Gottfried Wilhelm Leibniz Universität Hannover
Institut für Theoretische Informatik

Eine GUI zur Visualisierung von B-Bäumen

Bachelorarbeit

Jonas Kaiser
Matrikelnr. 10013838

Hannover, den 7. Juni 2024

Erstprüfer:	PD Dr. Arne Meier
Zweitprüfer:	Prof. Dr. Heribert Vollmer
Betreuer:	Nicolas Fröhlich

Erklärung der Selbstständigkeit

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, den 7. Juni 2024

_____

Jonas Kaiser

Inhaltsverzeichnis

1 Einleitung

B-Bäume sind eine Variante von Suchbäumen, die häufig für die Arbeit mit Blockspeichergeräten genutzt werden. Beispiele dafür sind SQL Datenbanken oder das Btrfs Dateisystem.

Sie sind besonders effizient auf diesen Blockgeräten, da ihre Knoten im Gegensatz zu Binärbäumen weit mehr als einen Schlüssel beinhalten können. Dadurch lässt sich jeweils ein Knoten auf einen gesamten Speicherblock abbilden, womit man deutlich weniger Zugriffe auf diesen vergleichsweise langsamen Blockspeicher benötigt als es bei herkömmlichen Binärbäumen der Fall wäre.

Entwickelt wurden die B-Bäume im Jahr 1972 von Rudolf Bayer und Edward M. McCreight [Edw72], und sind seitdem aus der Informatik nicht mehr wegzudenken. Daher sind sie auch ein fester Teil der Datenstrukturen und Algorithmen Vorlesungen an der Leibniz Universität Hannover. In dieser Arbeit wird eine graphische Anwendung entwickelt, die Studenten dabei helfen soll, die Funktionsweise der Bäume besser zu verstehen.

B-Bäume stellen sicher, dass die Knoten nicht über die Blockgröße hinaus wachsen können und beim Löschen mindestens bis zur Hälfte befüllt bleiben. Dafür nutzen sie die speziellen Operationen *Split*, *Fuse* und *Transfer*. Diese Operationen zu verstehen kann Studenten vor eine Herausforderung stellen.

Die Anwendung wird es ermöglichen, in Echtzeit Einfüge- und Löschoperationen durchzuführen, und deren Auswirkungen auf die Struktur des B-Baumes unmittelbar zu beobachten. Dies soll dazu beitragen, die theoretischen Konzepte, die in der Vorlesung vermittelt werden, durch praktische Erfahrungen zu untermauern und zu vertiefen.

Ein Beispiel für einen solchen B-Baum ist in Abb. 1.1 zu sehen. Zur Veranschaulichung nutzen wir hier Bäume mit wenigen Elementen pro Knoten. In der Realität ist die Knotengröße meist deutlich höher gewählt. Die Funktionsweise bleibt jedoch die selbe. Außerdem enthalten Knoten normalerweise nicht nur Schlüssel, sondern verweisen auch auf ein tatsächliches Objekt im Speicher. Die Schlüssel dienen lediglich der Sortierung. Weil dies für die Funktionsweise irrelevant ist, und bei der Visualisierung hinderlich wäre, arbeiten wir in dieser Arbeit ausschließlich mit den Schlüsseln.

Zu Beginn der Arbeit werden die Grundlagen der B-Bäume erklärt. Dazu werden alle Baumoperationen mit einem Pseudocode versehen, der später auch in der Anwendung angezeigt wird. Dieser Pseudocode spiegelt dann die Veränderung der Visualisierung wieder.

Anschließend gehen wir kurz auf die Programmiersprache Python ein, bevor wir uns der näheren Beschreibung des Quelltextes von unserer Anwendung widmen. Zum Abschluss der Arbeit wird noch auf einige Probleme, die bei der Implementierung aufgetreten, sind eingegangen.

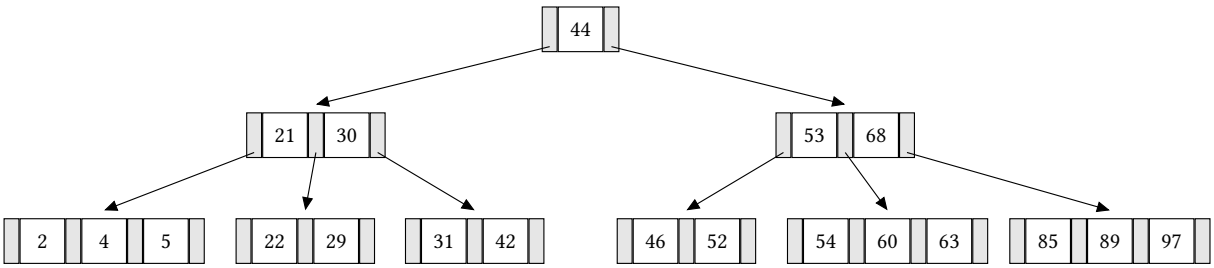


Abbildung 1.1: Beispiel B-Baum der Ordnung 4

2 Grundlagen B-Bäume

B-Bäume gehören zu den Suchbäumen. Das bedeutet, ihre Elemente sind so sortiert, dass alle Werte links von einem Schlüssel kleiner sind als dieser, und alle rechts davon größer. Dies gilt sowohl für Schlüssel innerhalb eines Knotens, als auch für Schlüssel in Teilbäumen, die „zwischen“ zwei Schlüsseln liegen. Darüber hinaus sind B-Bäume *balancierte* Bäume, was bedeutet, dass alle Pfade im Baum die gleiche Tiefe haben.

Wie viele Kinder ein Knoten im Baum haben kann, wird oft durch ein Zahlentupel angegeben. So hat ein $(2, 4)$-Baum mindestens 2 Kinder, maximal 4. Die zweite Zahl wird auch als *Ordnung* des Baumes bezeichnet. Die kleinste Ordnung, die ein B-Baum haben kann, ist $m = 3$. Sonst müsste er Knoten mit nur einem Kind und keinen Schlüsseln besitzen.

Da es in der Literatur verschiedene Definitionen für diesen Begriff gibt, werden wir ihn im Folgenden für den Verlauf der Arbeit festlegen:

> **Definition 1** (B-Baum)**.** Ein B-Baum der Ordnung $m \in \mathbb{N}$ erfüllt folgende Eigenschaften:
>
> 1. Alle Blätter haben die gleiche Tiefe und sind leer.
> 2. Jeder Knoten hat $\leq m$ Kinder.
> 3. Jeder innere Knoten hat $\geq \lceil \frac{m}{2} \rceil$ Kinder. Die Wurzel hat ≥ 2 Kinder.
> 4. Jeder Knoten mit $l \in \mathbb{N}$ Schlüsseln hat $l + 1$ Kinder.
> 5. Für alle Knoten mit Schlüsseln $s_1, \ldots, s_l$ und Kindern $v_0, \ldots, v_l$ gilt für $1 \leq i \leq l$:
>   - Alle Schlüssel in $T[v_{i-1}]$ sind $\leq s_i$ und
>   - $s_i \leq$ allen Schlüsseln in $T[v_i]$.
>
> $T[v]$ bezeichnet einen Teilbaum mit Wurzel v.

B-Bäume gehören zum Abstrakten Datentyp (ADT) Tree. Dieser besitzt einige die in Tabelle 2.1 dargestellten Methoden, die im weiteren Verlauf benötigt werden, um die Bäume zu bearbeiten.

2.1 Operationen

Um mit einem B-Baum arbeiten zu können, benötigt man drei primäre Operationen: *Suchen*, *Einfügen* und *Löschen*. Das Suchen unterscheidet sich nicht von der Suche in anderen Baumstrukturen, abgesehen davon, dass ein Knoten mehrere Elemente beinhaltet. Beim Einfügen

Operation	Beschreibung
`order()`	liefert Ordnung des Baumes
`isEmpty()`	`true` wenn Baum leer
`isRoot(Knoten` v`)`	`true` wenn v die Wurzel ist
`isLeaf(Knoten` v`)`	`true` wenn v ein Blatt ist
`aboveLeaf(Knoten` v`)`	`true` wenn Kinder von v Blätter sind
`root()`	liefert Wurzelknoten des Baumes
`parent(Knoten` v`)`	liefert Elternknoten von v
`children(Knoten` v`)`	liefert die Kinderknoten von v
`keys(Knoten` v`)`	liefert die Schlüssel in v

Tabelle 2.1: Methoden des ADT Tree

und Löschen müssen jedoch sowohl die Sortierung der Schlüssel, als auch die minimale und maximale Anzahl an Kinderknoten beibehalten werden. Deshalb beinhalten sie einige Sonderfälle, bei denen der Baum umstrukturiert werden muss.

2.1.1 Suche

Die Suche in einem B-Baum lässt sich rekursiv durchführen. Sie startet im Wurzelknoten des Baumes. Der gesuchte Wert x wird erst mit den Schlüsseln s_1 bis s_l verglichen. Wenn der Wert nicht gefunden wurde, wird die Suche auf den Teilbaum $T[v_i]$ ausgeführt, für den gilt $s_i < x < s_{i+1}$. Der Einfachheit halber wird hier lineare Suche genutzt. Für größere Ordnungen lohnt es sich, binäre Suche zu implementieren.

Die Suche endet, wenn der Schlüssel gefunden wurde, oder die Kinder externe Knoten sind. [1] Falls er nicht gefunden wurde, befinden wir uns nun an der Stelle, wo er beim Einfügen platziert werden müsste. Eine beispielhafte Suche wird in Abb. 2.2 dargestellt.

Im weiteren Verlauf wird die Suchfunktion für mehrere Operationen wiederverwendet. Dort wird nicht nur die Information benötigt, ob das Suchen erfolgreich war. Weiterhin müssen der Knoten, in dem die Suche geendet hat, und der letzte durchsuchte Index zurückgegeben werden. Selbst dann, wenn die Suche fehlgeschlagen ist, etwa um dort neue Elemente einzufügen (siehe Abschnitt 2.1.2 und Abschnitt 2.1.3). Daher gibt die Funktion nicht nur einen Knoten oder NULL zurück, sondern ein Tupel bestehend aus einem Wahrheitswert, dem letzten durchsuchten Knoten und dem letzten durchsuchten Index [2]. Der Pseudocode für diese Operation ist Abb. 2.1 zu entnehmen.

