TU Wien
Institut für Logic and Computation
Forschungsbereich Algorithms and Complexity

186.866 Algorithmen und Datenstrukturen VU

Programmieraufgabe P4

PDF erstellt am: 7. April 2022

1 Vorbereitung

Um diese Programmieraufgabe erfolgreich durchführen zu können, müssen folgende Schritte umgesetzt werden:

1. Laden Sie das Framework `P4.zip` aus TUWEL herunter.
2. Entpacken Sie `P4.zip` und öffnen Sie das entstehende Verzeichnis als Projekt in IntelliJ (nicht importieren, sondern öffnen).
3. Öffnen Sie die nachfolgend angeführte Datei im Projekt in IntelliJ. In dieser Datei sind sämtliche Programmiertätigkeiten durchzuführen. Ändern Sie keine anderen Dateien im Framework und fügen Sie auch keine neuen hinzu.
   `src/main/java/exercise/StudentSolutionImplementation.java`
4. Füllen Sie Vorname, Nachname und Matrikelnummer in der Methode `StudentInformation provideStudentInformation()` aus.

2 Hinweise

Einige Hinweise, die Sie während der Umsetzung dieser Aufgabe beachten müssen:

- Lösen Sie die Aufgaben selbst und nutzen Sie keine Bibliotheken, die diese Aufgaben abnehmen.
- Sie dürfen beliebig viele Hilfsmethoden schreiben und benutzen. Beachten Sie aber, dass Sie nur die oben geöffnete Datei abgeben und diese Datei mit dem zur Verfügung gestellten Framework lauffähig sein muss.

3 Übersicht

Diese Programmieraufgabe soll dabei helfen, das Wissen und die Intuition über Suchbäume aus der Vorlesung zu vertiefen und zu fördern. Gleichzeitig soll diese Aufgabe die Vorlesung um sogenannte Rot-Schwarz-Bäume ergänzen. Konkret sind sowohl einfache binäre Suchbäume als auch Rot-Schwarz-Bäume zu implementieren, damit diese in Folge verglichen werden können.

4 Theorie

Die notwendige Theorie für einfache binäre Suchbäume befindet sich in den Vorlesungsfolien „Suchbäume“.

Eine Art der balancierten binären Suchbäumen sind sogenannte Rot-Schwarz-Bäume. Im Gegensatz zu den höhenbalancierten AVL-Bäumen, besitzt ein Rot-Schwarz-Baum weniger strikte Vorgaben zur Einhaltung der Balance.

Ein Rot-Schwarz-Baum hat folgende Eigenschaften:

1. Jeder Knoten ist entweder rot oder schwarz.
2. Die Wurzel ist schwarz.
3. Jeder Blattknoten ist schwarz.
4. Ist ein Knoten rot, dann müssen seine Kindknoten schwarz sein.
5. Für jeden Knoten gilt, dass alle Pfade von diesem Knoten zu nachfolgenden Blattknoten die gleiche Anzahl an schwarzen Knoten beinhalten.

Diese Eigenschaften können Sie auch in ähnlicher Form aus den *aktuellen* Vorlesungsfolien „Praktische Datenstrukturen in Java — Ein Überblick“ entnehmen.

Insbesondere die Eigenschaften 4 und 5 der Rot-Schwarz-Bäume stellen dabei sicher, dass die Höhe eines Rot-Schwarz-Baumes durch $O(\log n)$ beschränkt ist. Beweise dazu stützen darauf, dass jeder Pfad von der Wurzel zu einem Blattknoten nur maximal zur Hälfte rote Knoten beinhalten kann.[1]

[1] Thomas H Cormen. *Introduction to Algorithms. Third Edition.* The MIT Press, 2009.

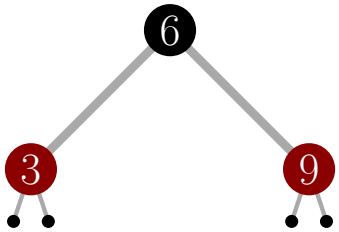


Abbildung 1: Rot-Schwarz-Baum

Ein Beispiel für einen Rot-Schwarz-Baum können Sie in Abbildung 1 sehen. Nicht vorhandene Kindknoten mit Schlüssel werden durch spezielle Nullknoten repräsentiert. Diese haben immer die Farbe Schwarz. In Abbildung 1 haben beispielsweise die Knoten mit den Schlüsseln „3“ und „9“ keine echten Kindknoten mehr. Die speziellen Nullknoten als deren Kindknoten sorgen dafür, dass Eigenschaft 3 erfüllt ist. Vergewissern Sie sich, dass alle fünf Eigenschaften im Baum in Abbildung 1 erfüllt sind.

