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 um externe Pointer – Finger – und wie man den Suchprozess in Datenstrukturen durch sie zu beschleunigen kann. Die Laufzeiten der Such-Funktion können durch extra Finger in einer Datenstruktur z.B. bei Bäumen von O(log n) auf O(log d) [1] reduziert. Diese Art der Suche heißt Distanzsuche, da von einer gewissen Distanz zur Wurzel der Suchauftrag erst startet.

Das Ziel dieser Arbeit, eine Simulation der Fingersuche zu programmieren, denn 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 Datenstruktur orientiert, als wenn man diese strukturelle Information nicht berücksichtigt und mit der konventionelle Wurzelsuche gesucht wird.

**Abstract - English**

TODO: HEURISTIK (finger) begriff einbauen

Inhaltsverzeichnis

[Abbildungsverzeichnis ix](#_Toc80022616)

[Einleitung 1](#_Toc80022617)

[1.1 Problemumfeld 1](#_Toc80022618)

[1.2 Zielsetzung 2](#_Toc80022619)

[Aktueller Forschungsstand: Übersicht über Finger 3](#_Toc80022620)

[1.3 Ein Finger 3](#_Toc80022621)

[1.4 Min-Max-Fingersuche 4](#_Toc80022622)

[1.5 Fingersuche mit Direktion 5](#_Toc80022623)

[1.6 Finger verwalten oder mit Fingern suchen 5](#_Toc80022624)

[1.7 Lazy Finger in Binär-Suchbäumen 6](#_Toc80022625)

[1.8 Splay-Tree und Fingersuche 6](#_Toc80022626)

[1.9 Splay-Trees, Zufall und Optimalität 6](#_Toc80022627)

[1.10 Fingersuche und weitere Datenstrukturen 7](#_Toc80022628)

[1.10.1 K-server Problem und multiple Finger 7](#_Toc80022629)

[1.10.2 SplayNets – Netzwerk Kommunikation 8](#_Toc80022630)

[1.10.3 Grover Algorithmus – Quantencomputer 9](#_Toc80022631)

[1.10.4 Prädiktion auf Input Streams für Lokalitätsprinzip 9](#_Toc80022632)

[1.11 Zusammenfassung des Forschungsstandes 9](#_Toc80022633)

[Modellierung des Problems 11](#_Toc80022634)

[1.12 Das Modell 11](#_Toc80022635)

[1.12.1 Arten des Fingermanagement 13](#_Toc80022636)

[1.12.2 Arten der Hauptdatenstruktur 13](#_Toc80022637)

[1.13 Einwände bezüglich des Modells 14](#_Toc80022638)

[Theoretische Betrachtung 16](#_Toc80022639)

[1.14 Anzahl der Finger 16](#_Toc80022640)

[1.15 Laufzeit Betrachtung 19](#_Toc80022641)

[1.16 Erkenntnisse für Fingerarten 19](#_Toc80022642)

[1.17 Bedeutung für Splay-Tree 19](#_Toc80022643)

[Programmierung der Simulation 21](#_Toc80022644)

[1.18 Vereinfachung des Modells 21](#_Toc80022645)

[1.19 Das Vorgehen - Spezifikation 21](#_Toc80022646)

[1.20 Klassendiagram 22](#_Toc80022647)

[1.21 Ausführen vom Code 24](#_Toc80022648)

[Evaluation der Ergebnisse 25](#_Toc80022649)

[1.22 Verteilungen 25](#_Toc80022650)

[1.22.1 Immer Zahl “1” suchen 25](#_Toc80022651)

[1.22.2 Immer Maximum Suchen 27](#_Toc80022652)

[1.22.3 Immer das mittelste Element suchen 29](#_Toc80022653)

[1.22.4 Immer Zick-Zack Suchen 31](#_Toc80022654)

[1.22.5 Random zwischen Minimalen und Maximalem Element 33](#_Toc80022655)

[1.22.6 Random vs. Arround Cluster Search 35](#_Toc80022656)

[1.23 Zusammenfassung: Direktvergleich der Graphen 35](#_Toc80022657)

[Einordnung der Ergebnisse Anhand des Forschungsstandes 37](#_Toc80022658)

[1.24 Trade-Off für Anzahl der Finger 37](#_Toc80022659)

[1.25 Informationen nutzen 37](#_Toc80022660)

[1.26 Historische Einordnung der Sicht auf die Finger 39](#_Toc80022661)

[1.27 Fazit 39](#_Toc80022662)

[Zusammenfassung 40](#_Toc80022663)

[Appendix 41](#_Toc80022664)

[1.28 Ausblick und Potential 41](#_Toc80022665)

[1.28.1 Fingersuche und Datenbanken 41](#_Toc80022666)

[1.28.2 Finger und Hardwaretechnology 43](#_Toc80022667)

[1.29 Grenzen der Fingersuche 44](#_Toc80022668)

[1.29.1 Stack 44](#_Toc80022669)

[1.29.2 Radix-Tree / Präfix-Tree 44](#_Toc80022670)

[Begriffe 45](#_Toc80022671)

[1.30 Lokalitätsprinzip 45](#_Toc80022672)

[1.31 Daten/Values 45](#_Toc80022673)

[1.32 Datenstruktur 46](#_Toc80022674)

[1.33 Baum 46](#_Toc80022675)

[1.34 Binärer Baum 47](#_Toc80022676)

[1.35 Rot-Schwarz-Baum 47](#_Toc80022677)

[1.36 Finger 48](#_Toc80022678)

[1.37 Splay-Tree 48](#_Toc80022679)

[Literaturverzeichnis 51](#_Toc80022680)

Abbildungsverzeichnis

[1 Abb.: Schematische Darstellung von einem Finger in einem Baum 3](file:///C:\Users\Anna\git\ba_schap\inBearbeitung_schriftlicheArbeit_v1.docx#_Toc80022532)

[2 Abb.: Schematische Darstellung: Distanzsuche mit Startposition 3](file:///C:\Users\Anna\git\ba_schap\inBearbeitung_schriftlicheArbeit_v1.docx#_Toc80022533)

[3 Abb.: Schematische Darstellung, Min-Max Finger 4](file:///C:\Users\Anna\git\ba_schap\inBearbeitung_schriftlicheArbeit_v1.docx#_Toc80022534)

[4 Abb.: Schematische Darstellung, Finger und Beweglichkeit 5](file:///C:\Users\Anna\git\ba_schap\inBearbeitung_schriftlicheArbeit_v1.docx#_Toc80022535)

[5 Abb.: Schematische Darstellung, Lazy-Finger 6](file:///C:\Users\Anna\git\ba_schap\inBearbeitung_schriftlicheArbeit_v1.docx#_Toc80022536)

[6 Abb.: Schematische Darstellung, k-Server mit k -Fingern 8](file:///C:\Users\Anna\git\ba_schap\inBearbeitung_schriftlicheArbeit_v1.docx#_Toc80022537)

[7. Abb.: links: Wurzelsuche, rechts: Fingersuche mit Fingermanagement 12](#_Toc80022538)

[8 Abb. Schematische Darstellung Fingermanagement 16](#_Toc80022539)

[9 Abb.: Abschätzung 18](#_Toc80022540)

[10 Abb.: Klassendiagram 23](#_Toc80022541)

[11 Abb.: Activitydiagram für User 24](file:///C:\Users\Anna\git\ba_schap\inBearbeitung_schriftlicheArbeit_v1.docx#_Toc80022542)

[12 Abb.: Minimum Search – Anzahl Nodes pro eine Suche 26](#_Toc80022543)

[13 Abb.: Minimum Search – Anzahl Nodes pro eine Suche (größerer Wertebereich) 27](#_Toc80022544)

[14 Abb.: Maximum Search – Anzahl Nodes pro eine Suche 28](#_Toc80022545)

[15 Abb.: Maximum Search – Anzahl Nodes pro eine Suche (größerer Wertebereich) 29](#_Toc80022546)

[16 Abb.: Middle Key Search - Anzahl Nodes pro Suche 30](#_Toc80022547)

[17 Abb.: Middle Key Search - Anzahl Nodes pro Suche (größerer Wertebereich) 31](#_Toc80022548)

[18 Abb.: Alternierend Minimum and Maximum - Anzahl Nodes pro Suche 32](#_Toc80022549)

[19 Abb.: Alternierend Minimum and Maximum - Anzahl Nodes pro Suche (größerer Wertebereich) 33](#_Toc80022550)

[20 Abb.: Random Key Search - Anzahl Nodes pro Suche 34](#_Toc80022551)

[21 Abb.: Abb.: Random Key Search - Anzahl Nodes pro Suche (größerer Maßstab) 35](#_Toc80022552)

[22 Abb.: Überblick Anzahl der Nodes pro eine Suche 37](#_Toc80022553)

[23 Abb.: häufigste Datenbank-Kategorien 42](#_Toc80022554)

[24 Abb.: Die vier beliebtesten Datenbankmodelle: Absolute Anzahl seit Januar 2013 43](#_Toc80022555)

[25 Abb: Radix, Prefix tree or Trie 45](file:///C:\Users\Anna\git\ba_schap\inBearbeitung_schriftlicheArbeit_v1.docx#_Toc80022556)

[26 Abb.: Nachdem 4 gefunden wurde 49](file:///C:\Users\Anna\git\ba_schap\inBearbeitung_schriftlicheArbeit_v1.docx#_Toc80022557)

[27 Abb.: Aktueller Splaytree 49](file:///C:\Users\Anna\git\ba_schap\inBearbeitung_schriftlicheArbeit_v1.docx#_Toc80022558)

[28 Abb.: Nachdem 7 gefunden wurde 49](file:///C:\Users\Anna\git\ba_schap\inBearbeitung_schriftlicheArbeit_v1.docx#_Toc80022559)

# Einleitung

Alle Datenstrukturen implementieren typischerweise immer die gleichen Operationen Search, Insert und Delete. Der Fokus dieser Arbeit ist es, die Search-Funktion mit externen Pointern, sogenannten Finger, zu beschreiben und wie der Suchprozess durch sie beschleunigt werden kann.

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

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

## Problemumfeld

Unbrauchbar wird der Lazy-Finger, wenn abwechselnd immer ein maximales und ein minimales Node in der Datenstruktur gesucht wird. Denn bei so einer Nutzung der Datenstruktur muss der Lazy-Finger einen besonders lange Distanz zum Suchergebnis zurücklegen und verhält sich schlechter als die Wurzelsuche.

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

Diese Fingersuche würde zwar bei den Suchanfragen, welche immer das Maximum und das Minimum suchen oder zumindest in deren Nähe suchen deutlich besser performieren, als der Lazy-Finger. Jedoch lässt sich auch bei dieser Fingersuche ein ungünstiger Fall konstruieren, innerhalb dessen die Wurzelsuche schneller ist.

Z.B. könnte man annehmen, dass die Datenstruktur ein Binärer-Suchbaum ist. Die am meisten gesuchten Nodes würden immer vom linken Teilbaum ganz rechts sein und anschließend im ganz rechten Teilbaum das linkeste Node gesucht werden.

Auch hier würde sich die Wurzelsuche im Vergleich zu der Min-Max-Suche schneller verhalten, denn diese Finger-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 hoch frequentierten Bereiche der Datenstruktur stellen sollte. Am sinnvollsten wären das Bereiche, von denen man aus der Vergangenheit weiß, dass in diesen Bereichen besonders oft gesucht wird.

Somit wäre somit die Aufgabe der Fingersuche eine neue weitere Datenstruktur zu sein, welche das Wissen über die besonders oft nachgefragten Startpositionen der Distanzsuche in sich enthält.

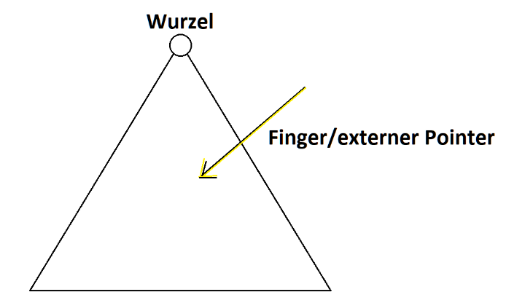
Ich möchte gerne untersuchen, ob verschiedenen Fingersuchen sich je nach Verteilung der Suchanfragen besonders nützlich oder träge verhalten.

Zunächst werde ich für die Fragestellung eine theoretische Überlegung diskutieren und anschließend mit der Simulation vergleichen.

# Aktueller Forschungsstand: Übersicht über Finger

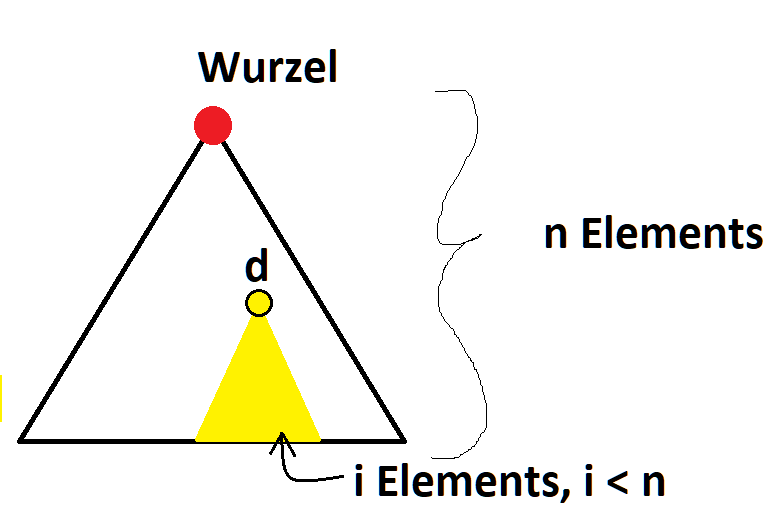
Dieses Kapitel ist ein Überblick über Varianten von Fingern, Fingersuchen und Datenstrukturen. In dieser Arbeit wird übersichtshalber davon ausgegangen, dass die Elemente einer gegeben Datenstruktur die ordinale Eigenschaft besitzen.