[1] Der Rekursionsanker ist nicht in den Blättern, da sonst die Einfügeposition nicht zurückgegeben werden kann.

[2] Zusätzlich könnte auch noch der Index im Elternknoten weitergereicht und zurückgegeben werden. Dies würde später die Over- und Underflow Behandlung effizienter machen. Wird hier nicht gemacht, da uns die Verständlichkeit wichtiger ist als Effizienz.

function SEARCH(Teilbaum T mit Wurzel u, Schlüssel x)
  $i \leftarrow 0, keys \leftarrow T.\text{keys}(u)$
  while $i < |keys|$ **und** $x > keys[i]$ **do** $i \leftarrow i + 1$
  if $i < |keys|$ **und** $keys[i] = x$ **then return** (true, u, i)
  else if T.aboveLeaf(u) **then return** (false, u, i) ▷ *Rekursionsende*
  else return SEARCH(T.children(u)$[i], x$)

Abbildung 2.1: Pseudocode für Suche

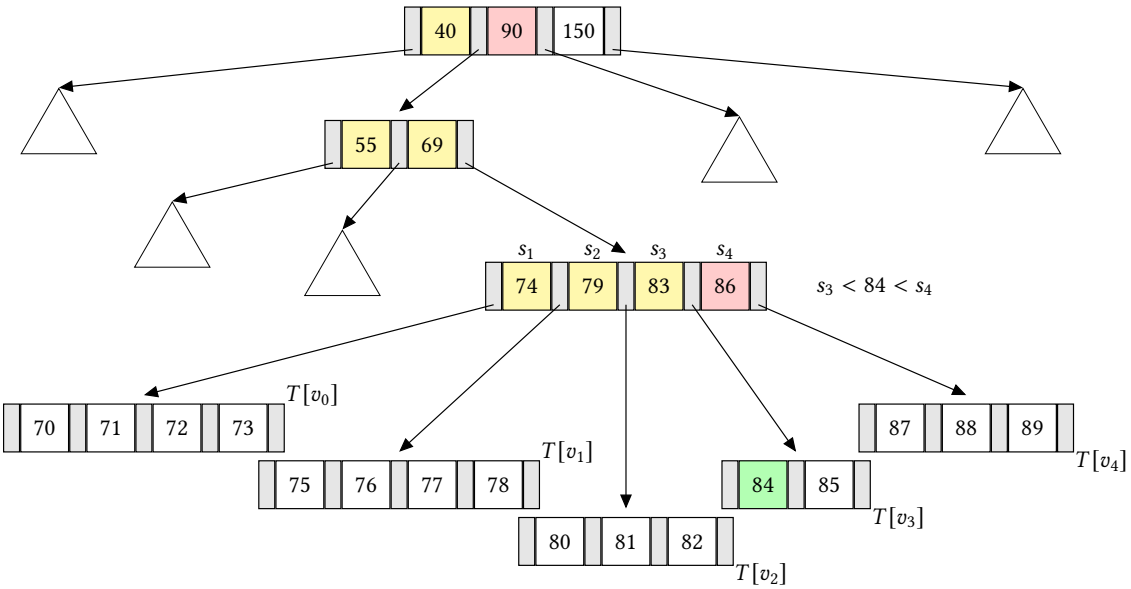


Abbildung 2.2: Suche nach 84 in einem B-Baum der Ordnung 5
gelb: $s_i < x$, rot: $s_i > x$, grün: $s_i = x$

2.1.2 Einfügen

Um einen neuen Schlüssel in den Baum einzufügen, wird zunächst mit der eben beschriebenen Suche der Knoten gefunden, in den er eingefügt werden muss. Dieser befindet sich immer auf der untersten Baumebene. Falls der Schlüssel schon existiert, wird abgebrochen. Anschließend wird der neue Schlüssel an der passenden Stelle in der Liste des Knotens eingefügt.

Sofern die Anzahl der Schlüssel $< m$ bleibt, ist das Einfügen abgeschlossen. Es kann aber passieren, dass der Knoten durch das Einfügen zu viele Schlüssel enthält. Dies wird *Overflow* genannt. Er muss dann in zwei kleinere Knoten aufgeteilt werden. Diese Operation bezeichnen wir als *Split*. (Pseudocode siehe Abb. 2.3)

```
function INSERT(Baum T, Schlüssel x)
    if T.isEmpty() then
        Erstelle neuen Wurzelknoten
        node ← T.root(), i ← 0
    else
        found, node, i ← SEARCH(T.root(), x)
        if found then return
    Füge x in T.keys(node) an Index i ein
    if |T.keys(node)| ≥ T.order() then SPLIT(node)
```

Abbildung 2.3: Pseudocode für Einfügen

Split

Vom Knoten v wird ein Teilknoten w abgespalten (Abb. 2.5). Dabei behält v die erste Hälfte der Schlüssel $s_1, \ldots, s_{\lfloor m/2 \rfloor}$ und w erhält die zweite Hälfte $s_{\lfloor m/2 \rfloor+2}, \ldots, s_m$. Bei gerader Ordnung erhält w einen Schlüssel weniger als v. Der mittlere Schlüssel $s_{\lfloor m/2 \rfloor+1}$ wird an den Elternknoten von v abgegeben. Die Referenzen auf Kinder werden ebenfalls aufgeteilt, v behält $T[v_0], \ldots, T[v_{\lfloor m/2 \rfloor}]$ und w erhält $T[v_{\lfloor m/2 \rfloor+1}], \ldots, T[v_m]$.

Der neue Knoten w wird hinter v im Elternknoten eingefügt. Somit bleibt das Verhältnis von Schlüsseln und Kindern dort im Gleichgewicht. Falls v der Wurzelknoten des Baumes ist, muss ein neuer Knoten erstellt und als Elternknoten von v eingesetzt werden. Der B-Baum wächst dadurch nach oben.

Beim Einfügen des mittleren Schlüssels in den Elternknoten kann es dort ebenfalls zu einem Overflow kommen. In diesem Fall wird die Split Operation noch auf den Elternknoten angewendet. Der Pseudocode für diese Operation ist Abb. 2.4 zu entnehmen.

2.1.3 Entfernen

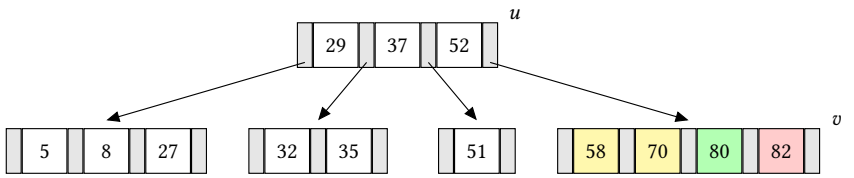
Für das Entfernen eines Schlüssels s wird, wie beim Einfügen, zunächst seine Position bestimmt. Falls der entsprechende Knoten keine Nachfolger hat, beziehungsweise all seine Kinder leer sind, kann der Schlüssel direkt gelöscht werden. Ist dies nicht der Fall muss der Schlüssel erst mit seinem *inorder Nachfolger* s' getauscht werden. Dieser ist der kleinste Schlüssel, der größer ist als s, und befindet sich daher immer auf der untersten Ebene. Dort kann s nun aus dem Knoten entfernt werden. Löschen aus einem Knoten, der noch weitere Teilbäume als Kinder besiztz, würde Eigenschaft 4 aus Definition 1 verletzten.

Ähnlich dem Overflow beim Einfügen, kann ein Knoten hier nach dem Löschen zu wenig Schlüssel besitzen. Dies tritt auf, wenn die Anzahl an Schlüsseln $l < \lceil \frac{m}{2} \rceil - 1$ ist, und wird *Underflow* genannt. Um einen Underflow zu beheben, gibt es zwei Möglichkeiten: *Transfer* und *Fuse*. Eine Reihe aus Underflowbehandlungen wird in Abb. 2.10 dargestellt.

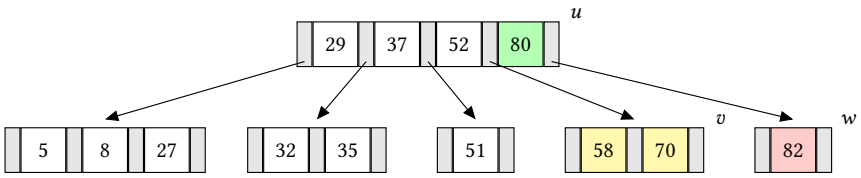
Ein Transfer kann ausgeführt werden, wenn der betroffene Knoten einen *direkten Geschwisterknoten* hat, der mehr als die minimale Anzahl an Schlüssel besitzt. Dabei wird einer der Schlüssel vom Geschwisterknoten über den Elternknoten an ihn weitergereicht. Dies

```
function SPLIT(Knoten v in Baum T)
    len  ← |T.keys(v)|
    mid ← ⌊len/2⌋
    if T.isRoot(v) then
        Erstelle neuen Wurzelknoten
        p ← T.root(), i ← 0
    else
        p ← T.parent(v)
        i ← Index von v in T.children(p)
    w ← Neuer Knoten, key_mid ← T.keys(v)[mid]
    T.keys(w) ← T.keys(v)[mid + 1, ..., len − 1]
    T.children(w) ← T.children(v)[mid + 1, ..., len]
    T.keys(v)  ← T.keys(v)[0, ..., mid − 1]
    T.children(v)  ← T.children(v)[0, ..., mid]
    Füge key_mid in T.keys(p) an Index i ein
    Füge w in T.children(p) an Index i + 1 ein
    if |T.keys(p)| ≥ T.order() then SPLIT(p)
```

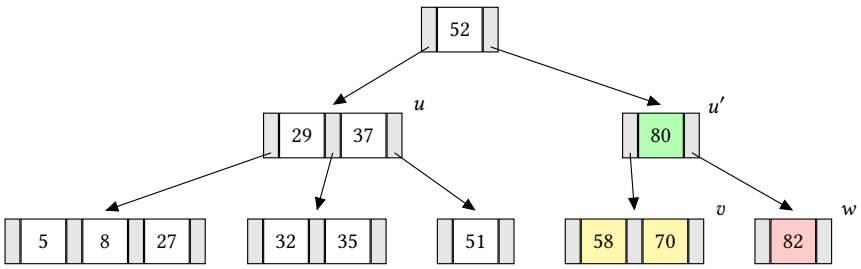
Abbildung 2.4: Pseudocode für Split



(a) vor *Split*



(b) nach *Split*, Wurzel zu voll



(c) nach weiterem *Split*

Abbildung 2.5: Nachdem 82 in B-Baum der Ordnung 4 eingefügt wurde ist Knoten v zu voll.

verändert den Grad des Elternknotens nicht.