4.1 Einfügen in einen Rot-Schwarz-Baum

Vor dem Einfügen hat ein Rot-Schwarz-Baum alle der oben beschriebenen Eigenschaften. Das Einfügen eines neuen Knotens q in einen Rot-Schwarz-Baum erfolgt zunächst ident zum Einfügen in einen einfachen binären Suchbaum. Der eingefügte Knoten q ist dabei immer rot gefärbt.

Im Anschluss kann es allerdings sein, dass der Baum nicht mehr alle Rot-Schwarz-Baum-Eigenschaften erfüllt. Folgende Probleme können auftreten:

- Ist der neu eingefügte (rote) Knoten q der erste Knoten, der eingefügt wurde, dann ist Eigenschaft 2 verletzt, denn die Wurzel muss schwarz sein.
- Ist der Elternknoten des neu eingefügten (roten) Knotens q rot, dann ist Eigenschaft 4 verletzt, denn ein roter Knoten darf keine roten Kindknoten haben.

Die Eigenschaften 1, 3 und 5 können nicht verletzt sein. Überlegen Sie, warum das der Fall ist und beantworten Sie Frage 1 in Abschnitt 8.

Nach dem Einfügen muss also ggf. ein Reparaturprozess durchgeführt werden, sodass der Baum wieder alle Kriterien eines Rot-Schwarz-Baum erfüllt. Dabei ist die Behebung der Verletzung von Eigenschaft 4 ein iterativer Prozess, bei

der vor und nach jeder Iteration folgender Zustand vorzufinden ist (dieser Zustand ist eine sogenannte Invariante):

- Der gerade betrachtete Knoten q ist rot.
- Eigenschaften 1, 3 und 5 sind erfüllt.
- Es kann keine oder eine der folgenden Eigenschaftsverletzungen vorhanden sein:
  - Knoten q ist rot und Knoten $q.parent$ existiert nicht (Verletzung von Eigenschaft 2).
  - Knoten q und $q.parent$ sind rot (Verletzung von Eigenschaft 4).

Der Ablauf des gesamten Reparaturprozesses wird grob in Algorithmus 1 dargestellt.

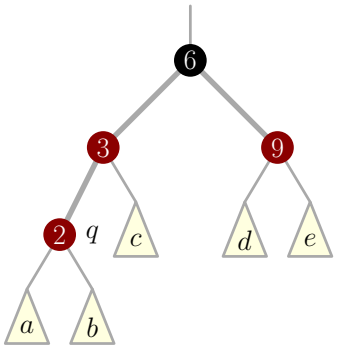
Algorithmus 1: Reparaturprozess nach dem Einfügen in einen Rot-Schwarz-Baum.

```
1 Input: Rot-Schwarz-Baum T und neu eingefügter Knoten q
2 while q.parent ≠ null und q.parent ist rot do
3 |   Behebe Verletzung von Eigenschaft 4
4 Behebe potentielle Verletzung von Eigenschaft 2
```

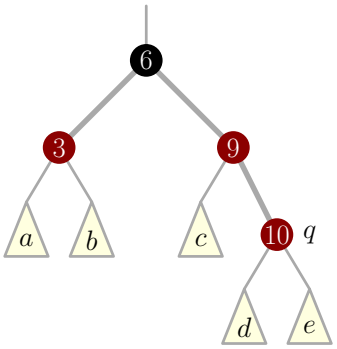
Da die Behebung einer potentiellen Verletzung von Eigenschaft 2 einfach ist (Wurzelknoten wird schwarz gefärbt), wird im Folgenden nur auf die Behebung der Verletzung von Eigenschaft 4 detailliert eingegangen.

Bei einer Verletzung der Eigenschaft 4 kann im Rahmen unseres Reparaturprozesses nur einer der sechs Fälle aus Abbildung 2 vorliegen. Beachten Sie dabei, dass die Fälle 1, 2 und 3 (Abbildungen 2a, 2c und 2e) gemein haben, dass $q.parent$ ein linker Kindknoten ist. Die zugehörigen gespiegelten Fälle (Abbildungen 2b, 2d und 2f) haben wiederum gemein, dass $q.parent$ ein rechter Kindknoten ist.

