Schwierigkeiten bei Implementierung und Evaluation von Datenstrukturen in Datenbanksystemen

Anton Rodenwald (19)

11. Januar 2024

Fachgebiet:	Mathematik/Informatik
Wettbewerbssparte:	Jugend Forscht
Bundesland:	Niedersachsen
Wettbewerbsjahr:	2024
Thema des Projektes:	In meinem Projekt wollte ich verschiedene in Datenbanksystemen genutzte Datenstrukturen implementieren und testen. Dabei betrachtete ich auch unterschiedliche Ansätze der Datenspeicherung und für den Datenzugriff. Bei den auftretenden Schwierigkeiten musste ich Lösungen finden bzw. etwas adaptieren.
Projektbetreuerin:	Birgit Ziegenmeyer
Institution:	Schillerschule Hannover

Kurzfassung

<Text>

Inhaltsverzeichnis

1 Motivation, Fragestellung und Ziel

Im Rahmen des Informatik Leistungkurses meiner Schule haben wir (Mitschüler und ich) uns mit Datenbanken und Modellierung beschäftigt. Da jedoch auf die Funktionsweise einer Datenbank nicht weiter eingegangen wurde und ich auch schon von Unterschiedlichen Datenbankansätzen, genauer Relationalen und Nicht-Relationalen, gehört hatte, wollte ich mich ihm Rahmen meines Projektes genauer damit beschäftigen. Dazu wollte ich eine einfache Datenbank umsetzen.

Meine Forschungsfrage ist deshalb, wie die Wahl der Datenstruktur die Geschwindigkeit einer Datenbank beeinflusst und inwiefern sich eine Datenstruktur für gewisse Daten besser oder schlechter eignet.

Mein Ursprüngliches Ziel war dabei die Programmierung einer Datenbank mit den Datenstrukturen R-Tree, B-Tree, Binary-Tree und Graphen. Davon konnte ich leider aufgrund der hohen Komplexität (Stand jetzt) vieles nicht umsetzen.

2 Hintergrund

2.1 Arten von Daten und passende Datenstrukturen

Um mir einen Überblick über das Themengebiet zu verschaffen recherchierte ich zuerst im Internet. Dabei fand ich viele Informationen über die Arten von Daten und Datenbanksystemen, allerdings nichts konkretes über Performance Vergleiche dieser. Verschiedene Datenarten sind dabei räumliche, zeitliche, als Objekte modellierbare und stark vernetzte Daten [Unk23] [JPA+12].

Räumliche Daten sind z. B. Kartendaten (Google Maps, OpenStreetMap). Bestimmte Landmarken oder Objekte (Bäume, Mulleimer) lassen sich beispielsweise als Punkte auf einer Fläche modellieren. Es müssen dann effizient Fragen wie z. B. “Wie viele Mülleimer gibt es in diesem Park?” oder “Was sind die 3 am besten bewerteten Restaurants in der Innenstadt?” beantwortet werden. Eine dafür ausgelegte Datenstruktur ist der R-Tree.

Zeitliche Daten können z. B. die von einem Sensor gemessene Temperaturen sein. Wichtig wäre dann z. B., dass man einfach Statistiken wie Durchschnittswerte bilden oder Anomalien erkennen könnte.

Daten, die klassischerweise in relationellen Datenbanken gespeichert werden, lassen

sich durch viele, gleich aufgebaute Objekte modellieren lassen. Beispielsweise speichert ein Unternehmen über Kunden immer die gleichen Daten (Name, E-Mail, ...). Jeder Datensatz kann dann eindeutig z. B. über eine ID identifiziert. Eine dafür geeignete Datenstruktur ist der B-Tree, eine dem Binary-Tree ähnliche Datenstruktur.

Wenn zwischen einzelnen Daten wie z. B. jedem Nutzer einer Social Media Plattform allerdings viele Beziehungen bestehen (Follower, Likes), so eignet sich ein Graph als Datenstruktur besser.

2.2 Arten von Datenbanksystemen

Moderne Datenbanksysteme ermöglichen meist, mit einem System verschiedene Speicherungsformen zu nutzen. Trotzdem muss man sich bei einem Multi-modalen System natürlich überlegen, welche Daten man nun in welcher Art von Datenstruktur speichern will. Die Website db-engines.com stellt z. B. eine Übersicht über die aktuell beliebtesten Datenbanksysteme und deren Modalitäten dar.