Ist ein Transfer nicht möglich, muss ein Fuse durchgeführt werden. Dieser verschmilzt den unterlaufenen Knoten mit einem seiner Geschwister unter Einbezug eines Schlüssels des Elternknotens. Da hierbei ein weiterer Underflow im Elternknoten auftreten kann, der mit einer weiteren Umstrukturierung verbunden wäre, werden Transfers einem Fuse bevorzugt. Damit die Pseudocodes nicht länger als nötig werden, nutzen wir eine Hilfsfunktion, die den passenden Tranfer oder Fuse auswählt (siehe Abb. 2.7).

Eine Ausnahme beim Underflow stellt der Wurzelknoten des B-Baumes dar. Dieser kann nicht zu wenig Schlüssel besitzen und wird lediglich gelöscht, wenn er keine Schlüssel mehr enthält. Sein einziges Kind ersetzt ihn dann gegebenenfalls. Der Pseudocode für die Löschoperation ist Abb. 2.6 zu entnehmen.

Transfer

Angenommen ein Knoten u hat zu wenig Schlüssel und besitzt einen direkten Nachbarknoten w, der mehr als die minimale Anzahl an Schlüsseln hat. Sei p der Elternknoten von u und w mit dem Schlüssel s_p zwischen ihnen. Sei weiterhin $T[w']$ das linke Kind von w. Mit s_w als kleinsten Schlüssel in w gilt dann: $s_u \leq s_p \leq s'_w \leq s_w$ für $s_u \in u, s'_w \in T[w']$.

Diese Reihenfolge wird auch eingehalten, wenn s_w statt s_p in den Elternknoten gesetzt wird. Damit weiterhin $s_u \leq s_p \leq s_w$ gilt, wird s_u rechts zu den Schlüsseln von u hinzugefügt. Um $s_p \leq s'_w \leq s_w$ einzuhalten, wird $T[w']$ rechts an die Kinder von u angehangen.

Auf diese Weise werden genau ein Schlüssel und ein Kind von w nach u verschoben. Auf die restlichen Schlüssel und Kinder von w und u hat dies keine Auswirkung, da sie nach wie vor $\geq s_w$ beziehungsweise $\leq s_u$ sind. Ein Beispiel für eine Transferoperation ist in den Abbildungen 2.10e und 2.10f zu sehen.

Äquivalent lässt sich ein Schlüssel aus dem linken Geschwisterknoten transferieren [3]. Dann wird statt dem kleinsten der größte Schlüssel aus w in den Elternknoten gesetzt, während s_p und das rechte Kind von w links in u angehangen werden. Um den Pseudocode zu vereinfachen, nutzen wir hier den Index -1 um auf die rechten Elemente der entsprechenden Listen zuzugreifen. Der Pseudocode für diese Operation ist Abb. 2.8 zu entnehmen.

Fuse

Wenn kein Geschwisterknoten vom unterlaufenen Knoten u mehr als $\lceil \frac{m}{2} \rceil - 1$ Schlüssel besitzt, muss u mit einem dieser Geschwister verschmolzen werden. Damit dabei kein Knoten entsteht, der zwei Kinder mehr als Schlüssel besitzt, wird ein Schlüssel aus dem gemeinsamen Elternknoten einbezogen. Dadurch wird auch in diesem das Verhältnis von Schlüsseln und Kindern gewahrt.

Knoten der Ordnung m benötigen mindestens $\lceil \frac{m}{2} \rceil - 1$ Schlüssel (siehe Definition 1). Der unterlaufene Knoten hat vor der Operation einen zu wenig. Nach dem Fuse kommt er daher auf $\lceil \frac{m}{2} \rceil - 2$ eigene Schlüssel, plus $\lceil \frac{m}{2} \rceil - 1$ seines Geschwisterknotens und +1 aus dem Elternknoten. Insgesamt ergibt das $\lceil \frac{m}{2} \rceil \cdot 2 - 2$ Schlüssel. Bei gerader Ordnung sind das einer

[3] Per Konvention wird erst versucht von rechts zu transferieren.

weniger als die maximal möglichen $m - 1$ Schlüssel. Bei ungerader Ordnung ist der Knoten nun maximal befüllt [4].

Der Schlüssel aus dem Elternknoten liegt wie beim Transfer zwischen den beiden Knoten. *Fuse* ist damit eine umgekehrte *Split* Operation.

Da der Elternknoten um ein Element kleiner wird, kann dort ein weiterer Underflow entstehen. In dem Fall wird erneut zuerst ein Transfer versucht. Falls dieser nicht möglich ist, muss ein weiterer Fuse durchgeführt werden. Sollte die Wurzel des Baumes durch den Fuse keine Schlüssel mehr enthalten, wird diese durch den verschmolzenen Knoten ersetzt. Der Pseudocode für diese Operation ist Abb. 2.9 zu entnehmen.

[4] Für das nächstkleinere m bleibt der $\lceil \frac{m}{2} \rceil$ Teil aufgrund des Aufrundens gleich groß

```
function DELETE(Baum T mit Schlüssel x)
    found, node, i ← SEARCH(T.root(), x)
    if found then return
    if not T.aboveLeaf(node) then
        next ← T.children(node)[i + 1]                ▷ Rechtes(tes) Kind von node
        while not T.aboveLeaf(next) do
            next ← T.children(next)[0]                ▷ Anschließend Linkes
        Vertausche T.keys(node)[i] mit T.keys(next)[0]
        node ← next, i ← 0
    Entferne Schlüssel an Index i aus T.keys(node)
    if |T.keys(node)| < ⌈T.order()/2⌉ − 1 then        ▷ Underflow
        if not T.isRoot(node) then FIX_UNDERFLOW(node)
```

Abbildung 2.6: Pseudocode für Löschen

```
function FIX_UNDERFLOW(Knoten u in Baum T)
    min ← ⌈T.order()/2⌉
    p ← T.parent(u)
    childs_p ← T.children(p)
    i ← Index von u in childs_p
    right ← childs_p[i + 1]
    left ← childs_p[i − 1]
    if right existiert und |T.keys(right)| ≥ min then
        TRANSFER(u, right, "nach links")
    else if left existiert und |T.keys(left)| ≥ min then
        TRANSFER(left, u, "nach rechts")
    else if right existiert then FUSE(u, right)
    else FUSE(left, u)
```

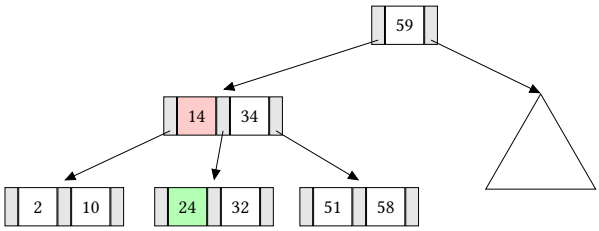
Abbildung 2.7: Hilfsfunktion um passende Underflowbehandlung zu starten

```
function TRANSFER(Knoten u, w in Baum T, dir)
    p ← T.parent(u)
    i ← Index von u in childs_p
    key_p ← T.keys(p)[i]
    if dir = "nach rechts" then
        Hänge key_p links an T.keys(w) an
        Hänge T.children(u)[−1] links an T.children(w) an
        Setze T.keys(u)[−1] in T.keys(p)[i]
    else
        Hänge key_p rechts an T.keys(u) an
        Hänge T.children(w)[0] rechts an T.children(u) an
        Setze T.keys(w)[0] in T.keys(p)[i]
```

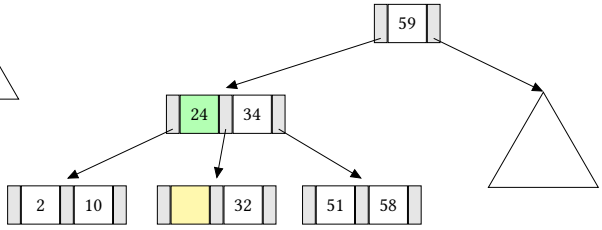
Abbildung 2.8: Pseudocode für Transfer

```
function FUSE(Knoten u und w in Baum T)
    p ← T.parent(u)
    i ← Index von u in T.children(p)
    key_p ← Entferne Index i aus T.keys(p)
    Entferne Index i + 1 aus T.children(p)
    Hänge key_p and T.keys(u) an
    Hänge T.keys(w) and T.keys(u) an
    Hänge T.children(w) and T.children(u) an
    if |T.keys(p)| < ⌈T.order()/2⌉ − 1 then          ▷ Weiterer Underflow
        if not T.isRoot(p) then FIX_UNDERFLOW(p)
```

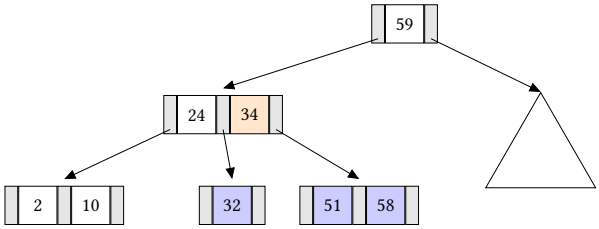
Abbildung 2.9: Pseudocode für Fuse



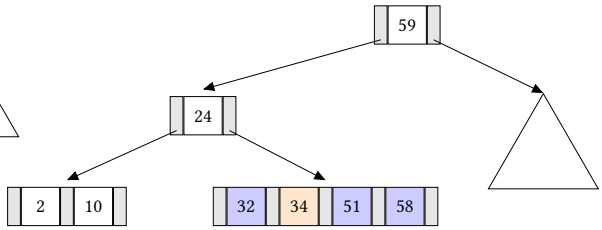
(a) 14 mit inorder Nachfolger tauschen und löschen