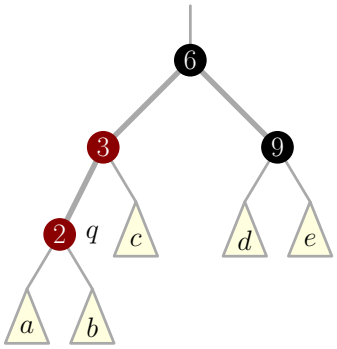
Umgang mit Fall 1: Da sowohl $q.parent$ als auch der Onkel von q ($q.parent.parent.right$) rot sind und der Großelternknoten von q ($q.parent.parent$) schwarz ist, können diese drei Knoten einfach die Farbe wechseln. Mit q bezeichnet man im Anschluss den Großelternknoten des bisherigen q (d.h. als Pseudocode: $q \leftarrow q.parent.parent$). Siehe Abbildung 3. Vergewissern Sie sich, dass die Invariante nach diesen Änderungen erfüllt ist.



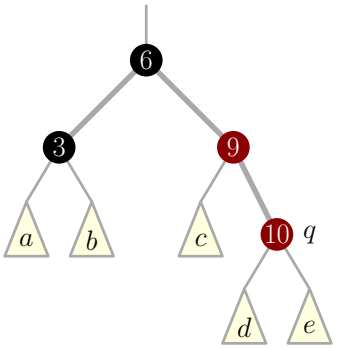
(a) Fall 1: Der Onkel von q ist rot. q kann auch ein rechtes Kind sein.



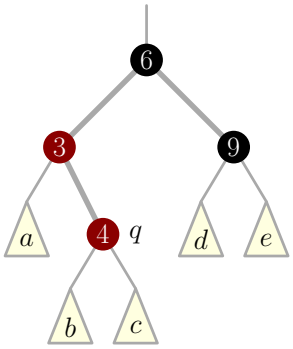
(b) Fall 1 (gespiegelt): Der Onkel von q ist rot. q kann auch ein linkes Kind sein.



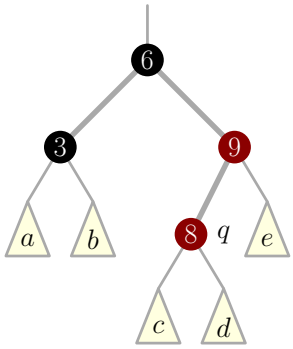
(c) Fall 2: Der Onkel von q ist schwarz und q ist ein linkes Kind.



(d) Fall 2 (gespiegelt): Der Onkel von q ist schwarz und q ist ein rechtes Kind.



(e) Fall 3: Der Onkel von q ist schwarz und q ist ein rechtes Kind.



(f) Fall 3 (gespiegelt): Der Onkel von q ist schwarz und q ist ein linkes Kind.

Abbildung 2: Mögliche Fehlerfälle im Reparaturprozess bei vorhandener Verletzung der Eigenschaft 4. Dabei repräsentieren a, b, c, d und e Teilbäume.

