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?

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

Abstract

In dieser Bachelorarbeit geht es um externe Pointer, auch Finger genannt, und wie man den Suchprozess in Datenstrukturen durch sie beschleunigen kann. Die Laufzeiten der Such-Funktion können durch zusätzliche Finger in einer Datenstruktur z.B. bei Bäumen von O(log n) auf O(log d) [1] reduziert werden. Diese Art der Suche heißt Distanzsuche, da der Suchauftrag mit einer gewissen Distanz zur Wurzel startet.

Das Ziel dieser Arbeit ist eine Simulation der Fingersuche zu implementieren, in der sich drei Fingersuchen in Abhängigkeit von fünf Suchaufträgen im Vergleich zum Benchmark der Wurzelsuche messen lassen.

Es wird außerdem eine theoretische Betrachtung geboten, wie die Anzahl der Finger bei der Fingersuche begrenzt werden kann. Es wird ein „gekürzter“ Splay-Tree als möglicher Algorithmus aus dem Model geschlussfolgert, welcher die Informationen aus den vergangenen Suchen besonders gut nutzen kann.

Abstract - English

This bachelor thesis is about external pointers, also called fingers, and how to use them to speed up the search process in data structures. The runtime of the search function can be reduced from O (log n) to O (log d) [1] using extra fingers in a data structure, e.g. in trees. This type of search is called a distance search, as the search task starts a certain distance from the root.

The aim of this work is to implement a simulation of the finger search, where 3 finger searches are performed based on 5 search requests which will be compared to the benchmark root search.

A section is dedicated to theoretical considerations of how the number of fingers used for a search can be limited. A “shortened” Splay-Tree is concluded from the model as a possible algorithm, which is able to use the information from previous searches in a very effective way.

Inhaltsverzeichnis

1 Einleitung 1

1.1 Problemumfeld 1

1.2 Zielsetzung 2

2 Aktueller Forschungsstand: Übersicht über Finger 3

2.1 Ein Finger 3

2.2 Min-Max-Fingersuche 4

2.3 Finger mit Suchrichtung 5

2.4 Finger verwalten oder mit Fingern suchen 5

2.5 Lazy-Finger in binären Suchbäumen 6

2.6 Splay-Tree und Fingersuche 6

2.7 Splay-Trees, Zufall und Optimalität 6

2.8 Fingersuche und weitere Datenstrukturen 7

2.8.1 K-server Problem und Finger 8

2.8.2 SplayNets – Netzwerk Kommunikation 8

2.8.3 Grover Algorithmus – Quantencomputer 9

2.8.4 Prädiktion auf Input Streams für Lokalitätsprinzip 9

2.9 Zusammenfassung des Forschungsstandes 9

3 Modellierung des Problems 10

3.1 Das Modell 10

3.1.1 Überlegungen zum Aufbau des Finger-Management 12

3.1.2 Überlegungen zu der Hauptdatenstruktur 12

3.2 Einwände bezüglich des Modells ... 13

4 Theoretische Betrachtung ... 14

4.1 Anzahl der Finger ... 14

4.2 Laufzeitbetrachtung ... 16

4.3 Erkenntnisse für Fingerarten ... 17

4.4 Der „gekürzte“ Splay-Tree ... 17

5 Programmierung der Simulation ... 19

5.1 Vereinfachung des Modells ... 19

6 Evaluation der Ergebnisse ... 20

6.1 Verteilungen in Suchanfragen ... 20

6.1.1 Verteilung in Suchanfragen - Immer Zahl “1” ... 21

6.1.2 Verteilung in Suchanfragen - Immer Maximum ... 22

6.1.3 Verteilung in Suchanfragen - Immer mittelstes Element ... 23

6.1.4 Verteilung in Suchanfragen - Alternierendes Minimum und Maximum ... 24

6.1.5 Zufällige Verteilung zwischen Minimalen und Maximalem Element ... 26

6.2 Zusammenfassung: Direktvergleich der Diagramme ... 27

7 Einordnung der Ergebnisse anhand des Forschungsstandes ... 29

7.1 Trade-Off für Anzahl der Finger ... 29

7.2 Informationen über die Beschaffenheit der Suchanfragen nutzen ... 30

7.3 Historische Einordnung der Finger ... 31

7.4 Fazit zur Einordnung der Ergebnisse ... 32

8 Zusammenfassung der Bachelorarbeit ... 33

9 Appendix 34
9.1 Ausblick und Potential 34
9.1.1 Fingersuche und Datenbanken 34
9.1.2 Finger und Hardwaretechnologie 36
9.2 Grenzen der Fingersuche 36
9.2.1 Stack 36
9.2.2 Radix-Tree / Präfix-Tree 37
9.3 Code für die Simulation 37
9.3.1 Spezifikation des Vorgehens 37
9.3.2 Klassendiagram 38
9.3.3 Ausführen des Codes 40
9.4 Begriffe 41
9.4.1 Lokalitätsprinzip 41
9.4.2 Daten/Values 41
9.4.3 Datenstruktur 42
9.4.4 Bäume und binäre Bäume 42
9.4.5 Rot-Schwarz-Baum 43
9.4.6 Finger 44
9.4.7 Splay-Tree 44

10 Literaturverzeichnis 47

Abbildungsverzeichnis

Abbildung 2-1 Schematische Darstellung, Distanzsuche mit Startposition 3
Abbildung 2-2 Schematische Darstellung, Finger in einem Baum 3
Abbildung 2-3 Schematische Darstellung, Min-Max Finger 4
Abbildung 2-4 Schematische Darstellung, Finger und Beweglichkeit 5
Abbildung 2-5 Schematische Darstellung, Lazy-Finger 6
Abbildung 2-6 Schematische Darstellung, k-Server mit k -Fingern 8
Abbildung 3-1 links: Wurzelsuche, rechts: Fingersuche mit Finger-Management 11
Abbildung 4-1 Schematische Darstellung, Finger-Management 14
Abbildung 4-2 Abschätzung für d 16
Abbildung 6-1 Minimum Search – Anzahl Nodes pro Suche 21
Abbildung 6-2 Maximum Search – Anzahl Nodes pro Suche 22
Abbildung 6-3 Middle Key Search - Anzahl Nodes pro Suche 23
Abbildung 6-4 Alternierend Minimum and Maximum - Anzahl Nodes pro Suche 25
Abbildung 6-5 Random Key Search - Anzahl Nodes pro Suche 26
Abbildung 6-6 Überblick Anzahl der Nodes pro Suche (größerer Wertebereich) und Suchaufträgen 28
Abbildung 9-1 häufigste Datenbankkategorien [18] 34
Abbildung 9-2 Die vier beliebtesten Datenbankmodelle: Absolute Anzahl seit Januar 2013 [18] 35
Abbildung 9-3 Radix-Tree, Präfix-Tree or Trie [21] 37
Abbildung 9-4 Klassendiagram der Simulation 38
Abbildung 9-5 Aktivitätsdiagram für Nutzer 40
Abbildung 10-1 Ordinäre Eigenschaft der Key 42
Abbildung 10-2 Schematische Darstellung Baum 43
Abbildung 10-3 Schematische Darstellung binär Baum 43
Abbildung 10-4 Schematische Darstellung Rot-Schwarz-Baum 44

1 Einleitung

Alle Datenstrukturen implementieren typischerweise immer die gleichen Operationen Search, Insert und Delete. Der Fokus dieser Arbeit liegt darin, die Search-Funktion mit externen Pointern, sogenannten Fingern, auszustatten und zu untersuchen, 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) reduziert [1]. Viele theoretische Ansätze versuchen diese Variable d möglichst klein zu halten und nennen diese Art der Suche Distanzsuche.

Ein gängiges Beispiel der 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 [2]. Er ist besonders praktisch, falls die Ergebnisse der Search-Funktion dem Lokalitätsprinzip folgen, strukturell nahe beisammen in der Datenstruktur anzutreffen sind.

1.1 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 eine besonders große Distanz zum Suchergebnis zurücklegen und verhält sich langsamer 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 [3]. Die Fingersuche könnte außerdem selbst entscheiden, ob die Startposition der Suche von dem maximalen oder von dem minimalen Node in der Datenstruktur beginnen soll.

Die Min-Max-Fingersuche würde bei den Suchanfragen, welche immer das Maximum und das Minimum suchen, oder zumindest in deren Nähe suchen, deutlich besser abschneiden als der Lazy-Finger.

Man könnte zum Beispiel annehmen, dass die Datenstruktur ein binärer Suchbaum ist. Die am meisten gesuchten Nodes würden immer den dem linken Teilbaum ganz rechts sein. Anschließend würden im rechten Teilbaum die ganz links gelegenen Nodes 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.

1.2 **Zielsetzung**

Man könnte aus den oberen Beispielen entnehmen, dass die Suche am besten Finger in allen hochfrequentierten Bereichen der Datenstruktur positionieren sollte. Am sinnvollsten wären Bereiche, von denen man aus der Vergangenheit weiß, dass in diesen Bereichen besonders oft gesucht wird.

Somit wäre die Aufgabe der Fingersuche eine neue weitere Datenstruktur zu sein, welche die Information über die besonders oft nachgefragten Startpositionen der Distanzsuche enthält.

Es wird untersucht, ob verschiedene Fingersuchen sich je nach Verteilung der Suchanfragen besonders nützlich oder träge verhalten.

Zunächst wird für die Fragestellung eine theoretische Überlegung diskutiert und anschließend eine Simulation der Fingersuche mit einem Lazy-Finger, einer Min-Max-Fingersuche und einem Splay-Tree angefertigt.

2 Aktueller Forschungsstand: Übersicht über Finger

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

2.1 Ein Finger

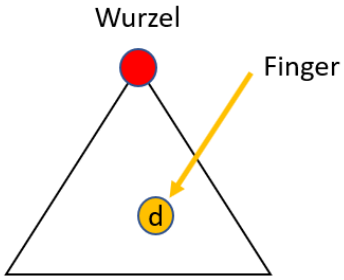

Abbildung 2-1 Schematische Darstellung, Finger in einem Baum

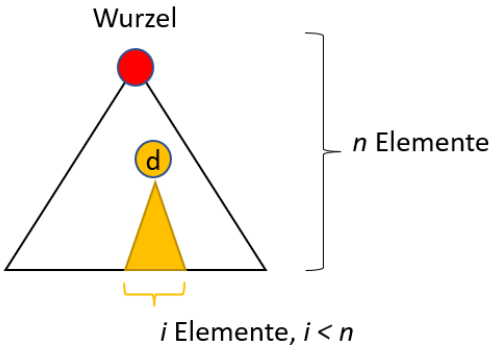

Abbildung 2-2 Schematische Darstellung, Distanzsuche mit Startposition

In Abbildung 2-1 ist eine Fingersuche mit genau einem Finger am Beispiel eines binären Suchbaums dargestellt. Finger sind externe Pointer, welche genutzt werden, um von ihrer jeweiligen Startposition d, statt von der Wurzel aus, zu suchen [1].

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 Abbildung 2-2 mit rot markiertem Startpunkt (genannt Root-Finger), in einem Binärbaum 2^{n-1} Elemente durchsuchen kann um das Ergebnis zu liefern [4].

Innerhalb des gelben Ausschnittes, welcher durch die Finger-Startposition d begrenzt wird, kann das zu suchende Node nach schon 2^{i-1} Vergleichen gefunden werden.

Der Vorteil ergibt sich daraus, dass *i* kleiner als *n* ist und je näher der Finger am Suchergebnis ist, desto exponentiell stärker reduziert sich die Suchdauer. Veranschaulicht bedeutet das, dass in Abbildung 2-2 die gelbe Fläche von dem Fingersuchbereich kleiner werden würde.