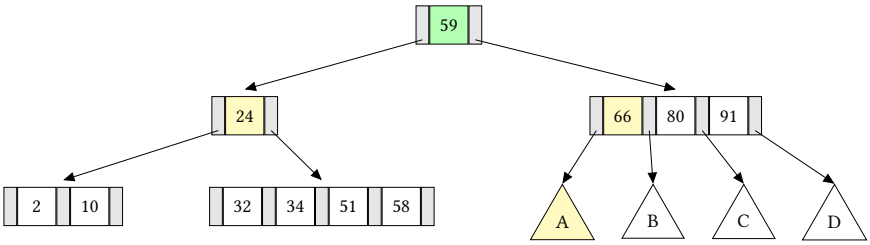
(b) 14 gelöscht. Underflow, *Transfer* nicht möglich



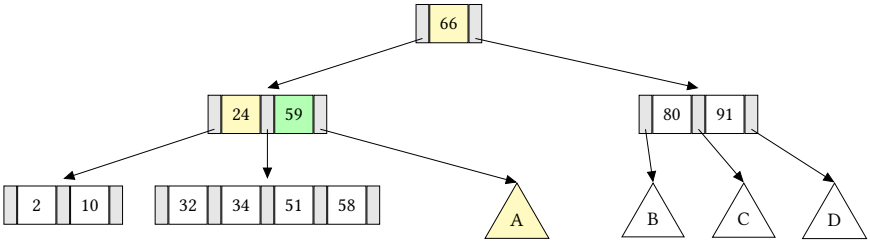
(c) *Fuse* mit rechtem Nachbarn



(d) *Fuse* abgeschlossen. Underflow in Elternknoten



(e) *Transfer* möglich



(f) *Transfer* abgeschlossen

Abbildung 2.10: Lösche 14 aus B-Baum der Ordnung 5

3 Anforderungen an die Nutzeroberfläche

Die Nutzeroberfläche benötigt eine Zeichnung der B-Bäume, ein Textfeld für den Pseudocode und Bedienflächen, um die Operationen zu starten. Die Bedienflächen sollen einem Debugger ähneln, sodass der Nutzer damit Schrittweise den Pseudocode ausführen kann.

Die Bedienelemente realisieren wir mit mehreren Reihen an Knöpfen. Einfügen, Löschen und Suchen brauchen dabei jeweils einen Startknopf. Die anderen Operationen werden jeweils als Unteraufruf gestartet, und können nicht vom Nutzer ausgewählt werden. Wir fügen auch Knöpfe zum Einfügen und Löschen von zufälligen Schlüsseln hinzu. Diese sind sowohl für den Nutzer, als auch zum schnelleren Testen der Programmlogik hilfreich. Um einen oder mehrere Schritte in der Ausführung zu machen, nutzen wir drei Knöpfe. Einen für einzelne Schritte, einen, um die ganze Operation auf einmal durchzuführen, und einen, der automatisch die Schritte durchläuft.

Die Textanzeige soll in der Lage sein, ein oder mehrere Reihen Pseudocode hervorzuheben. Diese müssen wiederspiegeln, in welchem Zustand sich die Zeichnung gerade befindet. Falls Unteraufrufe getätigt werden, beispielsweise bei der Suche innerhalb des Einfügens, soll der erste Pseudocode pausiert werden, und der Unteraufruf darunter angezeigt und durchlaufen werden.

Die Veränderungen, die im Pseudocode beschrieben werden, müssen in der Zeichnung animiert werden. Beispielsweise soll beim Splitten der mittlere Schlüssel aus dem Knoten gezogen werden, sodass daraus zwei neue Knoten entstehen. Die Suche könnte durch farbliche Hervorhebung angezeigt werden.

Es soll auch möglich sein, erstellte Bäume in einer Datei abzuspeichern, und diese später wieder in das Programm zu laden. Alternativ wäre auch denkbar, neue Bäume direkt nach dem Erstellen mit einigen zufälligen Elementen aufzufüllen, sodass der Nutzer nicht jedes mal mit einem leeren Baum startet.

4 Programmiersprache und Bibliotheken

4.1 Python

Die Anwendung ist in *Python 3.12* [Pyt24] geschrieben. Zum Ausführen wird mindestens Version *3.10* benötigt, da wir das dort eingeführte *Structural Pattern Matching* [Bra20] verwenden.

Die Programmiersprache Python zeichnet sich insbesondere durch simple und verständliche Syntax aus (Blöcke einrücken statt klammern, uvm.), sowie eine große Anzahl an verfügbaren Bibliotheken. Es werden verschiedene Programmierparadigmen unterstützt, sodass man für jeden Anwendungsfall selber entscheiden kann, ob zum Beispiel Objektorientierung von Nutzen ist. Python wird von allen gängigen Betriebssystemen unterstützt. Weiterhin hat die Sprache meist sehr ausführliche und klar zuzuordnende Fehlermeldungen, was beim Entwickeln überaus hilfreich ist. Lediglich in Einzelfällen, beispielsweise bei Vererbung, bleiben diese Fehlermeldungen manchmal aus.

Python Programme müssen nicht kompiliert werden. Stattdessen gibt es einen Interpreter, der den Quelltext während der Laufzeit liest und ausführt. Dies bringt den Vorteil mit sich, dass nicht für die jede unterschiedliche Plattform neu kompiliert werden muss. Ein Nachteil der Sprache ist, dass sie, aufgrund des Interpreters, vergleichsweise langsam ist. Dies betrifft jedoch hauptsächlich den selbstgeschriebenen Teil des Programmes. Bibliotheken sind meistens sehr gut optimiert und schnell, da sie auf auf C/C++ beruhen.

Da die Anwendung, die in dieser Arbeit entwickelt wird, lediglich zur Visualisierung dient, muss sie nicht besonders auf Resourcennutzung oder Laufzeit optimiert werden. Die Nachteile der Entwicklung in Python sind daher größtenteils irrelevant. Dazu kommt, dass die Studenten in der Vorlesung ebenfalls Python benutzen, und sie so die Möglichkeit haben, das hier entwickelte Programm nachzuvollziehen.

4.2 PyQt6 und pygraph

Es gibt viele Möglichkeiten in Python eine grafische Anwendung zu entwickeln. Eine der Bibliotheken mit den meisten Features ist PyQt [Riv24]. Das Qt Framework besitzt eine sehr große Auswahl an vordefinierten Klassen und Systemen. So gibt es hier beispielsweise schon Möglichkeiten zum Zeichnen und Animieren. Dazu kommt eine sehr ausgiebige Dokumentation. Weiterhin lässt sich die Qt Bibliothek auf allen gängigen Betriebssystemen installieren. Sie passt sich dort an das Aussehen der jeweiligen Nutzerumgebung weitestgehend an. So se-

hen die Knöpfe auf MacOs beispielsweise anders aus als auf Windows. Dies hilft dem Nutzer, sich in der Oberfläche zurechtzufinden.

Es gibt zwei Versionen des Qt Frameworks für Python: PyQt und PySide. Diese unterscheiden sich praktisch nicht voneinander, da beide nur die C++ API in Python zur Verfügung stellen. Sie nutzen beide frei verfügbare Lizenzen. Wir nutzen die PyQt Library. Die aktuellste Version zum Zeitpunkt dieser Arbeit ist v6.

Eine weitere Bibliothek, die auf PyQt aufbaut, ist pyqtgraph [dev24]. Sie wurde insbesondere für das Visualisieren von verschiedensten Daten entwickelt, und ähnelt matplotlib. Der Vorteil von pyqtgraph gegenüber nur PyQt ist, dass die Widgets, in denen gezeichnet werden kann, bereits Methoden zum Herein- bzw. Herauszoomen besitzen. Diese sind auch automatisch mit Mausevents verknüpft. Somit kann der Nutzer sich in der Baumzeichnung umherbewegen, ohne dass die Zeichnung von uns selber neu berechnet werden muss.

5 Quelltext

Die Anwendung lässt sich in drei Module aufteilen: Grafische Oberfläche, Implementierung der Baumstruktur und Zeichnen der Baumstruktur. Dementsprechend ist der Quelltext auch in die drei Dateien *gui.py*, *trees.py* und *draw.py* aufgeteilt. Einige Hilfsfunktionen wurden in *util.py* ausgelagert.

Kein eigenes Modul, aber ebenso wichtig, ist das Testen. Die Baumstruktur und alle Methoden, die darauf operieren, müssen während der Entwicklung getestet und auf Fehler durchsucht werden.

Die wichtigsten Klassen und Funktionen sind in den Abbildungen 5.1 bis 5.3 dargestellt. Der vollständige Quelltext befindet sich im `src/` Ordner der beiliegenden ZIP-Datei.

5.1 Baumstruktur

Bevor wir anfangen die Datenstruktur B-Baum zu implementieren, erstellen wir zwei allgemeinere Klassen: `Node` und `Tree`. Die Idee dabei ist, dass auch andere Baumstrukturen auf Grundlage dieser Klassen erstellt und alternativ zu den B-Bäumen in der Anwendung simuliert werden können. Dies ist kein Teil dieser Arbeit, bietet aber einen Ansatzpunkt, falls darauf aufgebaut werden soll.

Die Node Klasse wird anschließend als Grundlage für `BTNode` verwendet. Fast alle Methoden haben eine Auswirkung auf die Anzahl der Schlüssel und Kinder, weshalb die Methoden auch die Zeichnungen anpassen müssen. Daher sind die meisten Methoden in Node nur abstrakte Deklarationen, die von BTNode implementiert werden. Ähnlich verhält es sich mit der Tree Klasse. Sie dient als formelle Grundlage für `BTree`.