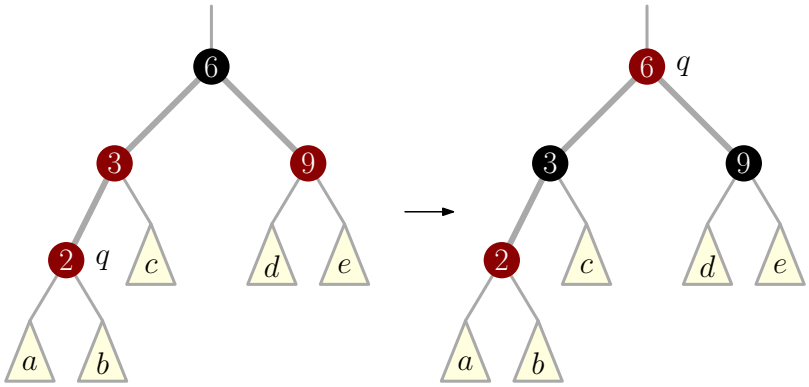


Abbildung 3: Umgang mit Fall 1

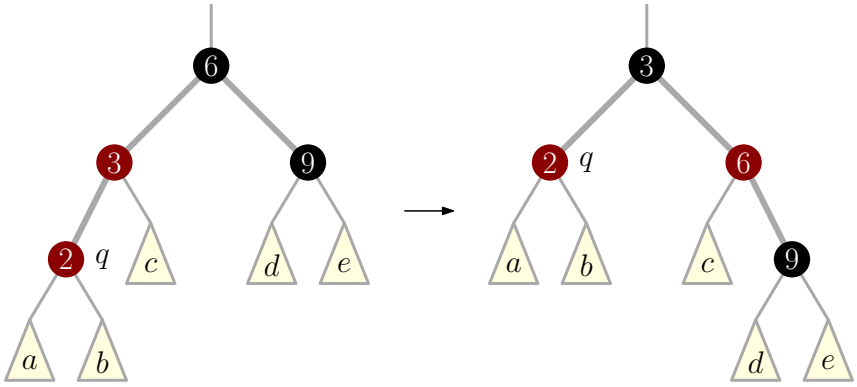


Abbildung 4: Umgang mit Fall 2

Umgang mit Fall 2: Zunächst führen wir eine einfache Rotation nach rechts am Großelternknoten von q ($q.parent.parent$) durch. Einfache Rotationen kennen Sie bereits aus der Theorie zu AVL-Bäumen aus den Vorlesungsfolien „Suchbäume“. Nun ist allerdings zusätzlich die Eigenschaft 5 verletzt. Daher färben wir **vor der Rotation** $q.parent$ schwarz und den Großelternknoten von q rot. Siehe Abbildung 4. Vergewissern Sie sich, dass die Invariante nach diesen Änderungen erfüllt ist.

Umgang mit Fall 3: Fall 3 ist kein eigenständiger Fall, sondern kann durch eine einfache Rotation nach links am Elternknoten von q ($q.parent$) in Fall 2 übergeführt werden. Mit q bezeichnet man im Anschluss den vorhergehenden Elternknoten des bisherigen q, welcher nach der Rotation der linke Kindknoten ist. Siehe Abbildung 5. Vergewissern Sie sich, dass die Invariante nach diesen Änderungen erfüllt ist und nun Fall 2 gegeben ist.

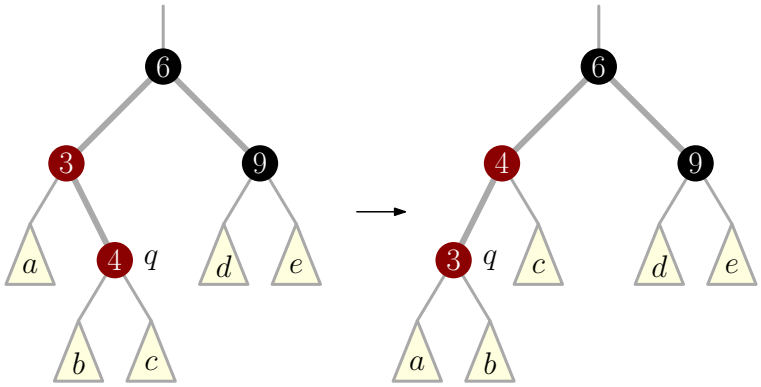


Abbildung 5: Umgang mit Fall 3

Umgang mit gespiegelten Fällen: Mit gespiegelten Fällen kann wie mit den zugehörigen, ungespiegelten Fällen umgegangen werden. Allerdings müssen links und rechts jeweils vertauscht werden.

5 Implementierung

Für die Umsetzung eines Algorithmus benötigen Sie eine Datenstruktur, in der Sie effizient nach Schlüsseln suchen können. Sie entscheiden sich für einen binären Suchbaum und vergleichen nun im Folgenden einfache binäre Suchbäume mit Rot-Schwarz-Bäumen.

5.1 Binäre Suchbäume

Implementieren Sie die Methode `void insertSimpleBinarySearchTree(Node root, Node newNode)`, in der anhand eines Integer-Schlüssels ein Knoten in einen einfachen binären Suchbaum eingefügt werden soll.

Der Parameter `Node root` stellt den Wurzelknoten des Suchbaumes dar, in dem der neue Knoten eingefügt werden soll. Ein `Node` stellt fünf Methoden zur Verfügung, die Sie zur Umsetzung benutzen dürfen:

- `Node getLeftChild()`: Gibt den linken Kindknoten zurück.
  Mittels `Node leftChild = root.getLeftChild()` kann so z.B. das linke Kind des Knotens `root` abgefragt werden. Wenn es kein linkes Kind gibt, wird `null` zurückgegeben.

- `Node getRightChild()`: Gibt den rechten Kindknoten zurück.
  Mittels `Node rightChild = root.getRightChild()` kann so z.B. das rechte Kind des Knotens `root` abgefragt werden. Wenn es kein rechtes Kind gibt, wird `null` zurückgegeben.