Formal ausgedrückt arbeitet die Wurzelsuche in der Laufzeitklasse O(log n), während die Distanzsuche auf i-Elementen arbeitet:

$n > i \quad => \quad O(\log n) > O(\log i).$

2.2 Min-Max-Fingersuche

Das Fast-Finger-Search Konzept beinhaltet, dass die Suche gleichzeitig von allen externen Fingern aus gestartet werden soll. Wenn einer der Finger ein Ergebnis findet, soll die Suche terminieren [3].

Als Spezialfall wird der Min-Finger und der Max-Finger genannt. Der Min-Finger steht immer auf dem minimalen Node und der Max-Finger steht immer auf dem maximalen Node im Tree. Die Fingersuche beginnt zeitgleich. Eine schematische Darstellung dieses Konzeptes ist in Abbildung 2-3 dargestellt [3].

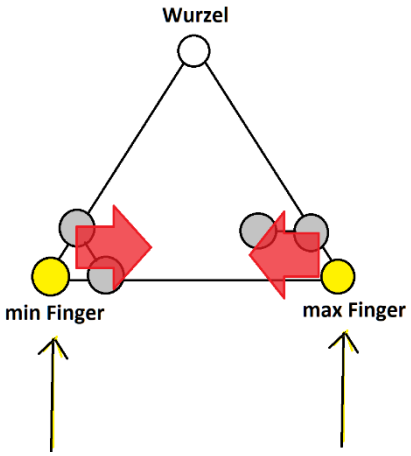

Abbildung 2-3 Schematische Darstellung, Min-Max Finger

Diese Laufzeit ergibt deshalb:

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

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

2.3 Finger mit Suchrichtung

Schon 1977 wird eine Fingersuche vorgestellt, welche eine Suchrichtung besitzt [5]. Wenn der Finger auf einer Position steht, kann es passieren, dass seine Nachbar-Nodes durchsucht werden müssen, bevor eine nach unten gerichtete Distanzsuche geschehen kann.

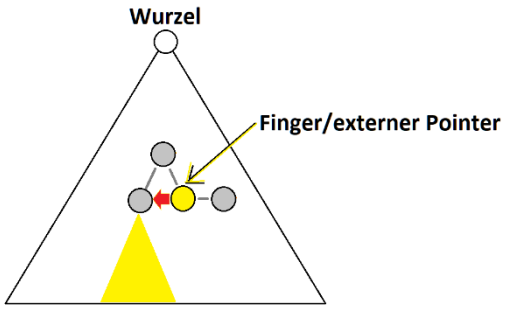

Abbildung 2-4 Schematische Darstellung, Finger und Beweglichkeit

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 beispielsweise im aktuellen Node auf dem kleinsten Element der Liste steht, dass das nächstkleinere bekannte Element das Maximum der Nachbarnode-Liste ist. Somit geht der Finger in den Nachbar-Node auf die gleiche Ebene wie in Abbildung 2-4 gezeichnet. [5]

In dieser Implementierung von Binärbaumfingern kann man also auf einer Ebene benachbarte Nodes durchsuchen, man sollte die Listen auf jeder Ebene somit als verlinkt betrachten.

Es wird auch der Fall betrachtet, dass Finger verschoben werden, also ein Update verlangen. Am Beispiel einer Insert-Operation wird festgestellt, dass Finger unter Umständen zu nahe beieinander liegen. [5]

2.4 Finger verwalten oder mit Fingern suchen

In einem Konzept aus dem Jahr 1982 werden auf Level-Linked-Binärbäumen Finger immer nur auf Leafs gesetzt [7]. 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. [7]

2.5 Lazy-Finger in binären Suchbäumen

In einem Conference Paper aus dem Jahr 2014 [2] 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 Abbildung 2-5 schematisch dargestellt.

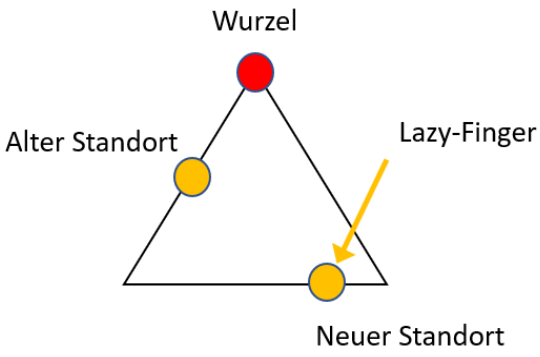

Abbildung 2-5 Schematische Darstellung, Lazy-Finger

Die Fragestellung der Abständen von dem Suchergebnis und dem Lazy-Finger wird im Zusammenhang von Entropien (Informationstheorie) diskutiert: *" finding the tree that minimizes the execution time of search sequence X using lazy finger takes time* $O(n^3)$*"* - das ist die Laufzeit, um eine optimale Lazy-Finger Position in einem binären Baum zu berechnen [2].

2.6 Splay-Tree und Fingersuche

„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[trees] keine expliziten Finger verwendet“ [6].

2.7 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. [7]

Die Eigenschaft von Splay-Trees, sich wie eine statische Datenstruktur, beispielsweise ein Binärbaum verhalten zu können, bezeichnet man als „statische Optimalität“ (siehe Kapitel „Splay-Tree und Fingersuche“) [7].

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“ [8].

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

2.8 Fingersuche und weitere Datenstrukturen

Das Konzept der Finger, als externe Pointer, hat keine Einschränkung für spezielle Datenstrukturen. Daher zählt dieses Kapitel auf, 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) [9]
- Treaps [1]
- Skip-List nur mit 1 Finger [1]
- (2,4)-Level-Linked-Tree [1]

Die (2,4)-Level-Linked-Trees sind besonders hervorzuheben, da sie maximal ausgebaut im Hinblick auf die Erreichbarkeit aller Nodes sind. Somit wird eine besonders flexible und auch effiziente Fingersuche ermöglicht [1].

2.8.1 K-server Problem und Finger

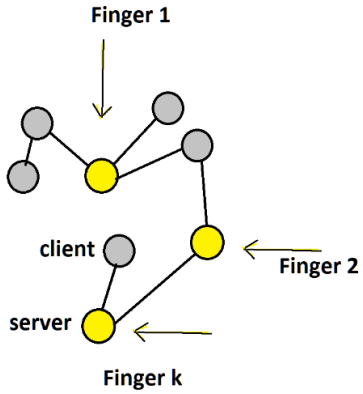

Abbildung 2-6 Schematische Darstellung, k-Server mit k -Fingern

Das k-server Problem besagt, dass k-viele Server deutlich mehr Clients bedienen müssen und das Ziel dabei ist, die Server möglichst nicht zu überlasten. [10]

Dem k-server Problem wird mit k-vielen Fingern begegnet. Es wird davon ausgegangen, dass alle Server und alle Clients eine „Karte aller Nodes“, also die komplette Information zum Entscheiden haben. [11]

Die grundsätzliche Idee ist, dass jedem Server ein Finger zugewiesen wird und anhand dieser Finger die nächstmöglichen Clients mithilfe der Fingersuche, gefunden und bedient werden, siehe Abbildung 2-6. [11]

Der Algorithmus gibt also die Information zurück, welcher Finger für welchen Client benutzt wurde. Das ist eine alternative Anwendung der Fingersuche. [11]

2.8.2 SplayNets – Netzwerk Kommunikation

In der Doktorarbeit von Belo Horizonte wird das Konzept der SplayNets präsentiert [12]. SplayNets arbeiten auf auf Peer-to-Peer (P2P) Netzwerken und versuchen die optimalen Kommunikationswege mit Hilfe von Splay-Trees als Fingersuche zu sogenannten SplayNets zu vereinfachen. Jeder Node erhält also einen Finger beziehungsweise 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 Netzwerke, sowie für interne Kommunikation in Facebooks-Datencentren vorgestellt, da 60% des Traffics interne Kommunikation darstellt [12].

SplayNets führen zu erheblicher Verbesserung der internen Kommunikationswege, da mit ihrer Hilfe irrelevante protokollbasierte Informationsaustauschversuche deutlich verringert werden können [12].

2.8.3 Grover Algorithmus – Quantencomputer

Der Grover Algorithmus arbeitet ausschließlich auf Quantencomputern, dabei werden quantenmechanische Effekte ausgenutzt und vorteilhaft ausgelesen. Er arbeitet auf unsortierten Datenstrukturen der Größe n und ermöglicht eine Suche in $O(\sqrt{n})$ [13].

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 [13].

2.8.4 Prädiktion auf Input Streams für Lokalitätsprinzip

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

Der Schwerpunkt ist die Vorhersage von Inputdaten für verschiedene Systeme, 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. [14]

2.9 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.

Obwohl der Begriff des Fingers nicht in allen Gebieten der Informatik etabliert ist und beispielsweise in der Netzwerkkommunikation kaum Erwähnung findet, wird das zugrundeliegende Konzept dennoch verwendet.

Die grundsätzliche wissenschaftliche Frage nach der dynamischen Optimalität der Fingersuche ist noch offen.

3 Modellierung des Problems

Dieses Kapitel beschäftigt sich mit der Frage, ob Finger zentral verwaltet werden sollten. Als Ausgangslage ist ein binärer Suchbaum gegeben, welcher um Finger erweitert wird.

3.1 Das Modell

Gegenstand dieser Arbeit ist, dass je nach Problemumfeld und Fragestellung (vorheriges Kapitel: „Aktueller Forschungsstand: Übersicht über Finger“) für die Fingersuche, jeweils unterschiedliche vorteilige oder nachteilige Verhaltensweisen der Suche resultieren können.

Daher sollte die Wahl eines Fingerkonzepts davon abhängen, wie die Datenstruktur, hier binärer Suchbaum, verwendet wird. Deswegen sollte die Nutzung der vorliegenden Datenstruktur im Vorfeld der Entscheidung untersucht werden.

Hierzu einige Beispiele:

- Z.B. wird beim Lazy-Finger indirekt angenommen, dass nur ein Bereich im binären Suchbaum besonders interessant ist. Jeder Wert außerhalb dieses Bereiches ist ein ‚ungünstiger‘ Fall. Was passiert, wenn die gesamten Suchaufträge ‚ungünstige‘ Fälle sind? Die Zick-Zack Verteilung der Keys ist so ein Beispiel, weil zwei verschiedene Enden der Datenstruktur alternierend abgefragt werden.
- Bei dem Min-Max Finger sind bei der Suche zwei Bereiche abgedeckt, jedoch würde diese Fingersuche bei einem dritten Bereich in der Mitte der Datenstruktur auch nicht besser als die Wurzelsuche abschneiden.

Im Folgenden wird deshalb die Fingersuche mit Hilfe einer Simulation untersucht: Statt der Wurzelsuche ohne Finger (Abbildung 3-1 links) soll ein Finger-Management vorgeschaltet werden (Abbildung 3-1 rechts). Dieses Finger-Management ist allgemeiner gefasst und speichert in sich selbst die Vorhersage wie die große Hauptdatenstruktur verwendet wird.