In einer Baumklasse, die als reine Datenstruktur dient, können Schlüssel (und zugehörige Items) sowie Referenzen auf Kinder einfach in jeweils einer Liste abgespeichert werden. Da die BTree Klasse jedoch zusätzlich für das organisieren der Zeichnung zuständig ist, müssen auch die Bilder von den Schlüsselelementen und den dazwischenliegenden Pointern abgespeichert werden. Damit nicht alle Elemente doppelt vorhanden sind — einmal als Wert und einmal als Zeichnung — kombinieren wir beides. Die Schlüssel werden in einer `BTreeItem` Klasse abgelegt und die Pointer in `BTreePtr`. Näheres zum Zeichnen und Animieren dieser Klassen wird in den Abschnitten 5.4 und 5.5 erläutert. Auf die Kinder eines Knotens wird über die BTreePointer zugegriffen.

5.1.1 BTree Klasse

Wie die BTNode Klasse, erhält auch die BTree Klasse einige extra Daten. Für den Zugriff auf alle Knoten einer Ebene, führen BTree Objekte eine Liste, in der alle benachbarten Knoten

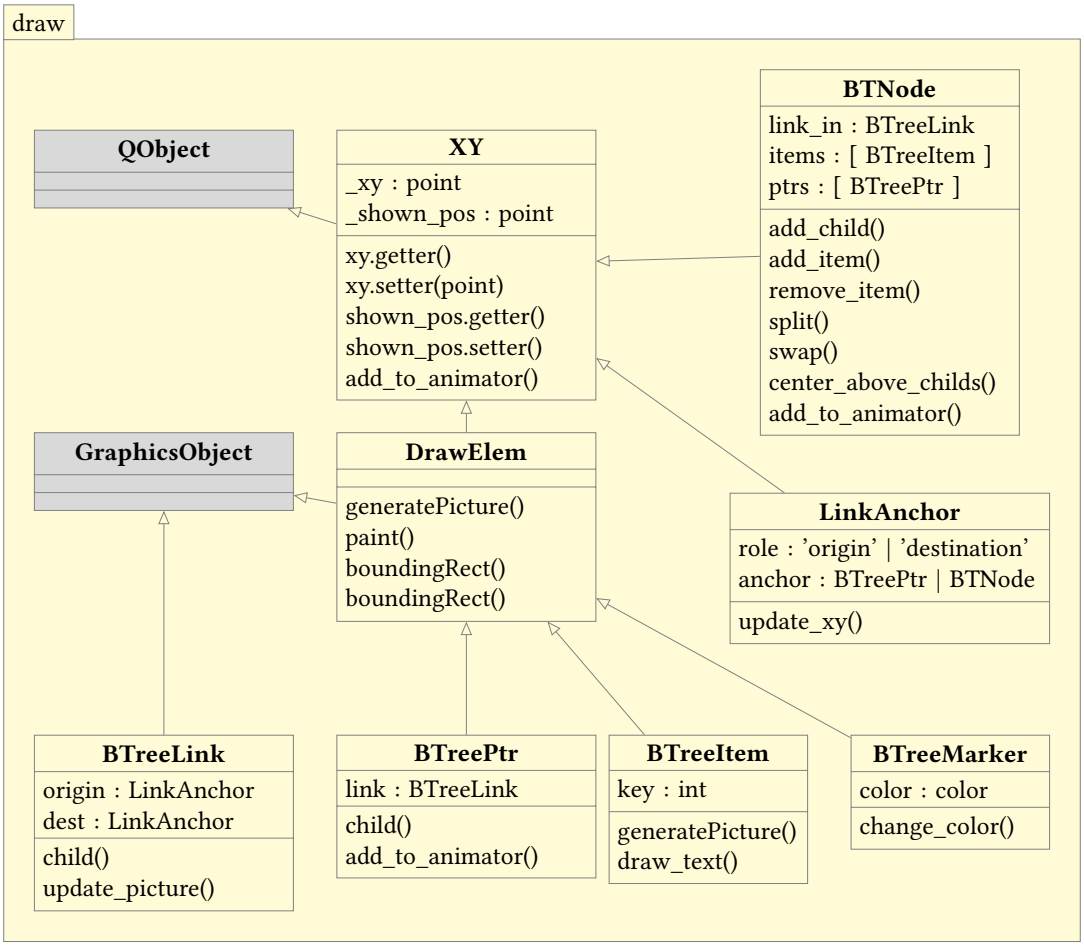


Abbildung 5.1: Wichtigste Klassen und Methoden des draw Moduls. Graue Klassen stammen aus Qt Bibliotheken

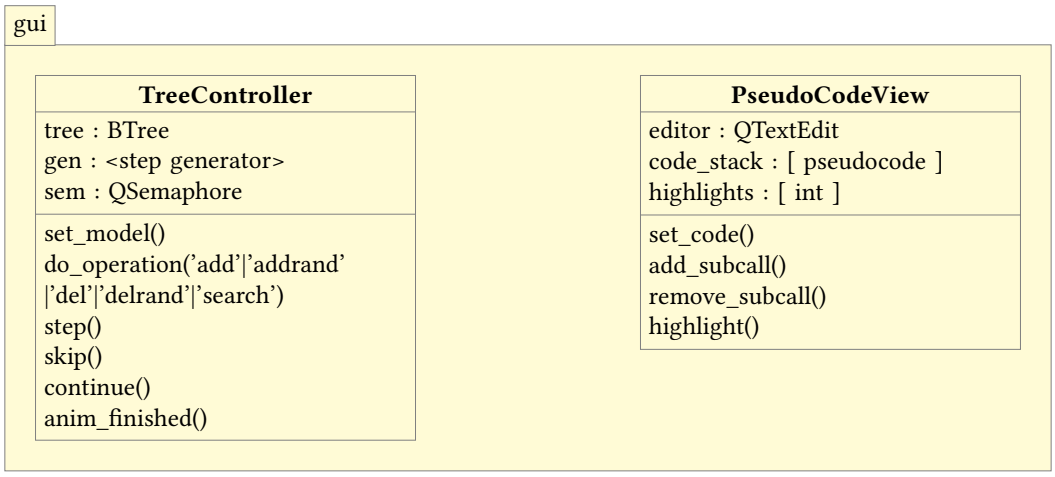


Abbildung 5.2: Wichtigste Klassen aus dem gui Modul

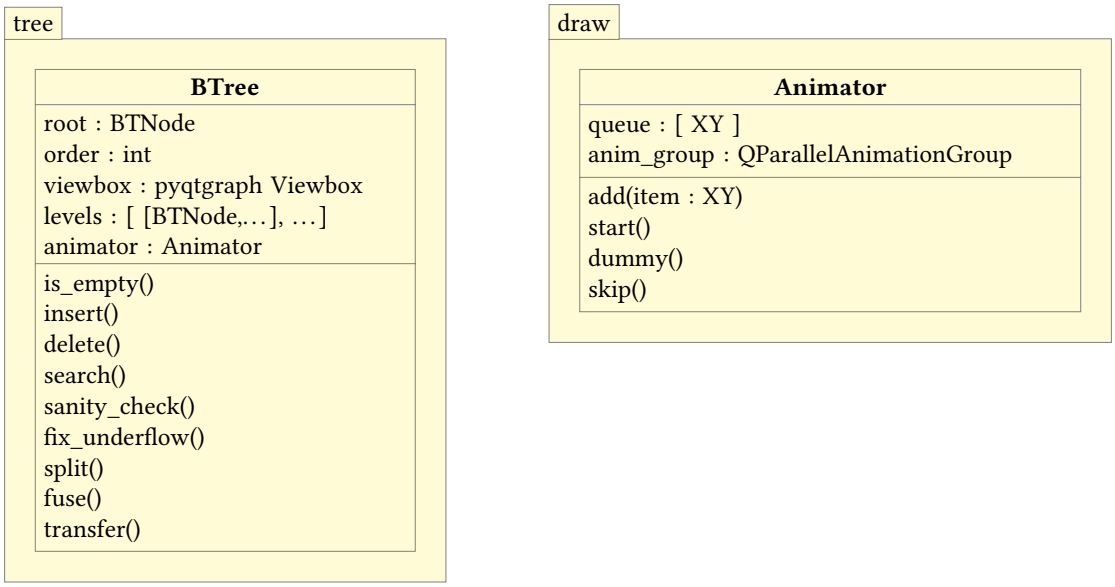


Abbildung 5.3: BTree und Animator Klasse

vermerkt werden. Diese wird beim Einfügen und Löschen aktualisiert. Für die Baumstruktur ist diese Liste nicht nötig, jedoch vereinfacht sie das Verschieben von Zeichnungen massiv, da nicht der ganze Baum durchlaufen werden muss. Weiterhin muss die Viewbox, welche als Zeichenfläche dient, bekannt sein, damit dort Elemente hinzugefügt oder gelöscht werden können. Außerdem erhält die Klasse ein `Animator` Objekt (siehe Abschnitt 5.5). Dieses kümmert sich um die Animation von allen Knoten- oder Schlüsselverschiebungen.

Um eine bessere Lesbarkeit zu ermöglichen, hat die BTree Klasse die Methoden `overflow`, `underflow` und `transfer_possible`. Sie geben einen einfachen Wahrheitswert zurück, der angibt, ob die jeweilige Eigenschaft erfüllt ist.

Sanity Check

Die erste Methode des B-Baumes, die wir implementieren, ist `sanity_check()`. Sie wird genutzt, um bei allen folgenden Schritten die Korrektheit der Baumstruktur zu garantieren. Dafür werden alle Punkte der Definition 1 durch Asserts geprüft. Alle Knoten müssen die korrekte Anzahl an Kindern für ihre Schlüssel haben. Die Kinder und Schlüssel müssen richtig sortiert sein. Dazu kommt, dass die Kinderknoten die korrekte Referenz auf ihren Elternknoten haben müssen, und der Wurzelknoten des Baumes keinen Elternknoten haben darf. Zusätzlich wird noch geprüft, ob die Zeichnungen aller Schlüssel und Pointer nebeneinander liegen.