## Ein Finger

Eine Fingersuche mit einem Finger ist besonders anschaulich in Abb. 2‑1 dargestellt, am Beispiel eines binären Suchbaums.

Finger sind externe Pointer, welche genutzt werden, um von ihrer jeweiligen Startposition - statt von der Wurzel aus zu suchen [1].

1 Abb.: Schematische Darstellung von einem Finger in einem Baum

Hier wird vorausgesetzt, dass der Finger schon so positioniert ist, dass die Finger-Position näher zum Ergebnis liegt, als die Wurzelsuche und somit das Lokalitätsprinzip für das Suchergebnis und den Finger gilt.

Das führt zur Überlegung, dass die Wurzelsuche, wie zu sehen in Abb. 2‑2 mit rot markiertem Startpunkt (genannt Root-Finger), in einem Binärbaum Elemente durchsuchen kann, um das Ergebnis zu liefern [3].

2 Abb.: Schematische Darstellung: Distanzsuche mit Startposition

Innerhalb des gelben Ausschnittes, welcher durch die Finger-Startposition d begrenzt wird, kann das zu suchende Node nach schon Vergleichen gefunden werden.

Da i kleiner ist als n, kann man von einem Vorteil sprechen.

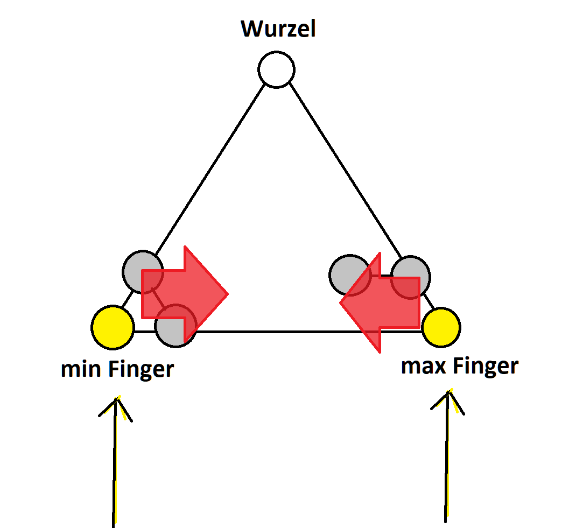
Man könnte sogar sagen, je näher der Finger am Suchergebnis ist, desto exponentiell stärker ist dieser Effekt - veranschaulicht bedeutet das, dass die Fläche von dem gelben Bereich kleiner werden würde.

Formal ausgedrückt arbeitet die Wurzelsuche in der Laufzeitklasse O(log n), während die Distanzsuche auf i-Elementen arbeitet, d.h. bei Verwendung eines Fingers:

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

## Min-Max-Fingersuche

Das Fast-Finger-Search Konzept beinhaltet, dass einfach von allen externen Fingern die Suche gleichzeitig gestartet werden soll. Wenn einer der Finger ein Ergebnis findet, soll die Suche terminieren. [4]

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. Eine Schematische Darstellung dieses Konzeptes ist in Abb. 2‑3 dargestellt.

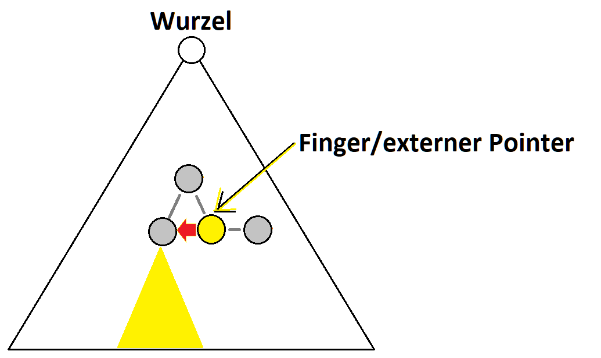
Diese Laufzeit ergibt deshalb:

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

In [5] wird dargestellt, dass versteckte Suchen wie jene in der Join-Operation oder Teile der Split-Operation auch von der Min-Max-Fingersuche profitieren können, da sie schon auf Teilbäume zugreifen und man deshalb keine temporären Wurzeln von Teilbäumen zusetzen braucht.

3 Abb.: Schematische Darstellung, Min-Max Finger

## Fingersuche mit Direktion

Schon 1977 [6] wird eine Fingersuche vorgestellt, welche eine Suchrichtung besitzt. Wenn der Finger auf einer Position steht, kann es passieren, dass seine Nachbar-Nodes durchsucht werden müssen, bevor eine nach unten gerichtete Suche Distanzsuche - wie bei der Wurzelsuche - im Baum geschehen kann.

4 Abb.: Schematische Darstellung, Finger und Beweglichkeit

Finger werden in diesem Paper vor allem dazu genutzt, um auf einzelne Nodes in Binären-Suchbäumen, welche Listen enthalten, zuzugreifen. So kann es vorkommen, dass wenn ein Finger z.B. auf dem kleinsten Element der Liste im aktuellen Node steht, dass das nächst bekannte kleinere Element das Maximum der Nachbarnode-Liste ist – also geht der Finger in den Nachbar-Node auf der gleichen Ebene. In dieser Binärbaum-Finger-Implementierung kann man also auf einer Ebene benachbarte Nodes durchsuchen, man könnte sogar sagen die Listen auf jeder Ebene sind verlinked.

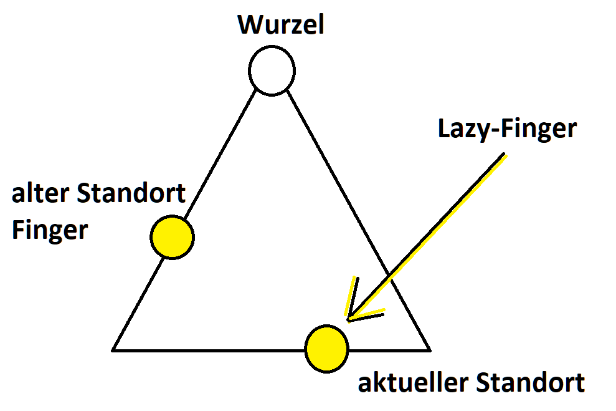
Es wird auch der Fall betrachtet, dass Finger verschoben, also ein Update verlangen; am Beispiel von einer Insert-Operation wird festgestellt, dass Finger unter Umständen zu nahe beieinander liegen.

## Finger verwalten oder mit Fingern suchen

In einem Konzept aus dem Jahr 1982 [7] werden auf Level-Linked-Binärbäumen Finger immer nur auf Leafs gesetzt. Die Autoren stellen fest, dass in der amortisierten Laufzeit die logarithmische Suche deutlich stärker zu tragen kommt als die konstante Laufzeit für die Erzeugung und Änderung der Finger.

Damit schlussfolgern die Autoren, dass der wichtige zu betrachtende Aspekt in der Laufzeit der Fingernutzung, die Fingersuche an sich ist und nicht die Verwaltung der Finger.

## Lazy Finger in Binär-Suchbäumen

In einem Conference Paper aus dem Jahr 2014 [8] 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 – wie in der Abb. 2‑5 schematisch dargestellt.

5 Abb.: Schematische Darstellung, Lazy-Finger

Die Fragestellung der Abstände vom Suchergebnis und Lazy-Finger wird im Zusammenhang von Entropien (Informationstheorie) diskutiert: *“Given the pairwise frequencies […] finding the tree that minimizes the execution time of search sequence X using lazy finger takes time ”* - das ist die Laufzeit, um eine optimale Lazy-Finger Position in einem Binären-Baum zu berechnen [8].

## Splay-Tree und Fingersuche

Hier [9] steht in der Einleitung übersetzt: “Im Jahr 2000 haben Cole [..] gezeigt, dass Splay-Trees (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“

## Splay-Trees, Zufall und Optimalität

Für viele Sequenzen in Suchanfragen von nicht-zufälligen Operationen arbeiten Splay-Tree- Datenstrukturen besser als andere Searchtrees - sogar besser als O(log n) für ausreichend nicht-zufällige Muster, selbst ohne das vorherige Kennen des Suchmusters. [10]

Die Eigenschaft von Splay-Trees, wie eine statische Datenstruktur – z.B. Binärbaum verhalten zu können, bezeichnet man als „statische Optimalität“ (siehe Kapitel „Splay-Trees und Fingersuche“) [10].

Eine ähnliche Vermutung gibt es für dynamische Datenstrukturen – also Datenstrukturen, welche sich 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“ [11].

Wenn man annimmt, dass die Suchanfragen an eine Datenstruktur nicht chaotisch sind, liegt es nahe, Splay-Trees, statt binären Suchbäumen einzusetzen.

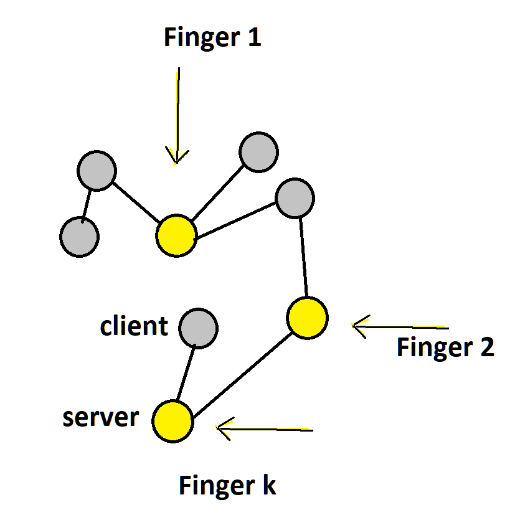
## Fingersuche und weitere Datenstrukturen

Das Konzept der Finger - als externe Pointer - hat keine Einschränkung für spezielle Datenstrukturen. Daher zählt dieses Kapitel, welche weitere Datenstrukturen bei der Recherche im Zusammenhang mit der Fingersuche aufgefallen sind:

* Listen (2 Finger auf den Extremwerten, um schneller Listen Operationen auszuführen, u.a. auch merge-Operation) [12]
* Treaps [2]
* Skip-List nur mit 1 Finger [2]
* (2,4) Level Linked Tree [2]

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

### K-server Problem und multiple Finger

In diesem Paper [13] wird versucht dem k-server Problem mit k vielen Fingern zu begegnen. Es wird davon ausgegangen, dass alle Server und alle Clients eine „Karte aller Nodes“, also die komplette Information zum Entscheiden haben.

6 Abb.: Schematische Darstellung, k-Server mit k -Fingern

Die grundsätzliche Idee ist, dass jedem Server ein Finger zugewiesen wird und anhand dieser Finger die nächstmöglichen Clients gefunden (Fingersuche) und 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 Doktorarbeit [14] wird das Konzept der SplayNets präsentiert. SplayNets arbeiten laut der Arbeit auf P2P Netzwerken und versuchen die optimalen Kommunikationswege mit Hilfe von Splay-Tree als Fingersuchen zu sogenannten SplayNets zu vereinfachen. Jeder Node erhält also einen Finger/Pointer. Die Verbindungen zwischen den Fingern werden gewichtet und mit Splay-Trees optimiert aufgespannt.

Es werden Anwendungsbereiche eines Overlay-Konzepts für Tor und weitere P2P Networks, sowie für interne Kommunikation in Facebooks-Datencentren vorgestellt, da 60% des Traffics interne Kommunikation ist. [14]

SplayNets führen zu erheblicher Verbesserung der internen Kommunikationswege, da mit ihrer Hilfe irrelevante protokollbasierte Informationsaustauschversuche deutlich verringert werden können. [14]

### Grover Algorithmus – Quantencomputer

Der Grover Algorithmus arbeitet ausschließlich auf Quantencomputern - es werden quantenmechanische Effekte ausgenutzt und geschickt ausgelesen. Er arbeitet auf unsortierten Datenstrukturen der Größe n und ermöglicht eine Suche in O().

Dem gegenüber stehen herkömmliche Algorithmen welche auf unsortierten Listen im Worstcase O(n) viele Vergleiche tätigen müssten, bis das gesuchte Element gefunden wird.

Als Anwendungsvorschlag ist Grovers Algorithmus auf großen unsortierten Datenbanken erwähnt. [15]

### Prädiktion auf Input Streams für Lokalitätsprinzip

In diesem Paper [16] wird der Sachverhalt der Vorhersage von Inputdaten auf verschiedene Systeme analysiert. Es wird explizit der Nutzen der Vorhersage im Sinne des Lokalitätsprinzips erwähnt, und es wird aus Geschwindigkeitsgründen in diesem Zusammenhang ein Splay-Tree verwendet. Es gibt auch abgewandelte Variationen mit gewichteten Kanten oder geclusterten Nodes im Splay-Tree.