3 Vorgehen

3.1 Schwierigkeiten bei der Implementierung

Mein geplantes Vorgehen bestand darin, ein Datenbanksystem mit diesen verschiedenen Datenstrukturen in C++ umzusetzen. Während dem Entwicklungsprozess und der Implementierung zeigte sich allerdings die zunehmende Komplexität, weswegen ich nur begrenzt die verschiedenen Datenstrukturen vergleichen konnte. Trotzdem werde die angestrebten Datenstrukturen zumindest in ihrer Funktion erläutern und meine Schwierigkeiten bei der Umsetzung beschreiben, sodass sie anderen vielleicht nicht passieren. Auch werde ich meine dann tatsächlich durchgeführten Tests beschreiben. Also ein normaler Entwicklungsprozess ganz nach dem Zitat

> »we do these things not because they are easy,
> but because we thought they were going to be easy «

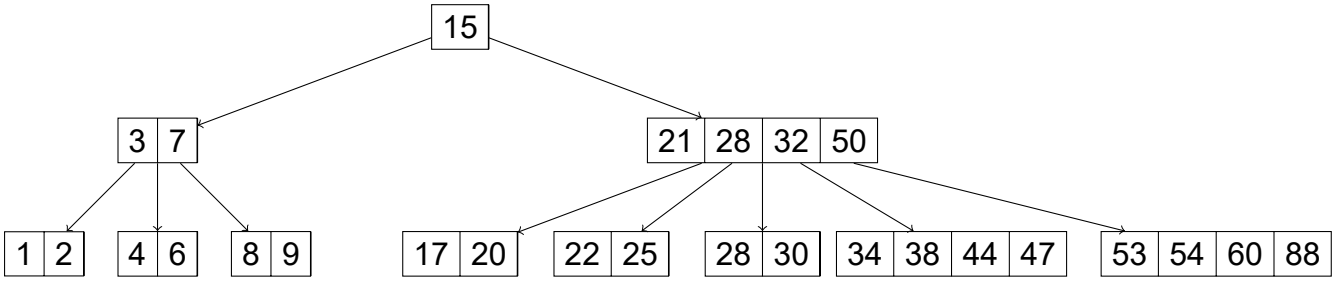
3.1.1 Beej und <sys/socket.h>

Zu Beginn meines Projektes folgte ich einem guten Tutorial von Brian Hall, um Kommunikation über Linux Websockets umzusetzen, da Datenbanksysteme eine Schnittstelle zu anderen Programmen benötigen. Da mein Datenbank-Backend allerdings nicht rechtzeitig ausgereift genug war, konnte ich diese Schnittstelle noch nicht in Kombination mit dem Rest testen.

3.1.2 Probleme bei B-Tree und Binary-Tree

Da der B-Tree eine in vielen Datenbanken verwendete Datenstruktur ist, wollte ich diesen zuerst implementieren. Als Grundlage nahm ich die erste Veröffentlichung über diese durch 2 Boeing Entwickler [BM70].

Der B-Tree ist dabei ähnlich zu einem Binären Suchbaum. Der Hauptunterschied ist, das jeder Node nicht nur 2 Child Nodes hat, wodurch der Baum weniger tief ist. Um Elemente zu Suchen, zu Löschen oder zu Ändern folgt man der Baumstruktur vom Root Node ausgehend. Durch Vergleiche innerhalb der Nodes und das traversieren der Äste findet man ein bestimmtes Elemente oder eine Position im Baum.



Bei der Implementation des B-Tree traten allerdings einige Probleme auf. Der Hauptgrund, warum ich mich dafür entschied, andere Datenstrukturen zu testen war, dass die Ausbalancierung beim B-Tree relativ komplex ist. Auch entschied ich mich, meine Tests ausschließlich mit im RAM liegenden Daten durchzuführen, da eine Speicherung auf einer Festplatte noch mehr Komplexität bedeutet hätte.