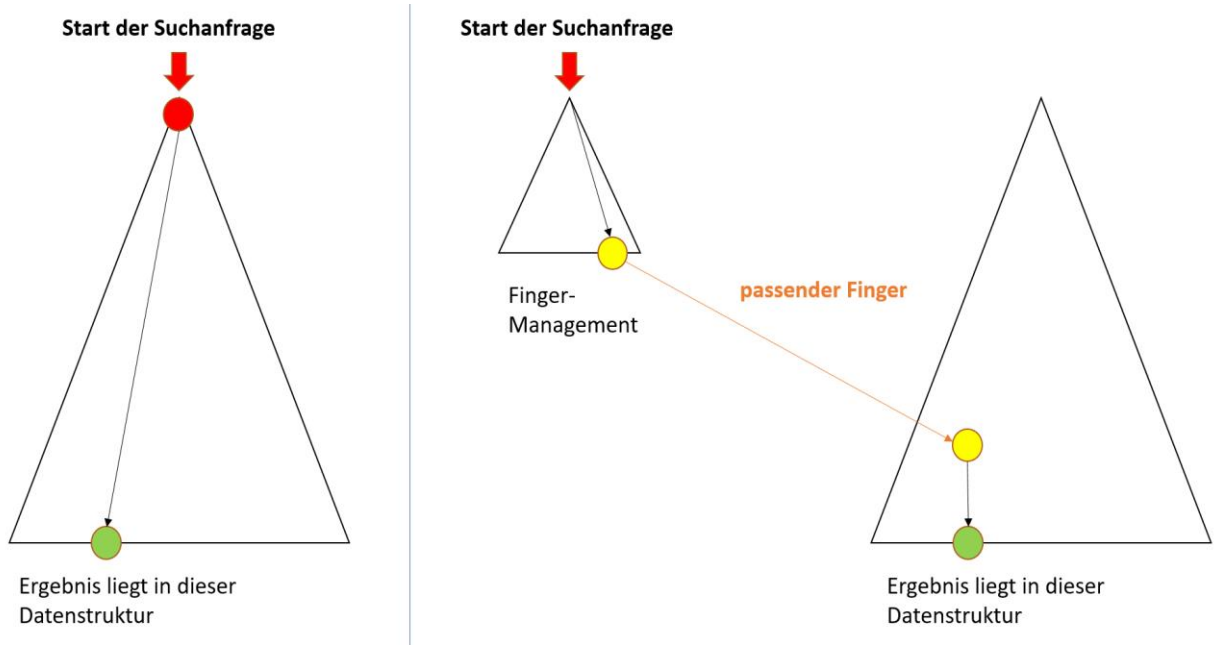

Abbildung 3-1 links: Wurzelsuche, rechts: Fingersuche mit Finger-Management

Im Gegensatz zur weitverbreiteten Wurzelsuche (Abbildung 3-1 links) wird in der Fingersuche (Abbildung 3-1 rechts) die Startposition der Suchanfragen anders gesetzt. Das Finger-Management wird beschrieben durch:

1) Eine separate Datenstruktur, die alle Finger/externe Pointer verwaltet
2) In dieser Finger-Management-Datenstruktur werden bei Suchanfragen die passenden Finger herausgesucht
3) Diese passenden 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 das Finger-Management – genauer die Search-Operation mit dem Finger-Management. Das Finger-Management ist entweder als ein Lazy-Finger, Min-Max-Finger oder als Splay-Tree implementiert.

3.1.1 Überlegungen zum Aufbau des Finger-Management

Das Finger-Management 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.

- Für den Lazy-Finger wäre das Finger-Management einfach ein Pointer mit Speicherplatz für einen Node.
- Für die Min-Max Fingersuche könnte z.B. das Finger-Management zwei Pointer beinhalten, welche in einer Liste verwaltet werden könnten.

Auch andere Datenstrukturen kann man in Betracht ziehen, wie beispielsweise binäre Suchbäume. Dann stellt man sich die Frage, welche Pointer in diesen Baum aufgenommen werden. Ein Vorschlag ist aus einem Rot-Schwarz Baum nur die Roten Nodes herauszunehmen und eine Red-Finger-Search zu implementieren. Man könnte sich an dieser Stelle auch eine Kostenfunktion für die beliebtesten Finger überlegen und diese dann berücksichtigen. Man könnte auch den Zufall entscheiden lassen, welche Finger in das Finger-Management-System aufgenommen werden.

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 [7]. 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-Trees, Zufall und Optimalität“ herausgearbeitet, besitzt der Splay-Tree außerdem die statische Optimalität.

3.1.2 Überlegungen zu 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 beliebig gewachsene und auch unsortierte Datenbank sein. Fingersuchen können außerdem auch auf Graphen angewendet werden, wie z.B. in P2P-Netzen [12].

In dieser Simulation wird, der Einfachhalt halber, ein binärer balancierter Suchbaum für die Hauptdatenstruktur implementiert.

3.2 Einwände bezüglich des Modells

Ein Kritikpunkt ist, dass man auch Balancierungs- und weitere baumtypische Operationen berücksichtigen muss, denn z.B. durch 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 Finger-Management-Datenstruktur erhalten. Zwangsläufig würden alle die Datenstruktur manipulierenden Operationen zur Neuaufsetzung der Finger führen.

Der Einfachheit- und Vergleichbarkeitshalber beschränkt sich diese Arbeit deswegen ausschließlich auf die Suchfunktion.

Wie im vorherigen Kapitel: “Splay-Tree und Fingersuche“ beschrieben, verhalten Splay-Tree und Finger-Suchen sich asymptotisch gleich, daher könnte davon ausgegangen werden, dass man einfach die Hauptdatenstruktur als Splay-Tree zu implementieren braucht und der Overhead für das Finger-Management entfällt.

Dieser Überlegung würde ich gerne einige Argumente entgegenstellen:

- Splay-Trees arbeiten oft im Cache [15], das bedeutet man versucht diese Datenstruktur möglichst klein zu halten. Damit wäre dieser Lösungsansatz, statt der Fingersuche Splay-Trees in der Hauptdatenstruktur zu verwenden, nur für kleine Datenansammlungen geeignet. Beim Splay-Tree kann es außerdem zu der nachteiligen linearen Formation kommen [7], was besonders bei sehr großen Splay-Trees ungünstig ist.
- Wenn man annimmt, dass die Hauptdatenstruktur auf verteilten Systemen arbeitet, dann ist es nicht sinnvoll diese Datenstruktur aufgrund von Suchanfragen jedes Mal zu restrukturieren. Das wäre unter Umständen sehr aufwändig. Wenn man außerdem bedenkt, dass viele Datenbanken historisch gewachsen sind, ist es eventuell einfacher, externe Finger anzubringen, als die komplette Datenbank jedes Mal umzustrukturieren.

4 Theoretische Betrachtung

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

4.1 Anzahl der Finger

Die Anzahl der möglichen zu durchsuchenden Elementen (Abbildung 4-1) wird ergänzend zu dem vorgestellten Modell erläutert:

- 2^n ist die Anzahl der Elemente in der Hauptdatenstruktur
- 2^x ist die Anzahl aller Finger im Finger-Management

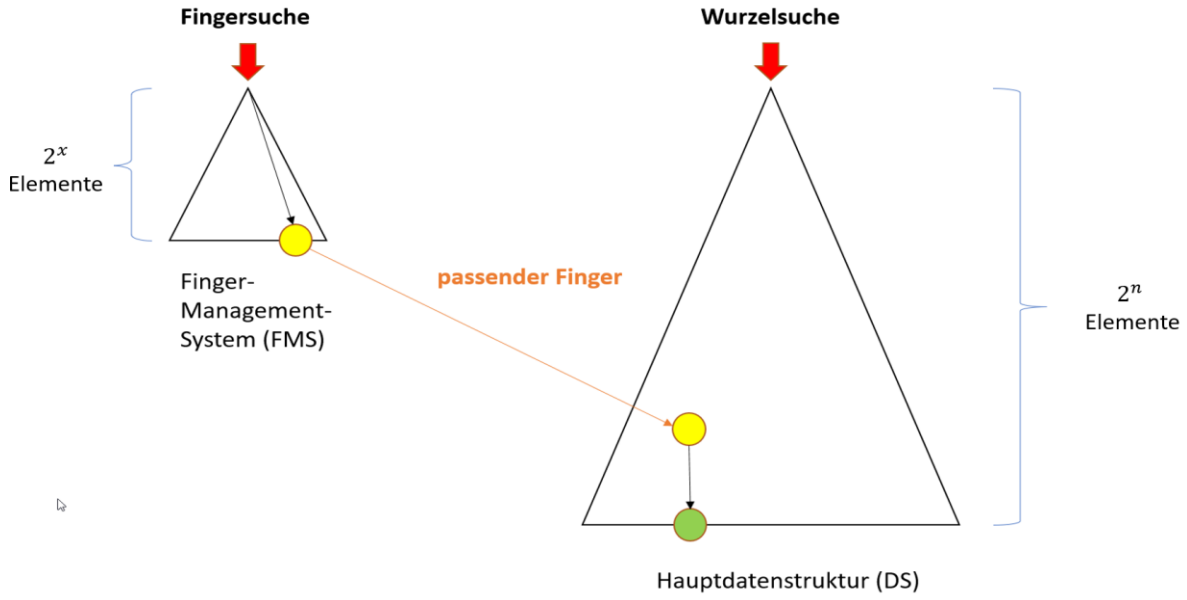

Abbildung 4-1 Schematische Darstellung, Finger-Management

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

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

O(Wurzelsuche) > O(Fingersuche)

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

2) Die Annahme für das weitere Vorgehen ist, dass das Finger-Managementsystem (FMS) und die Hauptdatenstruktur beides binäre Bäume sind. Damit kann log_2 einheitlich zur Berechnung der Anzahl der durchlaufenen Nodes verwendet werden.
Für die Anzahl der Finger x gilt dann:

O(Wurzelsuche)	> O(Fingersuche in FMS) + O(Distanzsuche)	
$log_2(n)$	> $log_2(x) + log_2(d)$	
$log_2(n)$	> $log_2(x \cdot d)$	
(n)	> $(x \cdot d)$	
$\frac{n}{d}$	> x	(I)

3) Nun lässt sich die Abschätzung tätigen: sei a der Unterschied zwischen der Wurzel in der Hauptdatenstruktur und der Distanzsuche-Startposition d. Graphisch betrachtet benutzt diese Abschätzung folgendes Konzept (Abbildung 4-2):

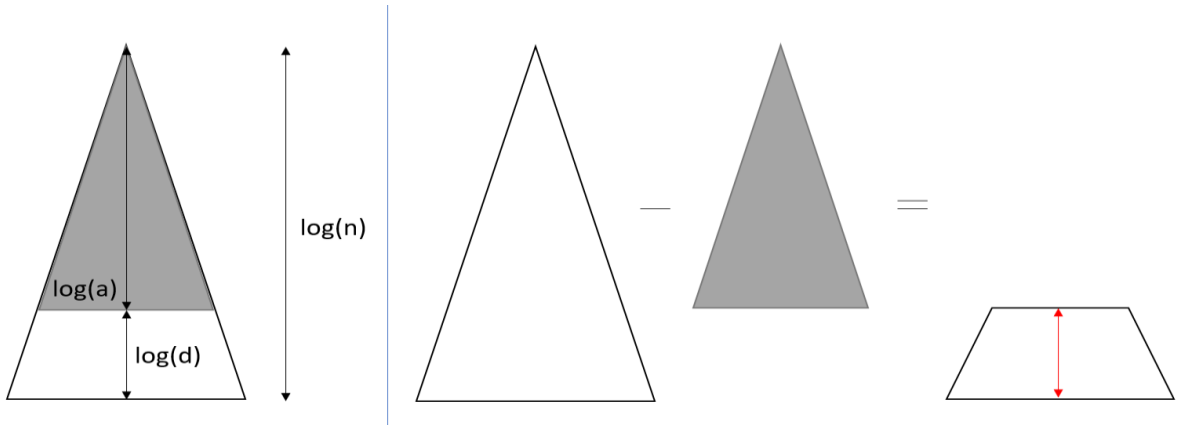

Abbildung 4-2 Abschätzung für d

$$log_2(n) - log_2(a) = log_2(d)$$

$$log_2(n \,/\, a) = log_2(d)$$

$$n \,/\, a = d$$

$$n = d \cdot a \qquad \text{(II)}$$

4) Nun kann man (I) und (II) kombinieren:

$$x < \frac{n}{d} \qquad \text{(I)}$$

$$x < \frac{d \cdot a}{d} \qquad \text{(II)}$$

$$x < a$$

Die Höhe bis zum Startpunkt der Distanzsuche gibt vor, was die maximale Anzahl der Finger x ist.

4.2 Laufzeitbetrachtung

Im vorherigen Kapitel wurde für binäre Suchbäume ermittelt, dass für die kürzere Laufzeit verglichen mit der Wurzelsuche, die Anzahl der Finger des Finger-Management-Systems maximal *a* sein soll. Dementsprechend dauert es maximal $log_2(a)$ Vergleiche um den richtigen

Finger zu finden. Für die Berechnung der gesamten Suche werden weitere $log_2(d)$ viele Vergleiche durchgeführt bis die Suche abgeschlossen ist.