In diesem Paper wird nicht explizit von Fingern gesprochen, jedoch werden die Ergebnisse der Vorhersage für den Splay-Tree wie für Finger eingesetzt.

## Zusammenfassung des Forschungsstandes

Sowohl Finger, Fingersuchen als auch die Fingersuche mit Splay-Trees sind in verschiedenen Fachgebieten präsent. So sind sie in der technischen Mathematik, im Bereich von Netzwerkkommunikation, im Bereich der Datenstrukturen, als auch in der 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.

Es gibt mindestens eine grundsätzliche offene wissenschaftliche Frage bei der dynamischen Optimalität der Fingersuche.

# Modellierung des Problems

## Das Modell

Gegenstand dieser Arbeit ist, dass je nachdem welches Konzept (vorheriges Kapitel: „Aktueller Forschungsstand: Übersicht über Finger“) für die Fingersuche implementiert wird, es zu nachteiligem Verhalten führen kann. Meines Erachtens sollte die Wahl eines Fingerkonzepts davon abhängen, wie Datenstruktur verwendet wird. Daraus folgt, dass der im Einzelfall angemessenen Wahl des Fingerkonzepts eine Prognose über die Nutzung der vorliegenden Datenstruktur vorhergehen sollte:

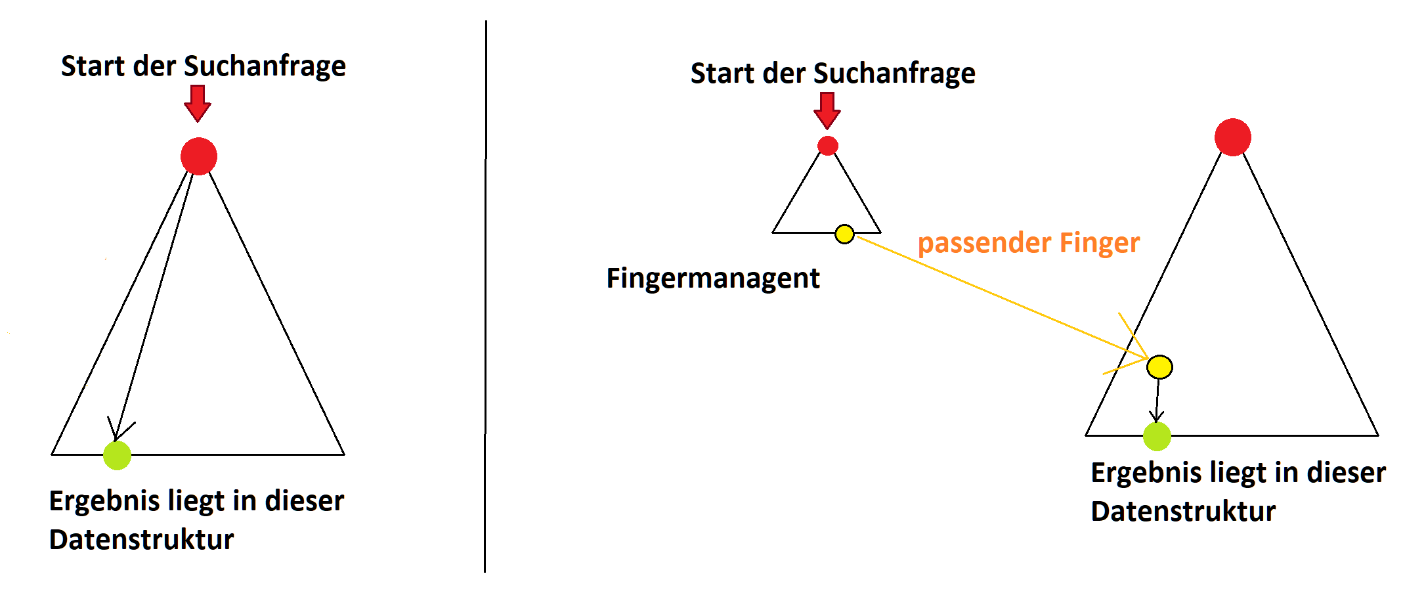
* Z.B. wird beim Lazy-Finger indirekt davon ausgegangen, dass nur ein Bereich im binären Suchbaum besonders interessant ist. Jeder Wert außerhalb dieses Clusters ist ein ungünstiger Fall. Was passiert, wenn die gesamten Suchaufträge ungünstige Fälle sind? Bei einer Zick-Zack Verteilung der Keys wäre das tatsächlich eine ungünstige Nutzung, weil zwei verschiedene Enden der Datenstruktur vermischt abgefragt werden.
* Bei dem Min-Max Finger sind schonmal zwei Cluster abgedeckt, jedoch würde diese Fingersuche bei einem dritten Cluster in der Mitte der Datenstruktur auch nicht besser als die Wurzelsuche performieren.

Es gibt theoretische Arbeiten, welche sich mit optimalen Fingerpositionen (vorheriges Kapitel: „Lazy-Finger in Binär-Suchbäumen“) befassen. Diese Lösungsvorschläge sind nicht trivial nachzuvollziehen.

Mein Ansatz ist daher die Simulation, man kann durch Programmierung eine Simulation und damit eine Verbesserung durch Finger belegen.

Um sich dieser Frage zu nähern, bietet sich folgender Versuchsaufbau an: statt üblicher Suche ohne die Finger (3‑1 Abb. links) würde ich ein Finger Management vorschalten (3‑1 Abb. rechts).

Dieses Finger-Management ist allgemeiner gefasst und es speichert in sich selbst die Vorhersage, wie die große Datenstruktur verwendet wird.



7. Abb.: links: Wurzelsuche, rechts: Fingersuche mit Fingermanagement

Im Unterschied zur weitverbreiteten Wurzel-Suche (3‑1 Abb. links) wird in der Fingersuche (3‑1 Abb. rechts) die Startposition der Suchanfragen anders sein.

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

Das Kernstück dieser Arbeit ist also das Fingermanagement – genauer die Search-Operation mit dem Fingermanagement.

Je nach Versuch ist die Fingermanagement Datenstruktur entweder als ein Lazy-Finger, Min-Max-Finger oder ein Splay-Tree implementiert.

### Arten des Fingermanagement

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

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

Man könnte auch weitere Datenstrukturen in Betracht ziehen, wie binäre Suchbäume, dann stellt man sich logischerweise die Frage welche Pointer genauso einen Baum aufgenommen werden sollten. Man könnte aus einem Rot-Schwarz Baum nur die Roten Nodes rausnehmen und eine Red-Finger-Search implementieren. Man könnte sich an dieser Stelle auch ein Kostenfunktion 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 nicht.

* Oder man entscheidet sich für einen Splay-Tree. Der Splay-Tree besitzt einen bubble-up Effekt für die am häufigsten verwendeten Finger – und einen sink-to-bottom Effekt für unbenutzte Nodes oder Finger. Mit einem Splay-Tree müsste man sich keine größeren Gedanken machen, wie man Finger einer wartungsaufwändigen Kostenfunktion unterzieht, sortiert und verwaltet - der Splay-Tree optimiert sich dahingehend selbst. Wie im Kapitel „Splay-Tree, Zufall und Optimalität“ herausgearbeitet, besitzt der Splay-Tree außerdem die statische Optimalität.

### Arten der Hauptdatenstruktur

Mein Modell beschränkt sich nicht nur – wie durch die Skizze angedeutet – auf Bäume. Die Hauptdatenstruktur kann auch eine Liste, Skipliste oder eine historisch seltsam gewachsene und auch unsortierte Datenbank sein. Fingersuchen können außerdem auch auf Graphen angewendet werden, wie z.B. in P2P-Netzen [14].

## Einwände bezüglich des Modells

Ein Kritikpunkt[[1]](#footnote-1) ist, dass man auch Balancierungs- und weitere baumtypische Operationen berücksichtigt werden müssen, denn durch z.B. Baumrotationen könnten Finger ungünstig verteilt werden.

In dieser Arbeit gehe ich davon aus, dass diese Operationen, welche eine versteckte Suche enthalten, jeweils ein eigenes Fingerkonzept sowie eine eigene Fingermanagement-Datenstruktur erhalten. Zwangsläufig würden alle die Datenstruktur manipulierenden Operationen zur Neuaufsetzung der Finger führen.

Der Einfachhalt und Vergleichbarkeit halber beschränkt sich diese Arbeit deswegen ausschließlich auf die Suchfunktion.

Wie im vorherigen Kapitel: “Splay-Tree und Fingersuche“ geschrieben, dass Splay-Tree und Finger-Suchen sich asymptotisch gleich verhalten, könnte daher davon ausgehen, dass man einfach die Hauptdatenstruktur als Splay-Tree zu implementieren braucht und sich den Overhead mit dem Fingermanagement sparen kann.

Dieser Überlegung würde ich gerne einige Argumente entgegenstellen:

* Splay-Tree‘s arbeiten oft im Cache, das bedeutet man versucht diese Datenstruktur möglichst klein zu halten. Damit wäre dieser Lösungsansatz, statt der Fingersuche Splay-Tree in der Hauptdatenstruktur zu verwenden, nur für kleine Datenansammlungen geeignet. Bei Splay-Tree kann es außerdem zu der nachteiligen linearen Formation kommen, das ist bei besonders bei sehr großen Splay-Trees ungünstig.
* Wenn man annimmt, dass die Hauptdatenstruktur auf verteilten Systemen arbeitet, dann macht es keinen Sinn diese Datenstruktur nur wegen Suchanfragen jedes Mal umzubauen, das wäre unter Umständen sehr aufwändig. Außerdem wenn man bedenkt, dass viele Datenbanken historisch gewachsen sind, ist es vielleicht einfacher externe Finger anzubringen als die komplette Datenbank jedes Mal umzustrukturieren.
* Ein weiteres Argument wäre theoretischer Natur, man kann generell Fingersuchen auf unendlich wachsenden Datenstrukturen anwenden, solange sie auf einem Bereich suchen, welcher sich im Moment der Suche nicht verändert.

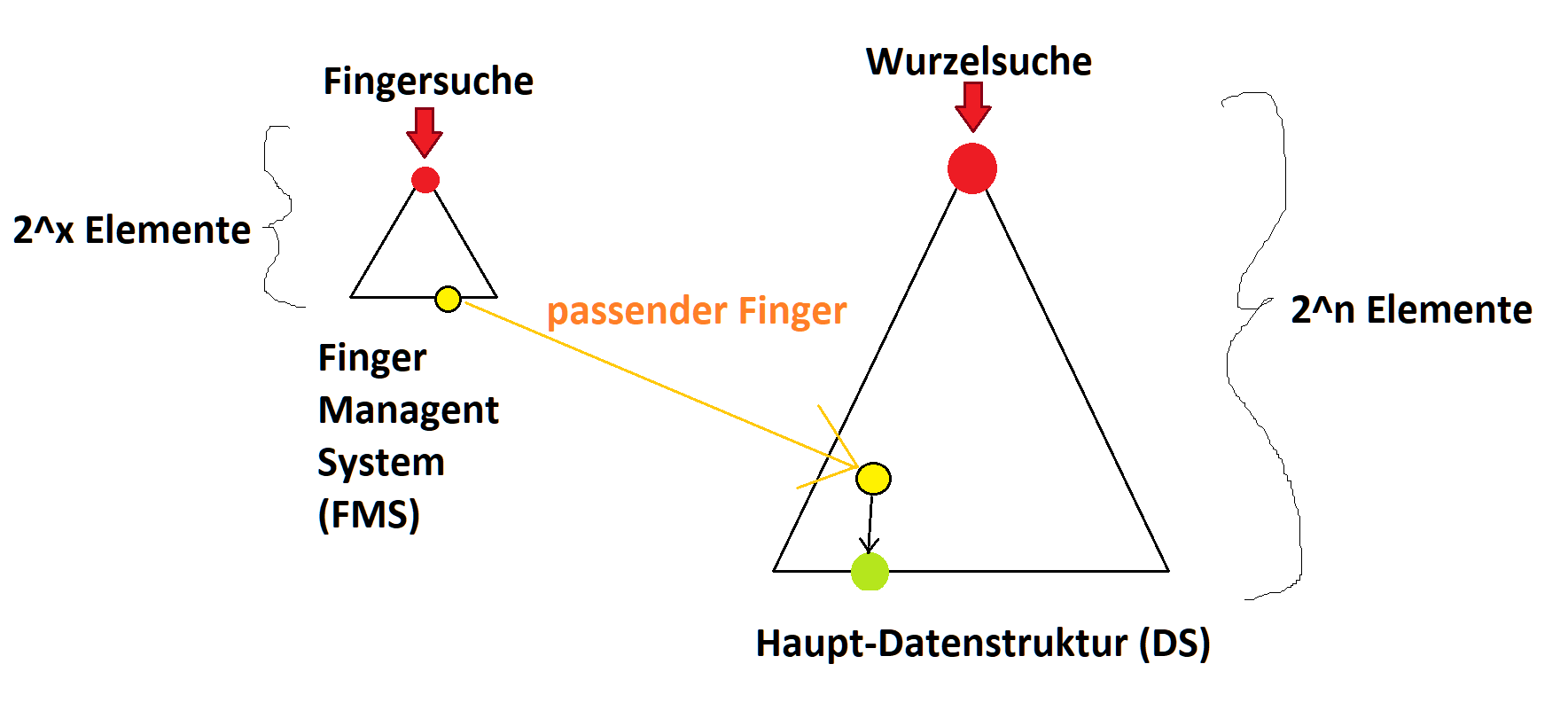
# **Theoretische Betrachtung**

In diesem Kapitel liegt der Schwerpunkt auf den durchsuchten Nodes in den Datenstrukturen und der Anzahl der Finger bei der Fingersuche beschäftigen, sowie der Laufzeitbetrachtung.