Anschließend implementierte ich als Alternative dann einen AVL-Tree, eine sich selbst ausbalancierende Variante des Binären Suchbaums (BST). Beim anschließenden konzeptionieren der Performance-Tests entdeckte ich allerdings noch einen beim einfügen von Elementen auftretenden Fehler, weswegen ich mich entschied, auch diesen nicht zu testen.

3.2 Testdaten und Test-Datenstrukturen

3.2.1 Testdaten

Bei den Testdaten handelt es sich um eine fiktive Schülerschaft, in der jeder Schüler Freundschaften und Feindschaften hat. Diese Testdaten generierte ich mir mit einem Python Script und speicherte diese in einer CSV-Datei. Jeder Schüler hat dabei eine ID und einen Namen, der eine zufällige Zeichenkette ist. Anschließend sind bei jedem Schüler noch 100 Freund- und Feindschaften eingetragen.

```
0, eszycidpyopumzgd, 255777-14862-..., 204372-226894-...
1, idixqgtnahamebxf, 233658-369416-..., 265451-355559-...
2, digztyrwpvlifrgj, 104446-129174-..., 188986-42660-...
...
```

3.2.2 Testanforderungen

Bei meinen Performance Tests verglich ich die Suchgeschwindigkeit der Datenstrukturen. Die gleichen Testdaten wurden in beide Datenstrukturen eingelesen und anschließend wurde gemessen, wie schnell auf einen bestimmten Teil der Daten jeweils zugegriffen werden konnte. Einmal sollten alle Freunde von Schüler Nr. 2 und einmal alle Feinschaften von Schüler Nr. 1 gefunden werden.

3.2.3 Such-Array

Auch aufgrund der vorigen Schwierigkeiten testete ich nun nicht mit einem B-Tree oder Binary-Tree, sondern wendete die Binäre Suche auf ein sortiertes Array an. Prinzipiell nutzen aber alle 3 Datenstrukturen die Binäre Suche. Die Testdaten sind in diesem Fall auf 3 Such-Arrays (vgl. Tabellen) aufgeteilt, jeweils eine für Schüler, Freundschaften und Feindschaften.

Schüler_1_ID	Schüler_2_ID
5	7
5	75
5	456
5	45

Schüler_1_ID	Schüler_2_ID
7	629
7	762
7	229
7	29435

Tabelle 1: Tabellenausschnitte Freundschaften/Feindschaften

Um alle Freunde/Feinde eines Schülers zu finden, müssen in den als Assoziationstabellen fungierenden, sortierten Such-Arrays alle Datensätze zwischen dem kleinstmög-

lichen und größtmöglichen Wert gefunden Werten. Beispielsweise für den Schüler 5 alles zwischen den Datensätzen $5|0$ und $5|m$ wenn m die Anzahl an Schülern ist. Da das Such-Array nach beiden Spalten sortiert ist, muss nur mit der Binären Suche das linke und rechte Ende des Bereichs bestimmt werden. Anschließend können in linearer Zeit alle gesuchten Datensätze ausgelesen werden.

```
std::vector<void*> Sorted_Array::search_between_keys(void* key_left, void* key_right) {
    Search_Result res_l = this->search_for_key(key_left);
    Search_Result res_r = this->search_for_key(key_right);

    int left_start = res_l.index;
    int right_start = res_r.index;

    if (res_l.status == 0 &&
        compare_keys(this->elements.at(res_l.index).key, key_left, this->key_attribute_lengths) == 1)
        ++left_start;
    if (res_r.status == 0 &&
        compare_keys(this->elements.at(res_r.index).key, key_right, this->key_attribute_lengths) == -1)
        --right_start;

    std::vector<void*> data_ptrs;
    for (int i = left_start; i < right_start + 1; ++i) {
        data_ptrs.emplace_back(this->elements.at(i).data);
    }
    return data_ptrs;
}
```