Um auf die Ausgangsformel zurückzukehren:

O_worst(Fingersuche) = O_worst(Fingersuche FMS) + O_worst(Distanzsuche)

$= log_2(a) + log_2(d)$

$= log_2(d \cdot a)$

4.3 Erkenntnisse für Fingerarten

Man kann an der Laufzeit-Formel gut erkennen, dass geringe Werte für (a) und tiefer gelegte Finger (d) in der Hauptdatenstruktur tendenziell besonders schnell werden lassen:

O_worst(Fingersuche) = $log_2(d \cdot a)$

Sowohl die Min- und Max-Finger als auch der Lazy-Finger würden das Kriterium besonders vorteilhaft erfüllen. Der Lazy-Finger wäre nur dahingehend vorteilhaft, als dass nur das Produkt $(d \cdot a)$ ausschlaggebend ist und wenn a klein ist, was der Lazy-Finger erfüllt, wird auch das Produkt klein sein.

Außerdem kann man ablesen, wie viele Bereiche in der Hauptdatenstruktur mit der Fingersuche maximal abgedeckt werden können. Wenn jeder Finger auf einen Bereich zeigt, sind das a-viele Bereiche. Des Weiteren könnte die Distanzsuche tiefer anfangen zu suchen, also *d* wird kleiner, dann bleibet mehr Spielraum für a und damit könnten noch mehr Bereiche durch Finger angesteuert werden.

4.4 Der „gekürzte“ Splay-Tree

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

Trotzdem sollte man den nicht chaotischen Fall, dass der Splay-Tree immer in eine lineare Formation [7] gebracht wird, untersuchen:

1. Die Laufzeitformel für Suche mit Fingern:

   O_worst(Fingersuche) = O_worst(Fingersuche FMS) + O_worst(Distanzsuche)

2. Ausführung mit Splay-Tree:

   O_worst(Fingersuche) = O_worst(Splay-Tree) + O_worst(Distanzsuche)

3. Worst-Case für Splay-Tree - Suchanfragen führen immer zur linearen Formation:

   O_worst(Splay-Tree) = O(a) = O($\log_2(2^a)$)

   ⇨ O_worst(Fingersuche) = O_worst(Splay-Tree) + O_worst(Distanzsuche)
   ⇨ O_worst(Fingersuche) = O_worst(O($\log_2(2^a)$)+ O_worst($log_2(d)$)
   ⇨ O_worst(Fingersuche) = $log_2(d \cdot 2^a)$

Man könnte somit schlussfolgern, dass es trotz der vorkommenden linearen Formation sinnvoll ist, einen Splay-Tree für die Fingersuche zu verwenden. Wenn dieser Splay-Tree wie hier dargestellt, 2^a-viele Finger statt n-viele Finger in sich speichert, nenne ich diese Struktur „gekürzter“ Splay-Tree.

5 Programmierung der Simulation

Dieses Kapitel befasst sich mit der Simulation der Fingersuche. Diese Simulation wurde mithilfe von Python programmiert. Es wurden möglichst wenige Libraries verwendet, um die Datenstruktur eigens modifizieren zu können und somit die maximale Kontrolle über die Simulation und das Vorgehen bei der Implementierung zu behalten.

5.1 Vereinfachung des Modells

Im vorherigen Kapitel wurde ein allgemeines Modell für die Fingersuche vorgestellt, welches die Verwaltung der Finger als eigenständige Datenstruktur versteht. Damit die Betrachtung der zuvor formulierten Untersuchung im Rahmen der Bachelorarbeit, aber auch möglichst präzise und fokussiert bleibt, wurden besondere Fälle und gesonderte Aspekte nicht weiter simuliert. Daher wurden bei der Hauptdatenstruktur und der Fingersuche folgende einfache Sachverhalte festgelegt:

- Die Hauptdatenstruktur wird ein balancierter binärer Baum sein.
- Das Finger-Management wird mit einem Lazy-Finger, Min-Max-Finger und mit einem Splay-Tree implementiert.

6 Evaluation der Ergebnisse

In diesem Kapitel werden drei verschiedene Fingersuchen sowie die Wurzelsuche als Benchmark gestartet. Das wird anhand von fünf verschiedenen Suchverteilungen verglichen und analysiert.

Die Durchführung der Simulation ist für alle Verteilungen gleich konzipiert:

- Es wird 16 Mal ein binärer Baum erstellt und mit natürlichen Zahlen (Keys) von 1 bis 25, 1 bis 50, … , 1 bis 400 befüllt.
- Dann wird in der jeweiligen Datenstruktur 25, 50, …, 400-mal der Suchauftrag mit den jeweiligen Verteilungen durchgeführt. So wird die absolute Anzahl der tangierten Nodes *t(n)* pro Suchauftrag ermittelt. Dadurch steigt die Funktion *t(n)* mit der Größe der Datenstruktur exponentiell an.
- Um diesen Effekt zu bereinigen, wird anschließend der durchschnittliche Wert pro Suche ermittelt $\frac{t(n)}{n}$.

Mit diesem durchschnittlichen Wert sind die Algorithmen Lazy-Finger, Min-Max-Finger und Splay-Tree Suche vergleichbar. Sowohl ihre Geschwindigkeit pro Suchauftrag, als auch die Geschwindigkeiten, welche sich mit steigender Größe der Datenstruktur ändern.

Durch die theoretische Betrachtung wäre unter den fünf Verteilungen in den Suchanfragen anzunehmen, dass $\frac{t(n)}{n}$ sich entweder logarithmisch oder konstant verhält.

6.1 Verteilungen in Suchanfragen

6.1.1 Verteilung in Suchanfragen - Immer Zahl “1”

In dem Plot Abbildung 6-1 ist zu sehen:

- x-Achse: Anzahl der Elemente im binären Baum.
- y-Achse: Diese Zahl setzt sich aus den besuchten Nodes während der Fingersuche und der Distanzsuche zusammen. Im Diagramm ist $\frac{t(n)}{n}$ dargestellt.

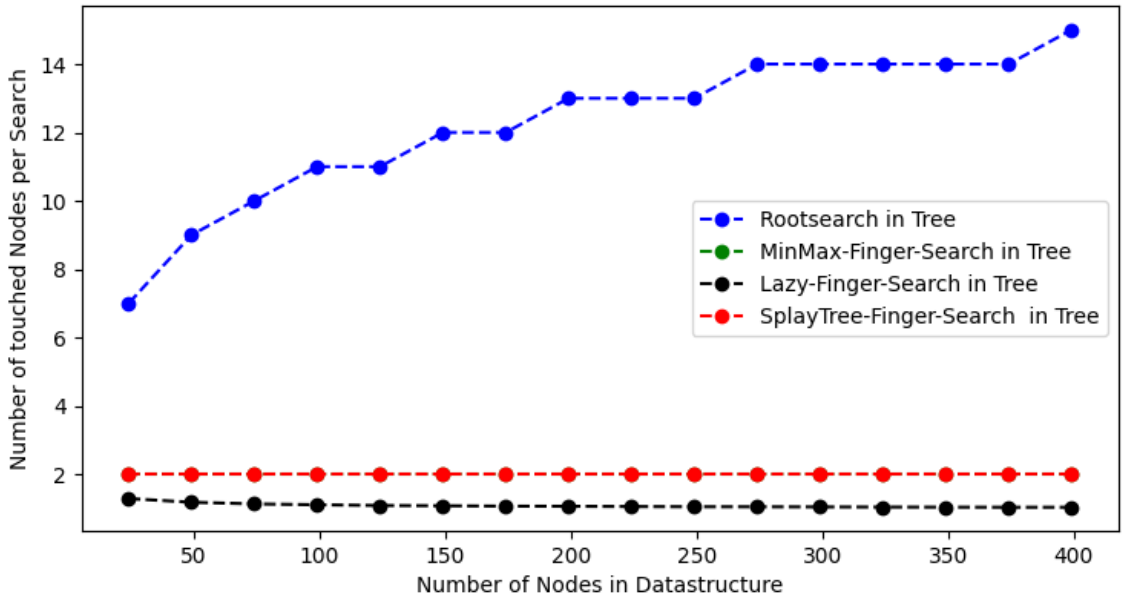

Abbildung 6-1 Minimum Search – Anzahl Nodes pro Suche

Die Min-Max-Fingersuchkurve ist hinter der roten Kurve des Splay-Trees verborgen.

Zu erwarten wäre, dass Lazy-Finger und Splay-Tree sich die Position des Nodes mit der 1 merken und jedes weitere Mal die Lokalitätseigenschaft der Finger nutzen. Die Min-Max-Fingersuche hat in diesem Fall den Vorteil, dass der statische Min-Finger schon das Node mit der 1 kennt, somit würde auch diese Fingersuche die Lokalitätseigenschaft nutzen.

Dem gegenüber steht der Benchmark, die Wurzelsuche, welche keinen Memory-Effekt oder Lokalitätseigenschaften nutzen kann. Sie muss deshalb die langsamste Laufzeit aufweisen, welche logarithmisch ist. Die anderen Fingersuchen müssen durch den Lokalitätseigenschaft eine nahezu konstante Laufzeit aufweisen.

Diese Kurven entsprechen somit den theoretischen Erwartungen.

6.1.2 Verteilung in Suchanfragen - Immer Maximum

In diesem Plot Abbildung 6-2 ist zu sehen:

- x-Achse: Anzahl der Elemente im binären Baum.
- y-Achse: Diese Zahl setzt sich aus den besuchten Nodes während der Fingersuche und der Distanzsuche zusammen. Im Diagramm ist $\frac{t(n)}{n}$ dargestellt.

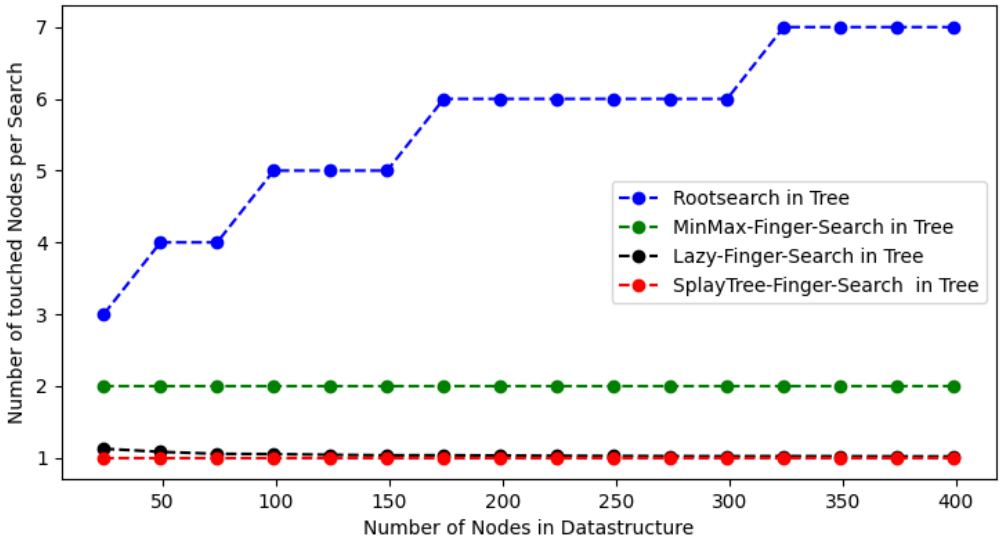

Abbildung 6-2 Maximum Search – Anzahl Nodes pro Suche

In dieser Simulation würde man erwarten, dass die gleichen Werte, wie bei der Suche nach dem Minimum zu sehen sind.

