

Graphisches Tool zur Visualisierung eines balancierten Baumes

Datenbank Projekt

des Studiengangs Informatik

an der Dualen Hochschule Baden-Württemberg Stuttgart

von

Tim Steiner, Marius Wergen und Thorben Schabel

Juni 2023

Bearbeitungszeitraum
Matrikelnummer, Kurs
Ausbildungsfirma
Gutachter

08.03.2022 bis 28.05.2023
7828738, 1128309, 6317097, INF21E
Mercedes-Benz Group AG, Nokia Corporation,
Abbas Rashidi

Inhaltsverzeichnis

Abkürzungsverzeichnis	II
Glossar	III
Abbildungsverzeichnis	IV
1 Einleitung	1
1.1 B-Baum	2
1.2 Vorgehen	3
1.3 Installation	4
2 Architektur	5
3 Backend	7
3.1 Aufbau	7
3.2 Algorithmen	8
3.2.1 Suchen	8
3.2.2 Einfügen	8
3.2.3 Löschen	9
4 Frontend	10
4.1 Aufbau	10
4.1.1 Schichten	10
4.1.2 Bedienelemente	11
4.1.3 Informationen für den User	14
4.2 Animationen	15
4.2.1 Darstellung des B-Baums	15
4.2.2 Suchen und Löschen	16
4.2.3 Einfügen	16
4.2.4 Darstellung breiter Graphen	17
5 Ausblick	18

Abkürzungsverzeichnis

GUI	Graphical User Interface
CSV	Comma-Separated value file
DBS	Datenbanksystem

Glossar

B-Baum Balancierter Baum.

balancieren Eigenschaften des B-Baumes wiederherstellen.

Id Identifikationsnummer anhand welcher jede Node eindeutig identifiziert werden kann.

iGraph Python Library zur Visualisierung von Graphen.

Indexstrukturen Struktur in welcher der Index eines Datensatzes abgelegt ist. Kann durch einen B-Baum realisiert werden.

Jira Organisationstool für Softwareentwicklung.

Matplotlib Plotting Framework für Python.

Overflow Knoten in einem B-Baum hat mehr als $2k$ Schlüssel.

Python Programmiersprache.

Root Wurzelknoten des B-Baumes.

Tkinter GUI Framework für Python.

Abbildungsverzeichnis

1.1	Korrektter B-Baum	2
1.2	.exe Datei	4
2.1	Klassendiagramm	5
4.1	Drei Schichten: <i>a)</i> Tkinter -Fenster, <i>b)</i> Matplotlib -Frame, <i>c)</i> iGraph -Canvas	11
4.2	Einfacher Modus	11
4.3	Comma-Separated value file (CSV)-Modus	12
4.4	CSV -Bestätigungs-Fenster	12
4.5	Zufallsmodus	13
4.6	Ordnung verändern	13
4.7	Fehlermeldung bei falscher Eingabe	14
4.8	Anzeige der aktuell ausgeführten Operation	14
4.9	Anzeige, dass gesuchter Schlüssel gefunden wurde	14
4.10	Anzeige der Anzahl an Seitenzugriffen einer Operation	15
4.11	Anzeige des Dateinamens beim Speichern des B-Baums	15
4.12	Visualisierung des B-Baums	16
4.13	Anzeige der Schlüssel des Knotens, auf dem sich der Mauszeiger befindet	17

1 Einleitung

In jeder Anwendung, die eine Datenbank nutzt, ist es essentiell, dass Datenoperationen möglichst effizient ablaufen. Um diese Eigenschaft auch bei großen Datenbanken erfüllen zu können, nutzen viele Datenbanksysteme [Indexstrukturen](#).

Ein Index ist eine Hilfsstruktur, um Datenoperationen zu beschleunigen. Indiziert werden immer ein oder mehrere Attribute eines Datensatzes. Unterschieden werden kann bei diesen zwischen einem dichten und einem spärlichen Index. Ein dichter Index indiziert jeden Datensatz einer Seite, während ein spärlicher Index pro Seite einmal indiziert. Ein spärlicher Index kann beispielsweise ein Index über die Anfangsbuchstaben der Nachnamen der Personen in einer Tabelle sein, während ein dichter Index auf das Geburtsjahr indiziert (vgl. Tabelle 1.1).

M	→Mustermann	Max	16.08.1999	←	1999
S	→Steiner	Tim	06.07.2001	←	2001
W	Schabel	Thorben	14.09.2003	←	2003
	→Wergen	Marius	17.02.2003	←	

Tabelle 1.1: Beispiel spärlicher und dichter Index

Der Vorteil bei der Nutzung von Indizes ist, dass je nach Anfrage und Indizes deutlich schneller auf die Datensätze zugegriffen werden kann. Im obigen Beispiel wäre eine Abfrage nach allen Einträgen mit Personen, welche 2003 geboren sind, effizient durchzuführen. Auch nach Erweiterung der Tabelle um weitere Einträge, da diese ebenfalls indiziert werden. Indizes verwaltet ein Datenbanksystem in Indexstrukturen. Dies sind spezielle Datenstrukturen, um besonders schnell auf gewünschte Indizes zugreifen zu können. Eine dieser Datenstrukturen ist ein [B-Baum](#). Aufgrund der besonderen Eigenschaften dieser, eignen sie sich besonders gut als Indexstrukturen und finden häufig Anwendung in aktuellen Datenbanksystemen. Im folgenden soll eine eben solche Indexstruktur in einem Programm realisiert und dessen Nutzung und Betrieb visualisiert werden.

1.1 B-Baum

Wie bereits eingangs erläutert handelt es sich bei einem **B-Baum** um eine Indexstruktur. In ihr werden somit keine Daten gespeichert, sondern Indizes, die auf Daten verweisen. Der Grund, warum sich ein **B-Baum** als Indexstruktur besonders gut eignet, ist seine Zugriffszeit. Ein **B-Baum** hat bei der Suche nach jeglichem Eintrag eine Zugriffszeit von $\mathcal{O}(\log n)$. Somit ist auch für eine große Datenmenge eine gute Performance gewährleistet. Je nach Wahl der Indizierung können verschiedene Zugriffsarten effizient umgesetzt werden. Dazu zählen Single Match, Exact Match und Range Query.

