sich hierarchische Strukturen abbilden oder erstellen lassen. Dabei können ausgehend von der Wurzel mehrere gleichartige Objekte mit den Keys miteinander verkettet werden, sodass eine lineare Struktur einer Liste aufgebrochen wird und eine Verzweigung stattfindet. Da Bäume zu den meistverwendeten Datenstrukturen in der Informatik gehören, gibt es viele Spezialisierungen und Deutungen. (<https://de.wikipedia.org/wiki/Baum_(Datenstruktur)> 04.05.2021 14:47 Uhr)



#### Binär-Baum

Binärbäume sind in der Informatik die an der häufigsten verwendeten Unterart der Bäume. Im Gegensatz zu anderen Arten von Bäumen können die Knoten eines Binärbaumes nur höchstens zwei direkte Nachkommen haben. Meistens werden an den jeweiligen Verbindungen oder Pointern ordinäre Vergleiche verwendet, wie größer oder kleiner als der aktuelle Knoten.

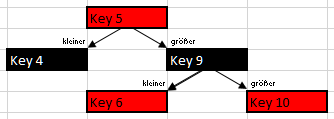


Die Binärsuche startet in der Wurzel – dem obersten Knoten - und folgt den Pointern zu den Kindern, den untergeordneten Knoten. Um diese Suche von der Fingersuche zu unterscheiden, nenne ich diese Suche Wurzelsuche.

Die Wurzelsuche hat eine Worst-case Laufzeit von O (log n), wobei n die Anzahl der Knoten ist. In meiner implementierung haben die Funktionen Insert und Delete eine Worst-case Laufzeit von O (1), wobei diesen beiden Operationen eine Suche vorgeschaltet ist.

#### Rot-Schwarz-Baum

Rotschwarz-Bäume sind eine spezielle Art der Binärbäume. Die enthaltenen Knoten werden je nach Ebene des Baumes entweder rot oder schwarz gefärbt. Diese Datenstrucktur hat den Vorteil, dass es einen Algorythnus der balancierung gibt, welcher die zusätzliche Information der unterscheidliochen Knotenfarben ausnutzt und deshlab schneller ist.



(<https://en.wikipedia.org/wiki/Red%E2%80%93black_tree> 04.05.2021 15:06 Uhr)

### Skipliste

### Finger

Finger sind Pointer. In dieser Arbeit haben Finger die Besonderheit dass sie extern auf die jeweilige Datenstrucktur

### Fingersearch und Fingersuche

### Splay tree

Original paper https://dl.acm.org/doi/pdf/10.1145/3828.3835

# Literaturverzeichnis

Brink, A. (2005). *Anfertigung wissenschaftlicher Arbeiten.* Oldenbourg.

Forbes, M. H. (1994). An exact algorithm for multiple depot bus scheduling. *European Journal of Operational Research, 72 (1)*, S. 115 - 124.

Gopen, G., & Swan, J. (1990). The Science of Scientific Writing If the reader is to grasp what the writer means, the writer must understand what the reader needs. *American Scientist 78 (6)*, S. 550 - 558.

Kalyan T. Talluri, G. J. (2005). *Revenue Management.* New York: Springer.

Kliewer, N. (2005). Optimierung des Fahrzeugeinsatzes im öffentlichen Personennahverkehr. *Dissertation*. Universität Paderborn.

Kohm, M., & Morawski, J.-U. (2014). KOMA-Script, Die Anleitung. Abgerufen am 29. 10 2014 von https://www.rrzn.uni-hannover.de/fileadmin/kurse/

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbständig und ohne unerlaubte fremde Hilfe angefertigt, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Quellen und Hilfsmitteln wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Berlin, den 1. Februar 2042

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(Unterschrift des Verfassers)