- `int getKey()`: Gibt den Integer-Schlüssel zurück.
  Mittels `int key = root.getKey()` kann so z.B. der Schlüssel des Knotens `root` abgefragt werden.

- `void attachNodeLeft(Node node)`: Fügt einen Knoten als linkes Kind ein.
  Mittels `root.attachNodeLeft(node)` wird der Knoten `node` zum linken Kind von `root`. Gibt es bereits einen linken Kindknoten, kommt es zu keinen Änderungen.

- `void attachNodeRight(Node node)`: Fügt einen Knoten als rechtes Kind ein.
  Mittels `root.attachNodeRight(node)` wird der Knoten `node` zum rechten Kind von `root`. Gibt es bereits einen rechten Kindknoten, kommt es zu keinen Änderungen.

Der zweite Parameter `Node newNode` ist jener Knoten, der in den Baum mit der Wurzel `root` eingesetzt werden soll. Stellen Sie sicher, dass für alle Knoten gilt, dass die Schlüssel der linken Kinder kleiner und die Schlüssel der rechten Kinder größer sind als der eigene Schlüssel.

In den Testinstanzen wird `void insertSimpleBinarySearchTree(Node root, Node newNode)` mehrmals in Folge aufgerufen, um einen Baum zu konstruieren. Stellen Sie dabei sicher, dass gleiche Schlüssel nicht mehrfach eingefügt werden. Achten Sie weiters darauf, nach dem Einfügen einen korrekten binären Suchbaum zu hinterlassen, da Folgeaufrufe sonst mit einem fehlerhaften Baum arbeiten.

5.2 Rot-Schwarz-Bäume

Implementieren Sie die Methode `void insertRedBlackTree(RBNode root, RBNode newNode)`, in der anhand eines Integer-Schlüssels ein Knoten in einen Rot-Schwarz-Baum eingefügt werden soll. Beachten Sie dabei, dass der Rot-Schwarz-Baum

entsprechend der Theorie in Abschnitt 4 alle Eigenschaften beibehalten muss.

Der Parameter `RBNode root` stellt einen Knoten eines Rot-Schwarz-Baumes dar, unter dem der neue Knoten eingefügt werden soll. Ein `RBNode` stellt folgende Methoden zur Verfügung, die Sie zur Umsetzung benutzen dürfen:

- `RBNode getLeftChild()`: Gibt den linken Kindknoten zurück.
  Mittels `RBNode leftChild = root.getLeftChild()` kann so z.B. das linke Kind des Knotens `root` abgefragt werden. Wenn es kein echtes linkes Kind gibt, wird `null` zurückgegeben. Hierbei repräsentiert `null` den speziellen Nullknoten entsprechend der Theorie aus Abschnitt 4.

- `RBNode getRightChild()`: Gibt den rechten Kindknoten zurück.
  Mittels `RBNode rightChild = root.getRightChild()` kann so z.B. das rechte Kind des Knotens `root` abgefragt werden. Wenn es kein echtes rechtes Kind gibt, wird `null` zurückgegeben. Hierbei repräsentiert `null` den speziellen Nullknoten entsprechend der Theorie aus Abschnitt 4.

- `RBNode getParent()`: Gibt den Elternknoten zurück.
  Mittels `RBNode parent = node.getParent()` kann so z.B. der Elternknoten des Knotens `node` abgefragt werden. Wenn es keinen Elternknoten gibt, wird `null` zurückgegeben.

- `RBNode getGrandparent()`: Gibt den Großelternknoten zurück.
  Mittels `RBNode grandparent = node.getGrandparent()` kann so z.B. der Großelternknoten des Knotens `node` abgefragt werden. Wenn es keinen Großelternknoten gibt, wird `null` zurückgegeben.

- `int getKey()`: Gibt den Integer-Schlüssel zurück.
  Mittels `int key = root.getKey()` kann so z.B. der Schlüssel des Knotens `root` abgefragt werden.

- `boolean isRed()`: Gibt zurück, ob ein Knoten rot ist.
  Mittels `boolean isRed = root.isRed()` kann so z.B. abgefragt werden ob der Knoten `root` rot ist.

- `boolean isBlack()`: Gibt zurück, ob ein Knoten schwarz ist.
  Mittels `boolean isBlack = isBlack()` kann so z.B. abgefragt werden ob der Knoten `root` schwarz ist.

