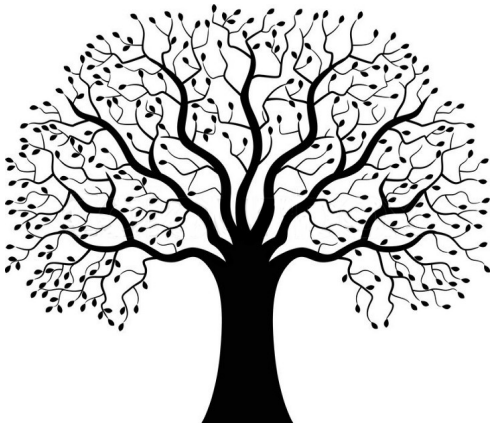
Hochschule Darmstadt
Prof. Dr. Benjamin Meyer

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

29. Juni 2021

Programmieren, Algorithmen und Datenstrukturen II, SoSe 2021

Aufgabenblatt 5



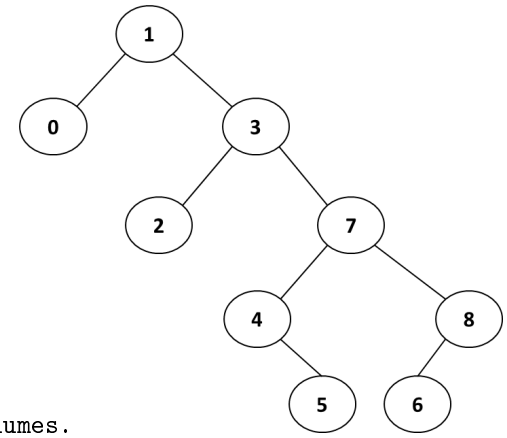
Im letzten Aufgabenblatt werden Sie sich mit der Umsetzung eines binären Suchbaums beschäftigen. Sie werden merken, die Implementierung eines Baumes ist der einer Liste sehr ähnlich, viele Aspekte werden Sie, leicht angepasst, von den letzten Aufgabenblättern übernehmen können.

5.1 Das Grundgerüst des Suchbaums

Als einzelne Elemente des Baumes benötigen Sie sog. *Knoten*, die den zu speichernden Inhalt darstellen. Diese Knoten sollen nach einem numerischen *Schlüssel* in dem Suchbaum eingeordnet werden, aber zudem auch einen beliebigen *Inhalt* speichern können. Erstellen Sie daher eine Klasse `Node`, die, neben dem `key` und einem Template-Datenfeld `data`, Zeiger auf das linke sowie rechte Kindknoten speichert. Diese rekursive Datenstruktur ähnelt dabei stark der einer Liste, nur dass auf zwei weitere Elemente verwiesen wird.
Die Klasse `BinarySearchTree` stellt den binären Suchbaum dar. Anfangs enthält sie einzig einen Zeiger auf den Wurzelknoten. Implementieren Sie zudem in beiden Klassen get- und set-Methoden, Konstruktoren und Destruktoren, wie Sie es für sinnvoll erachten.
Im letzten Aufgabenblatt hatten Sie eine grafische Ausgabe eines Baumes auf der Konsole entwickelt. Die Ausgabe war zwar visuell gut interpretierbar, aber leider auf wenige Ebenen des Baumes begrenzt gewesen. Wir wollen hier zu Testzwecken eine andere Darstellungsart wählen, vgl. Listing 1.
Diese Ausgabe lässt sich erzeugen, indem Sie in der `Node`-Klasse eine `print`-Methode schreiben, die bereits einen String als Einrückung (engl. indent) übergeben bekommt. Geben Sie erst den indent und dann den Schlüssel der `Node` auf der Konsole aus (beispielsweise mit einem voranstehenden >), und rufen Sie rekursiv die `print`-Methode für die Kinderknoten auf, nachdem Sie den indent um ein Trennzeichen (wie |) erweitert haben - falls es überhaupt Kinderknoten gibt.
Der `BinarySearchTree` kann nun eine eigene `print`-Methode bekommen, welche `print()` des `root`-Knoten startet.

Erzeugen Sie in der main die entsprechenden Nodes und verbinden Sie diese händisch, um die Struktur aus Listing 1 herzustellen. Übergeben Sie als data-Eingräge Strings Ihrer Wahl. Erzeugen Sie einen Baum mit dieser Node-Struktur und testen Sie Ihre Ausgabe.

```
> 1
| > 0
| > 3
| | > 2
| | > 7
| | | > 4
| | | |
| | | | > 5
| | | > 8
| | | | > 6
| | | |
```



Listing 1: Konsolendarstellung des rechten Baumes.

Neben dieser strukturierten Ausgabe wollen wir die Schlüssel der `Nodes` auch einfach direkt als Aufzählung ausgeben können. Implementieren Sie hierfür verschiedene Ausgabemöglichkeiten, entsprechend der Pre-Order-, In-Order- und Post-Order-Ausgabe.
Letztlich benötigt der `BinarySearchTree` auch die Möglichkeit, nach einzelnen Knoten anhand des Schlüssels zu suchen. Wurde der richtige Schlüssel gefunden, geben Sie einen Zeiger auf den Node als Ergebnis zurück (sodass der Nutzer Zugriff auf das data-Feld erhält). Geben Sie zudem die Daten des Knotens auf der Konsole aus. Konnte der Knoten nicht gefunden werden, werfen Sie eine Fehlermeldung in der Such-Funktion und fangen diese beim Aufruf ab.