## Anzahl der Finger

Der Übersicht halber würde ich das zuvor vorgestellte Modell um die Anzahl der möglichen zu durchsuchenden Elemente (Abb. 4‑1) ergänzen:

* ist die Anzahl der Elemente in der Hauptdatenstruktur
* ist die Anzahl aller Finger im Fingermanagement



8 Abb. Schematische Darstellung Fingermanagement

1. Jetzt kann man das Ziel formulieren.

Das Ziel ist, dass die Fingersuche schneller ist als die Wurzelsuche, wobei die Fingersuche aus zwei Teilsuchen der Fingermanagement-Suche und der Distanzsuche besteht.

In dem Moment, in dem die zusammengesetzte Suche die Laufzeit der Wurzelsuche überschreiten sollte, verliert das Fingermanagement seinen Mehrwert, daher wird die Wurzelsuche als Schranke gesehen:

O(Wurzelsuche) > O(Fingersuche)

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

1. Man kann die obere Formel für beliebige Datenstrukturen verwenden. Allerdings wäre die Annahme, dass das Fingermanagementsystem (FMS) und die Hauptdatenstruktur beides binäre-Suchbäume sind.

Damit kann 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)

> + /ODER Durchschnitt(d)?

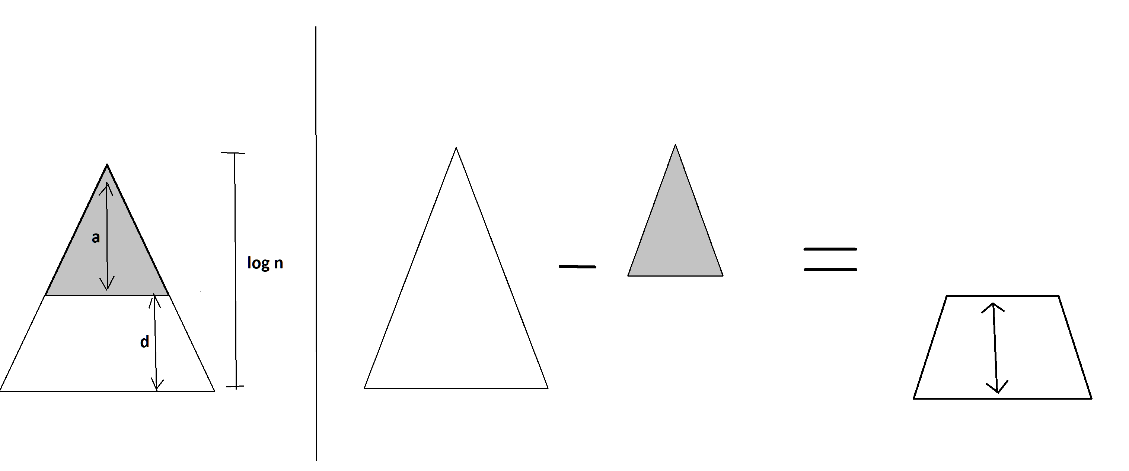
>

>

>

1. Nun könnte man eine Abschätzung tätigen.

a sei der Unterschied zwischen der Wurzel in der Hauptdatenstruktur und der Distanzsuche-Startposition d. Graphisch gesprochen benutzt meine Abschätzung (Abb. 4‑2) folgendes:



9 Abb.: Abschätzung

=

=

=

=

=

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

< (1)

< (3)

<

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-Managementsystem (FMS) nach der Höhe a „abgeschnitten“ oder gekürzt werden sollte, um einen absoluten Vorteil zu bieten.

## Laufzeit Betrachtung

Im vorherigen Kapitel wurde ermittelt, dass die Anzahl der Finger vom Finger-Management-System maximal a-viele sind - dementsprechend dauert es maximal Nodes um den richtigen Finger zu finden. Und dann nochmal d-viele Nodes um das richtige Endergebnis zugeben.

Um auf die Ausgangsformel zurückzukehren:

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

= +

=

## Erkenntnisse für Fingerarten

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

O\_worst(Fingersuche) = =

Sowohl Min- und Max-Finger als auch der Lazy-Finger würden das vorteilhaft erfüllen. Der Lazy-Finger wäre dahingehend vorteilhaft, als dass nur das Produkt d\*a ausschlaggebend ist und wenn a klein ist – was der Lazy-Finger erfüllt - wird auch das Produkt klein.

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

## Bedeutung für Splay-Tree

Wie in der Recherche im Kapitel 2.6 „“ herausgefunden, verhält sich der Splay-Tree bei genug chaotischen Anfragen asymptotisch wie ein statischer binärer Suchbaum - unter weniger chaotischen Anfragesequenzen - sogar deutlich besser.

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 Splay-Tree:

O\_worst(Fingersuche) = O\_worst(Splay-Tree) + O\_worst(Distanzsuche)

1. Worst-Case für Splay-Tree - Suchanfragen sind chaotisch genug:

O\_worst(Splay-Tree) = O\_worst(Binärsuchbaum)

= O()

* O\_worst(Fingersuche) = O\_worst(Splay-Tree) + O\_worst(Distanzsuche)
* O\_worst(Fingersuche) = O\_worst() + O\_worst()
* O\_worst(Fingersuche) =

Man könnte schlussfolgern, dass es nicht schadet, einen Splay-Tree für die Fingersuche zu verwenden.

# Programmierung der Simulation

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

## Vereinfachung des Modells

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

* Meine Hauptdatenstruktur wir ein balancierter Baum sein
* Mein Fingermanagement wird mit einem Lazy-Finger, Min-Max-Finger und mit einem Splay-Tree implementiert

## Das Vorgehen – Spezifikation

Erster Schritt – Hauptdatenstruktur initiieren:

* Ein bi-direktional verketteter Binärbaum (rotschwarz-Baum) mit den funktionierenden Methoden: insert, delete und down-search sowie two-directional-search erstellen
* Die Hauptdatenstruktur wird mit natürlichen Zahlen befüllt

Zweiter Schritt – Finger initiieren:

* Je nach Finger-Managementsystem (Lazy-Finger, Min-Max-Finger, Splay-Tree) werden die passenden Finger gesetzt

Dritter Schritt – Suchaufträge für Finger initiieren:

* Suchverteilung erstellen: Listen mit Zahlenfolgen erstellen
* Fingersuche auf der Datenstruktur mit den Listen ausführen

Vierter Schritt – Wissensgewinn

* Geschwindigkeitsvergleich bei unterschiedlichen Fingermanagements
* Graphische Veranschaulichung

## Klassendiagram

Im Folgenden ist ein Klassendiagram, welches die relevanten Methoden und Klassen dargestellt hat:

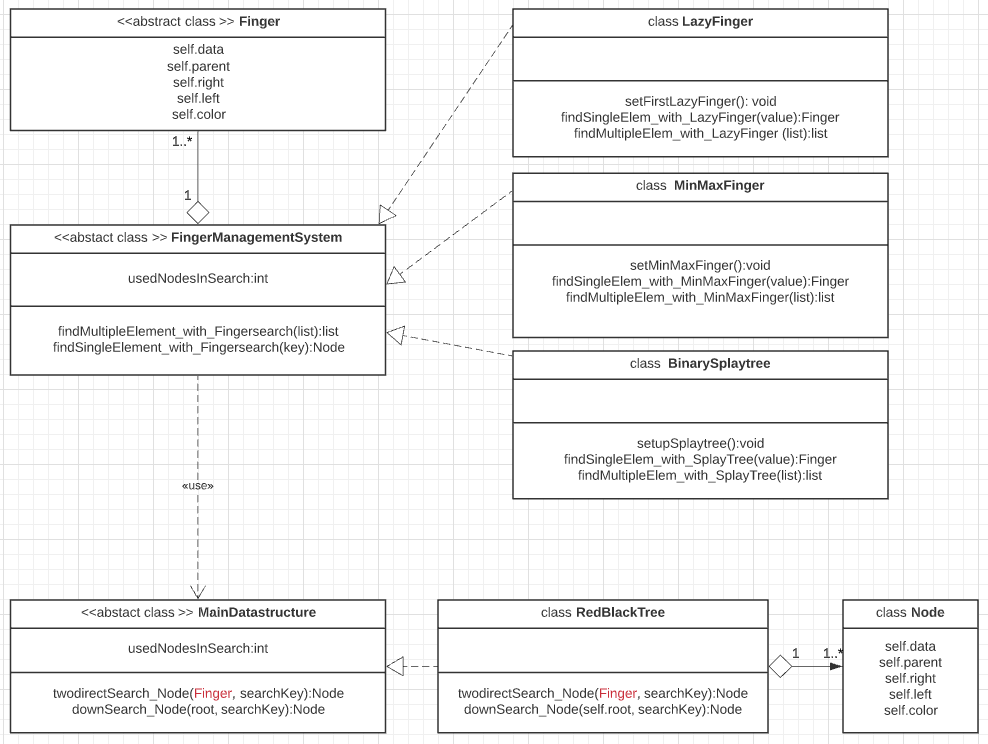


Abb.: Klassendiagram

Die interessantesten Methoden sind:

* BinarySplaytree*.findMultipleElem\_with\_SplayTree*,
* LazyFinger.*findMultipleElem\_with\_LazyFinger*
* MinMaxFinger*.findMultipleElem\_with\_MinMaxFinger*

Denn sie alle verwenden die Fingersuche und anschließend die Distanzsuche, welche sowohl hoch wie auch runter in die Subtrees suchen kann: RedBlackTree*.twoDirectSearch\_Node*. Das „twodirect“im Namen soll das bi-direktionale Suchen unterstreichen und das „*Node*“ im Namen soll hervorheben, dass ein *Node* zurückgegeben wird, denn die gleiche Methode gibt’s auch in der Version, dass nur der *Key (Node.data)* zurückgegeben wird.

Es ist auch hervorzuheben, dass jede Klasse das Attribut *usedNodesInSearch* besitzt, dieses Attribut ist einfach ein Counter, welcher hochzählt, wenn in der Klasse während der Suche das Node.data Attribut in einer Vergleichsoperationen entweder von Fingern oder von Nodes verwendet wurde („touched“).

## Ausführen vom Code

Um meinen Code auszuführen, empfehle ich die Vorgehensweise aus dem Activity-Diagramm:

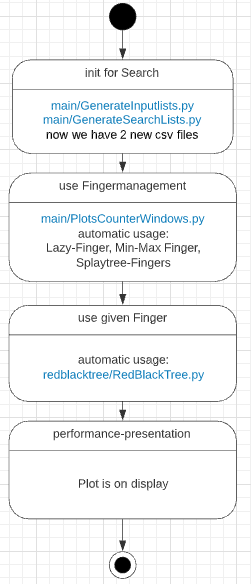
1. Bitte den Ordner *ba\_schap/main* öffnen.

Abb.: Activitydiagram für User

1. Diese 2 files auszuführen: *GenerateInputLists.py* und *GenerateSearchLists.py*, nun haben die Fingersuchen zwei .csv files für die Initialisierung.
2. Dann im selben Ordner *PlotsCounterWindows.py* auszuführen, dieses Skript verwendet die verschiedenen Arten der Fingersuche.
3. Jetzt erscheint ein Plot im neuen Fenster

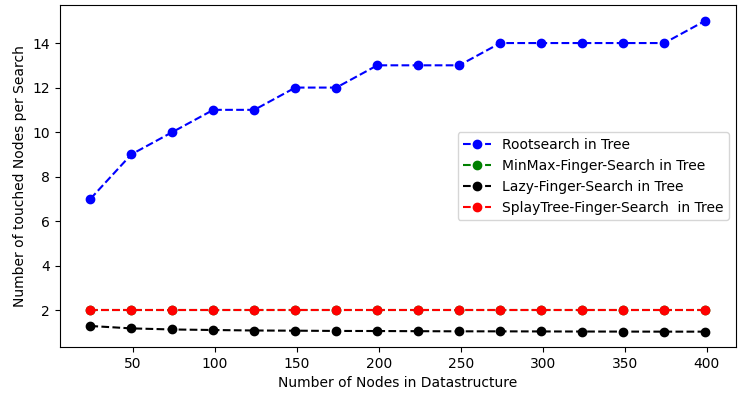
Diese Plots sind auch die gleichen verwendeten Plots in dieser Bachelorarbeit – und im nachfolgenden Kapitel diskutiert.

# Evaluation der Ergebnisse

In diesem Kapitel werden 3 verschiedene Fingersuchen und mit der Wurzelsuche als Benchmark gestartet, das wird anhand von 5 verschiedenen Suchverteilungen verglichen und analysiert.