Abbildung 1: C++ Code der Such-Array-Suchfunktion

3.2.4 Graph

Bei meiner Implementierung einer Graph-Datenstruktur sind in jedem Node die Daten und die Verbindungen zu anderen Nodes gespeichert. Um alle Freundschaften oder Feindschaften zu erhalten muss nur über diese Daten iteriert werden und man erhält schnell die Daten der befreundeten oder befeindeten Schüler. Die Art der Verbindung (Freund/Feind) kann dabei durch das $link\_schema$ festgelegt werden.

```
std::vector<Link> Graph::query_links(Node* node, int link_schema){
    std::vector<Link> matching_links;
    for(const auto link : node->links){
        if (link.link_schema == link_schema){
            matching_links.push_back(link);
        }
    }
    return matching_links;
}
```

Abbildung 2: C++ Code der Graph-Suchfunktion

3.3 Komplexitäts Analyse

4 Messwerte

Meine Messungen führte ich auf meinem Desktop PC und meinem Laptop (Plugged-In) durch. Auf jedem Computer nahm ich 2 Messreihen auf. Jede der 4 Anfragen and die 2 Datenstrukturen wurde jeweils 50-mal direkt hintereinander ausgeführt. SA beschreibt die Datenstruktur Such-Array. G stetht für Graph. Das F beschreibt die Suche nach allen Freunden des Schülers Nr. 2. Das H beschreibt die Suche nach allen Feindschaften (Hostilities) des Schülers Nr. 1.

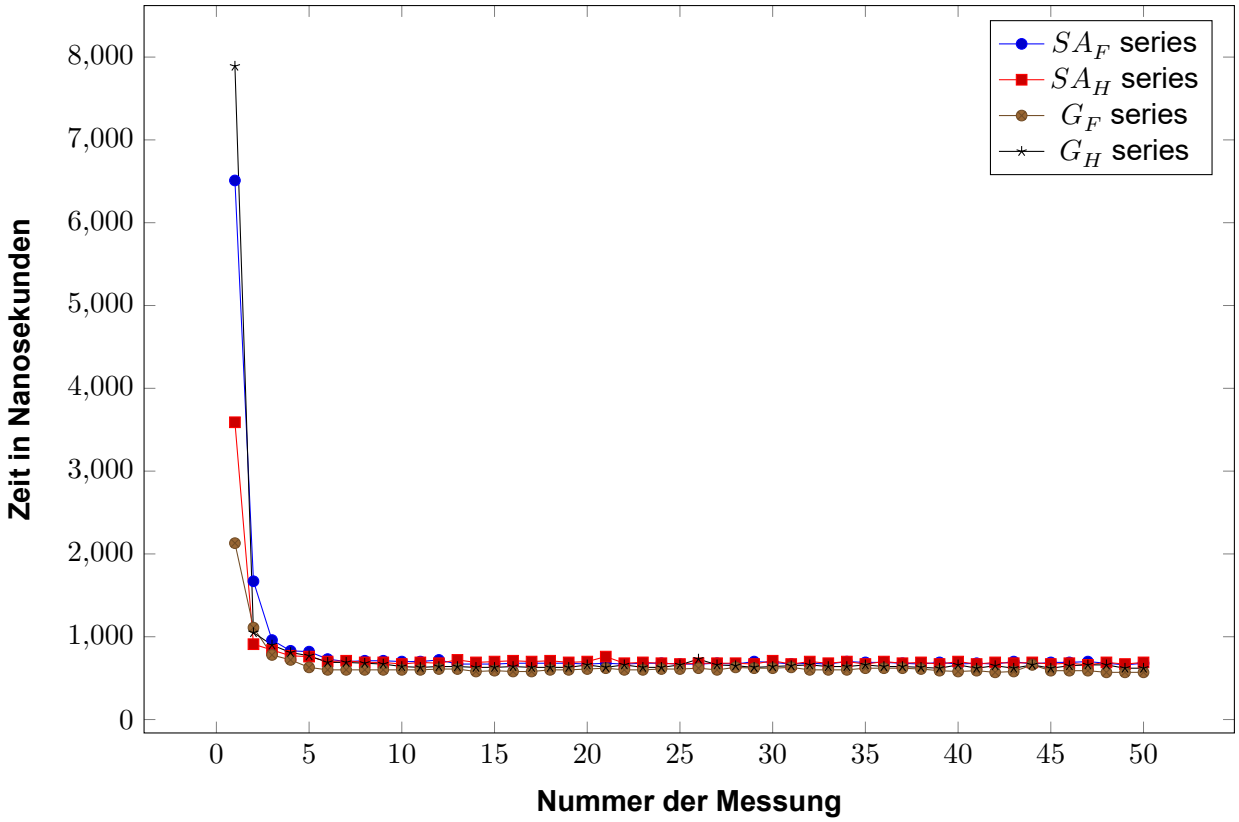


Abbildung 3: 1. Messreihe am PC (Ryzen 7 2700)