- `void attachNodeLeft(RBNode node)`: Fügt einen Knoten als linkes Kind ein.
  Mittels `root.attachNodeLeft(node)` wird der Knoten `node` zum linken Kind von `root`. Gibt es bereits einen linken Kindknoten, kommt es zu keinen Änderungen.

- `void attachNodeRight(RBNode node)`: Fügt einen Knoten als rechtes Kind ein.
  Mittels `root.attachNodeRight(node)` wird der Knoten `node` zum rechten Kind von `root`. Gibt es bereits einen rechten Kindknoten, kommt es zu keinen Änderungen.

- `void setColorRed()`: Setzt die Farbe auf rot.
  Mittels `root.setColorRed()` kann so der Knoten `root` rot gefärbt werden.

- `void setColorBlack()`: Setzt die Farbe auf schwarz.
  Mittels `root.setColorBlack()` kann so der Knoten `root` schwarz gefärbt werden.

Der zweite Parameter `RBNode newNode` ist jener (rote) Knoten, der in den Baum mit der Wurzel `root` eingesetzt werden soll. Beachten Sie, dass sich die Wurzel des Baumes im Rahmen des Reparaturprozesses ändern kann. Stellen Sie sicher, dass für alle Knoten gilt, dass die Schlüssel der linken Kinder kleiner und die Schlüssel der rechten Kinder größer sind als der eigene Schlüssel.

Als weitere Hilfestellung müssen die Rotationen nicht selbstständig implementiert werden. Ein `RBNode` stellt weitere Methoden zur Verfügung, um Rotationen zu ermöglichen. Angenommen `node` ist die Wurzel eines Teilbaumes der rotiert werden soll, dann können Sie

- mittels `node.rotateToRight()` eine einfache Rotation nach rechts und
- mittels `node.rotateToLeft()` eine einfache Rotation nach links

durchführen. Diese Methoden haben weder Parameter noch Rückgabewerte. Alle relevanten Aspekte der Datenstruktur werden für Sie aktualisiert. Beispielsweise muss die neue Wurzel des rotierten Teilbaumes nicht mehr mit dem Elternknoten verknüpft werden.

In den Testinstanzen wird `void insertRedBlackTree(RBTree tree, RBNode newNode)` mehrmals in

Folge aufgerufen, um einen Baum zu konstruieren. Stellen Sie dabei sicher, dass gleiche Schlüssel nicht mehrfach eingefügt werden. Achten Sie weiters darauf, nach dem Einfügen einen korrekten binären Suchbaum zu hinterlassen, da Folgeaufrufe sonst mit einem fehlerhaften Baum arbeiten.

6 Testen

Führen Sie zunächst die `main`-Methode in der Datei `src/main/java/framework/Exercise.java` aus.

Anschließend wird Ihnen in der Konsole eine Auswahl an Testinstanzen angeboten, darunter befindet sich zumindest `abgabe.csv`:

```
Select an instance set or exit:
[1] abgabe.csv
[0] Exit
```

Durch die Eingabe der entsprechenden Ziffer kann entweder eine Testinstanz ausgewählt werden oder das Programm (mittels der Eingabe von `0`) verlassen werden. Wird eine Testinstanz gewählt, dann wird der von Ihnen implementierte Programmcode ausgeführt. Kommt es dabei zu einem Fehler, wird ein Hinweis in der Konsole ausgegeben.

Relevant für die Abgabe ist das Ausführen der Testinstanz `abgabe.csv`.

Die Testinstanzen `simple-binary-search-tree-small.csv`, `simple-binary-search-tree.csv`, `rb-tree-small.csv` und `rb-tree.csv` können Sie nutzen, falls Sie bisher nur eine der beiden Unteraufgaben implementiert haben und diese testen möchten. Unter Umständen kann dies ein paar Minuten in Anspruch nehmen. Mithilfe von `rb-tree-debug.csv` können Sie überprüfen, ob Ihr Rot-Schwarz-Baum die erwarteten Rotationen durchführt. Dabei handelt es sich um kleine Bäume, die beim Debuggen der Fälle 1, 2, 3, 1 (gespiegelt), 2 (gespiegelt) und 3 (gespiegelt) aus Abbildung 2 helfen. Es gibt vier Bäume zu Fall 1 (gespiegelt, ungespiegelt sowie jeweils q linkes Kind, q rechtes Kind), zwei Bäume zu Fall 2 (gespiegelt, ungespiegelt) und zwei Bäume zu Fall 3 (gespiegelt, ungespiegelt). Jeder Fall tritt dabei nur einmal pro Baum auf; bei den Bäumen zu den Fällen 2 und 3 tritt jedoch zunächst einmal der Fall 1 auf. Die Testinstanz `interactive.csv` wird für Frage 7 in Abschnitt 8 benötigt.