Es ist für die Fingersuchen unerheblich, ob die Finger sich am Minimum oder Maximum positionieren. Der Lazy-Finger muss sich am anderen Ende der Datenstruktur positionieren. Der Splay-Tree wird einen anderen Finger in die Wurzel speichern und die Min-Max Fingersuche würde den Max-Finger statt dem Min-Finger verwenden. Bei der Wurzelsuche wäre analog zu erwarten, dass sie keine Lokalitätseigenschaften nutzt.

Da für die Wurzelsuche jedoch halb so viele Nodes angezeigt werden, wie bei der Suche nach dem Minimum-Node im Kapitel vorher, könnte man vermuten, dass der rotschwarz-Baum nicht ganz balanciert ist.

Im Wesentlichen verhält sich die Wurzelsuche wie erwartet logarithmisch und die anderen drei Fingersuchen nahezu konstant. Die Kurven entsprechen somit der theoretischen Betrachtung.

6.1.3 Verteilung in Suchanfragen - Immer mittelstes Element

In diesem Plot Abbildung 6-3 ist zu sehen:

- x-Achse: Anzahl der Elemente im binären Baum
- y-Achse: Diese Zahl setzt sich aus den besuchten Nodes während der Fingersuche und der Distanzsuche zusammen. Im Diagramm ist $\frac{t(n)}{n}$ dargestellt.

Von dem mittelsten Key-Element wurde angenommen, dass es wurzelnah in der Hauptdatenstruktur ist.

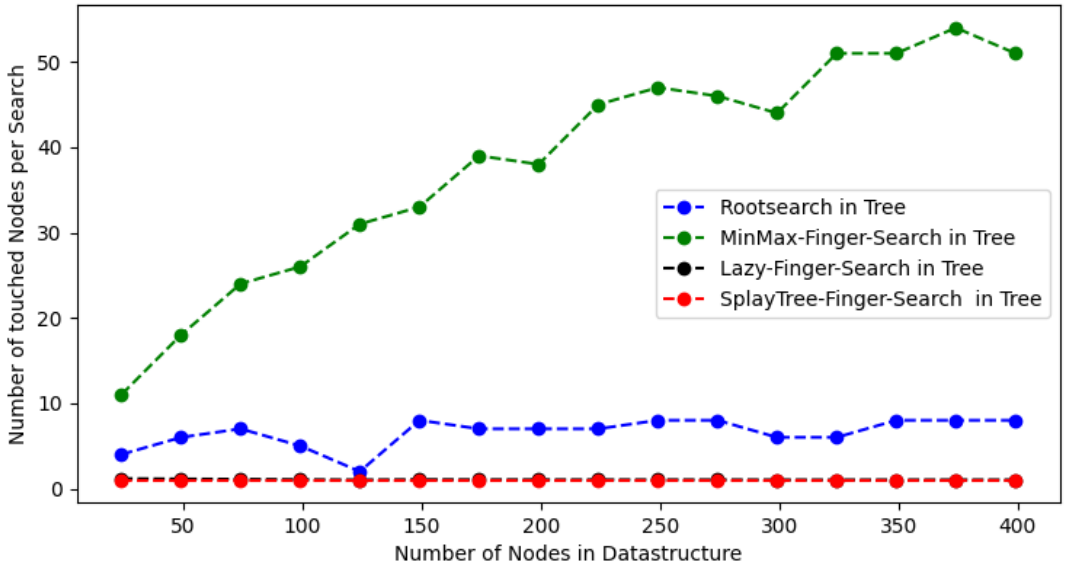

Abbildung 6-3 Middle Key Search - Anzahl Nodes pro Suche

Die schwarze Lazy-Fingerkurve liegt hinter der roten Kurve des Splay-Trees verborgen.

Bei dieser Verteilung der Suchanfragen erwartet man, dass der Lazy-Finger sich auf das mittelste Element wurzelnah positioniert und dort die Lokalitätseigenschaft wieder nutzt. Der Splay-Tree muss ein neues Node in seiner Wurzel speichern und ist somit laufzeittechnisch gleich zum Lazy-Finger einzuschätzen.

Bei der Min-Max-Fingersuche erwartet man, dass die statisch gesetzten Finger und damit die Vermutung der statischen Lokalitätseigenschaft der Suchergebnisse ungünstig zutragen kommt. Weshalb diese Fingersuche besonders langsam ist.

Die Wurzelsuche wird auch Root-Fringer-Search genannt, siehe Kapitel „Ein Finger“. Damit wird suggeriert, dass die Wurzelsuche die Lokalitätseigenschaft nahe der Wurzel ausnutzt. Hier vermutet man daher eine nahezu konstante Laufzeit anzutreffen, weil das mittelste Element in einem balancierten Baum meistens in der Wurzel ist.

Die Messwerte bei einem 200 Node großen Binärbaum für die Wurzelsuche ergeben:

1592/200 = 7,96.

Es werden also 7,96 Vergleiche benötigt, um ein angeblich wurzelnahes Node zu finden. Es wurde festgestellt, dass das ungefähr so viele Vergleiche sind, wie man zu einem beliebigen Leaf gebraucht: $log_2(n) = log_2(200) = 7{,}64$.

Deshalb kann man Schlussfolgern, dass das mittelste Element eher auf Leafhöhe anzutreffen ist, als nahe der Wurzel. Daraus ergibt sich erneut der Verdacht, dass der implementierte Rotschwarzbaum nicht perfekt balanciert ist. Diese Vermutung würde auch davon gestützt werden, dass es einen „Zacken“ bei 125-Nodes im Diagramm gibt. Dort ist vermutlich das mittelste Element näher zur Wurzel gefunden worden.

Davon unberührt verhält sich die Min-Max-Fingersuche erwartungsgemäß langsam logarithmisch und der Lazy-Finger sowie der Splay-Tree fast konstant.

6.1.4 Verteilung in Suchanfragen - Alternierendes Minimum und Maximum

In diesem Plot Abbildung 6-4 ist zu sehen:

- x-Achse Anzahl der Elemente im Rotschwarz Baum
- y-Achse Diese Zahl setzte sich aus den besuchten Nodes während der Fingersuche und der Distanzsuche zusammen. Im Diagramm ist $\frac{t(n)}{n}$ dargestellt.

Es wurde jeweils immer Suchanfragen als Zick-Zack-Muster gestellt, also Minimum und Maximum alternierend:

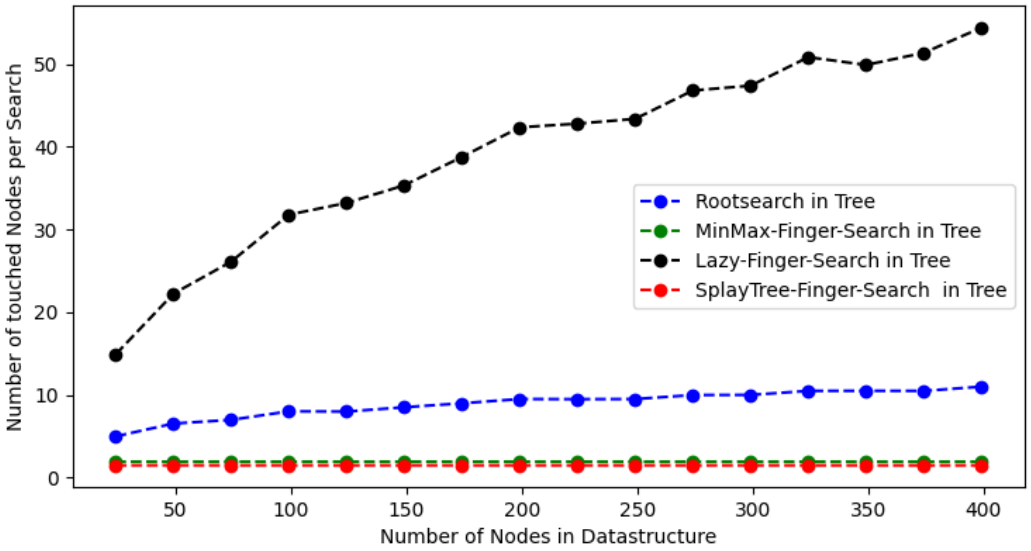

Abbildung 6-4 Alternierend Minimum and Maximum - Anzahl Nodes pro Suche

In diesem Anwendungsfall wäre zu erwarten, dass der Lazy-Finger durch seine Memory Eigenschaft behindert wird. Es ist erwartungsgemäß im Plot zu erkennen, dass der Lazy-Finger deutlich langsamer als die anderen Fingersuchen und auch deutlich langsamer als die Wurzelsuche ist.

Der Splay-Tree begegnet dieser Suchverteilung hingegen damit, dass das Maximum und das Minimum nahe der Splay-Tree-Wurzel gespeichert werden. Damit ist zu erwarten eine nahezu konstante Laufzeit zu sehen. Ähnliches Verhalten wäre für die Min-Max-Fingersuche zu vermuten, da hier das Minimum und Maximum schon bereit liegen.

Dieses Verhalten der Min-Max-Fingersuche und des Splay-Trees ist somit erwartungsgemäß. Es war zu erwarten, dass der Lazy-Finger die höchsten Werte haben wird, die Wurzelsuche im Mittelfeld liegen wird und die anderen Splay-Tree Suche und Min-Max-Fingersuche die schnellsten sein werden.

Die Erwartungen aus den theoretischen Betrachtungen des logarithmischen Verhaltens des Lazy-Fingers, dem konstanten Verhalten des Splay-Trees und der Min-Max-Fingersuche konnten nachgewiesen werden.

6.1.5 Zufällige Verteilung zwischen Minimalen und Maximalem Element

In diesem Plot Abbildung 6-5 ist zu sehen:

- x-Achse: Anzahl der Nodes im rotschwarzen Baum.
- y-Achse Diese Zahl setzte sich aus den besuchten Nodes während der Fingersuche und der Distanzsuche zusammen. Im Diagramm ist $\frac{t(n)}{n}$ dargestellt.

Das Suchmuster sind zufällige Keys zwischen dem minimalen und dem maximalen Node:

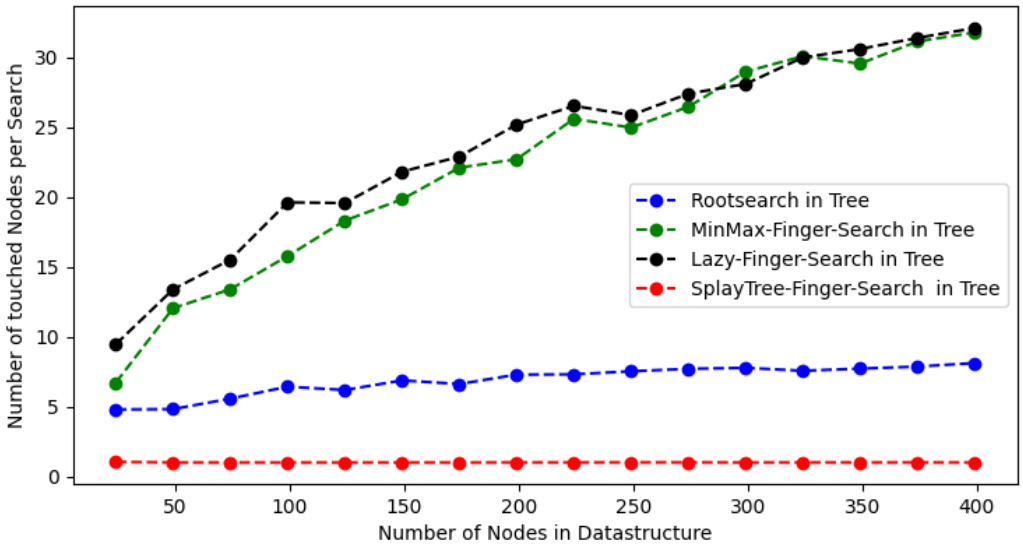

Abbildung 6-5 Random Key Search - Anzahl Nodes pro Suche

Laut meiner Literaturrecherche im Kapitel „Splay-Trees, Zufall und Optimalität“ wäre genau diese zufällige Suchverteilung der ungünstigste Fall für den Splay-Tree, da es kein Suchmuster gibt. Dieses Verhalten kann ich in meiner Simulation leider nicht zeigen. Wie man in Abbildung 6-5 sieht ist der Splay-Tree konstant schnell.