Baumoperationen

Alle Funktionen, die die Baumoperationen aus Abschnitt 2.1 implementieren, sind grundsätzlich Implementierungen der entsprechenden Pseudocodes in Python. Wir fügen nur an

den passenden Stellen Pausen mithilfe von `yield` ein (siehe Abschnitt 5.2). Außerdem erstellen wir, insbesondere bei der Suche, ein `BTreeMarker` Objekt (siehe Abschnitt 5.4.2). Dieses wird immer an den Schlüssel bewegt, der gerade betrachtet wird.

Alle Operationen auf einem Knoten verändern jedoch die Zeichnung und Positionen eines oder sogar aller Knoten im Baum. Selbst, wenn nur in einem Knoten etwas eingefügt wird, muss dafür alles verschoben werden, um Platz für die neue Zeichnung zu machen. Damit die Logik für dieses Verschieben nicht in jeder Funktion erneut eingefügt werden muss, und damit die BTree Klasse so übersichtlich wie möglich bleibt, lagern wir das meiste in Methoden von BTNode aus. Diese ist sowieso für die Zeichnungen der zugehörigen Schlüssel und Pointer verantwortlich. Wir greifen daher nicht von BTree Methoden aus direkt auf beispielsweise die Liste mit Kindern zu, sondern verwenden die BTNode Methode `add_child`. Die verwendeten Methoden sind in dem Diagram Abb. 5.1 zu finden.

Die meisten dieser Methoden erschließen sich selbst, daher werden hier nicht alle einzeln erläutert. Sie alle sorgen dafür, dass sobald eine Zeichnung verändert oder verschoben wurde, dies dem Animationsobject des B-Baumes mitgeteilt wird.

5.2 Ausführung Pausieren

Während auf den nächsten Schritt des Nutzers gewartet wird, müssen die Baumoperationen unterbrochen werden. Dies kann auf verschiedene Weisen implementiert werden. Eine Möglichkeit wäre, die in Abschnitt 5.1 beschriebenen Funktionen einfach in viele kleine Funktionen aufzuteilen und alle Variablen in einer gemeinsamen Datenstruktur zu speichern. Die Funktionen könnten beispielsweise der Reihe nach in eine Liste eingefügt werden. Dadurch ließe sich für jeden Schritt, den der Nutzer durchführen will, die nächste Funktion der Liste ausführen. Hierbei müsste jedoch sehr viel mehr Quelltext geschrieben werden, als für die eigentliche Funktion nötig wäre. Jeder Schritt bräuchte einen eigenen Funktionskopf und Rückgabewert, und auf alle Funktionen müsste zusätzlich eine Referenz in eine Liste gelegt werden. Außerdem ist die Fehleranfälligkeit bei sehr vielen kleinen Funktionen, die alle die selben Daten manipulieren, sehr hoch. Wir verwerfen diese Idee daher.

Eine weitere Möglichkeit ist die Nutzung von Threads. Jede Baumoperation braucht dabei nur eine Funktion, die jeweils in einem seperaten Kontrollfluss abgearbeitet wird. Der Thread mit der aktuellen Operation kann einfach unterbrochen werden, bis der Nutzer den nächsten Schritt starten will. Allerdings ist auch diese Variante schwerer umzusetzen als erwartet. Jeder Thread bräuchte eine Möglichkeit, sich selber an einer bestimmten Stelle zu unterbrechen, und müsste garantiert den Kontrollfluss wieder an das Hauptprogramm geben. Die Umsetzung dieser beiden Punkte resultiert in deutlichem Mehraufwand bei der Programmierung.

Python bietet uns noch eine dritte Möglichkeit: *Generatoren*. Ein Generator ist eine Funktion, die durch das Schlüsselwort `yield` unterbrochen wird und einen Wert zurückgibt. Um die Werte zu erhalten, kann der Generator in einer For-Schleife durchlaufen werden. Alternativ kann die eingebaute Pythonfunktion `next()` benutzt werden [1]. Bei der Nutzung von Generatoren bleibt der Kontrollfluss unseres Programms garantiert linear. Dazu kommt, dass über

[1] For-Schleifen nutzen intern auch `next()`

die yields kommuniziert werden kann, an welcher Stelle des Pseudocodes sich die Funktion gerade befindet.

Allen Klassen, die von der Nutzeroberfläche pausiert werden müssen, werden daher an den Stellen, wo der nächste Pseudocodeschritt beginnt, durch yield pausiert. Der zurückgegebene Wert gibt an, ob eine neue Zeile Pseudocode markiert oder eine weitere Funktion aufgrund eines Unteraufrufs hinzugefügt werden muss. Zu Beginn einer neuen Funktion gibt diese einen String mit dem zugehörigen Pseudocode an die Nutzeroberfläche. Wenn eine Funktion Unteraufrufe tätigt, die ebenfalls Generatoren sind, werden die Werte mithilfe von `yield from` weitergereicht.

5.3 Tests

Beim Implementieren der Baumlogik und Knoten werden unweigerlich Fehler passieren, die erst gefunden, und anschließend mühsam lokalisiert und behoben werden müssen. Um den Mehraufwand, der dadurch entsteht, zu minimieren, wird von Beginn an ein Skript zum Testen mitentwickelt.

In Python gibt es die Möglichkeit, das eingebaute Modul `unittest` zu verwenden. Mit diesen Unittests können verschiedene Tests von einer Klasse oder Methode in einem *TestCase* zusammengefasst werden. Die Testcases lassen sich dann regelmäßig automatisch durchlaufen. Um Testcases zu erstellen, muss man allerdings von vornerein wissen, welche Randfälle auftreten und Probleme machen könnten. Außerdem müssen alle Testfälle erst manuell erstellt werden. Unittests sind daher vor allem geeignet, um sicherzugehen, dass ein vorhandenes Programm nach einigen Änderungen noch gleich funktioniert, und keine neuen Fehler eingebaut wurden.

Eventuelle Randfälle und Probleme sind vor Beginn dieser Arbeit jedoch noch nicht bekannt. Weiterhin kann das Schreiben einer geeigneten Anzahl von Testfällen sehr viel Zeit in Anspruch nehmen. Daher nutzen wir stattdessen zufällig generierter Bäume als Tests, die sich schnell und in großer Anzahl generieren lassen. Dazu werden eine Vielzahl an Assert Statements überall im Programm verteilt, die fehlerhafte Programmzustände aufdecken. Ziel ist es, in jeder Methode die ein- und ausgegangenen Werte auf Plausibilität zu prüfen. Somit lässt sich beispielsweise sofort erkennen, dass ein Kindknoten im Baum eine falsche Referenz auf seinen Elternknoten besitzt. Ohne Asserts würde dieser Fehler erst dann auffallen, wenn der Knoten irgendwann voll genug ist, um gesplittet zu werden. Dann hätte auf einmal ein falscher Knoten den verschobenen Schlüssel erhalten.

Mithilfe der Asserts konstruieren wir eine Methode, die für jeden Knoten im Baum alle Eigenschaften eines B-Baumes auf Richtigkeit überprüft. Innerhalb der zufällig generierten Testbäume kann diese Methode dann nach jedem Schritt aufgerufen werden.

Um die zufälligen Bäume zu generieren, nutzen wird das eingebaute Python Modul `random`. In einem seperaten Python Skript werden unsere `draw` und `tree` Module importiert. Dort werden dann Bäume mit Zufallszahlen befüllt. Mithilfe der Asserts, und der `sanity_check` Methode (siehe Abschnitt 5.1.1), werden die Bäume auf Korrektheit überprüft.

5.4 Zeichnen

Die Nutzeroberfläche (siehe Abschnitt 5.6) nutzt eine `pyqtgraph.Viewbox` in der die Baumstruktur gezeichnet werden soll.

Das *pyqtgraph* Framework bringt einige Möglichkeiten mit sich, verschiedenste Arten von Diagrammen und Graphen anzuzeigen. Da es jedoch keine für B-Bäume oder ähnliche Baumstrukturen gibt, müssen wir die Bäume manuell zeichnen. Dazu müssen `pyqtgraph.GraphicsObject`(s) erstellt werden, die dann in der Viewbox angezeigt und bewegt werden können.

GraphicsObject dient als Elternklasse. Es müssen lediglich die Methoden `paint()` und `boundingRect()` implementiert werden. Damit die Objekte nicht bei jeder Veränderung der Viewbox neu berechnet werden müssen, ist es sinnvoll, sie bei der Initialisierung einmal zu berechnen und abzuspeichern. Dann kann die Zeichnung bei jedem Aufruf von `paint()` immer wiederverwendet werden.

Um das Bild zu berechnen, erhält die Klasse eine `generatePicture()` Methode. In dieser wird dann ein `QPicture` erstellt, in welchem mit `QPainter`-Objekten gezeichnet werden kann. Die `QPainter` Klasse kann eine Vielzahl an geometrischen Formen erstellen. Für das Zeichnen eines B-Baumes reichen dabei `drawRect()`, `drawText()` und `drawLine()`.

Da die einzelnen Schlüssel und Pointer des Baumes getrennt voneinander bewegt werden müssen, und nicht dauerhaft in der selben Knotenformation verweilen, werden für sie einzelne Bilder erstellt. Verwaltet werden sie in unserer `BTNode` Klasse (siehe Abschnitt 5.1).

5.4.1 XY

PyQt und darauf aufbauend pyqtgrpah stellen die Methoden `pos()` und `setPos()` zur Verfügung. Mit diesen wird die aktuelle Position einer Zeichnung in der Viewbox verändert. Die Koordinaten liegen allerdings nicht in einem einheitlichen Koordinatensystem, sondern nutzen immer die Position, an der die Objekte ursprünglich gezeichnet wurden als Ursprung. Deshalb können wir mit diesen Koordinaten nicht direkt arbeiten. Stattdessen speichern wir in jedem Objekt die aktuelle Position, und berechnen die Differenz zur gewünschten Position. Diese Differenz wird dann auf den von `pos()` zurückgegebenen Punkt addiert und an `setPos()` weitergereicht. Um unsere aktuelle Position zu speichern, nutzen wir einen `QPoint`, der einfach ein Tupel aus x- und y-Koordinate ist.