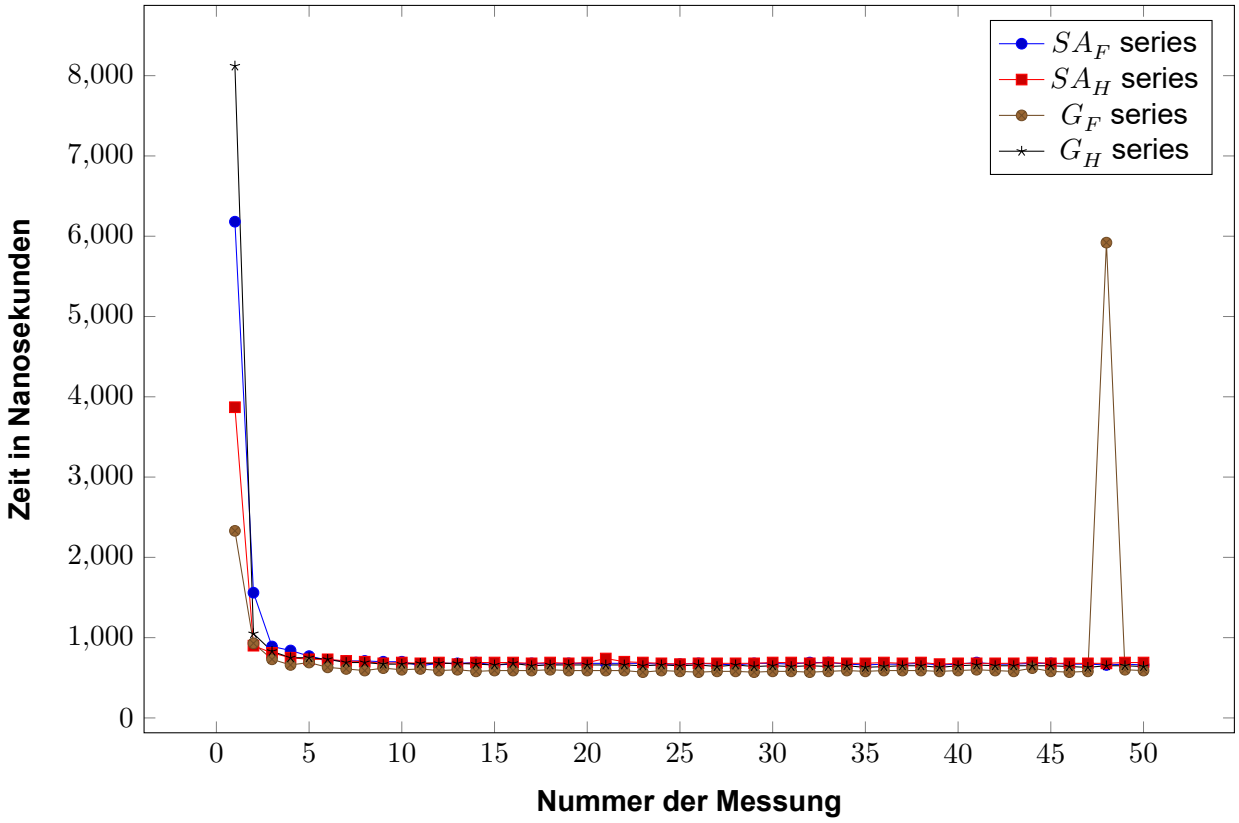


Abbildung 4: 2. Messreihe am PC (Ryzen 7 2700)