Bei einem **B-Baum** handelt es sich um einen balancierten Baum, welcher besondere Eigenschaften erfüllt. Da nicht jede davon für die Realisierung des **B-Baumes** notwendig sind, werden nur die Nötigsten erläutert. Ein **B-Baum** besteht aus einer Menge Knoten und Kanten. Die Knoten speichern die Schlüssel und die Kanten repräsentieren die Referenzen. Die Schlüssel in einem Knoten werden in aufsteigender Reihenfolge gespeichert. Der Verzweigungsgrad k des Baumes gibt an wie viele Schlüssel mindestens in einem Knoten sein müssen. Maximal dürfen in einem Knoten $2k$ Schlüssel vorhanden sein. Zudem muss jeder Knoten mit n Schlüsseln $n + 1$ Kinder haben. Ausgenommen hier von ist die **Root** des Baumes. Diese darf zusätzlich lediglich einen Schlüssel beinhalten. Sobald eine dieser Eigenschaften nach einer Operation verletzt wurde, muss der Baum rebalanciert werden. Wann und wie diese Balancierungsoperationen funktionieren werden in einem späteren Abschnitt erläutert. Die Grundsätzliche Idee eines **B-Baumes** ist die geschickte Ausnutzung von Referenzen. Jeder Knoten speichert seine Schlüssel in aufsteigender Reihenfolge. Nun kann ein Schlüssel jeweils zwei Kinder haben. Ein Kindknoten in welchem kleinere Schlüsselemente gespeichert sind und ein Kindknoten mit größeren. Daraus ergibt sich, dass jeder Knoten insgesamt $2k + 1$ Kinder haben kann. Folgende Grafik zeigt einen korrekten **B-Baum**:

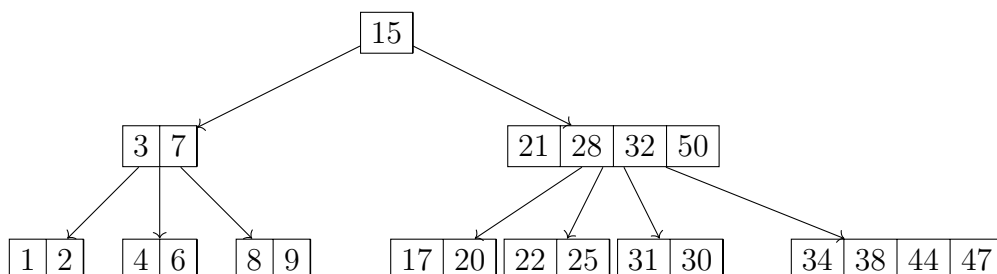


Abbildung 1.1: Korrekter **B-Baum**

Wird nun ein Schlüssel gesucht beginnt die Suche stets in der **Root** des Baumes. Von dort wird der Suchschlüssel solange mit den Schlüsseln im Knoten verglichen, bis der Suchschlüssel kleiner ist. Die Suche geht nun im linken Kindknoten des Schlüssel weiter, welcher größer ist, als der Suchschlüssel. Sollte kein Schlüssel größer sein, wird im rechtesten Kindknoten weitergesucht. Dieser Vorgang wiederholt sich so oft, bis der Schlüssel gefunden wurde, oder man in einem Knoten ohne Kinder angekommen ist.

1.2 Vorgehen

Für die Realisierung der B-Baum Visualisierung wurde **Python** gewählt. Python bietet viele eingebaute Funktionen und hilfreiche Bibliotheken. Jegliche GUI Elemente wurden mit **Tkinter** implementiert, während die Baum Visualisierung an sich aus **iGraph** und **Matplotlib** besteht. Für die Organisation und Aufgabenverteilung wurde **Jira** verwendet. Nach anfänglicher Anforderungsanalyse wurden initiale Aufgaben in Jira erstellt und verteilt. Das Projekt teilt sich in zwei große Blöcke: Frontend und Backend. Im späteren Verlauf der Dokumentation wird genauer auf diese eingegangen.

Da der Fokus des Programms auf der visuellen Darstellung eines B-Baumes liegt, wurde besonders viel Augenmerk auf das Frontend gelegt. Somit sind die Algorithmen im Backend so aufgebaut, dass sie dem Frontend die nötigen Daten liefern kann. Dies spiegelt sich jedoch in der Performance des **B-Baumes** wieder. Je höher der Baum wird, desto mehr Rechenkraft wird benötigt, um das Programm problemlos laufen zu lassen.

1.3 Installation

Um das Programm in **Python** starten zu können sind einige **Python** Bibliotheken notwendig. Vor dem ersten Start müssen folgende Befehle ausgeführt werden:

```

1  # matplotlib
2  pip install matplotlib
3
4  # tkinter
5  pip install tk
6
7  # igraph
8  pip install igraph
9
10 # numpy
11 pip install numpy

```

Um das Programm zu starten muss *main.py* ausgeführt werden.

Alternativ kann auch die *.exe* Datei ausgeführt werden, welche sich im Ordner *BTree* befindet. Dort findet sich die *BTree.exe* (vgl.1.2). Es dauert einige Sekunden bis sich die Datei öffnet.

