TU Wien
Institut für Logic and Computation
Forschungsbereich Algorithms and Complexity

186.866 Algorithmen und Datenstrukturen VU

Programmieraufgabe P3

PDF erstellt am: 19. März 2024

1 Vorbereitung

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

1. Laden Sie das Framework `P3.zip` aus TUWEL herunter.
2. Entpacken Sie `P3.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 für einige der Suchbäume aus der Vorlesung zu vertiefen und zu fördern. Konkret sind sowohl einfache binäre Suchbäume als auch AVL-Bäume zu implementieren, damit diese in Folge verglichen werden können.

4 Theorie

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

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 AVL-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.

5.2 AVL-Bäume

Implementieren Sie die Methode `void insertAVLTree(AVLNode p, AVLNode newNode)`, in der anhand eines Integer-Schlüssels ein Knoten in einen AVL-Baum eingefügt werden soll. Beachten Sie dabei, dass alle Knoten des Baumes entsprechend der Lehrveranstaltung balanciert bleiben müssen.

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

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

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

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

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

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

- `void setHeight(int height)`: Hinterlegt die Höhe eines Teilbaumes.
  Mittels `p.setHeight(10)` kann so z.B. die Höhe 10 des Teilbaumes mit der Wurzel `p` im Knoten `p` hinterlegt werden. Der hinterlege Wert kann dann mit einer Hilfsmethode (siehe unten `static int height(AVLNode node)`) ausgelesen werden.

Der zweite Parameter `AVLNode newNode` ist jener Knoten, der in den Baum mit der Wurzel `p` 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.

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

- mittels `p.rotateToRight()` eine einfache Rotation nach rechts,
- mittels `p.doubleRotateLeftRight()` eine Doppelrotation links-rechts,
- mittels `p.rotateToLeft()` eine einfache Rotation nach links und

- mittels `p.doubleRotateRightLeft()` eine Doppelrotation rechts-links

durchführen. Diese Methoden haben weder Parameter noch Rückgabewerte. Alle relevanten Aspekte der Datenstruktur werden für Sie aktualisiert. Beispielsweise muss die neue Wurzel des rebalancierten Teilbaumes nicht mehr mit dem Elternknoten verknüpft werden. Auch die Höhen der (Teil-)bäume werden für Sie entsprechend der Vorlesungsfolien aktualisiert.

Weiters können Sie zur Ermittlung der Höhe eines (Teil-)baumes die Methode `static int height(AVLNode node)` nutzen. Mittels `int height = AVLNode.height(p)` erhalten Sie die Höhe des Teilbaumes mit der Wurzel p, die zuvor mithilfe von `void setHeight(int height)` hinterlegt wurde. Wird ein leerer (Teil-)baum übergeben (`null`), ist der Rückgabewert -1.

In den Testinstanzen wird `void insertAVLTree(AVLNode p, AVLNode newNode)` mehrmals in Folge aufgerufen, um einen Baum zu konstruieren. Stellen Sie dabei sicher, dass gleiche Schlüssel nicht mehrfach eingefügt werden.

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`, `avl-tree-small.csv` und

`avl-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 `avl-tree-debug.csv` können Sie überprüfen, ob Ihr AVL-Baum die erwarteten Rotationen durchführt. Dabei handelt es sich um jene Bäume, die als Fälle 1.1, 1.2, 2.1 und 2.2 in den Vorlesungsfolien (in genau derselben Reihenfolge) präsentiert werden. Darüber hinaus könnte es bei hartnäckigen Fehlern nützlich sein, eine eigene Methode anzufertigen, die zur besseren Nachvollziehbarkeit den root-Knoten als weiteres Argument weitergibt. Auch das Setzen von Conditional Breakpoints könnte möglicherweise hilfreich sein (in IntelliJ z.B. sind solche konfigurierbar durch Rechtsklick auf einen Breakpoint).

Die Testinstanz `interactive.csv` wird für Frage 6 im 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. Erklären Sie die Funktionsweise Ihrer Implementierung. Gehen Sie speziell auf Ihre Umsetzung des AVL-Baumes und deren Unterschiede zu dem in der Vorlesung vorgestellten Pseudocode ein.

2. 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 können potentiell mehr Schlüssel in den AVL-Baum eingefügt werden, bevor es zu Problemen kommt?

3. 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 *AVL 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 entsprechendem AVL-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.

4. Blenden Sie nun alles bis auf *Simple Binary Search Tree - ordered* und *AVL 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.

5. Sehen Sie sich die konstruierten Bäume der Instanzen *Simple Binary Search Tree - shuffled: 4* und *AVL 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. Sind in beiden Bäumen alle Knoten balanciert? Können Sie aus den beiden Visualisierungen die Einfügereihenfolge rekonstruieren? Begründen Sie Ihre Antwort.

Erstellen Sie im Anschluss Screenshots der beiden Bäume.

6. 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 AVL-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 P3* 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 1* ab.

10 Nachwort

Balancierte binäre Suchbäume sind eine ganz fundamentale Datenstruktur 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 erreichen Sie das Ziel der Suche bei einer Datenstruktur mit n Elementen nach nur $\log_2 n$ 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, so dass Sie trotzdem noch sehr effizient nach den Elementen suchen können. 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.