Dieses Messergebnis lässt jedoch mehrere Interpretationen zu. Einmal, dass die vorliegende Implementation des Splay-Trees fehlerhaft ist, oder dass die *random*-Funktion in Python nicht chaotisch genug ist, um das ungünstige Verhalten zu verursachen oder vielleicht ist tatsächlich der Splay-Tree ein besonders schneller Algorithmus.

Interessant ist außerdem zu beobachten, dass der Lazy-Finger und auch die Min-Max-Suche eine ähnliche Anzahl an Vergleichen tätigen.

Es ist ebenfalls gut zu erkennen, dass die Wurzelsuche schneller ist als die Min-Max-Fingersuche und die Lazy-Fingersuche. Es ist naheliegend, da die Wurzelsuche unabhängig der Suchanfrageverteilungen arbeitet.

Theoretisch zu erwarten wären vier logarithmische Kurven, davon konnten drei logarithmisch gezeigt werden. Der Splay-Tree verhält sich auffällig konstant.

6.2 Zusammenfassung: Direktvergleich der Diagramme

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

In der Zusammenstellung siehe Abbildung 6-6, wo die Finger- und Wurzelsuche pro Suchauftrag dargestellt ist, ist gut zu erkennen, dass je nach Fingersuche und Größe der Datenstruktur die Kurven unterschiedlich stark steigen. Die folgenden Diagramme sind wie im vorherigen Kapiteln erstellt worden, nur mit größerem Wertebereich.

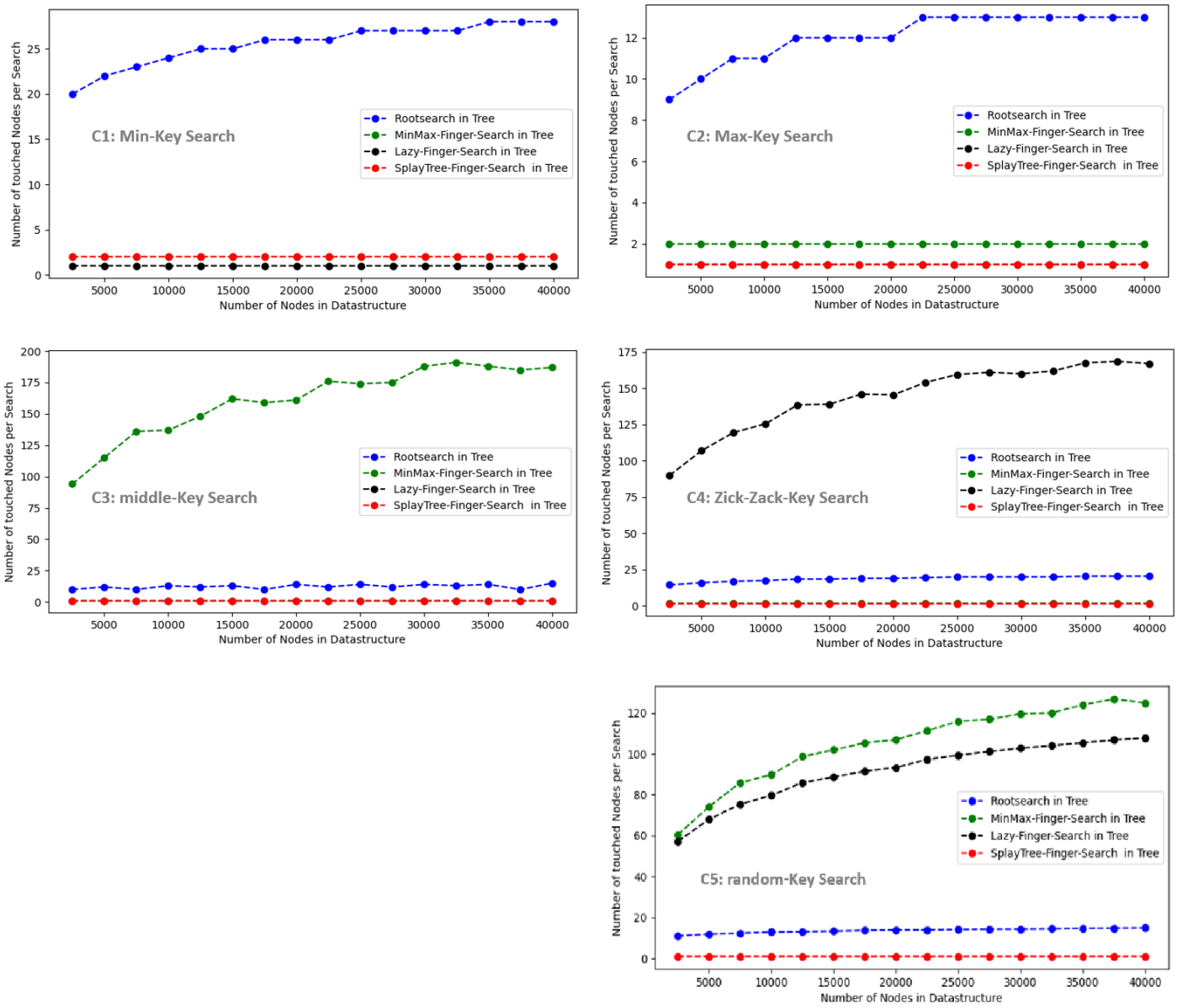

Abbildung 6-6 Überblick Anzahl der Nodes pro Suche (größerer Wertebereich) und Suchaufträgen

Je höher und steiler die Kurve ist, desto schlechter verhält sich der Algorithmus im jeweiligen Anwendungsfall. Wie in der theoretischen Betrachtung ermittelt, ist das logarithmische oder konstante Verhalten der jeweiligen Suche zusehen.

Zusammenfassend konnte gezeigt werden, dass durch die Verteilungen der abzufragenden Keys in den Suchanfragen ein Einfluss auf die Schnelligkeit der unterschiedlichen Fingersuchen gezeigt werden konnte.

7 Einordnung der Ergebnisse anhand des Forschungsstandes

Somit konnte gezeigt werden, dass sich die Fingersuchen in Abhängigkeit der Konfiguration der Suchanfragen, entweder konstant oder logarithmisch verhalten.

7.1 Trade-Off für Anzahl der Finger

Anhand der theoretischen Abhandlungen ist der folgende Trade-Off als nützlich herausgearbeitet worden. Es geht nicht darum eine optimale Anzahl von Finger und eine optimale Position zu ermitteln, sondern es geht darum einen absoluten Vorteil durch Fingernutzung zu erzielen. Z.B. beim Splay-Tree hätte man 2^a-viele Finger gespeichert - statt n-vielen. Der Lösungsvorschlag ist wie folgt, die Finger-Management-Datenstruktur wird ab einer gewissen Größe und Höhe „gekürzt".

Als Benchmark für den absoluten Vorteil der Fingersuche werden die Werte der Größe und Laufzeit der Wurzelsuche genommen.

Mit den Diagrammen aus Abbildung 6-6 konnte allerdings gezeigt werden, dass wenn man nur ein oder zwei Finger hat, also die Finger-Management-Datenstruktur sehr stark „gekürzt" ist, sich die Fingersuche unter ungünstigen Suchanfragen langsam verhält. Umgekehrt gibt es auch Suchanfragen wo die Fingersuche deutlich schneller als die Wurzelsuche arbeitet. Es kommt stark auf die Verteilung der Suchanfragen an, wenn man besonders wenige Finger zur Verfügung hat.

Mit dem Splay-Tree konnte jedoch gezeigt werden, dass man einen Vorteil über alle vorgestellten Anwendungsfälle hinweg hat. Für den Splay-Tree ist auch die Trade-off-Betrachtung deutlich einfacher, da er zuerst mit *n*-vielen Nodes startet und anschließend auf 2^a- viele „gekürzt" werden könnte.

Damit würde der „gekürzte" Splay-Tree für die Fingersuche immer einen absoluten Vorteil gegenüber der Wurzelsuche garantieren, auch bei ungünstigen Suchanfragen.

7.2 Informationen über die Beschaffenheit der Suchanfragen nutzen

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

Dass die vorgestellten Fingersuchen die Information über das Wesen der Suchanfragen nutzen konnten, konnte durch die Implementierung bestätigt werden. Wie in der Einleitung dieser Arbeit diskutiert, konnte das ungünstige Verhalten einiger Suchen mit weniger Fingern nachgewiesen werden.

Die Abbildung 6-6 bestätigt außerdem die Sicht aus der Informationstheorie, dass es einen großen Einfluss auf die Fingersuche hat, wie chaotisch die Suchanfragen sind (siehe Kapitel “Zufällige Verteilung zwischen Minimalen und Maximalem Element“). Hier konnte gezeigt werden, dass mit chaotischen Anfragen der Lazy-Finger und die Min-Max-Fingersuche ähnlich schlecht verhalten, im Vergleich zum Benchmark. Das ist besonders hervorzuheben, da der Lazy-Finger dynamisch und die Min-Max-Fingersuche, wie auch die Wurzelsuche, statisch sind.

Eine der interessanteren Finger-Management-Datenstrukturen ist der Splay-Tree, jedoch entgegen der Theorie aus dem Kapitel „Splay-Trees, Zufall und Optimalität“, dass chaotische Suchanfragen den Splay-Tree besonders negativ beeinflussen, war durch die Implementierung zusehen, dass dieser schnell ist, siehe Abbildung 6-6: Plot C5: random-key search.

Dieses ermittelte Ergebnis lässt mehrere Interpretationen zu. Einmal, dass die vorliegende Implementation des Splay-Trees fehlerhaft ist, oder dass die Zufallsfunktion in Python nicht chaotisch genug ist, um das ungünstige Verhalten zu verursachen oder vielleicht ist tatsächlich der Splay-Tree ein besonders schneller Algorithmus.

Ausgehend von der letzten Vermutung, dass der Splay-Tree auch bei chaotischen Anfragen besonders performant ist, könnte man sagen, dass diese Datenstruktur die Informationen über die

Beschaffenheit 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 einen absoluten Vorteil gegenüber der herkömmlichen Wurzelsuche bietet: der „gekürzte“ Splay-Tree.

7.3 Historische Einordnung der Finger

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

Mit dem aus der Softwaretechnik inspirierten Modell dieser Arbeit, der ausgelagerten Finger-Management-Datenstruktur, wird für jede funktionale Anforderung ein einzelnes Objekt geschaffen. Bei Anwendungen der Fingersuche ist es relevant, dass man die Verwaltung der Finger und die Teilabschnitte der Suche klar voneinander 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.

In älteren Arbeiten zu Fingern wurde bis in die 90‘er Jahre diese Trennung nicht vorgenommen und stattdessen wurde versucht durch die Finger z.B. Baumrotationen, Tree-Joins und andere aufwändige Baumoperationen einen universellen Vorteil zu erwirtschaften, siehe Kapitel “Aktueller Forschungsstand: Übersicht über Finger“.

Die neueren Arbeiten zu dem Thema der Finger hingegen versuchen sich ausschließlich auf die Fingersuche zu konzentrieren, siehe Kapitel “Aktueller Forschungsstand: Übersicht über Finger“.

Diese Arbeit reiht sich somit in die modernere Sicht auf die Finger ein. Ein Kapitel dieser Arbeit „Prädiktion auf Input Streams für Lokalitätsprinzip“ zeigt, dass es eine Sensibilisierung für das

Thema der Prädiktion der Suchanfragen gibt und dass daran gearbeitet wird das Lokalitätsprinzip zu nutzen. [14]

7.4 Fazit zur Einordnung der Ergebnisse

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

Anhand des vorgestellten Modells aus Kapitel „Das Modell“ konnte ein Fingersuchalgorithmus entworfen werden, welcher immer einen absoluten Vorteil gegenüber der Wurzelsuche bietet: der „gekürzte“ Splay-Tree.