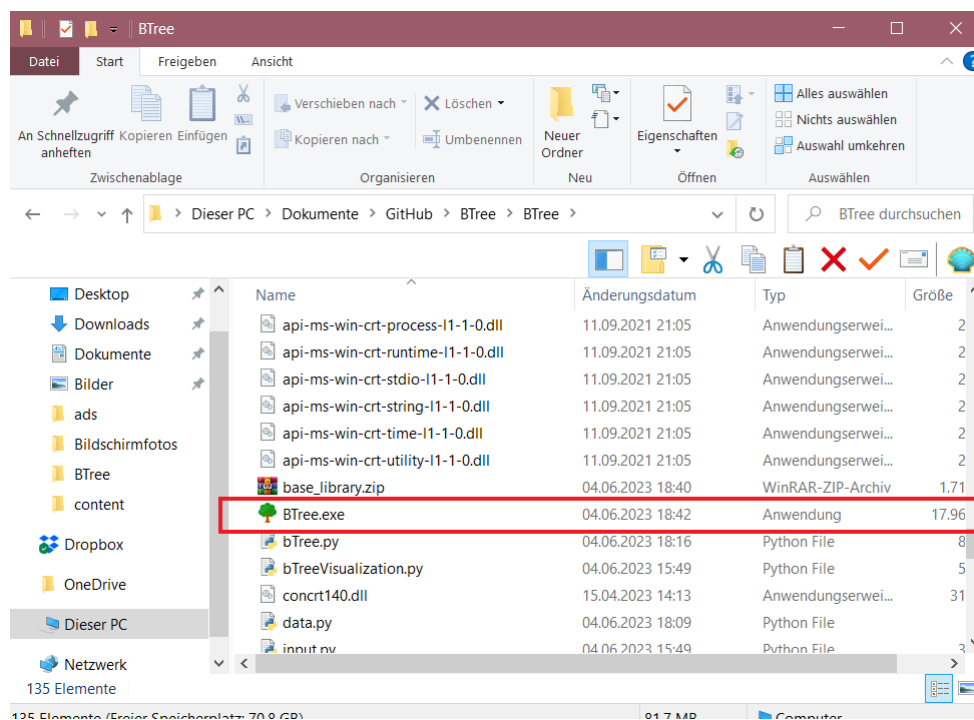


Abbildung 1.2: .exe Datei

2 Architektur

Wie bereits eingangs erläutert gliedert sich das Programm in zwei große Teile. Das Frontend übernimmt die Aufgabe der Visualisierung und Verarbeitung der Usereingaben, während das Backend den **B-Baum** mit seinen Operationen zu Verfügung stellt. In folgendem Klassendiagramm soll das Zusammenspiel der Komponenten gezeigt werden (vgl. Abbildung 2.1).

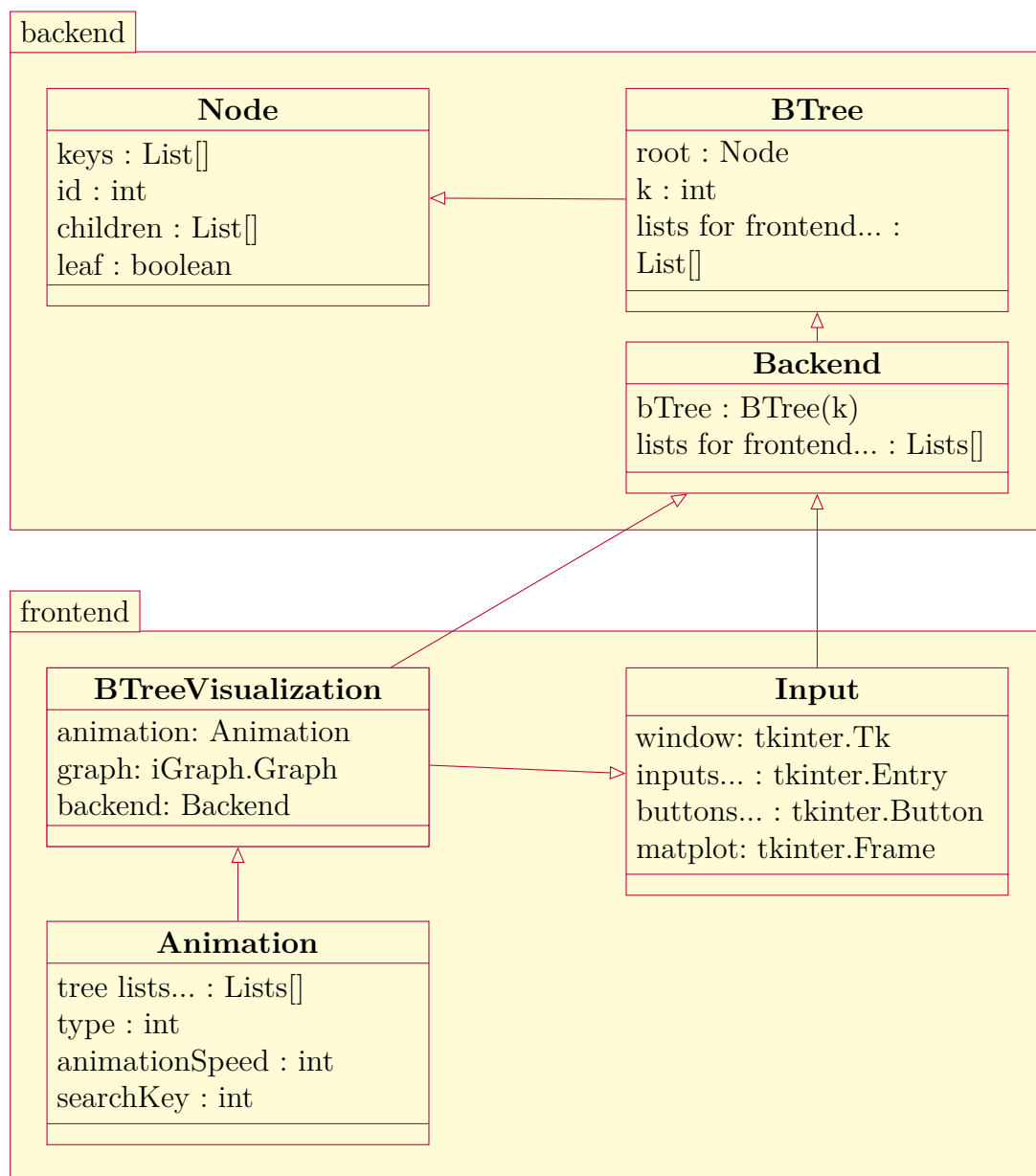


Abbildung 2.1: Klassendiagramm

Aus dem Klassendiagramm wird deutlich, dass die Schnittstelle des Programms sich in der Backendklasse findet. Diese Schnittstelle wird im Abschnitt Backend genauer definiert. Durch die klare Trennung der Zuständigkeiten konnte sichergestellt werden, dass die einzelnen Komponenten sich nicht überschneiden, sondern genau ihre Aufgabe erfüllen.