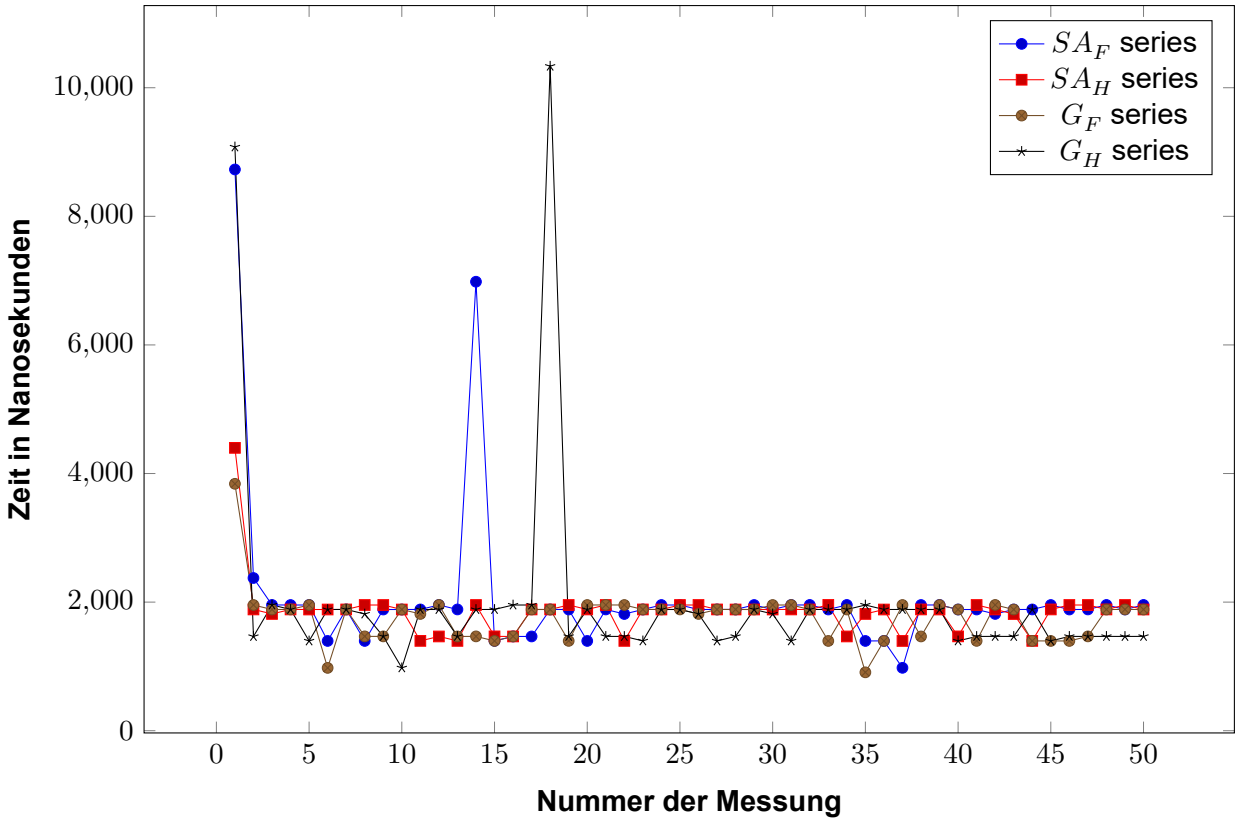


Abbildung 5: 1. Messreihe am Laptop (Ryzen 5 5500U)