Für die Animation wird dieser Punkt um einen weiteren ergänzt, der die momentan sichtbare Position speichert. Den ersten Punkt nennen wir `xy`, den sichtbaren `shwon_pos`. Mit diesem System können wir die tatsächliche Position von Elementen verändern und berechnen, ohne die Zeichnung direkt verschieben zu müssen. Das kann im weiteren Verlauf genutzt werden, um das Verschieben zu animieren (siehe Abschnitt 5.5).

Außerdem ist es so möglich, mehrere Baumoperationen nacheinander durchzuführen, und erst am Ende die Anzeige zu aktualisieren. Die Schritte nehmen dann deutlich weniger Rechenleistung und Zeit in Anspruch.

In der XY Klasse sind die xy und shown_pos Punkte als Properties implementiert. Für shown_pos nutzen wir `pyqtProperty` statt der normalen Python Properties. Diese verhalten sich wie normale Python Property, sind aber in das Qt System eingebunden. Das ermöglichen beispielsweise das Animieren durch `QPropertyAnimation`. Properties verhalten sich

nach außen hin wie normale Klassenattribute. Bei Zuweisungen wird jedoch eine Settermethode aufgerufen, beim Abrufen ein Getter. Dadurch können wir sichergehen, dass der Punkt nicht versehentlich als Referenz übergeben wird. Wir erstellen in beiden Methoden immer eine Kopie. Durch das Kopieren der Objekte wird unser Programm zwar minimal langsamer, allerdings ist es deutlich weniger fehleranfällig.

Es wäre auch möglich statt der QPoint Klasse ein simples Tupel aus zwei Integern zu verwenden. Das wäre auch minimal effizienter. Das Programm ist mit QPoints allerdings deutlich übersichtlicher, da sie schon alle benötigten Operationen wie beispielsweise Addition von zwei Punkten unterstützen. Da Effizienz für diese Anwendung nur eine nebensächliche Rolle spielt, ist der geringe Rechenoverhead zu vernachlässigen.

Dadurch, dass wir eigene Getter und Setter definieren können, ist es außerdem möglich, aus dem Setter heraus noch andere Methoden zu starten. Dies nutzen wir vor allem für den shown_pos Punkt, um falls nötig automatisch die Zeichnungen neu zu berechnen.

5.4.2 DrawElem

Als Grundlage für das Zeichnen von sowohl BTreeLink als auch BTreePtr Elementen dient die Klasse DrawElem. Diese erbt, wie in Abschnitt 5.4 beschrieben, von der pyqtgraph.GraphicsObject Klasse. Die generatePicture Methode bleibt hier abstrakt, da die verschiedenen Elemente unterschiedlich gezeichnet werden. Zusätzlich zu GraphicsObject wird von unsere eigenen Klasse XY geerbt (siehe Abb. 5.1). Für die Animation wird hier von dieser der Setter für `shown_pos` überschrieben. In diesem wird die Zeichnung mithilfe von GraphicsObject Methoden verschoben.

BTreeItem und BTreePtr

Um ein Schlüsselelement des Knotens zu zeichnen, brauchen wir lediglich ein Rechteck und einen Text. Da große Zahlen nicht über das Rechteck hinauswachsen sollen, prüfen wir das vorher und veringern gegebenenfalls die Schriftgröße.

Ein Pointer braucht wie der Schlüssel auch lediglich ein Rechteck. Dieses färben wir grau ein.

BTreeMarker

Der Marker wird ähnlich gezeichnet wie das BTreeItem. Jedoch wird das Rechteck farbig ausgemalt. Wir wollen die Farbe des Markers jedoch an manchen Stellen ändern. Beispielsweise in der Suche, wenn ein Schlüssel gefunden wurde. Daher fügen wir eine `change_color()` Methode hinzu. Diese muss ein neues Bild mithilfe von `generatePicture()` zeichnen und anschließend der Viewbox mitteilen, dass der Inhalt der Zeichnung sich geändert hat. Dies geschieht über Aufrufen der von GraphicsObject geerbten `update()` Methode.

5.4.3 BTreeLink

Um die Pfeile zu zeichnen, die Pointer und Knoten verbinden, können wir nicht einfach auf eine ferige Methode zurückgreifen. Stattdessen nutzen wir Qts `draw_line` um alle nötigen

Linien selber zu ziehen.

Pfeile zwischen den Knoten können nicht einfach mitverschoben werden können, wenn ein Knoten die Position wechselt, da Anfang und Ende sich in verschiedene Richtungen bewegen können. Ähnlich wie beim BTreeMarker müssen wir daher die Zeichnung neu berechnen.

Um zu wissen, wo Anfang und Ende des Pfeils liegen, nutzen wir eine weiter Klasse `LinkAnchor`, mit der die Ankerpunkte bewegt werden. Von diesen nutzen wir dann jeweils den shown_pos Punkt.

Durch die neue LinkAnchor Klasse ist es möglich, ebenfalls von `XY` zu erben, und somit die Voraussetzungen für die QPropertyAnimation zu erfüllen (siehe Abschnitt 5.4.1 und Abschnitt 5.5).

5.5 Animation

In PyQt können verschiedenste QWidgets mithilfe der `QPropertyAnimation` Klasse animiert werden. Diese kann über alle Möglichen `pyqtProperty` Werte interpolieren. Dabei ist lediglich ein Start- und Endwert nötig. Wir nutzen daher das `shown_pos` Attribut von unserer `XY` Klasse um eine Zeichnung von ihrer momentanen Position zu der Zielposition, die im xy Attribut steht, zu bewegen.

Um mehrere Zeichnung gleichzeitig zu bewegen, kann die `QParallelAnimationGroup` Klasse genutzt werden. In dieser werden mehrere Animationsobjekte gesammelt und gleichzeitig gestartet.

Animator

In praktisch jeder Methode von BTNode und BTree müssen eine Vielzahl von Zeichnungen bewegt werden. Außerdem müssen manche Animationen nicht direkt gestartet werden sondern erst dann, wenn für alle benachbarten Zeichnungen auch eine neue Position berechnet wurde. Daher sammeln wir alle Elemente, die bewegt werden müssen, in einer Liste im BTree. Diese Liste ist Teil unserer `Animator` Klasse. Durch ihre Methoden `start()` werden alle gesammelten Zeichnungen in eine neue QParallelAnimationGroup gelegt und gleichzeitig bewegt. Über die Animator Klasse können wir die Animation auch überspringen, zum Beispiel, falls der Nutzer direkt den nächsten Schritt einer Baumoperation startet.

Eine Zusätzliche Methode des Animators ist `dummy()`. Um Animationen nacheinander ablaufen zu lassen verbindet die GUI sich mit dem finished-Signal der AnimationGroup. Wenn im Pseudocode nur Variablen zugewiesen werden, und keine Zeichnungen bewegt werden muss, würde der entsprechende Schritt ohne Animation direkt übersprungen werden. Da wir den Pseudocode aber auch Schrittweise durchlaufen wollen, starten wir eine leere Animation, die nur dazu dient, das finished-Signal nach verstrichener Zeit zu senden.

Es wäre auch denkbar eine andere Lösung für dieses Problem zu implementieren, beispielsweise mit Threads. Da wir aber schon ein System haben, dass mithilfe der finished-Signale funktioniert, erweitern wir dieses stattdessen. Damit sparen wir Aufwand und vermeiden

potentielle Fehlerquellen durch Threads. Eine minimale Verschwendung an Rechenleistung nehmen wir dabei in Kauf.

5.6 Grafische Oberfläche

Die Nutzeroberfläche besteht hauptsächlich aus der Baumanzeige, der Pseudocodeanzeige und den Baumoperationen. Dazu kommen noch eine Menüleiste und ggf. Popupfenster. Sie alle werden von einem `QMainwindow` gemanaged. Um die Baum- und Pseudocodeanzeigen mit den Buttons für die Operationen zu verknüpfen, bietet sich das Model View Controller Muster an. Das Modell ist dabei eine Instanz der B-Baum Klasse. Die `TreeController` Klasse muss sich darum kümmern, dass die Buttons mit den entsprechenden Methoden des Baumes verknüpft werden.

5.6.1 Pseudocode

Um den Pseudocode anzuzeigen und die aktuellen Zeilen zu markieren, wird ein `QTextEdit` Widget genutzt. Der Pseudocode wird als String mit Markdown Syntax verfasst. Das Textedit Widget besitzt eine `setMarkdown` Methode, mit der dieser String automatisch formattiert wird.

Der große Nachteil dieser Methode ist, dass jedes Wort einzeln mit Markdown Syntax versehen werden muss. Dabei werden leicht Wörter übersehen oder vergessen. Da der Pseudocode der B-Bäume sich innerhalb dieser Arbeit nicht oft ändern sollte, wird diese Methode trotzdem implementiert. Eine andere Möglichkeit wäre die Nutzung eines `QSyntaxHighlighter`. Diese Option wird im Abschnitt 6.6 auf Seite 28 beschrieben.

5.6.2 Debugging Buttons

Die Knöpfe zur Operationsauswahl und Schrittweisen Ausführung werden mit dem TreeController verbunden. Dieser besitzt die Methode `do_operation()`, mit der neue Operationen gestartet werden. Wenn ein valider Wert im Eingabefeld vorliegt, wird im Baummodell die entsprechende Operation gestartet. Dabei erhält der TreeController einen Generator, den er zwischenspeichert.

Bevor ein Generator vorliegt, werden die Knöpfe zum Schrittweisen Ausführen ausgegraut. Sobald ein neuer Generator erstellt wurde, werden die Knöpfe aktiviert. Wenn der Nutzer diese nun anklickt, wird die `step()` Methode ausgeführt. Durch Aufrufen von `next()` wird der nächste Wert des Generators geholt. Wie in Abschnitt 5.2 beschrieben, wird ein Schritt der Baumoperation durchgeführt, anschließend wartet der Generator wieder.