3 Backend

Im Backend des Projekts werden die Algorithmen des [B-Baumes](#) realisiert. Des Weiteren bereitet das Backend die nötigen Listen für das Frontend vor. Der [B-Baum](#) nutzt als Kenngröße den bereits definierten Verzweigungsgrad k . Zu den unterstützten Operationen auf dem [B-Baum](#) gehören:

- Suchen
- Einfügen
- Löschen

3.1 Aufbau

Das Backend besteht aus drei Klassen:

- BTree
- Backend
- Node

Die Klasse Node repräsentiert die Knoten des Baumes und hat als Attribute eine Liste an Schlüsseln, eine [Id](#), eine Liste an Kindreferenzen und eine bool Variable, die angibt, ob es sich um ein Blatt handelt.

BTree besteht aus dem Verzweigungsgrad k , der [Root](#) und Listen, welche das Frontend für die Visualisierung benötigt.

Bei der Backendklasse handelt es sich um die Schnittstelle zwischen dem Frontend und dem Backend. Die Backendklasse selbst hat ein BTree Objekt auf welchem es die entsprechenden Operationen ausführt. Nach Ausführung der Operationen füllt die Klasse alle Listen, die das Frontend benötigt, um den Baum darzustellen.

3.2 Algorithmen

Alle Operationen auf den **B-Baum** sollen dessen Eigenschaften nicht verletzen. So soll sowohl nach Einfügen oder Löschen der Verzweigungsgrad für jeden einzelnen Knoten korrekt sein. Die hat zur Folge, dass beim Einfügen Knoten gespalten werden müssen und beim Löschen rotiert oder gemerged werden muss.

3.2.1 Suchen

Sowohl die Einfüge- als auch die Löschoperation haben eine vorgeschaltete Suchoperation. Somit ist die Suche eine ausschlaggebende Funktion eines **B-Baumes**. Wie bereits erläutert beginnt jede Suche in der **Root** des Baumes. Von dort wird jeder Schlüssel mit dem Suchschlüssel verglichen bis entweder der Suchschlüssel, oder ein größerer Schlüssel gefunden wurde. Sollte der Suchschlüssel sich nicht in der **Root** befinden so geht die Suche im linken Kindknoten des ersten größeren Schlüssels weiter. Sollte kein Schlüssel größer sein, als der Suchschlüssel so wird im rechtesten Kind der **Root** gesucht. Dieser Vorgang wiederholt sich nun so lange bis der Suchschlüssel gefunden wurde, oder der Algorithmus sich in einem Blatt befindet und den Suchschlüssel noch immer nicht gefunden hat. Je nach Ausgang der Suche gibt der Algorithmus True oder False zurück.

3.2.2 Einfügen

Die erste Aufgabe des Einfüge-Algorithmus ist es den richtigen Knoten für den einzufügenden Schlüssel k zu finden. Hierzu traversiert der Algorithmus durch den **B-Baum** nach dem oben beschriebenen Schema, um den richtigen Knoten zu finden. Für das Einfügen in den **B-Baum** gibt es zwei Szenarien:

1. Der Knoten in welcher der Schlüssel eingefügt werden soll, hat weniger als $2k$ Schlüssel.
2. Der Knoten in welcher der Schlüssel eingefügt werden soll, hat genau $2k$ Schlüssel und es kommt zu einem **Overflow**.

Das erste Szenario ist recht simpel abgehandelt. Der einzufügende Schlüssel wird mit den Schlüsseln des Knotens verglichen, bis sich ein größerer Schlüssel an der Stelle n findet. Der einzufügende Schlüssel wird daraufhin an Stelle $n - 1$ eingefügt. Sollte kein größerer Schlüssel innerhalb des Knoten vorhanden sein, so wird der einzufügende Schlüssel an die letzte Stelle des Knoten eingefügt.

Beim zweiten Szenario wird der Schlüssel ebenfalls an die korrekte Stelle im Knoten

eingefügt. Nun hat der Knoten $2k + 1$ Schlüssel und verletzt somit die Eigenschaften des **B-Baumes**! Da der Knoten eine ungerade Anzahl an Schlüssel hat, kann der mittlere Schlüssel bestimmt werden. Dieser wandert eine Ebene höher und spaltet die restlichen $2k$ Schlüssel in zwei Knoten mit jeweils k Schlüsseln. Der hochgewanderte Schlüssel wird im nächsten Knoten ebenfalls an die korrekte Stelle eingefügt. Sollte nun erneut ein Overflow stattfinden, wiederholt sich der Prozess bis alle Knoten höchstens $2k$ Schlüssel haben.

3.2.3 Löschen

Beim Löschen gibt es zwei grundlegende Herangehensweisen, um den Baum nach der Löschoperation wieder zu **balancieren**:

- Rotation
- mergen

Die Rotation beschreibt das borgen von Schlüsseln von Nachbarknoten. Dies setzt voraus, dass der Nachbarknoten selbst mindestens $k + 1$ Schlüssel hat. Da bei dieser Operation nicht umverkettet werden muss, sondern nur Schlüssel getauscht werden, ist es ratsam stets die Rotation zu wählen.

Gemerged wird erst wenn es nur Nachbarn mit k Schlüsseln gibt. Dabei wird dann in den Underflow Knoten die restlichen Schlüssel des Nachbarn gepackt. Der leere Nachbar ist dann kein Teil des Baumes mehr.

Sollte der Knoten mit dem zu löschenden Schlüssel mehr als k Schlüssel haben, so kann der Schlüssel einfach gelöscht werden. Handelt es sich bei dem Knoten um kein Blatt muss sich ebenfalls um die Kindreferenzen gekümmert werden. Je nach Anzahl der Schlüssel in den Kindern kann dann entweder die Rotation angewandt oder beide Kinder gemerged werden. Wie bereits erläutert wird erst gemerged, wenn beide Knoten eine Minimalbelegung von k Schlüsseln haben.

4 Frontend

Im Frontend gibt es zwei Themenbereiche: Den Aufbau des Frontends und die Animationen des B-Baums. Im Folgenden wird das Vorgehen für beide Bereiche erläutert.

4.1 Aufbau

Die Interaktion mit dem Benutzer wird in der *Input*-Klasse realisiert. Darin sind drei Schichten kombiniert: Die Visualisierung des Graphen, die Animationen des Graphen und die Graphical User Interface (GUI)-Elemente für die Interaktion mit dem Benutzer.