## Verteilungen

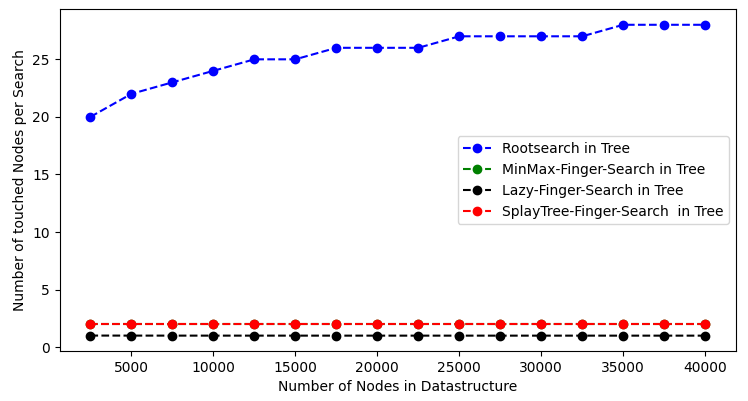
### Immer Zahl “1” suchen

* In diesem Plot ist zu sehen: x-Achse: Anzahl der Elemente im Rotschwarz Baum
* y-Achse: Diese Summe besteht aus Anzahl der besuchten Nodes während der Fingersuche und der Distanzsuche. Im Diagramm sind die Anzahl der durchschnittlich Durchsuchten Nodes für einen Suchauftrag dargestellt.

12 Abb.: Minimum Search – Anzahl Nodes pro eine Suche

Die MinMax-Fingersuchkurve ist hinter der roten kurve des Splaytrees verborgen. Es wurde jeweils immer so viele Anfragen wie Nodes in der Hauptdatenstruktur als 1‘er – was auch das Minimum ist – gestellt. Anschließend die Anzahl der touched Nodes durch die Anzahl der Suchanfragen geteilt.

So kann man auf einen Blick erkennen, ohne dass die Größe der Datenstruktur zu exponentiellem Wachstum führt, welcher Algorithmus pro Suchauftrag wie viele Nodes verwendet. Aus dieser Kennzahl kann man ablesen, welcher Algorithmus der schnellste oder auch der langsamste ist. Der zweite Plot zeigt den gleichen Versuch bloß mit größerer Wertebereich:



13 Abb.: Minimum Search – Anzahl Nodes pro eine Suche (größerer Wertebereich)

### Immer Maximum Suchen

* In diesem Plot ist zu sehen:
* x-Achse Anzahl der Elemente im Rotschwarz Baum
* y-Achse: Diese Summe besteht aus Anzahl der besuchten Nodes während der Fingersuche und der Distanzsuche. Im Diagramm ist die durchschnittliche Anzahl der durchsuchten Nodes pro Suchauftrag dargestellt:

Es wurde jeweils immer so viele Anfragen wie Nodes in der Hauptdatenstruktur gestellt. Die Diagramme haben verschiedene Wertebereiche, um das Verhalten der Fingersuchen im größeren Maßstab einschätzen zu können.

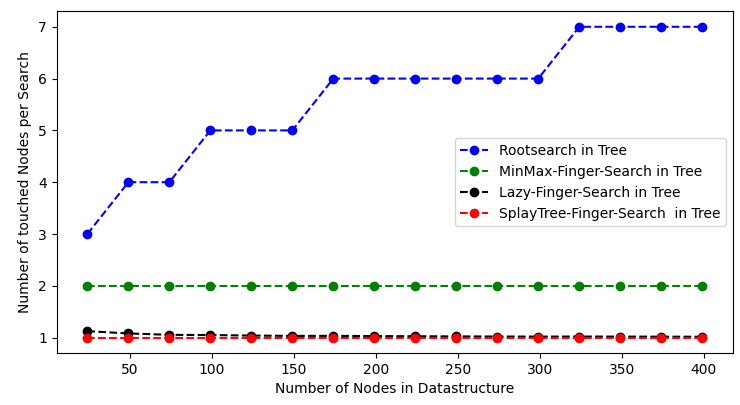
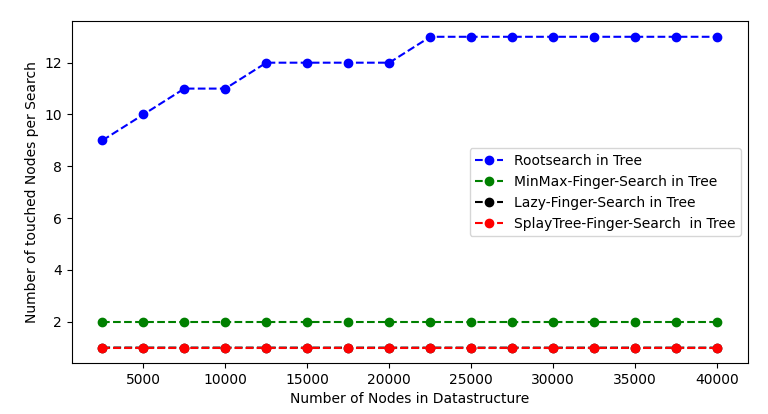
Man kann wieder gut erkennen, dass jede Suche schneller ist als die Wurzelsuche. Die Lazy-Fingersuche und die Splay-Tree Suche sind wieder zusammengefallen. Diese beiden Fingersuchen haben 1 bis 2 -Vergleiche gebraucht, um die gesuchte Node zu finden.

Abb.: Maximum Search – Anzahl Nodes pro eine Suche

Das folgende Diagramm stellt den gleichen Suchauftrag für große Wertebereiche da:

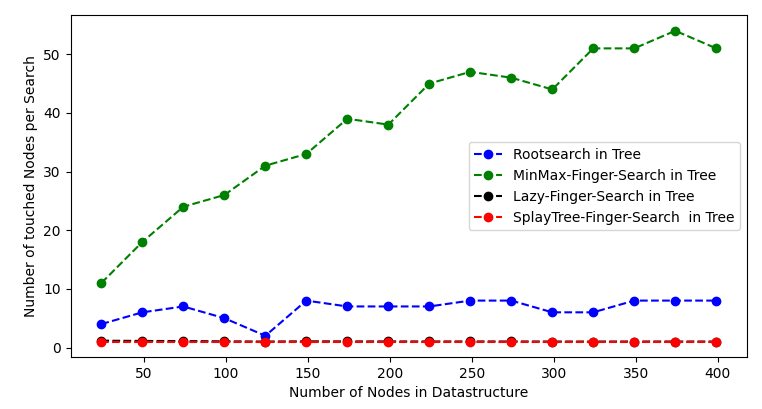
15 Abb.: Maximum Search – Anzahl Nodes pro eine Suche (größerer Wertebereich)

### Immer das mittelste Element suchen

In diesem Plot ist zu sehen:

* x-Achse Anzahl der Elemente im Rotschwarz Baum
* y-Achse Anzahl der Nodes welche bei der Suche angefasst wurden. Diese Zahl ist die Summe der Fingersuche und der Suche in der Hauptdatenstruktur

Es wurden jeweils immer so viele Anfragen wie Nodes in der Hauptdatenstruktur gestellt mit immer dem mittelsten Key Element – von dem ich angenommen habe, dass es Wurzel nah in der Hauptdatenstruktur ist:



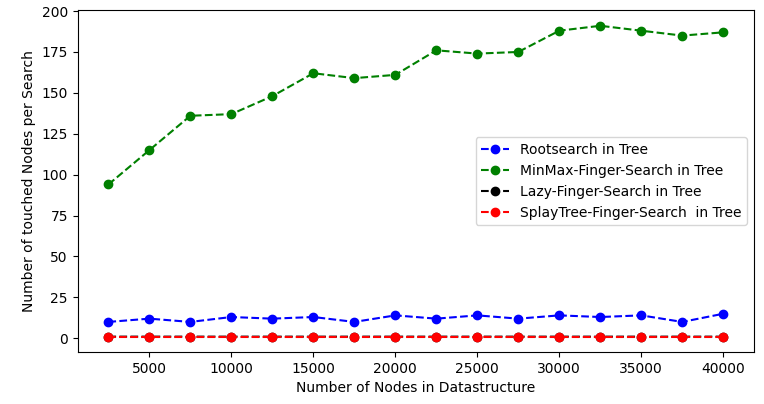
16 Abb.: Middle Key Search - Anzahl Nodes pro Suche

Hier kann man wieder die gleiche Formel verwenden: und rausfinden, dass z.B. für den letzten Wert für den Splay Tree gilt: 200/200 = 1, also ein Vergleich notwendig war, um das Ergebnis zu liefern. Im Gegensatz zur Wurzelsuche: 1592/200 = 7,96 Vergleiche benötig wurden.

Ich hätte hier vermutet, dass die Wurzelsuche sich gleich gut verhält, wie der Splay Tree und der Lazy-Finger - und nicht wie dargestellt. Typischerweise wäre das mittelste Element naher der Wurzel bei balancierten Bäumen.

Da bei der Wurzelsuche die Anzahl der Vergleiche so viele sind, wie man zu einem beliebigen Leaf brächte: = = 7,64 dann könnte man Schlussfolgern, dass das mittelste Element eher auf Leafhöhe anzutreffen wäre als nahe der Wurzel. Daraus ergibt sich der Verdacht, dass der implementierte Rotschwarzbaum nicht perfekt balanciert ist.

Die Diagramme haben verschiedene Wertebereiche, um das Verhalten der Fingersuchen im größeren Maßstab einschätzen zu können:

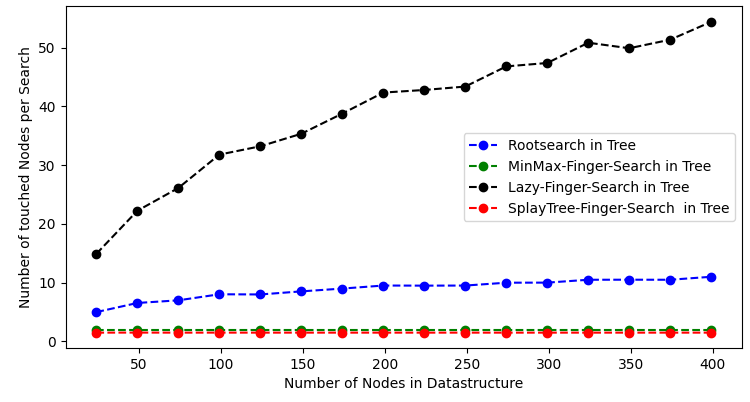


17 Abb.: Middle Key Search - Anzahl Nodes pro Suche (größerer Wertebereich)

### Immer Zick-Zack Suchen

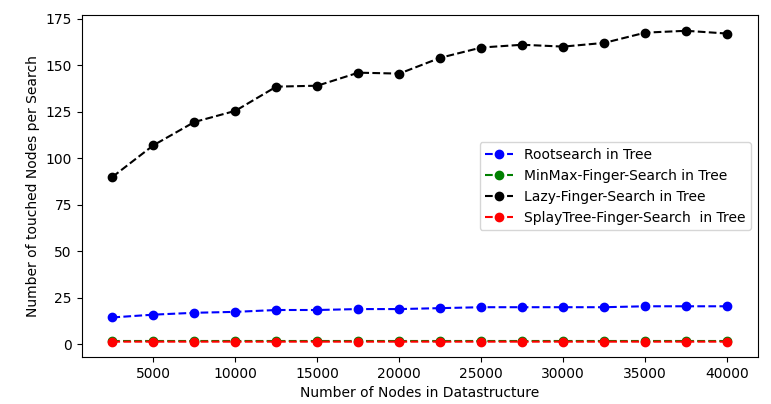
In diesem Plot ist zu sehen:

* x-Achse Anzahl der Elemente im Rotschwarz Baum
* y-Achse Anzahl der Nodes welche bei der Suche angefasst wurden. Diese Zahl ist die Summe der Fingersuche und der Suche in der Hauptdatenstruktur
* Es wurde jeweils immer so viele Anfragen wie Nodes in der Hauptdatenstruktur als Zick-Zack-Muster gestellt, also Minium und Maximum alternierend gesucht:

 Abb.: Alternierend Minimum and Maximum - Anzahl Nodes pro Suche

Im Plot ist gut zu erkennen, dass die Min-Max-Suche sich besonders gut verhält, im Vergleich zur Wurzelsuche. Auch ist gut zu erkennen, dass der Lazy-Finger einen besonders langen Weg zurücklegen muss, um das Ergebnis zu finden – das leite ich aus der Anzahl der tangierten Nodes auf dem Weg der Suche ab. Und auch wenn man sich das Beispiel aus der Einleitung anschaut, wo dieser Fall als besonders ungünstig erscheint, konnte das mit diesem Plot gezeigt werden.