5.2 Einfügen und Löschen

Erweitern Sie den `BinarySearchTree` um Funktionen zum Hinzufügen und Löschen eines Nodes. Stellen Sie sicher, dass die Eigenschaften eines binären Suchbaums nicht verletzt werden!
Das Löschen erwartet als Parameter einen zu löschenden Schlüssel. Ist der Schlüssel nicht im Baum enthalten, wird wiederum ein Fehler geworfen. Achten Sie zudem darauf, ggf. auch den root-Eintrag neu zu setzen, sollte dieser bei einer Lösch-Operation verändert werden.
Übernehmen Sie aus den vorherigen Aufgabenblättern eine minimalistische GUI, um die Such-, Einfüge- und Löschoperationen besser testen zu können.

5.3 Rotieren

Aus der Vorlesung bekannt sind zudem die Methoden des rechtsseitigen und linksseitigen Rotierens. Setzen Sie diese für einen einzelnen Node um. Achten Sie darauf, die neuen Zeiger für den linken und rechten Teilbaum korrekt zu setzen. Sollte eine Rotation nicht möglich sein, erzeugen Sie wiederrum eine Fehlermeldung.
Bei einer Rotation muss zusätzlich auch der Verweis des Vaterknotens auf den zu rotierenden Knoten angepasst werden. Dies geschieht meist außerhalb, also an der Stelle, wo die Rotation aufgerufen wurde. Geben Sie daher in den Rotationsmethoden der Node einen Zeiger auf den neuen Wurzelknoten des Teilbaums zurück.

5.4 Balancieren

Die Rotationsmethoden können genutzt werden, um degenerierte Bäume wieder zu balancieren. Schreiben Sie dafür in der `Node` eine Methode, die für diesen Knoten (rekursiv) die Höhe des (Teil-) Baums berechnet. Starten Sie bei 1, falls der Knoten keine Kinder mehr haben sollte.
Die Methode `balance()` des `BinSearchTrees` soll nun den Baum ausbalancieren. Die Methode startet bei der Wurzel des `BinSearchTrees` und berechnet sich die Höhen der beiden Teilbäume der Kinder (falls diese existieren). Liegen die beiden mehr als 1 auseinander, rotiert sie den Baum an der Wurzel entsprechend, um den Unterschied der Teilbaumhöhen zu verringern. Dieser Schritt wird wiederholt, bis die Teilbäume ungefähr gleich hoch sind (also der Unterschied max. 1 beträgt). Danach kann mit beiden Teilbäumen rekursiv auf gleiche Weise fortgefahren werden.
Ein Problem hierbei ist, dass der Vaterknoten über die Rotation seiner Kinder informiert werden muss. Ohne zusätzlichen Aufwand ist der Vaterknoten zum Zeitpunkt der Rotation jedoch nicht (mehr) bekannt. Dies kann umgangen werden, indem die Methode `balance()` drei Parameter bekommt:

1. Einen Zeiger auf den aktuellen Knoten, um den ggf. rotiert werden soll
2. Einen Zeiger auf dessen Vaterknoten
3. Einen boolean, der aussagt, ob der aktuelle Knoten linkes oder rechtes Kind vom Vater gewesen ist

So kann nach jeder Rotation der Verweis vom Vaterknoten auf das linke bzw. auf das rechte Element angepasst werden. Ist der Vaterknoten `null`, handelt es sich bei dem aktuellen Element um die Wurzel des kompletten Baumes. In diesem Fall muss statt des Zeigers des Vaters der Zeiger des Baumes auf den root-Knoten angepasst werden (der boolean kann hier ignoriert werden).

Eine Alternative zu diesem Vorgehen wäre es gewesen, zusätzlich in der Node einen Zeiger auf den Vater zu speichern. Diese Referenz hätte jedoch bei jeder Rotations-, Einfüge- und Löschoperation aktualisiert werden müssen.
Testen Sie Ihre balance-Methode an verschiedenen Bäumen, um sicherzugehen, dass sie korrekt funktioniert (optional können Sie auch wieder UnitTests schreiben, um die korrekte Funktionalität sicherzustellen).
Diese Balancing-Methode ist nicht sonderlich effizient. Ähnliche (und etwas ausgefeiltere) Ansätze werden jedoch vom AVL-Tree und vom RB-Tree direkt bei der Einfüge- und Löschoperation verfolgt, um eine Degenerierung des Baumes zu verhindern.

5.5 Optional: Datenrettung

Diese Datenstruktur kann beispielsweise genutzt werden, um Kundendaten einer Firma abzuspeichern. Anhand einer eindeutigen Kundennummer (als Schlüssel) können größere Datensätze schnell durchsucht werden.
Als Wiederholung: Überlegen Sie sich, wie Sie sämtliche Daten des Baumes in eine Datei speichern und wieder laden können unter Einhaltung der folgenden Struktur:

```
#Kundennummer: 123456
#Name: Max Mustermann
***************
```

Listing 2: Beispielcodierung eines Kunden der Datenbank.