4.1.1 Schichten

Die Unterste Ebene des Frontends ist ein *Tkinter*-Fenster, das beim Ausführen der *main.py* Datei automatisch geöffnet wird (vgl. Abbildung 4.1a). Auf dem Fenster befinden sich oben Bedienelemente, um den B-Baum zu konfigurieren. Unten sind Buttons mit denen der Nutzer den B-Baum zurücksetzen und speichern kann.

Dazwischen ist ein *Matplotlib*-Frame (vgl. Abbildung 4.1b). Dieses beinhaltet einen Graphen in Form eines *iGraph*-Objektes. Das Objekt wird in ein Canvas umgewandelt und kann somit auf dem *Matplotlib*-Frame dargestellt werden (vgl. Abbildung 4.1c).

Das *iGraph*-Objekt muss in ein *Matplotlib*-Frame eingefügt werden, damit die *FuncAnimation*-Klasse für *Matplotlib* nutzbar ist. Dadurch wird der B-Baum für jeden Frame neu dargestellt, ohne dass sich das Fenster bei jedem Frame schließt und wieder öffnet. Das *FuncAnimation*-Objekt ruft für jeden Frame eine Methode auf, die Position und Inhalt der Knoten des B-Baums für diesen Frame berechnet und aktualisiert.

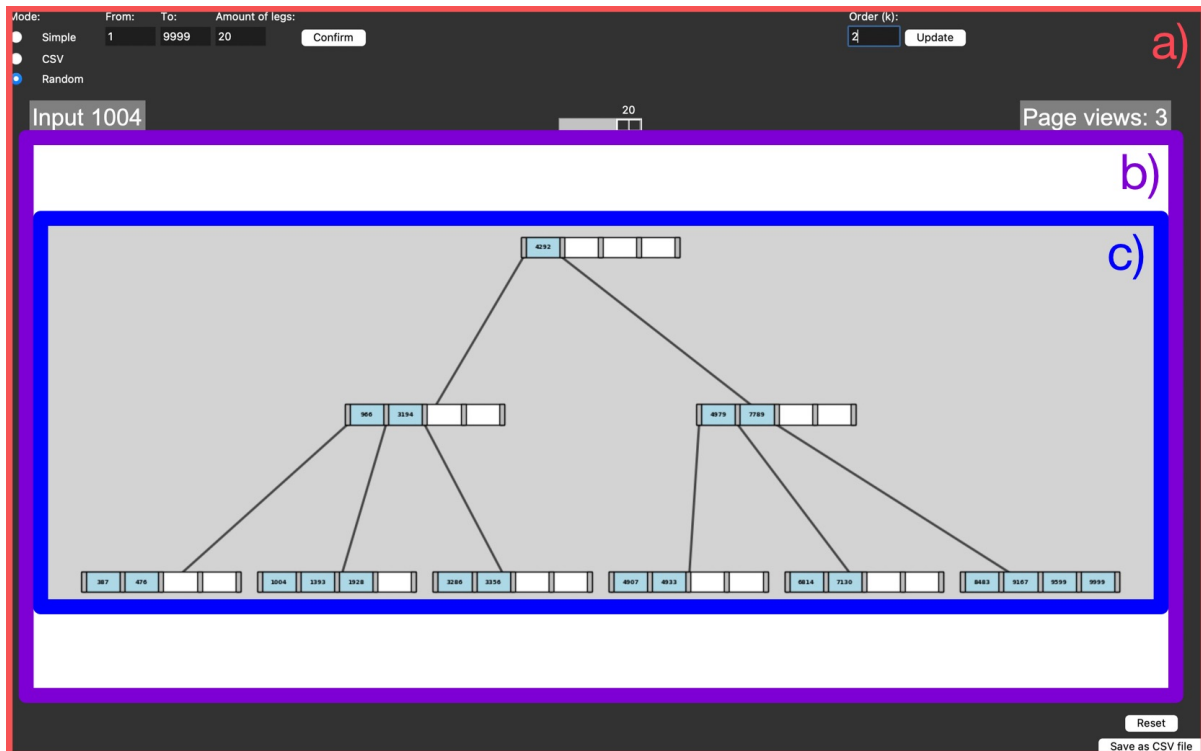


Abbildung 4.1: Drei Schichten: a) Tkinter-Fenster, b) Matplotlib-Frame, c) iGraph-Canvas

Wenn der Nutzer die Größe des Fensters anpasst, wird der Graph immer komplett dargestellt, indem sich die Größe des Graphen verändert. Wenn das Fenster zu klein wird, überlappt der Graph die Bedienelemente. Der Fokus liegt darauf den Graphen immer sichtbar zu halten.

4.1.2 Bedienelemente

Im linken oberen Bereich des Fensters kann der Nutzer zwischen drei Modi für die Modifikation des Graphen entscheiden:

1. Beim einfachen Input kann der Nutzer einen Wert zwischen 1 und 9999 eingeben. (vgl. Abbildung 4.2) Dieser wird je nach Auswahl des Nutzers eingefügt, gesucht oder gelöscht. Der Zahlenbereich endet bei 9999, um die Darstellung der Schlüssel in den Knoten zu verbessern. So kann sichergestellt werden, dass ein Schlüsselwert den Kasten eines Schlüssels nicht überschreitet.