Die Diagramme haben verschiedene Wertebereiche, um das Verhalten der Fingersuchen im größeren Maßstab einschätzen zu können:

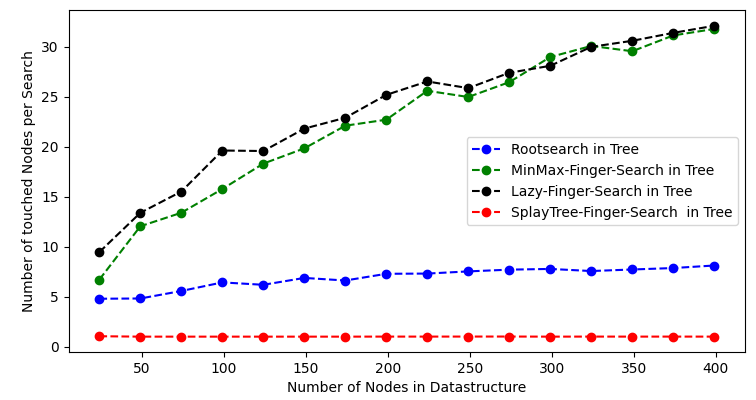


19 Abb.: Alternierend Minimum and Maximum - Anzahl Nodes pro Suche (größerer Wertebereich)

### Random zwischen Minimalen und Maximalem Element

In diesem Plot ist zu sehen:

* x-Achse Anzahl der Elemente im Rotschwarz Baum
* y-Achse Anzahl der Nodes welche bei der Suche angefasst wurden. Diese Zahl ist die Summe der Fingersuche und der Suche in der Hauptdatenstruktur
* Es wurde jeweils immer so viele Anfragen wie Nodes in der Hauptdatenstruktur gestellt.
* Das Suchmuster sind randomisierte Keys zwischen dem Minimalen und dem Maximalen Element.
* Es wurden wieder so viele Anzahl an Anfragen wie Nodes in der Datenstruktur gestellt:



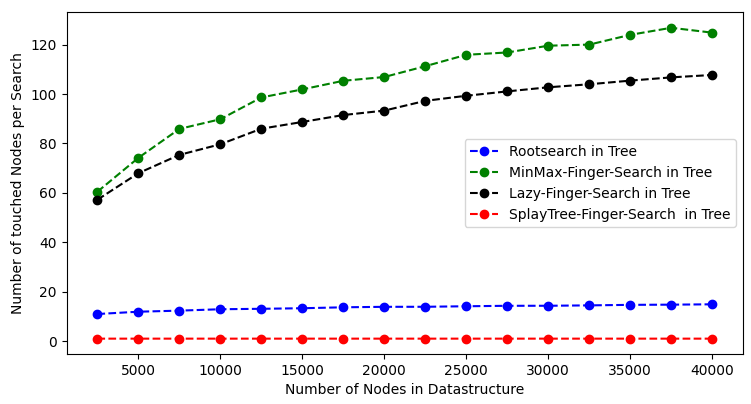
20 Abb.: Random Key Search - Anzahl Nodes pro Suche

Laut meiner Literaturrecherche wäre genau das der ungünstigste Fall für den Splay Tree, da der Zufall den Splay Tree nicht balanciert erscheinen lässt und damit die Such-Wege eigentlich gleich oder ein bisschen größer werden können als in der Hauptdatenstruktur. Dieses Verhalten kann ich in meiner Simulation leider nicht zeigen.

Jetzt würde ich mich fragen, ob ich den Splay-tree an irgendeiner Stelle falsch implementiert habe, oder ich könnte mich fragen ob vielleicht der Splay-Tree sich tatsächlich so verhält. Ich kann die Frage leider weder in die eine noch in die andere Richtung beantworten.

Interessant ist auch zu sehen, dass der Lazy-Finger und auch die Min-Max-Suche eine ähnliche Anzahl an Vergleichen tätigen - das ist auch überraschend, denn es passt auch nicht zu meiner Recherche. Es ist jedoch gut zu erkennen, dass die Wurzelsuche schneller ist als die Min-Max und die Lazy-Fingersuche.

Die Diagramme haben verschiedene Wertebereiche, um das Verhalten der Fingersuchen im größeren Maßstab einschätzen zu können:



21 Abb.: Abb.: Random Key Search - Anzahl Nodes pro Suche (größerer Maßstab)

## Zusammenfassung: Direktvergleich der Graphen

In diesem Kapitel möchte ich einen Überblick geben, wie sich die von mir programmierten Algorithmen je nach Suchanfragen verhalten. Genauere Details sind in den Kapiteln vorher beschrieben.

Es ist gut zu erkennen, dass je nach Fingersuche die Kurven unterschiedlich stark steigen. Dafür gibt es im Folgenden eine Diagrammübersicht, wo die gleiche Suche pro Suchauftrag dargestellt ist:

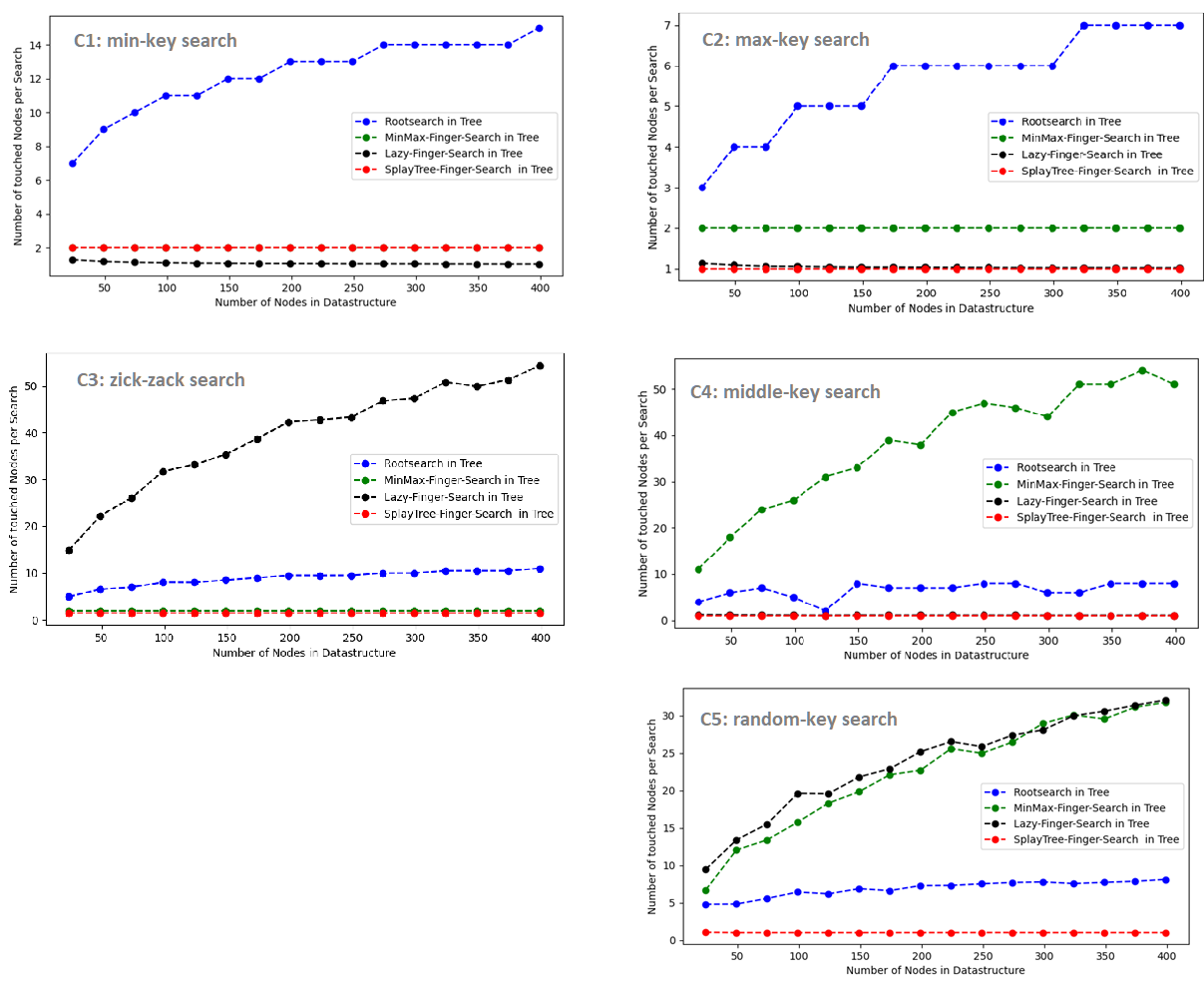


Abb.: Überblick Anzahl der Nodes pro eine Suche

Man könnte allgemein sagen, je höher und steiler die Kurve ist, desto schlechter verhält sich der Algorithmus im jeweiligen Anwendungsfall.

Man kann allerdings auch gut erkennen, dass sich die Fingersuche mit der Zunahme der Größe der Datenstruktur logarithmisch verhält.

Zusammenfassend kann man sagen, dass in den Suchanfragen ein Einfluss auf die Schnelligkeit der unterschiedlichen Fingersuchen gezeigt werden konnte.

# Einordnung der Ergebnisse Anhand des Forschungsstandes

Ich konnte mit meinem Versuchen zeigen, dass in Abhängigkeit der Konfiguration der Suchanfragen, sich die Fingersuchen entweder konstant oder logarithmisch verhalten.

## Trade-Off für Anzahl der Finger

Das ist in meinen Augen der beste Trade-off: Es geht nicht darum optimale Anzahl von Finger und deren optimale Positionen zu ermitteln, sondern es geht darum einen absoluten Vorteil durch Fingernutzung zu erzielen, z.B. beim Splay-Tree hätte man a-viele Finger gespeichert - statt n-viele. Im Prinzip ist mein Lösungsvorschlag deshalb die Fingermanagement-Datenstruktur ab einer gewissen Größe und Höhe einfach „zu kürzen“.

Als Benchmark für den absoluten Vorteil werden die Werte der Wurzelsuche genommen.

Mit den Plots konnte allerdings auch gezeigt werden, dass wenn man nur 1 oder 2 Finger hat – also die Fingermanagementdatenstruktur sehr stark kürzt – unter ungünstigen Suchanfragen sich diese Fingersuche langsam verhält. Umgekehrt gibt es auch Suchanfragen wo die Fingersuche deutlich besser als die Wurzelsuche performiert. Es kommt also stark auf den Anwendungsfall an, wenn man besonders wenige Finger zur Verfügung hat.

Mit dem Splaytree konnte jedoch gezeigt werden, dass man einen Vorteil über alle vorgestellten Anwendungsfälle hinweg hat. Für den Splaytree ist auch die Trade-off-Betrachtung deutlich einfacher, da er zuerst mit n-vielen Nodes starten könnte und anschließend auf a viele „gekürzt“ werden könnte.

Damit würde der Splaytree für die Fingersuche immer einen absoluten Vorteil gegenüber der Wurzelsuche garantieren – auch bei ungünstigen Suchanfragen.

## Informationen nutzen

Im Kapitel „Evaluation der Ergebnisse“ konnte gezeigt werden, dass sich die verschieden Fingersuchen je nach Suchanfragen im Vergleich zu der Wurzelsuche besser oder schlechter verhalten.

Dass die vorgestellten Fingersuchen sich die Information über das Wesen der Suchanfragen merken und für die Zukunft zunutze machen können, konnte durch die Implementierung bestätigt werden, auch konnte das ungünstige Verhalten einiger Fingersuchen, wie in der Einleitung dieser Arbeit diskutiert, nachgewiesen werden.

Meine Plots bestätigen außerdem die Sicht aus der Informationstheorie, dass es einen großen Einfluss auf die Fingersuche hat, wie chaotisch die Suchanfragen sind (siehe Kapitel “Random zwischen Minimalen und Maximalem Element“). Hier konnte gezeigt werden, dass mit chaotischen Anfragen der Lazy-Finger und die Min-Max-Fingersuche ähnlich schlecht performiern – im Vergleich zum Benchmark. Das ist erstaunlich da der Lazy-Finger dynamisch und die Min-Max-Fingersuche - wie auch die Wurzelsuche- statisch sind.

Eine der interessanteren Fingermanagement-Datenstrukturen ist der Splaytree - jedoch entgegen der Theorie, dass chaotische Suchanfragen den Splaytree besonders negativ beeinflussen (C5: random-key search), war durch die Implementierung zusehen, dass er sehr performant ist, sogar schneller als die Wurzelsuche.

Dieses ermittelte Ergebnis lässt mehrere Interpretationen zu. Einmal, dass meine Implementation des Splaytrees fehlerhaft ist oder dass die random-funktion in Python nicht chaotisch genug ist, um das ungünstige Verhalten zu verursachen oder vielleicht ist tatsächlich der Splaytree ein besonders schneller Algorithmus, um die Fingersuche zu implementieren.

Ausgehend von der letzten Vermutung, dass der Splaytree auch bei chaotischen Anfragen besonders gut performiert, könnte man sagen, dass diese Datenstruktur die Informationen über das Wesen der Suchanfragen besonders effizient nutzt, um die zukünftigen Anfragen besser als alle anderen vorgestellten Algorithmen verarbeiten zu können.

Kombiniert mit der Betrachtung aus dem Vorherigen Kapitel „Trade-Off für Anzahl der Finger“ konnte ein Fingersuch-Algorithmus gefunden werden, welcher selbst im Worstcase einen absoluten Vorteil gegenüber der herkömmlichen Wurzelsuche bietet : der „gekürzte“ Splaytree.