8 Zusammenfassung der Bachelorarbeit

In dieser Arbeit wurde der aktuelle Forschungsstand betrachtet, es wurde ein Modell vorgestellt, um sich der Fingersuche zu nähern. Außerdem wurde sich theoretisch mit dem Thema der Laufzeiten auseinandergesetzt und zum Ende dieser Arbeit wurden einige Anwendungsfälle mithilfe des Modells programmiert. Es konnten einige Vermutungen, wie sich einzelne Fingersuch-Algorithmen in der Simulation verhalten werden, belegt werden. Es gab auch eine kurze historische Trend-Einordnung der Sicht auf die Finger.

Es wurde die Frage beantwortet, dass der beste Trade-off für die Anzahl der Finger darin liegt, die Finger-Management-Datenstruktur zu „kürzen“. Der beste Trade-off wird durch die Höhe der Startpunkte in der Distanzsuche beschränkt. In dem Zusammenhang wurde der „gekürzte“ Splay-Tree als ein Algorithmus, welcher einen absoluten Vorteil gegenüber der Wurzelsuche bietet, aus dem Modell abgeleitet.

Es konnte gezeigt werden, dass die drei Fingersuch-Algorithmen Lazy-Finger, Min-Max-Finger und Splay-Tree Fingersuche die Information über die Struktur der vorhergegangenen Suchaufträgen in sich speichern und für die zukünftigen Suchaufträge nutzen. Anhand von fünf Suchverteilungen wurde dieses Verhalten gezeigt. Der Splay-Tree hat sich hierbei als besonders vorteilhaft erwiesen.

9 Appendix

9.1 Ausblick und Potential

9.1.1 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 um 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 [18] am häufigsten genutzt werden:

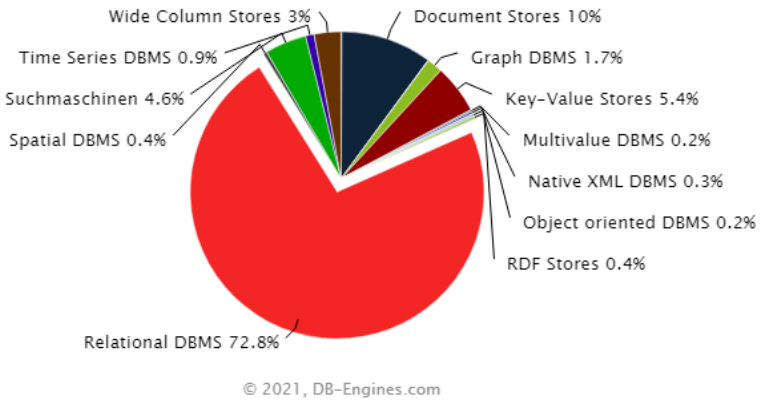

Abbildung 9-1 häufigste Datenbankkategorien [18]

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:

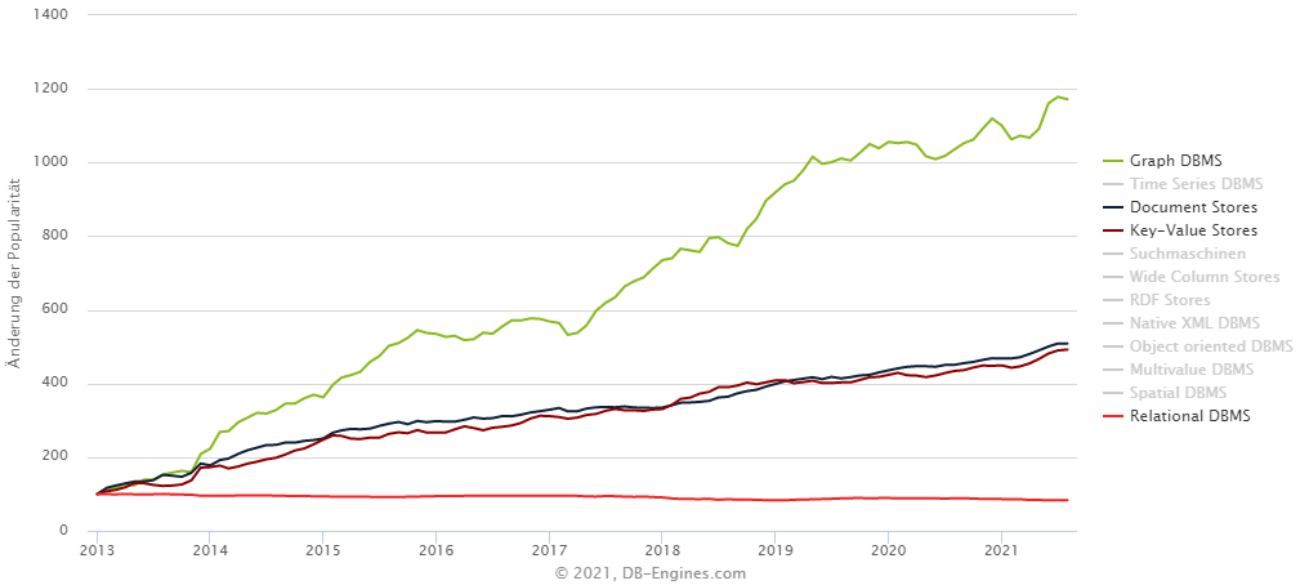

Abbildung 9-2 Die vier beliebtesten Datenbankmodelle: Absolute Anzahl seit Januar 2013 [18]

Man kann aus der Abbildung 9-2 ablesen, 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. Die Daten in Nodes werden mit „eindeutig bezeichneten und identifizierbaren Datenentitäten“ und Kanten gespeichert [19]. Solche Nodes könnten demnach einen unique Identifier oder Key generiert bekommen und damit wäre die Fingersuche ebenfalls nutzbar.

Aus dieser Analyse kann geschlussfolgert werden, dass die Fingersuche ein hervorragendes Konzept ist, um es auf den beliebtesten und zukunftsträchtigsten Datenbankmodellen anzuwenden.

9.1.2 Finger und Hardwaretechnologie

Angesichts des Trends [20] 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 Finger-Managementsystem auf GPU, CPU laufen lassen oder sogar die Anwendung im Bereich von Quantencomputern evaluieren.

Beispielsweise laufen Splay-Trees besonders gut auf GPU's, da diese besonders viele parallele Vergleiche ausführen können [15]. 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 [13] 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.

9.2 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.

9.2.1 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 erscheinen daher als kontraintuitiv für eine minimalistische Datenstruktur.

9.2.2 Radix-Tree / Präfix-Tree

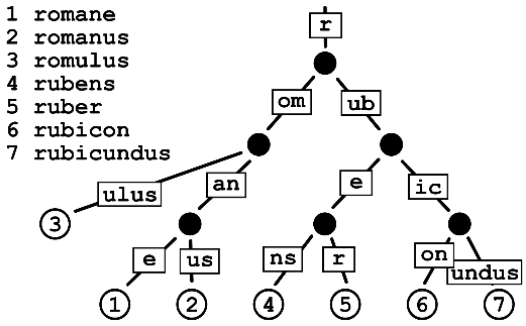

Abbildung 9-3 Radix-Tree, Präfix-Tree or Trie [21]

Der Präfix-Tree ist, wie der Name schon andeutet, ist ein Tree der sich den Pfad durch die Datenstruktur zunutze macht wie in Abbildung 9-3 [21]. Hier einen Finger mitten in der Datenstruktur suchen zulassen, wäre kontraproduktiv, da man den Pfadanfang mit vergleichen muss.

Im Prinzip würde der Finger die übergebenen Präfixe mit dem zu suchenden Node abgleichen, was einer Menge von Vergleichsoperationen, wie dem Durchlaufen von der Wurzel aus, stark ähnelt.

9.3 Code für die Simulation

Dieser Code wurde konform zur Prüfungsordnung mit einer GNU lesser general public license veröffentlicht:

https://github.com/AnnaHannah/ba_abgabe.git

9.3.1 Spezifikation des Vorgehens

Erster Schritt – Hauptdatenstruktur initiieren:

- Einen balancierten bi-direktional verketteten Binärbaum (rotschwarz-Baum) mit den Methoden: insert, delete und down-search sowie two-directional-search implementieren.
- Die Hauptdatenstruktur wird mit natürlichen Zahlen, den Keys, befüllt (1,2,3 …, n).

Zweiter Schritt – Finger initiieren:

- Je nach Finger-Managementsystem (Lazy-Finger, Min-Max-Finger, Splay-Tree) werden die passenden Finger in der Hauptdatenstruktur gesetzt.

Dritter Schritt – Suchaufträge für Finger initiieren:

- Verteilung der zu suchenden Keys, also Listen mit Zahlenfolgen, erstellen

Vierter Schritt – Wissensgewinn

- Fingersuche mit den Listen ausführen
- Geschwindigkeitsvergleich mit unterschiedlichen Finger-Management-Systemen
- Graphische Veranschaulichung

9.3.2 Klassendiagram

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

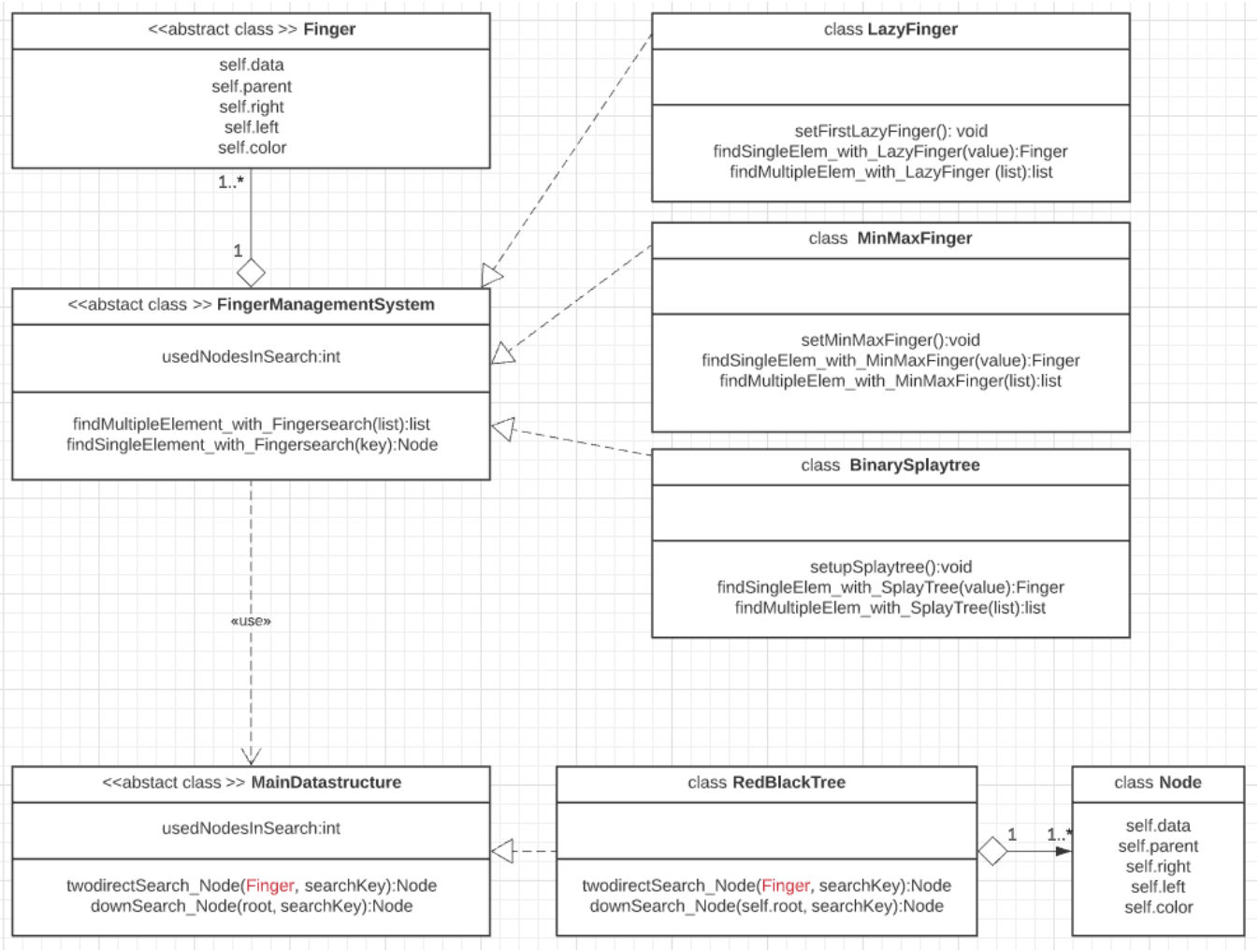

Abbildung 9-4 Klassendiagram der Simulation

Die interessantesten Methoden sind:

- BinarySplay-Tree.*findMultipleElem_with_Splay-Tree*,
- 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 es in der Version, dass nur das *Value*, entspricht *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 durch Finger oder Methoden verwendet wurde („touched“).