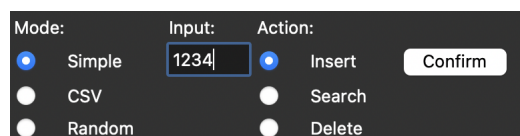
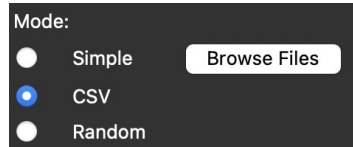
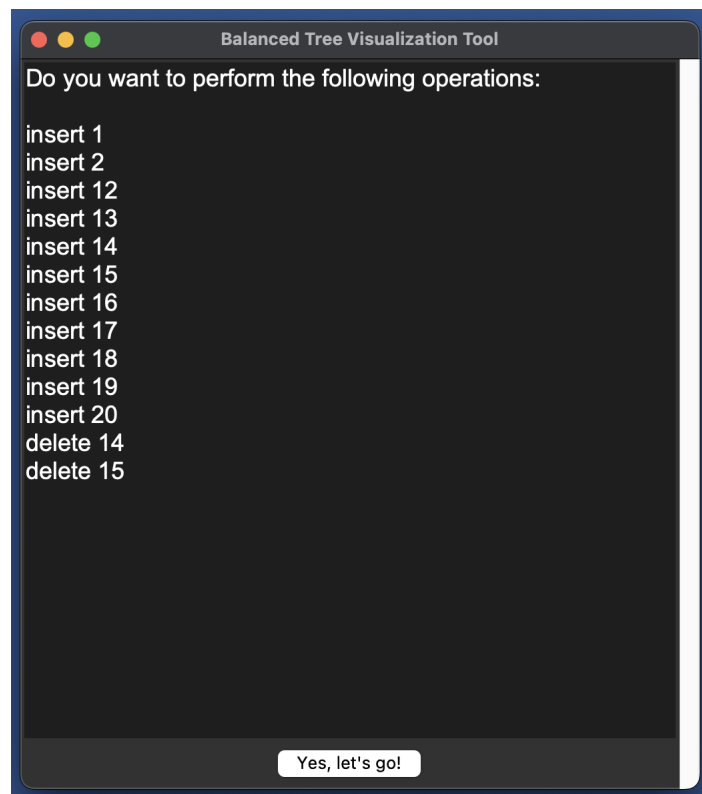


Abbildung 4.2: Einfacher Modus

2. Beim **CSV**-Modus kann eine Datei im **CSV**-Format ausgewählt werden (vgl. Abbildung 4.3). Dabei wird die Datei ausgelesen und Einfüge- und Löschoperationen werden erkannt. Sie werden dem Nutzer in einem weiteren Fenster angezeigt und er kann die Operationen mit einem Button bestätigen (vgl. Abbildung 4.4).

Abbildung 4.3: **CSV**-ModusAbbildung 4.4: **CSV**-Bestätigungs-Fenster

Die **CSV**-Datei wird zeilenweise eingelesen. Dabei muss eine Zeile einem bestimmten Format entsprechen: Operation, Komma, Zahl zwischen 1 und 9999. Die Einfüge-Operation wird mit i (engl. *insert*) abgekürzt, die Löschoperation mit d (engl. *delete*).

Zum Beispiel wäre folgende Zeile syntaktisch und semantisch korrekt:

```
1 i , 420
```

3. Beim Einfügen von Zufallszahlen kann der Nutzer den Zahlenbereich festlegen und wie viele Zufallswerte aus diesem Bereich eingefügt werden sollen (vgl. Abbildung 4.5).

Abbildung 4.5: Zufallsmodus

Die Zufallswerte werden mit folgender Funktion generiert:

```

1  # library to generate random numbers
2  import random
3
4  # generates a number between from - to
5  random.randint(from, to)

```

Oben rechts kann der Nutzer die Ordnung des **B-Baums** verändern (vgl. Abbildung 4.6). Mit Ordnung ist das k gemeint, also die minimale Anzahl an Schlüsseln in einem Knoten, außer der Wurzel. Die maximale Anzahl an Schlüsseln in einem Knoten beträgt $2k$.

Abbildung 4.6: Ordnung verändern

Wenn der Nutzer die neue Ordnung bestätigt, wird der **B-Baum** neu initialisiert. Er wird mit neuer Ordnung zurückgesetzt und die Werte, die vorher eingefügt wurden, werden neu eingefügt, angepasst auf die neue Ordnung.

Zentral oben befindet sich ein Schieberegler, mit dem der Nutzer die Geschwindigkeit der Animationen festlegen kann. Geschwindigkeit 1 ist dabei langsam, Geschwindigkeit 20 ist schnell.

Unten rechts Befinden sich zwei weitere Buttons:

1. Per *reset*-Button wird der Inhalt des **B-Baums** gelöscht und die Struktur wird auf einen leeren Knoten zurückgesetzt.
2. Mit dem *Save as CSV file*-Button werden alle Einfüge- und Löschoptionen gesichert und im zuvor beschriebenen Format zeilenweise in einer **CSV**-Datei gespeichert.

Je nach Eingabe werden dem Nutzer Hinweise angezeigt. Diese werden im Folgenden Unterkapitel erläutert.

4.1.3 Informationen für den User

Um die Bedienung der Applikation anwenderfreundlich zu machen, erhält der Nutzer bei bestimmten Interaktionen Informationen in Form von Text. Beispielsweise, wenn die Eingabe in einem Feld nicht den Vorgaben entspricht (vgl. Abbildung 4.7). Dadurch erkennt der Nutzer, dass ein Fehler vorliegt und kann diesen korrigieren.

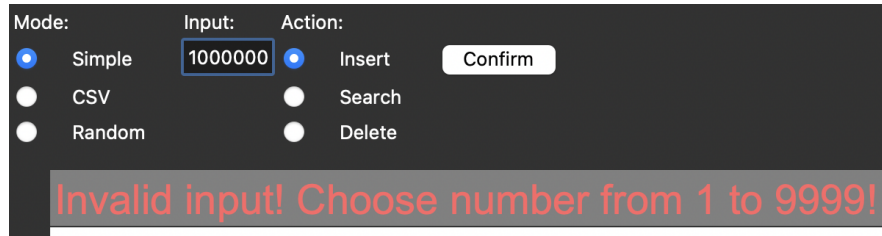


Abbildung 4.7: Fehlermeldung bei falscher Eingabe

Ebenfalls wird die aktuell durchgeführte Operation angezeigt, zum Beispiel beim Einfügen eines Wertes (vgl. Abbildung 4.8) oder beim Ende einer Suche, wenn dem Nutzer mitgeteilt wird, ob der gesuchte Wert gefunden wurde, oder nicht (vgl. Abbildung 4.9).