## Historische Einordnung der Sicht auf die Finger

Es gibt einige modernere Arbeiten, die von den Vorteilen der Splay-Tree für gewisse Anwendungen berichten (Kapitel „Aktueller Forschungsstand: Übersicht über Finger“). In diesen Anwendungen werden Splay-Tree gleichgesetzt mit der Fingersuche – was meiner Meinung nach zu einschränkend für die Fingersuche und auch für den Splay-Tree ist.

In meinem kleinen Modell – aus der Softwaretechnik inspiriert – wird für jede funktionale Anforderung ein einzelnes Objekt geschaffen. Es spielt meiner Meinung nach eine Rolle, gerade bei Anwendungen der Fingersuche, dass man die Verwaltung der Finger und die Teilabschnitte der Suche klar von einender trennt. Denn nur durch die Trennung kann man einzelne Schritte optimierter gestalten und neu zusammensetzen – vor allem wenn man aus dem Gebiet der Finger nur die Fingersuche verwenden möchte.

Ich sehe aber gleichzeitig, dass in älteren Arbeiten bis in die 80er Jahre diese Trennung nicht vorgenommen wurde und stattdessen hat man einen universellen Vorteil durch die Finger z.B. auch für Baumrotationen, Tree-Joins und andere aufwändige Baumoperationen versucht mit zu nutzen.

Die neueren Arbeiten zu dem Thema hingegen versuchen sich ausschließlich auf die Suche zu konzentrieren. Diese Arbeit reiht sich somit in die modernere Sicht auf die Finger ein. Auch zeigt ein Kapitel dieser Arbeit „Prädiktion auf Input Streams für Lokalitätsprinzip“, dass es eine Sensibilisierung für das Thema der Prädiktion der Suchanfragen gibt und dass daran gearbeitet wird es auszunutzen.

## Fazit

Zusammenfassend konnte ich mit der Modellierung zeigen, dass einige Fingersuchalgorithmen geeignete Anwendungsfälle in sich vereinen.

Auch konnte ein Fingersuchalgorithmus gefunden werden, welcher immer einen absoluten Vorteil gegenüber der Wurzelsuche bietet: der „gekürzte“ Splaytree.

# Zusammenfassung

In dieser Arbeit wurde der aktuelle Forschungsstand betrachtet, es wurde ein Modell vorgestellt, um sich der Fingersuche zu nähern, außerdem wurde auch theoretisch sich mit dem Thema der Laufzeiten auseinandergesetzt und zum Ende dieser Arbeit wurden einige Anwendungsfälle mithilfe des Modells programmiert. Es gab auch eine kurze historische Trend-Einordnung der Sicht auf die Finger.

Es konnten einige Vermutungen, welche am Anfang dieser Arbeit getätigt wurden, belegt werden. Auch ein überraschendes Ergebnis, bezüglich der Random Key Search, konnte veranschaulicht werden – nämlich, dass der Splaytree sich bei zufälligen Suchanfragen besser als die Wurzelsuche verhält.

Es wurde die Frage beantwortet, dass der beste Trade-off für die Anzahl der Finger darin liegt die Fingermanagement-Datenstruktur zu „kürzen“. Der beste Trade-off wird durch die Höhe der Startpunkte in der Distanzsuche beschränkt.

Es konnte gezeigt werden, dass einige Finger-Algorithmen die Information über die Struktur der vorhergegangenen Suchaufträge in sich speichern und für die zukünftige Suchaufträge nutzen.

Es wurde der „gekürzte“ Splaytree als ein Algorithmus, welcher einen absoluten Vorteil gegenüber der Wurzelsuche bietet, aus dem Modell abgeleitet.

# Appendix

## Ausblick und Potential

### Fingersuche und Datenbanken

Ich würde gerne an dieser Stelle erwähnen, dass die Fingersuche, so wie sie in meinem Modell vorgestellt ist, eigentlich eine einfache Anwendung ist, Datenstrukturen egal welcher Art - auch die unsortierten oder unstrukturierten - nachzurüsten. Man hätte eine externe Fingerdatenstruktur, die man nur anzubinden braucht und schon hat man eine mehr oder minder starke Beschleunigung bei oft durchsuchten Bereichen z.B. in Datenbanken. Wie in meinen Plots gezeigt, muss natürlich der Anwendungsfall vorteilhaft abgestimmt werden.

Um sich der Frage zu nähern, ob die Fingersuche konzeptuell geeignet ist, um Datenbanken aufzurüsten, kann man sich ansehen, welche Arten der Datenbanken aktuell [17] am häufigsten genutzt werden:

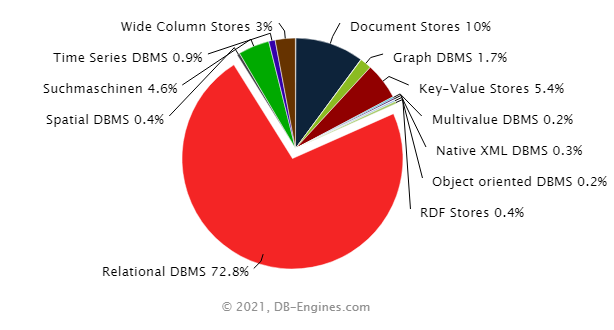
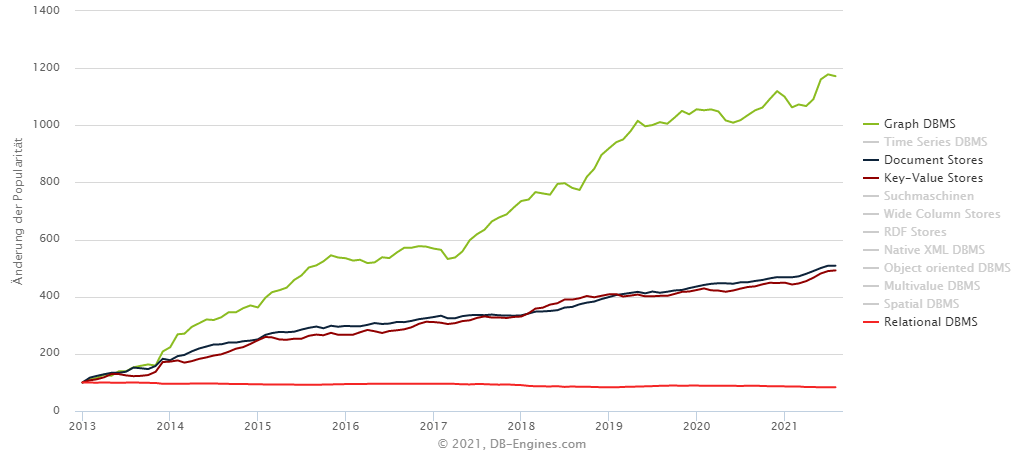


Abb.: häufigste Datenbank-Kategorien

Hier sieht man schnell, dass die relationale-Datenbanken besonders gerne genutzt werden. In diesem Datenbank Modell werden Daten nach dem Relationalen Modell in Tabellen gespeichert und jede Zeile erhält einen einzigartigen Key. Mit diesem Key könnte man eine Fingersuche implementieren.

Wenn man beurteilen möchte, ob die Fingersuche auch in der Zukunft potenzial hätte, kann man sich den größten Zuwachs an Popularität von verschiedenen Datenbank-Modellen ansehen:



24 Abb.: Die vier beliebtesten Datenbankmodelle: Absolute Anzahl seit Januar 2013

Man kann aus dem Diagramm8‑2ablesen, dass die Beliebtheit der Relationalen Datenbanken sich kaum verändert hat - das könnte bedeuten, dass diese Datenbanken weiterhin im Einsatz sind und jedes Jahr weiter mit Daten befüllt werden.

Man kann ebenfalls gut erkennen, dass die Nutzung der graphenbasierten-Datenbanken in den letzten 9 Jahren besonders stark zugenommen hat. Laut dieser Quelle [18] werden Daten in Nodes mit „eindeutig bezeichneten und identifizierbaren Datenentitäten“ und Kanten gespeichert. Solche Nodes könnten demnach einen unique Identifier oder Key generiert bekommen und damit wäre die Fingersuche ebenfalls nutzbar.

Aus dieser Analyse würde ich Schlussfolgern, dass die Fingersuche ein hervorragendes Konzept ist, um es auf den beliebtesten und zukunftsträchtigsten Datenbankmodellen anzuwenden.

### Finger und Hardwaretechnology

Angesichts des Trends [19] weg von General Purpose Technology hin zu spezialisierter Hardware für spezielle Anwendungen könnte man sich überlegen, ob das Finger-Managementsystem nicht eine spezialisierte Hardware nur für den jeweiligen Algorithmus bräuchte. Man könnte das Fingermanagementsystem auf GPU, CPU laufen lassen oder sogar Quantencomputern eine Chance geben.

Z.B. laufen Splay-Trees besonders gut auf GPU’s - da GPU’s besonders viele parallele Vergleiche ausführen können. Deshalb liegt es nahe, den ersten Teil der Fingersuche speziell auf diese Art der Hardware auszulagern und den zweiten Teil der Fingersuche, die Distanzsuche, auf den meist speichernahen General Purpose Technologies z.B. CPU auszuführen.

Der Grover Algorithmus arbeitet mit der Quantentechnologie auf unsortierten Datenstrukturen, wie im Kapitel “Grover Algorithmus – Quantencomputer“ beschrieben. Man könnte einfach Finger in eine Liste speichern und darauf den Grover Algorithmus laufen lassen, so könnte man eine Fingersuche mit Quantencomputern realisieren.

#### Skipliste

Ein weiterer Interessanter Fall wäre eine Skipliste, welche durch ein Splay Tree optimaler durchsucht werden kann.

Ein Grund für die Nutzung von Splay-Tree aus Skipliste wäre, dass die Skipliste sich niemals neu organisieren müsste und die Fingersuche sich selbst optimierend verhält – eine praktische wartungsarme Konstruktion. Auch, dass nur die Fingersuche im Cache zu verorten wäre und die eigentliche Skipliste z.B. im Speicher liegt, ist ein Vorteil für besonders große Datenstrukturen.

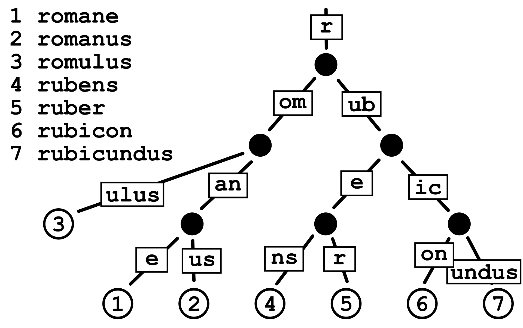
## Grenzen der Fingersuche

Dieses Kapitel widmet sich den Grenzen der Fingersuche und bei welchen Datenstrukturen es meines Erachtens keinen Mehrwert bieten würde. Z.B. bei einem Stack oder einem Radix tree / Präfix-Tree.

### Stack

An sich hat der Stack schon eine Prädiktion, wie er verwendet wird, nämlich first-in first-out. Man könne sogar sagen, dass der oberste Stack-Pointer schon ein Finger ist. Mehr Finger und auch eine Fingersuche scheint Zuviel für so eine minimalistische Datenstruktur.

### Radix-Tree / Präfix-Tree

Der Präfix-Tree ist, wie der Name schon andeutet, ein Tree der sich den Pfad durch die Datenstruktur zunutze macht [20]. Hier einen Finger mitten in der Datenstruktur suchen zulassen, wäre kontraproduktiv - da man den Pfadanfang mit vergleichen muss.

Im Prinzip würde der Finger den übergebenen Präfix abgleichen, was einer Menge von Vergleichsoperationen stark ähnelt, wie dem durchlaufen von der Wurzel aus.

25 Abb: Radix, Prefix tree or Trie

# Begriffe

## Lokalitätsprinzip

In der Informatik ist das Lokalitätsprinzip die Tendenz eines Prozessors, über einen kurzen Zeitraum wiederholt auf denselben Satz von Speicherplätzen zuzugreifen. Es gibt zwei grundlegende Arten von Referenzlokalitäten – die zeitliche und die räumliche Lokalität. [21]

Lokalität ist eine Art von vorhersagbarem Verhalten, das in Computersystemen auftritt. Systeme, die dem Lokalitätsprinzip folgen, sind großartige Kandidaten für die Leistungsoptimierung durch die Verwendung von Techniken wie Caching, Pre-fetching für Speicher und fortschrittliche Verzweigungsprädiktoren in der Pipeline-Phase eines Prozessorkerns. [21]

Besonders genutzt wird in dieser Arbeit die räumliche Lokalität: Wenn ein bestimmter Speicherort zu einem bestimmten Zeitpunkt referenziert wird, ist es wahrscheinlich, dass in naher Zukunft nahegelegene Speicherorte referenziert werden. [21]

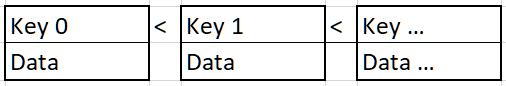
In diesem Fall wird häufig versucht, die Größe und Form des Bereichs um die aktuelle Referenz herum zu erraten, für den es sich lohnt, einen schnelleren Zugriff für eine spätere Referenz vorzubereiten. [21]

In dieser Arbeit werden für die Referenzen Finger eingesetzt und das Lokalitätsprinzip sollte sollte auf abstrakten Datenstrukturen gelten.