7 Evaluierung

Wenn der von Ihnen implementierte Programmcode mit der Testinstanz `abgabe.csv` ohne Fehler ausgeführt werden kann, dann wird nach dem Beenden des Programms im Ordner `results` eine Ergebnis-Datei mit dem Namen `solution-abgabe.csv` erzeugt.

Die Datei `solution-abgabe.csv` beinhaltet erstellte Bäume und Zeitmessungen der Ausführung der Testinstanz `abgabe.csv`, welche in einem Web-Browser visualisiert werden können. (Auch Ergebnis-Dateien anderer Testinstanzen können zu Testzwecken visualisiert werden.) Öffnen Sie dazu die Datei `visualization.html` in Ihrem Web-Browser und klicken Sie rechts oben auf den Knopf *Ergebnis-Datei auswählen*, um `solution-abgabe.csv` auszuwählen.

Beantworten Sie basierend auf der Visualisierung die Fragestellungen aus dem folgenden Abschnitt.

8 Fragestellungen

Öffnen Sie `solution-abgabe.csv` und bearbeiten Sie folgende Aufgaben- und Fragestellungen:

1. Wieso können Eigenschaften 1, 3 und 5 nicht verletzt sein, nachdem ein Knoten naiv (also vor der Durchführung des Reparaturprozesses) in einen Rot-Schwarz-Baum entsprechend der Theorie in Abschnitt 4 eingefügt wurde?

   Beantworten Sie weiters, wie viele Rotationen im Rahmen eines Reparaturprozesses maximal durchgeführt werden können. Analysieren Sie hierzu, welche Fälle aufeinander folgen können.

2. Erklären Sie die Funktionsweise Ihrer Implementierung. Gehen Sie speziell auf Ihre Umsetzung des Rot-Schwarz-Baumes ein.

   Optionale, ungewertete Bonusaufgabe: Erklären Sie weiters, inwiefern sich die Balancierung von denen in der Vorlesung vorgestellten AVL-Bäumen unterscheidet.

3. Nehmen Sie an, dass `void insertSimpleBinarySearchTree(Node root, Node newNode)` als rekursive Methode umgesetzt wurde. Warum kann das speziell bei

jenen Testinstanzen, die Schlüssel in sortierter Reihenfolge einfügen, zu einem `StackOverflowError` beim einfachen binären Suchbaum führen? Wieso ist eine potentiell rekursive Implementierung des Rot-Schwarz-Baumes später betroffen?

4. Durch Klicken auf Gruppennamen in der Legende neben der Plots, lassen sich einzelne Gruppen aus- bzw. einblenden. Unter Umständen müssen Sie nach dem Klicken kurz warten. Blenden Sie alles bis auf *Simple Binary Search Tree - shuffled* und *Red Black Tree - shuffled* aus. Die erste Gruppe zeigt die durchschnittlichen Laufzeiten von Suchen in Ihrem einfachen binären Suchbaum in Abhängigkeit von der Schlüsselanzahl. Die zweite Gruppe zeigt die durchschnittlichen Laufzeiten von Suchen in einem entsprechenden Rot-Schwarz-Baum. Die Schlüssel wurden zuvor in zufälliger Reihenfolge eingefügt. Vergleichen Sie die Laufzeiten: Beschreiben Sie die Messungen und argumentieren Sie deren Zustandekommen.

   Drücken Sie im Anschluss in der Menüleiste rechts über dem Plot auf den Fotoapparat, um den Plot als Bild zu speichern.

5. Blenden Sie nun alles bis auf *Simple Binary Search Tree - ordered* und *Red Black Tree - ordered* aus. Die Plots unterscheiden sich nur dahingehend zu den vorhin betrachteten, dass die Schlüssel in sortierter Reihenfolge eingefügt wurden. Vergleichen Sie die Laufzeiten: Beschreiben Sie die Messungen und argumentieren Sie deren Zustandekommen.

   Erstellen Sie im Anschluss wieder mithilfe der Menüleiste ein Bild des Plots.