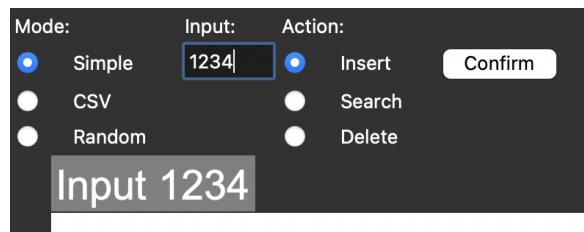


Abbildung 4.8: Anzeige der aktuell ausgeführten Operation

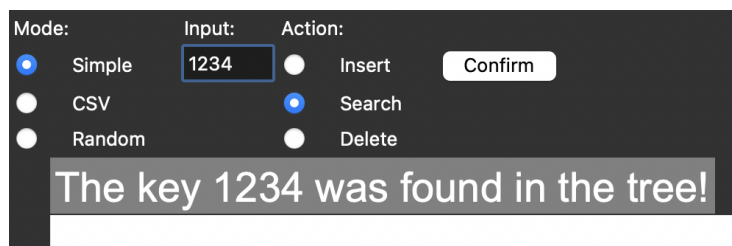


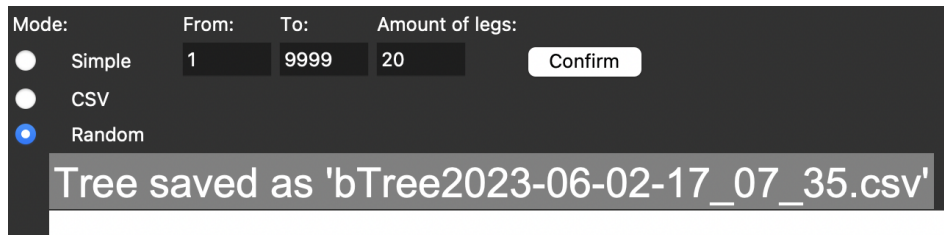
Abbildung 4.9: Anzeige, dass gesuchter Schlüssel gefunden wurde

Bei allen Animationen wird die Anzahl der benötigten Seitenzugriffe, sprich Knotenzugriffe, mitgezählt (vgl. Abbildung 4.10). Die Anzahl wird während der Animation der Operation dynamisch mitgezählt.

Page views: 3

Abbildung 4.10: Anzeige der Anzahl an Seitenzugriffen einer Operation

Wenn der [B-Baum](#) als [CSV](#)-Datei gespeichert wird, bekommt der Nutzer den Dateinamen angezeigt (vgl. [Abbildung 4.11](#)). Dabei wird der einmalige Zeitstempel der Sekunde, in der der Baum gespeichert wurde, verwendet.



Mode:	From:	To:	Amount of legs:	
<input type="radio"/> Simple	1	9999	20	<input type="button" value="Confirm"/>
<input type="radio"/> CSV				
<input checked="" type="radio"/> Random				

Tree saved as 'bTree2023-06-02-17_07_35.csv'

Abbildung 4.11: Anzeige des Dateinamens beim Speichern des [B-Baums](#)

Hinzu kommen weitere Hinweise und Fehlermeldungen ähnlich der zuvor beschriebenen.

4.2 Animationen

Für die Visualisierung des [B-Baums](#) wurde [iGraph](#) genutzt. Damit können Knoten und Kanten einfach gezeichnet werden. Zudem kann deren Design nach Belieben angepasst werden. [iGraph](#) kann in Kombination mit [Matplotlib](#)-Animationen verwendet werden, um die Veränderung eines Graphen zu animieren.

4.2.1 Darstellung des B-Baums

Der [B-Baum](#) besteht aus mehreren Knoten, die jeweils k bis $2k$ Schlüssel beinhalten. Die Schlüssel werden hellblau dargestellt (vgl. [Abbildung 4.12](#)).

Ein [B-Baum](#) mit x Schlüsseln hat $x+1$ Kinder. Alle Schlüssel im Ast des linken Kindes eines Eltern-Schlüssels sind kleiner als der Eltern-Schlüssel. Alle Schlüssel im Ast des rechten Kindes sind größer. Die Verweise von Eltern-Knoten auf ihre Kinder sind Referenzen. Diese sind grau dargestellt und befinden sich zwischen zwei Schlüsseln in einem Knoten oder links, bzw. rechts neben dem äußersten Schlüssel eines Knotens (vgl. [Abbildung 4.12](#)). Von einer Referenz geht eine ungerichtete Kante auf den entsprechenden Kind-Knoten (vgl. [Abbildung 4.12](#)).

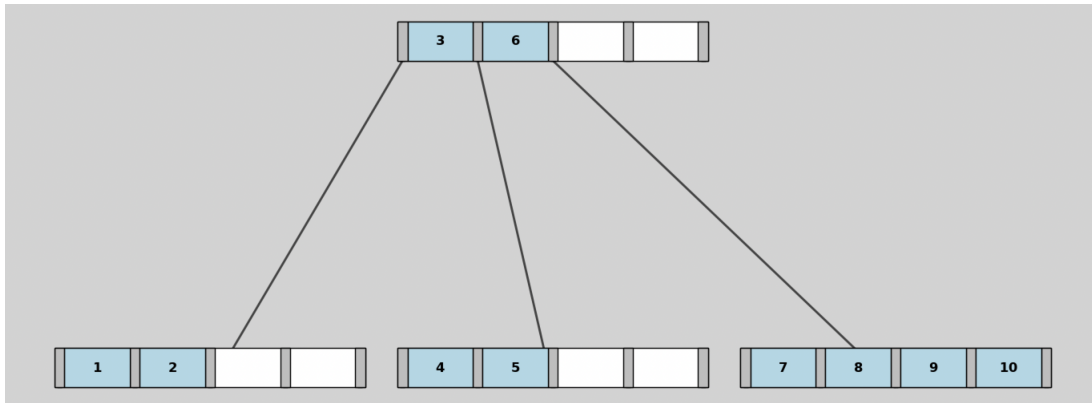


Abbildung 4.12: Visualisierung des B-Baums

Bei der Visualisierung wird immer der gesamte Knoten mit $2k$ Schlüsselpositionen dargestellt. In einem realen Datenbanksystem (DBS) werden Daten in Blöcken gespeichert. Die Blöcke haben eine feste Speichergröße, z.B. 64K. Der Speicherplatz existiert auch, wenn er nicht voll ausgenutzt wird. Der freie Speicherbereich in den Knoten des B-Baums ist durch weiße, anstatt blaue Schlüsselpositionen gekennzeichnet (vgl. Abbildung 4.12).