## 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. [21] Daten und Values werden in dieser Arbeit analog sowohl in der schriftlichen Ausarbeitung als auch im Code für die Simulation verwendet.

In meiner Arbeit wird vor allem die Ordinale Eigenschaft der Daten/Values vorausgesetzt. Ordinal bedeutet, dass man die Daten ordnen oder einer Reihenfolge zuweisen kann. In dieser Arbeit haben Datenwerte/Values deshalb 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 Verhältnis größer, gleich oder kleiner zueinander zustellen. [22] [23]



Die Kombination aus Datenwerten/Values mit Keys heißen Nodes. Nodes können in einer Liste wie in der oberen Abbildung angeordnet werden.

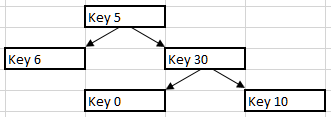
## Datenstruktur

„Wie für viele fundamentale Begriffe der Informatik gibt es auch für […] Algorithmen und Datenstrukturen, nicht eine einzige, scharfe, allgemein akzeptierte Definition“ [24, p. 1]. Daher halte ich mich an 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.“ [25]

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

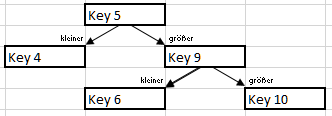
## Baum

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. [26]



## Binärer 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.

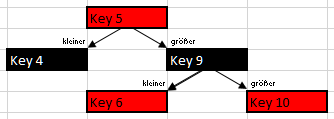


Die Binärsuche startet in der Wurzel – dem obersten Knoten - und folgt den Pointern zu den Kindern, den untergeordneten Knoten. Um diese Suche von der Fingersuche zu unterscheiden, nenne ich diese Suche Wurzelsuche.

Die Wurzelsuche hat eine Worstcase Laufzeit von O (log n), wobei n die Anzahl der Knoten ist. In meiner Implementierung haben die Funktionen Insert und Delete eine Worstcase 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 Datenstruktur hat den Vorteil, dass es einen Algorithmus der Balancierung gibt, welcher die zusätzliche Information der unterschiedlichen Knotenfarben ausnutzt und deshalb schneller ist. [27]



## Finger

Finger sind Pointer oder Referenzen. In dieser Arbeit speichert jeder Finger eine Node in sich selbst – diese Node kann dann als Start-Position oder Start-Node für eine Suche genutzt werden.

Finger haben meistens die Besonderheit, dass sie extern auf die jeweilige Datenstruktur zeigen und sie durchsuchen können.

## Splay-Tree

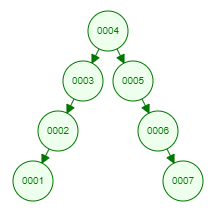
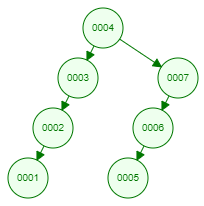
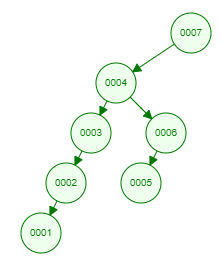
Im Originalpaper [28] wird der Splaytree als “eine sich selbst anpassende Form des binären Suchbaums“ benannt. „Die Effizienz von Splaytrees kommt [..] von der Anwendung einer einfachen Umstrukturierungsheuristik, genannt *splaying* wenn auf den Baum zugegriffen wird“. Die Splay-Operation positioniert das zuletzt gesuchte Element in die Wurzel des Baumes, wie in den unteren Abbildungen dargestellt, ist einmal der Splaytree vor der Suche nach der 7 und nach der Suche nach der 7 und anschließend nach der Suche nach der 4 dargestellt. Man kann erkennen, dass die Splay-Operation das jeweilige Suchergebnis in die Wurzel gestellt hat. [29]

Abb.: Nachdem 4 gefunden wurde

Abb.: Aktueller Splaytree

Abb.: Nachdem 7 gefunden wurde

Auf einem n-Node Splay-Tree haben alle Standardbaumoperationen eine amortisierte Laufzeitklasse von O(log n) pro Operation, wobei mit "amortisierte Zeit" pro Operation gemeint ist auch gemittelt über alle Worstcase-Sequenzen von Operationen. [28]

Daher sind Splaytrees genauso effizient wie balancierte Bäume, wenn die amortisierte Laufzeit von Interesse ist. Außerdem sind Splaytrees für ausreichend lange Zugriffsfolgen bis auf einen konstanten Faktor genauso effizient wie statische optimale binäre Suchbäume. [28]

# Literaturverzeichnis

|  |  |
| --- | --- |
| [1] | Brodal, Gerth Stølting by CRC Press, LLC , „Finger Search Trees,“ 2005. [Online]. Available: https://www.cs.au.dk/~gerth/papers/finger05.pdf. [Zugriff am 12. 08. 2020 (16:40 Uhr)]. |
| [2] | Brodal, Gerth Stølting by CRC Press, LLC, „Finger Search Trees,“ 2005. [Online]. Available: https://www.cs.au.dk/~gerth/papers/finger05.pdf. [Zugriff am 15. 07. 2020]. |
| [3] | Hee-Kap Ahn and Chan-Su Shin, „Algorithms and Computation,“ 25th International Symposium, ISAAC 2014, Jeonju, Korea, December 15-17, 2014, 12 2014. [Online]. Available: https://link.springer.com/content/pdf/10.1007%2F978-3-319-13075-0.pdf. [Zugriff am 12. 08. 2021 (16:53 Uhr)]. |
| [4] | R. Seidel and C. Aragon, „Randomized search trees,“ 10. 1996. [Online]. Available: https://link.springer.com/content/pdf/10.1007/BF01940876.pdf. [Zugriff am 12. 08. 2021 (16:39 Uhr)]. |
| [5] | R. Seidel and C. Aragon, „Randomized search trees,“ 10 1996. [Online]. Available: https://link.springer.com/content/pdf/10.1007/BF01940876.pdf. [Zugriff am 12. 08. 2021 (16:39 Uhr)]. |
| [6] | Leo J. Guibas, Edward M. McCreight, Michael F. Plass, Janet R. Roberts, „A NEW REPRESENTATION FOR LINEAR LISTS,“ 1977. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=5A00C23223BF564B3107486113A2D1BE?doi=10.1.1.527.7294&rep=rep1&type=pdf. [Zugriff am 12. 08. 2021 (16:48 Uhr)]. |
| [7] | Mehlhorn, Scott Huddlestonl and Kurt, „A New Data Structure for Representing Sorted Lists\*,“ 1982. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.217.5131&rep=rep1&type=pdf. [Zugriff am 12. 08. 2021 (16:50 Uhr)]. |
| [8] | Hee-Kap Ahn and Chan-Su Shin, „Algorithms and Computation,“ 25th International Symposium, ISAAC 2014, Jeonju, Korea, December 15-17, 2014, 12 2014. [Online]. Available: https://link.springer.com/content/pdf/10.1007%2F978-3-319-13075-0.pdf. [Zugriff am 12. 08. 2021 (16:51 uhr)]. |
| [9] | Chalermsook, Goswami, Kozma, Mehlhorn and Saranurak, „Multi-Finger Binary Search Trees,“ 29th International Symposium on Algorithms and Computation (ISAAC 2018)., 2018. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2018/10003/pdf/LIPIcs-ISAAC-2018-55.pdf. [Zugriff am 12. 08. 2021 (16:43 Uhr)]. |
| [10] | Wikipedia, „Splay\_tree,“ [Online]. Available: https://en.wikipedia.org/wiki/Splay\_tree . [Zugriff am 03. 06. 2021 (13:50 Uhr)]. |
| [11] | Wikipedia , „Splay-Baum,“ [Online]. Available: https://de.wikipedia.org/wiki/Splay-Baum. [Zugriff am 03. 06. 2021 (14:00 Uhr)]. |
| [12] | Leo J. Guibas, Edward M. McCreight, Michael F. Plass, Janet R. Roberts, „A NEW REPRESENTATION FOR LINEAR LISTS,“ 1977. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=5A00C23223BF564B3107486113A2D1BE?doi=10.1.1.527.7294&rep=rep1&type=pdf. [Zugriff am 12. 08. 2021 (16:48 Uhr)]. |
| [13] | Chalermsook, Goswami, Kozma, Mehlhorn and Saranurak, „Multi-Finger Binary Search Trees,“ 29th International Symposium on Algorithms and Computation (ISAAC 2018)., 2018. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2018/10003/pdf/LIPIcs-ISAAC-2018-55.pdf. [Zugriff am 12. 08. 2021 (16:43 Uhr)]. |
| [14] | Horizonte, Belo, „CONCURRENT SELF-ADJUSTING DISTRIBUTED TREE NETWORKS,“ 05 2017. [Online]. Available: https://arxiv.org/pdf/1705.09555.pdf. [Zugriff am 12. 08. 2021 (16:47 Uhr)]. |
| [15] | Figgatt, Maslov, Landsman, Linke, Debnath and Monroe, „Complete 3-Qubit Grover Search on a Programmable Quantum Computer,“ 31 03 2017. [Online]. Available: https://arxiv.org/pdf/1703.10535.pdf. [Zugriff am 12. 08. 2021 (16:41 Uhr)]. |
| [16] | Chen Ding and Yutao Zhong, „Predicting whole-program locality through reuse distance analysis,“ PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementationJune 2003 Pages 245–257, 09. 05. 2003. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/781131.781159. [Zugriff am 12. 08. 2021 (16:58 Uhr)]. |
| [17] | db-engines.com, „DBMS Popularität pro Datenbankmodell,“ solid IT gmbh, 2021. [Online]. Available: https://db-engines.com/de/ranking\_categories. [Zugriff am 12. 08. 2021 (16:46 Uhr)]. |
| [18] | 1&1 & ionos, 07. 11. 2019. [Online]. Available: https://www.ionos.at/digitalguide/hosting/hosting-technik/graphdatenbank/. [Zugriff am 04. 08. .2021 (17:06 Uhr)]. |
| [19] | Neil Thompson and Svenja Spanuth, „The Decline of Computers as a General Purpose Technology,“ Communications of the ACM, March 2021, Vol. 64 No. 3, Pages 64-72, 10.1145/3430936, 03. 2021. [Online]. Available: https://cacm.acm.org/magazines/2021/3/250710-the-decline-of-computers-as-a-general-purpose-technology/fulltext. [Zugriff am 04. 08. 2021 (18:24 Uhr)]. |
| [20] | Wikipedia, „Radix tree, Patricia tree, Trie,“ [Online]. Available: https://en.wikipedia.org/wiki/Radix\_tree. [Zugriff am 04. 08. 2021 (21:13 Uhr)]. |
| [21] | Duden, „Daten,“ [Online]. Available: https://www.duden.de/rechtschreibung/Daten. [Zugriff am 04. 05. 2021 (14:00 Uhr)]. |
| [22] | Wikipedia, „Ordinalzahl,“ [Online]. Available: https://de.wikipedia.org/wiki/Ordinalzahl. [Zugriff am 04. 05. 2021 (14:07 Uhr)]. |
| [23] | Universität der Bundeswehr München, „Deskriptive Statistik - Ordinalskala,“ [Online]. Available: https://www.unibw.de/hum-bildungswissenschaft/professuren/swm/methodenskripte/deskriptive-statistik.pdf. [Zugriff am 04. 05. 2021 (14:11 Uhr)]. |
| [24] | „Datenstrukturen II,“ [Online]. Available: https://www.fernuni-hagen.de/mi/studium/module/pdf/Leseprobe-komplett\_01662.pdf. [Zugriff am 04. 05. 2021 (13:50 Uhr)]. |
| [25] | Wikipedia, „Datenstruktur,“ [Online]. Available: https://de.wikipedia.org/wiki/Datenstruktur. [Zugriff am 04. 05. 2021 (13:54 Uhr)]. |
| [26] | Wikipedia, „Baum (Datenstruktur),“ [Online]. Available: https://de.wikipedia.org/wiki/Baum\_(Datenstruktur). [Zugriff am 04. 05. 2021 (14:47 Uhr)]. |
| [27] | Wikipedia, „Redblack-Tree,“ [Online]. Available: https://en.wikipedia.org/wiki/Red%E2%80%93black\_tree. [Zugriff am 04. 05. 2021 (15:06 Uhr)]. |
| [28] | Daniel Dominic Sleator and Robert Endre Tarjan, „Self-adjusting binary search trees (THE SPLAY-TREE),“ Journal of the ACMVolume 32 Issue 3 July 1985 pp 652–686, 01 07 1985. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/3828.3835 . [Zugriff am 12. 08. 2021 (17:35 Uhr)]. |
| [29] | David Galles, „Splay Tree - Visualisation,“ University of San Francisco, 2011. [Online]. Available: https://www.cs.usfca.edu/~galles/visualization/SplayTree.html. [Zugriff am 12. 08. 2021 (18:01 Uhr)]. |
| [30] | Statista, „Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2025,“ [Online]. Available: https://www.statista.com/statistics/871513/worldwide-data-created/. [Zugriff am 04. 08. 2021 (15:48 Uhr)]. |

Hiermit erkläre ich eines 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)

1. Danke liebe Komillitonen [↑](#footnote-ref-1)