Der Rückabewert des Generators gibt an, ob eine neue Zeile des Pseudocodes markiert, eine Animation gestartet, oder ein komplett anderer Code angezeigt werden muss. Falls ausschließlich ein neuer Schritt markiert werden soll, wird trotzdem eine Animation gestartet. Diese bewegt jedoch keine Elemente. Sie ist alleine dafür da, die Continue Methode anzuschubsen, sofern der entsprechnde Knopf momentan aktiv ist.

Wenn mehrere Schritte auf einmal durchgeführt werden sollen, weil der Nutzer auf Skip geklickt hat, wird einfach solange die step Methode durchgeführt, bis der Generator am Ende angekommen ist. Zu Beginn von step werden vorherige Animation immer übersprungen, sodass die Zeichnung aktuell bleibt.

Der Continue Knopf funktioniert ähnlich wie Skip, nur dass auf das Ende der Animation gewartet werden muss. Praktischerweise besitzen PyQt Animationen ein `finished` Signal, mit dem man sich verbinden kann. Wir schreiben daher eine Methode `anim_finished()`, die am Ende einer Animation gestartet wird.

Da dies zu einem beliebigen Zeitpunkt passieren kann, und es möglich ist, dass gerade eine vom Nutzer gestartete step Methode läuft, müssen wir den TreeController um einen Mutex erweitern, der garantiert, dass nur ein Kontrollfluss gleichzeitig auf den Generator zugreifen kann. Dafür nutzen wir die von Qt bereitgestellte `QSemaphore`. Zu Beginn von step und anim_finished wird die Semaphore reserviert, was den Kontrollfluss gegebenenfalls blockiert, bis sie wieder frei ist. Die Semaphore nutzen wir auch, wenn ein neuer Baum generiert werden soll.

6 Probleme und Verbesserungspotential

6.1 Garbage Collector

PyQt6 ist nicht nativ in Python geschrieben sondern in C++. Daher konkurrieren beim Nutzen der Bibliothek der PyQt interne Garbage Collector mit dem von Python. Konflikte, die hierbei auftreten können, sollten normalerweise von der Bibliothek aufgelöst werden. Dies funktioniert allerdings nicht immer, wenn unklar ist, ob ein Objekt zu PyQt oder zu Python gehört.

Da für die Implementierung der grafischen Darstellung von einigen PyQt bzw. pyqtgraph Klassen geerbt wurde, scheint es auch bei dieser Anwendung zu einem Konflikt zu kommen. Dieser tritt erst auf, wenn die Anwendung beendet wird, und auch nur dann, wenn mindestens ein Element aus einem Baum gelöscht wurde. Python meldet dabei einen `Memory Access Error`. Dabei ist egal, ob noch eine Referenz auf das Baumobjekt existiert, oder bereits ein neuer Baum erstellt wurde und angezeigt wird. Ebenso hat es keine Auswirkung, ob der Baum überhaupt in einer Viewbox angezeigt wird und eine Referenz auf diese besitzt. Lediglich das Abspeichern der Elemente, beispielsweise in einer Liste, die nicht gelöscht wird, umgeht diese Fehlermeldung. Selbst dann, wenn die Liste nur innerhalb des Baum Objektes gespeichert wird, das beim Erstellen eines neuen Baumes zerstört werden müsste.

Die Python Garbage Collection ist extrem undurchsichtig und darüber hinaus auch nicht deterministisch, daher ist es uns nicht gelungen den Ursprung von diesem Fehler zu finden. Während der Laufzeit des Programms scheint dieses Problem keine Auswirkung zu haben. Der Speicherabdruck des Programms steigt bei vielfachem Einfügen und Löschen deutlich an (von ca 100MB auf 250MB nach 8000 insert und delete Aufrufen). Ein Memory Leak im Programm oder der Bibliothek ist daher nicht ausgeschlossen.

Es ist möglich, dass die Verwendung von `weakrefs` den Garbage Collector Konflikt beheben könnte. Aufgrund des begrenzten Bearbeitungszeitraumes werden wir diese Idee jedoch nicht weiter verfolgen.

6.2 Visuelle Artefakte

Gelegentlich bleiben in der Viewbox Artifakte von verschobenen oder gelöschten Zeichnungen übrig. Diese verschwinden jedoch sobald die Viewbox durch ihre enableAutoRange() Methode aktualisiert wird. Die Ursache für diese Artefakte konnte nicht festgestellt werden. Alle gezeichneten Elemente informieren die Viewbox darüber, wenn ihre Größe sich

verändert hat, und eine Verschiebung von Elementen wird nur durch Viewbox Methoden bewerkstelligt.

6.3 Schriftgrößen

Je nach Standardschrift oder Displayauflösung wird eine andere Schriftgröße für den Text in den Boxen der Baumelemente benötigt [1]. Da nicht auf allen Systemen die selben Schriftarten vorhanden sind, zeichnen wir den Text erst in einer Größe, die auf allen getesteten Systemen entweder passend oder zu groß ist. Gegebenenfalls wird die Größe dann reduziert.

Es wäre auch denkbar, bei jedem Programmstart einmalig die passende Größe auszurechnen. Dies spart jedoch nur marginal Rechenleistung ein und ist für die Funktionsweise in keiner Weise relevant. Aufgrund der begrenzten Bearbeitungszeit wird diese Lösung nicht implementiert.

6.4 Skalierung der Anwendung

Qt erkennt von alleine, ob es auf einem Display mit sehr hoher Pixeldichte, beispielsweise einem 4k Monitor, läuft. Die Größe seiner Elemente und Texte passt es dann automatisch an. Gerade auf kleinen Laptop Bildschirmen kann dies jedoch nicht ausreichend sein. Je nach Desktopumgebung hat der Nutzer eventuell die Möglichkeit, das Aussehen von Qt in den Einstellungen zu konfigurieren. Falls dies nicht der Fall ist, kann Qt durch die Umgebungsvariable `QT_SCALE_FACTOR` skaliert werden. Ein Programmaufruf wäre dann statt `$ python src/gui.py` etwa `$ QT_SCALE_FACTOR=1.5 python src/gui.py`.

Um die Anwendung dynamisch zu skalieren, müssten alle Elemente der Nutzeroberfläche mit expliziten Größen versehen werden. Diese könnten dann während der Laufzeit neu berechnet und angepasst werden. Wir nutzen in dieser Arbeit die Lösung über Umgebungsvariablen, da sie deutlich simpler und einfacher umzusetzen ist.

6.5 Sprache in Nutzeroberfläche

Um zu den Funktionsnamen im Pseudocode zu passen, sind die Beschriftungen der Bedienelemente in Englisch gehalten. Der Rest der Nutzeroberfläche ist der Einfachheit halber daran angepasst. Wenn für den Nutzer mehrere Sprachen zur Auswahl stehen sollen, ist es in PyQt möglich, alle Texte übersetzen zu lassen. Dafür müssen alle Strings durch in einen `tr()` Aufruf an einen Übersetzer gegeben werden. Dieser braucht dafür eine Wörterbuchdatei mit allen Übersetzungen. Da Mehrsprachigkeit für diese Anwendung keine Priorität hat und Englisch vollkommen ausreichen sollte, wurde dies nicht gemacht.

[1] Unter anderem: Macbook Retina 23pt, Linux Full HD Laptop 16pt, windows mit 1440p Monitor, 20pt

6.6 Syntax Highlighter

Wenn man den Pseudocode einfacher ändern möchte oder öfters neuen Pseudocode hinzufügt, wäre es sinnvoll, ihn als Plaintext abzuspeichern. Mithilfe eines `QSyntaxHighlighter` kann dann die Formatierung von Qt übernommen werden. Diesem Syntaxhighlighter werden Muster in Form von regulären Ausdrücken gegeben. Die gefundenen Wörter werden dann automatisch mit dem zugehörigen Format versehen.

Die Herausforderung dabei ist es, einen regulären Ausdruck zu schreiben, welcher zwischen Variablennamen und normalem Text unterscheiden kann. Das ist nicht ohne weiteres möglich, da es keine Merkmale gibt, die die beiden unterscheiden. Einzig bei der Zuweisung von Variablen lassen sich diese anhand des Pfeils, der auf sie zeigt, erkennen. Denkbar wäre, sich gefundene Variablen in einer Liste zu merken. Aus dieser kann dann ein neues Muster gemacht werden, das wiederholtes Auftreten der Variablen erkennt. Für Variablen, die nur im Funktsionskopf definiert werden, bräuchte es auch ein weiteres Muster.

6.7 Speicherfunktion

Die simpelste Art, ein Pythonobjekte auf eine Festplatte zu speichern, ist das `pickle` Modul. Damit können Daten automatisch serialisiert und auch wieder eingelesen werden. Pickle ist allerdings nicht in der Lage, PyQt Objekte zu serialisieren, weshalb für diese Anwendung ein eigener Speicheralgorithmus geschrieben werden müsste. Aufgrund der Tatsache, dass sich sehr einfach und schnell neue Bäume erstellen und mit Daten befüllen lassen, wird die Speicherfunktion nicht implementiert.

Literatur

[Bra20] Guido van Rossum <guido at python.org> Brandt Bucher <brandt at python.org>. *Structural Pattern Matching: Specification*. PEP 634. 12. Sep. 2020. URL: https://peps.python.org/pep-0634/.

[dev24] PyQtGraph developers. *API Reference*. Version 0.13.7. 2024. URL: https://pyqtgraph.readthedocs.io/en/pyqtgraph-0.13.7/api_reference/index.html.

[Edw72] Rudolf Bayer und Edward M. McCreight. *Organization and Maintenance of Large Ordered Indices*. 1972.

[Pyt24] Python Software Foundation. *The Python Language Reference*. Version 3.12.3. 6. Juni 2024. URL: https://docs.python.org/3.12/reference/index.html.

[Riv24] Riverbank Computing Limited, The Qt Company. *PyQt Documentation*. Version 6.7.0. 2024. URL: https://www.riverbankcomputing.com/static/Docs/PyQt6/.