4.2.2 Suchen und Löschen

Sucht der Nutzer nach einem bestimmten wert y , beginnt die Visualisierung der Suche in der Wurzel. Hier wird in einem regelmäßigen Abstand ein Schlüssel nach dem Anderen rot gefärbt, um zu signalisieren, dass dieser Schlüssel z gerade mit y verglichen wird. Sobald $z > y$ gilt, wird die Suche im linken Kind von z fortgesetzt. Wenn der Schlüssel gefunden wurde, wird er grün gefärbt. Ansonsten wird dem Nutzer angezeigt, dass der Schlüssel nicht gefunden wurde.

Beim Löschen erscheint dieselbe Animation. Allerdings wird der zu löschende Schlüssel aus dem Baum entfernt, nachdem er gefunden wurde und der Baum wird ggf. umverkettet, um die Eigenschaften des B-Baums zu bewahren.

4.2.3 Einfügen

Bei der Einfüge-Animation wird der einzufügende Schlüssel y wie bei Suchen und Löschen mit den im Baum vorhandenen Schlüsseln verglichen. Die Einfüge-Position befindet sich in einem untersuchten Blatt links von einem Schlüssel z , wenn gilt $z > y$. Diese Position wird grün markiert und der Schlüssel wird anschließend dort eingefügt.

Der Schlüssel y wandert von der Wurzel aus entlang aller Knoten, mit deren Schlüsseln er verglichen wird. Hat ein Blatt nach Einfügen eines Schlüssels $2k + 1$ Schlüssel, verletzt es die **B-Baum**-Eigenschaften. Folglich wandert der mittlere der $2k + 1$ -Schlüssel in den Eltern-Knoten. Dieses Wandern wird dadurch visualisiert, dass sich der Schlüssel entlang der Kante bewegt, die Eltern- und Kind-Knoten miteinander verbindet. Der Schlüssel wird im Eltern-Knoten einsortiert und der Kind-Knoten wird auf zwei Knoten aufgeteilt.

4.2.4 Darstellung breiter Graphen

Der **B-Baum** hat die Eigenschaft, dass er bei steigender Anzahl an Schlüsseln immer breiter wird. Dabei steigt auch der Unterschied zwischen Breite und Höhe. Folglich ist ein Baum mit vielen Schlüsseln breit und niedrig. Daraus folgt, dass die Höhe des **iGraph**-Canvas (vgl. Abbildung 4.1c) bei zunehmender Schlüsselanzahl sinkt, während die Breite gleich bleibt. Je höher die Ordnung des Baums, desto breiter wird der Baum, da jeder Knoten mehr Schlüssel fasst und somit breiter ist.

Ab ca. 100 Schlüsseln (13-Zoll-Bildschirm und $k = 2$) lassen sich die Schlüssel in den Knoten kaum noch erkennen, da der **B-Baum** zu breit geworden ist und die Größe der Schlüssel dadurch sinkt. Damit der Nutzer den Inhalt der Knoten trotzdem erkennt und dabei der gesamte Baum sichtbar bleibt, wurde folgender Kompromiss getroffen: Bewegt der Nutzer den Mauszeiger über einen Knoten, werden die Schlüssel des Knoten unterhalb des **B-Baums** als Liste in konstanter Größe angezeigt (vgl. Abbildung 4.13).

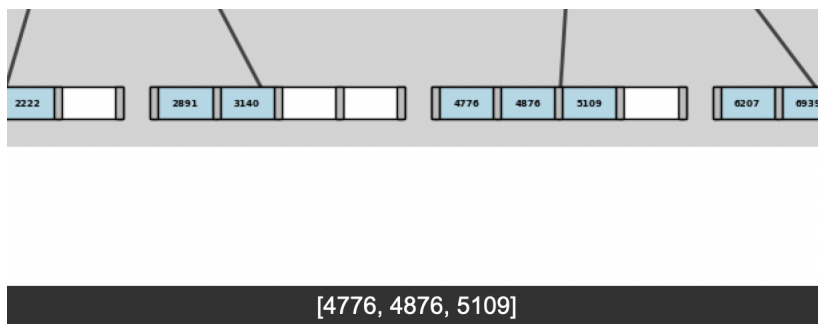


Abbildung 4.13: Anzeige der Schlüssel des Knotens, auf dem sich der Mauszeiger befindet

Dadurch sieht der Nutzer die gesamte Struktur des **B-Baums** und kann sich den Inhalt jedes Knotens anzeigen lassen. Eine Zoom-Funktion ist mit der Kombination aus **Tkinter**-Fenster, **Matplotlib**-Frame und **iGraph**-Canvas nicht möglich.

5 Ausblick

Zusammenfassend lässt sich festhalten, dass das in dieser Arbeit entwickelte Werkzeug das Verhalten eines balancierten Baumes auf den Operationen Einfügen, Suchen und Löschen visualisiert.

Dabei wurde besonders auf Benutzerfreundlichkeit geachtet. So erhält der Nutzer textuelle Hinweise und kann Funktionen nutzen, die die Bedienung des Werkzeuges vereinfachen, z.B. das Speichern des [B-Baumes](#) als [CSV](#)-Datei.

Durch aufwändige Animationen muss das Backend bei zunehmender Schlüsselanzahl immer mehr Listen für das Frontend generieren, wodurch die Animationen bei vielen Schlüsseln ruckelig werden. Dafür gilt es im nächsten Schritt eine Verbesserung zu finden. Beispielsweise in Form einer schlankeren Schnittstelle zwischen Backend und Frontend für die Übergabe von Animationen.

Die Animationen sollen aber so aufwändig bleiben, wie sie sind. Der Fokus des Werkzeugs liegt auf der Visualisierung der Operationen auf einem [B-Baum](#) und darauf, wie der Baum balanciert wird. Dadurch können Nutzer, die das Verhalten eines [B-Baumes](#) nicht kennen, die Einzelnen Schritte einer Operation nachvollziehen und visuell lernen, wie ein [B-Baum](#) funktioniert.