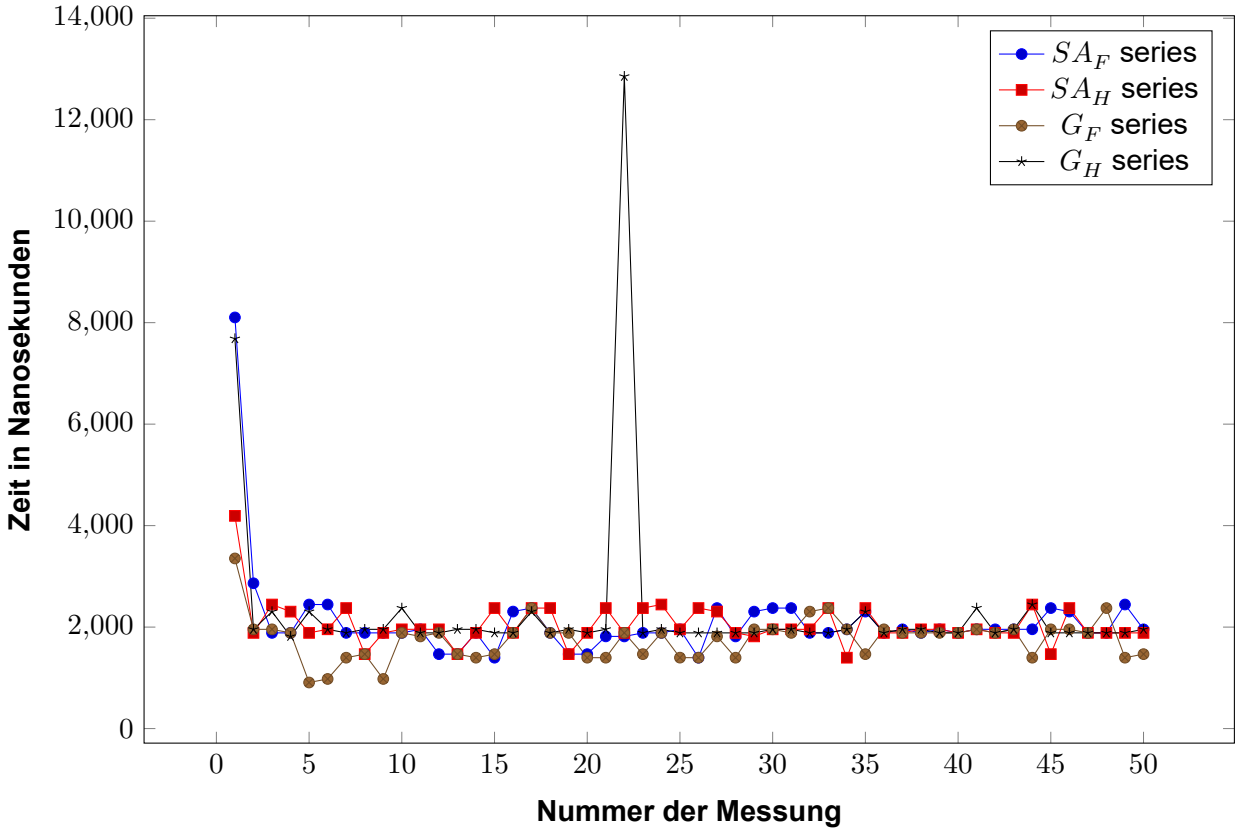


Abbildung 6: 2. Messreihe am Laptop (Ryzen 5 5500U)

5 Deutung der Messwerte

Bei den Messwerten am PC lässt sich erkennen, dass in den ersten paar Messungen die benötigte Zeit deutlich höher ist als bei den restlichen, nachfolgenden Messungen. Die benötigte Zeit aller Tests pendelt sich auf einem niedrigeren Niveau ein. Gleiches ist auch am Laptop zu beobachten, wo die Messpunkte jedoch unregelmäßiger sind. Dies lässt sich vermutlich auf eine Speicherung von Werten im CPU-Cache zurückführen, in dem nach den ersten Durchläufen Daten vorgehalten werden, was zu besseren Messergebnissen führt. Vereinzelt gibt es auch in diesen Bereichen bei PC und Laptop ausreißende Werte. Diese sind wahrscheinlich Messanomalien, erzeugt durch einmahlige Lastspitzen verursacht durch andere Prozesse auf dem Testsystem oder durch Hardwarebedingte Faktoren. Betrachtet man die 1. Messung, wo die Werte noch auseinanderliegen, so lässt sich nicht feststellen, dass eine der Datenstrukturen schneller ist als die anderen. Interessant ist bei diesen ersten Messwerten, dass die Messungen auf gleich strukturierte Daten (Freundschaften und Feindschaften) deutlich voneinander Abweichen. Nicht überraschend sind die abweichenden Niveaus von PC und Laptop, da bei CPUs mit unterschiedlichem Leistungsniveau logischerweise auch andere Performance-Werte erzeugt werden.

6 Beantwortung der Forschungsfrage

Abschließend lässt sich die Forschungsfrage insofern beantworten, als dass die Wahl der Datenstruktur in diesem Fall keinen Einfluss auf die Zugriffsgeschwindigkeit hat. Keine der Datenstrukturen scheint der anderen Überlegen.