6. Sehen Sie sich die konstruierten Bäume der Instanzen *Simple Binary Search Tree - shuffled: 4* und *Red Black Tree - shuffled: 4* an, indem Sie im Dropdown links unter den Plots die entsprechenden Einträge auswählen und nach Notwendigkeit etwas hinunter scrollen. Den Wurzelknoten können Sie jeweils an einer schwarzen Umrahmung erkennen. Linke Kindknoten sind durch eine durchgehende graue Kante verbunden. Rechte Kindknoten sind durch eine strichlierte rote Kante verbunden. Durch Klicken und Ziehen mit dem Mauszeiger können Sie Knoten verschieben, um sich die Bäume zurechtzurücken. Können Sie aus den beiden Visualisierungen die gemeinsame Einfügereihenfolge rekonstruieren? Begründen Sie Ihre Antwort.

   Erstellen Sie im Anschluss Screenshots der beiden Bäume.

7. Führen Sie die Testinstanz `interactive.csv` aus und geben Sie 10 unterschiedliche Schlüssel in solch einer Reihenfolge ein, dass der resultierende einfache binäre Suchbaum dem Rot-Schwarz-Baum entspricht. Öffnen Sie `solution-interactive.csv` und sehen Sie sich die Visualisierungen der Bäume an. Geben Sie die gewählte Einfügereihenfolge an.

   Erstellen Sie im Anschluss einen Screenshot von einem der beiden Bäume.

Falls sich im Zuge der Evaluierung die Darstellung der Plots auf ungewünschte Weise verändert (z.B. durch die Auswahl eines zu kleinen Ausschnitts), können Sie mittels Doppelklick auf den Plot oder Klick auf das Haus in der Menüleiste die Darstellung zurücksetzen.

Fügen Sie Ihre Antworten in einem Bericht gemeinsam mit den zwei erstellten Bildern der Plots sowie den drei Screenshots der Bäume zusammen.

9 Abgabe

Laden Sie die Datei `src/main/java/exercise/StudentSolutionImplementation.java` in der TUWEL-Aktivität *Hochladen Source-Code P4* hoch. Fassen Sie diesen Bericht mit den anderen für das zugehörige Abgabegespräch relevanten Berichten in einem PDF zusammen und geben Sie dieses in der TUWEL-Aktivität *Hochladen Bericht Abgabegespräch 2* ab.

10 Nachwort

Balancierte binäre Suchbäume sind ganz fundamentale Datenstrukturen in der Informatik, die sich immer dann anwenden lassen, wenn geordnete Objekte gespeichert und verarbeitet werden sollen. Anders als in einer einfachen Liste oder einem Array lassen sich mit Suchbäumen nämlich die Suchzeiten nach Elementen signifikant beschleunigen, denn in jedem Suchschritt wird die noch relevante Menge von Elementen in etwa halbiert. Damit erreicht eine Suche in einem balancierten binären Suchbaum mit n Elementen nach nur $O(\log_2 n)$ Schritten, und das weiterhin mit linearem Platzbedarf $O(n)$. Implizit verwenden Sie ein solches Suchverfahren zum Beispiel auch, wenn Sie nach einem bestimmten Buch in den Regalen der

Bibliothek suchen. Auch Computer verbringen einen großen Teil ihrer Aufgaben mit der Suche nach Elementen, von Reisebuchungen mit Reservierungsnummern hin zu Kundendatenbanken mit User-IDs. Für die Effizienz der Suche in solchen großen Datenmengen sind (dynamische) Binärbäume und verwandte Datenstrukturen ausschlaggebend.

Eine wichtige Anwendung von binären Suchbäumen findet sich in der algorithmischen Geometrie. Dort geht es darum, räumliche Daten zu verarbeiten und zu speichern, beispielsweise wichtige Punkte auf einer Landkarte, dreidimensionale Messpunkte in den Geowissenschaften oder hochdimensionale Daten mit den unterschiedlichsten Attributen. Solange jede Dimension eine Ordnung besitzt, sind Binärbäume dafür hervorragend geeignet. Stellen Sie sich vor, Sie wollen bestimmte Elemente filtern, die in einem gegebenen mehrdimensionalen Suchbereich liegen – hierzu lassen sich die AVL-Bäume aus der Vorlesung geschickt erweitern und über mehrere Dimensionen verknüpfen, sodass trotzdem noch sehr effizient nach den Elementen gesucht werden kann. Oder auch beim Entwurf von so genannten Sweep-Line Algorithmen, die beispielsweise alle Schnittpunkte von n Strecken berechnen sollen, helfen Binärbäume, um zu effizienteren Algorithmen zu kommen. Diese und weitere Beispiele sehen Sie z.B. in der LVA *Algorithmics* oder der LVA *Algorithmic Geometry* im Masterstudium.