9.3.3 Ausführen des Codes

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

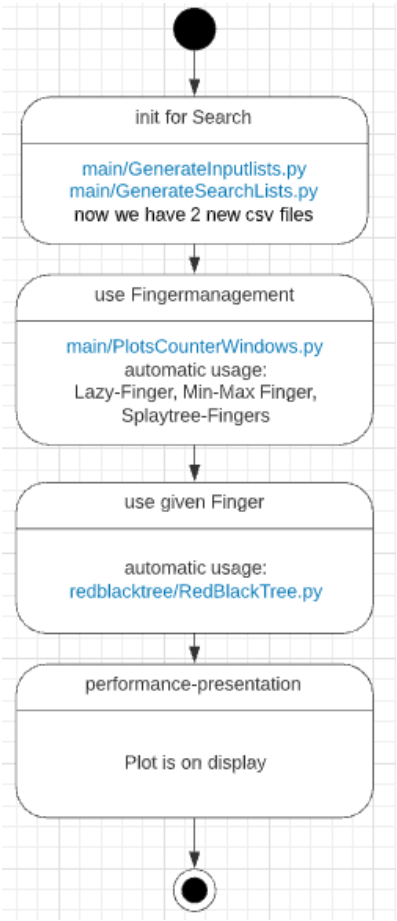

Abbildung 9-5 Aktivitätsdiagram für Nutzer

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

Diese Plots sind die gleichen verwendeten Plots in dieser Bachelorarbeit.

9.4 Begriffe

9.4.1 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. [22]

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 [22].

Besonders häufig 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 [22].

In diesem Fall wird häufig versucht, die Größe und Form des Bereichs um die aktuelle Referenz herum zu nutzen, für den es sich lohnt, einen schnelleren Zugriff für eine spätere Referenz vorzubereiten. [22]

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

9.4.2 Daten/Values

Daten sind durch Beobachtungen, Messungen, statistische Erhebungen u.a. gewonnene Zahlenwerte, welche auf Beobachtungen, Messungen, statistischen Erhebungen oder Angaben formulierbare Befunde sind. [23]

In meiner Arbeit wird vor allem die ordinale Eigenschaft von Daten/Values vorausgesetzt. Ordinal bedeutet, dass man die Daten ordnen oder einer Reihenfolge zuweisen kann. Deshalb haben Datenwerte/Values jeweils einen Key, diese Keys unterliegen der ordinalen Eigenschaft und sind in dieser Arbeit natürliche Zahlen, wie in Abbildung 9-6 angedeutet.

Insbesondere kann man über die Keys aussagen, dass wenn sie verglichen werden, es möglich ist, dass die Keys entweder im Verhältnis größer, gleich oder kleiner zueinander zustellen. [24] [25]

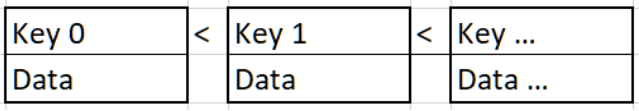

Abbildung 9-6 Ordinäre Eigenschaft der Key

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

9.4.3 Datenstruktur

„Wie für viele fundamentale Begriffe der Informatik gibt es auch für […] Algorithmen und Datenstrukturen, nicht eine einzige, scharfe, allgemein akzeptierte Definition“ [26, 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.“ [27]

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

9.4.4 Bäume und binäre Bäume

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, siehe Abbildung 9-7 [28].

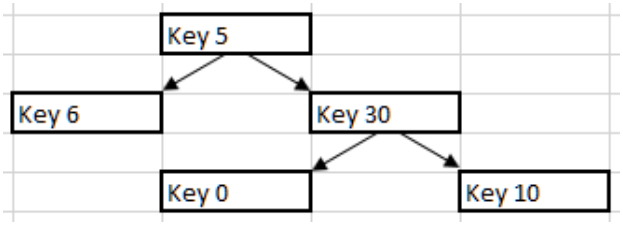

Abbildung 9-7 Schematische Darstellung Baum

Binärbäume sind die am häufigsten verwendete Unterart der Bäume. Im Gegensatz zu anderen Arten von Bäumen können die Nodes eines Binärbaumes nur höchstens zwei direkte Nachkommen haben. Meistens werden an den jeweiligen Verbindungen oder Pointern ordinäre Vergleiche verwendet, wie größer oder kleiner als die aktuelle Node, siehe Abbildung 9-8.

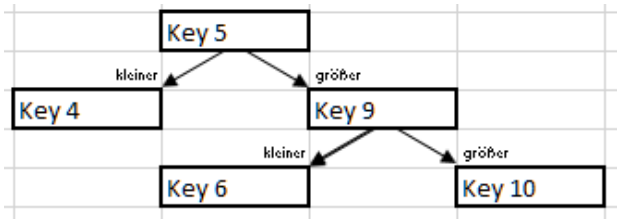

Abbildung 9-8 Schematische Darstellung binär Baum

Die Binärsuche startet in der Wurzel, dem obersten Node, und folgt den Pointern zu den Kindern, den untergeordneten Nodes. 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 Nodes ist. In meiner Implementierung haben des Weiteren die Funktionen Insert und Delete eine Worstcase Laufzeit von O (1), wobei diesen beiden Operationen u.U. eine Suche vorgeschaltet ist.

9.4.5 Rot-Schwarz-Baum

Rotschwarz-Bäume sind eine spezielle Art der Binärbäume, wie in Abbildung 9-9 skizziert. Die enthaltenen Nodes 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 Nodefarben ausnutzt und deshalb schneller ist. [29]

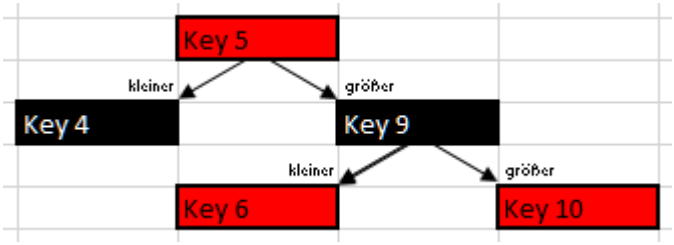

Abbildung 9-9 Schematische Darstellung Rot-Schwarz-Baum

9.4.6 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.

9.4.7 Splay-Tree

Im Originalpaper von Daniel Sleator and Robert Tarjan [16] wird der Splay-Tree als “eine sich selbst anpassende Form des binären Suchbaums“ benannt. „Die Effizienz von Splay-Trees 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. In den unteren Abbildungen ist dargestellt, einmal der Splay-Tree vor der Suche nach der 7, nach der Suche nach der 7 und anschließend nach der Suche nach der 4.

Man kann erkennen, dass die Splay-Operation das jeweilige Suchergebnis in die Wurzel gestellt hat. [30]

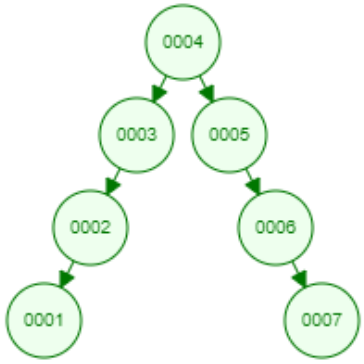

Abbildung 9-5 Aktueller Splay Tree [30]

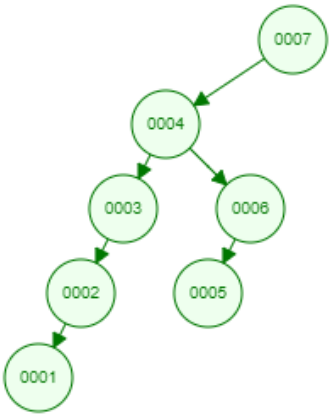

Abbildung 9-6 Nachdem 7 gefunden wurde [30]

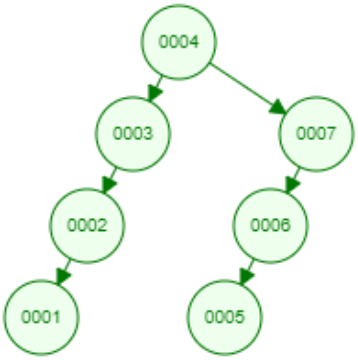

Abbildung 9-7 Nachdem 4 gefunden wurde [30]

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, dass auch über alle Worstcase-Sequenzen von Operationen gemittelt wurde. [16]

Daher sind Splay-Trees genauso effizient wie balancierte Bäume, wenn die amortisierte Laufzeit von Interesse ist. Außerdem sind Splay-Trees für ausreichend lange Zugriffsfolgen bis auf einen konstanten Faktor genauso effizient wie statische optimale binäre Suchbäume. [16]

10 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] Hee-Kap Ahn and Chan-Su Shin, „Algorithms and Computation,“ 25th International Symposium, ISAAC 2014, Jeonju, Korea, 15-17 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)].

[3] 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)].

[4] Hee-Kap Ahn and Chan-Su Shin, „Algorithms and Computation,“ 25th International Symposium, ISAAC 2014, Jeonju, Korea, 15-17 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)].

[5] 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)].

[6] 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)].

[7] Wikipedia, „Splay_tree," [Online]. Available: https://en.wikipedia.org/wiki/Splay_tree . [Zugriff am 03. 06. 2021 (13:50 Uhr)].

[8] Wikipedia , „Splay-Baum," [Online]. Available: https://de.wikipedia.org/wiki/Splay-Baum. [Zugriff am 03. 06. 2021 (14:00 Uhr)].

[9] 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)].

[10] Wikipedia, „k-server Problem," [Online]. Available: https://en.wikipedia.org/wiki/K-server_problem#:~:text=The%20k%20-server%20problem%20is%20a%20problem%20of,a%20set%20of%20k%20servers%2C%20represented%20as%20 . [Zugriff am 19 08 2021 (20:01 Uhr)].

[11] 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)].

[12] 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)].

[13] 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)].

[14] 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 implementation p.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)].

[15] Eric K. Lee and Charles U. Martel, „When to use splay trees,“ 22 11 2006. [Online]. Available: http://www2.ee.ntu.edu.tw/~yen/courses/ds07/splay-tree-exp.pdf. [Zugriff am 19 08 2021 (14:11 Uhr)].

[16] 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)].

[17] 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)].

[18] 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)].

[19] Wikipedia, „Radix tree, Patricia tree, Trie,“ [Online]. Available: https://en.wikipedia.org/wiki/Radix_tree. [Zugriff am 04. 08. 2021 (21:13 Uhr)].

[20] Wikipedia, „Lokalitätsprinzip - Locality of reference,“ [Online]. Available: https://en.wikipedia.org/wiki/Locality_of_reference. [Zugriff am 12. 08. 2021 (18:10 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] 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)].

[31] 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)].

[32] 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)].

[33] 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)].