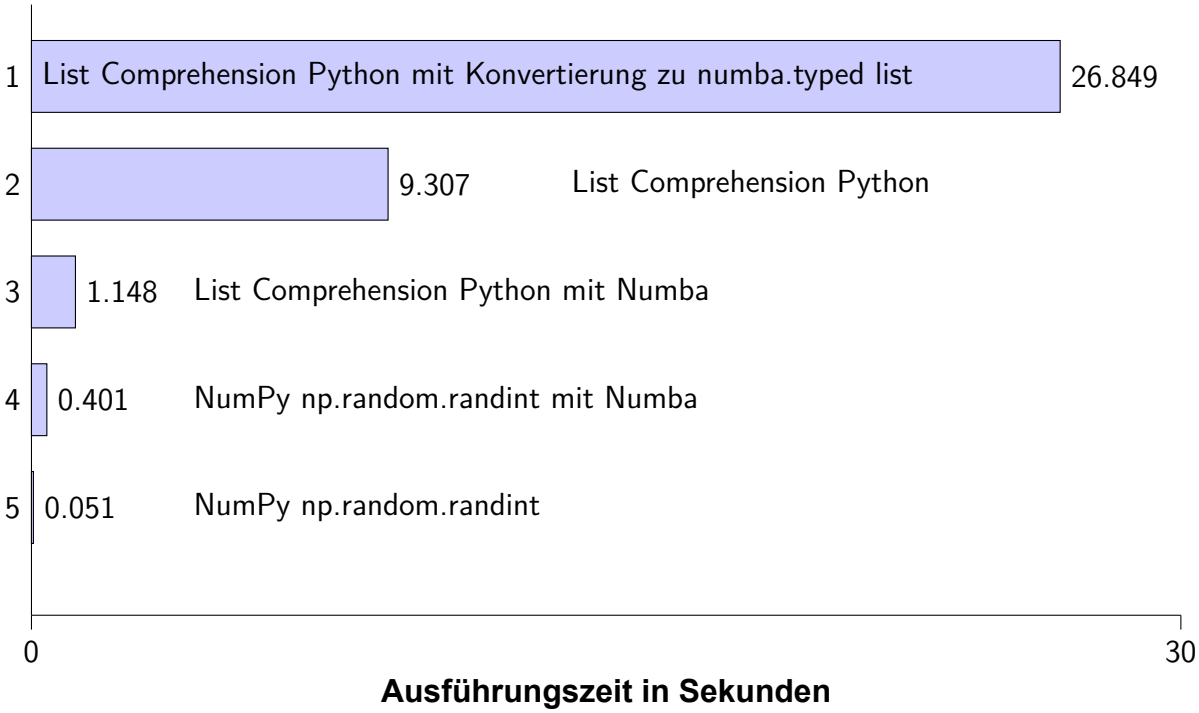
7 Schwierigkeiten und Limitationen

implementation nur ram AMDuProf hat nicht funktioniert, weil keine ahnung

8 fazit

gelernt gdb C/C++ experience mit gdb b tree socket api linux

9 ausblick interessen geweckt



[BM70] [JPA+12] [Hip08] [Unk24]

10 Quellenangaben

https://beej.us/guide/bgnet/html/

Literaturverzeichnis

[BM70] R. Bayer und E. McCreight. „Organization and Maintenance of large ordered Indices“. In: (1970). URL: https://dl.acm.org/doi/10.1145/1734663.1734671.

[Hip08] D. Richard Hipp. *How SQL Database Engines Work*. 2008. URL: https://www.youtube.com/watch?v=Z_cX3bzkExE (besucht am 28. 12. 2023).

[JPA+12] Nishtha Jatana u. a. „A Survey and Comparison of Relational and Non-Relational Database“. In: (2012).

[Unk23] Unknown. *NoSQL*. 2023. URL: https://de.wikipedia.org/wiki/NoSQL (besucht am 27. 12. 2023).

[Unk24] Unknown. *AVL tree*. 2024. URL: https://en.wikipedia.org/wiki/AVL_tree.