TU Kaiserslautern – FB Informatik
Dr. Patrick Michel

21.01.2020

Algorithmen und Datenstrukturen (WS 2019)

Aufgabenblatt 10

zu bearbeiten bis: 27.01.20 - 29.01.20

Aufgabe 10.1 (Binäre Suchbäume - Theorie)

a) Fügen Sie in einen (anfangs leeren) Suchbaum die Werte $6, 8, 1, 3, 10, 12, 8, 5, 2, 4, 11$ ein. Skizzieren Sie den Baum. Löschen Sie die 10 und die 6. Skizzieren Sie den Baum erneut.

b) Richtig oder falsch? "Fügt man dieselben Werte in unterschiedlicher Reihe in einen Suchbaum ein, erhält man immer denselben Baum" Begründen Sie.

c) Gegeben einen Suchbaum mit n Knoten, geben Sie den Aufwand für das Suchen und Löschen eines Wertes im Best Case und Worst Case an. Begründen Sie jeweils.

Aufgabe 10.2 (`TreeMap` (Binäre Suchbäume) - Praxis)
Der `TreeMap` Datentyp aus der Vorlesung ist eine binäre Suchbaum Implementierung, mit getrennten Schlüsseln (Typ `K`) und Werten (Typ `V`). Er implementiert damit sehr natürlich den abstrakten Datentyp `Map`.

Bei der Implementierung der `put` Methode müssen schon einige Fälle unterschieden werden, nämlich (a) Ersetzen des Wertes eines bestehenden Knotens, (b) Einfügen an der Wurzel, (c) Einfügen als linkes Kind eines Knotens und (d) Einfügen als rechtes Kinds eines Knotens. Bei der Methode `remove` sind es noch mehr Fälle:

- Es gibt den Knoten nicht.
- Der Knoten hat nur **ein Kind links** und ist selbst die **Wurzel**.
- Der Knoten hat nur **ein Kind rechts** und ist selbst die **Wurzel**.
- Der Knoten hat nur **ein Kind links** und ist das **linke Kind** eines anderen Knotens.
- Der Knoten hat nur **ein Kind links** und ist das **rechte Kind** eines anderen Knotens.
- Der Knoten hat nur **ein Kind rechts** und ist das **linke Kind** eines anderen Knotens.
- Der Knoten hat nur **ein Kind rechts** und ist das **rechte Kind** eines anderen Knotens.
- Der Knoten hat **zwei Kinder** und das größte Kind im linken Baum ist **sein linkes Kind**.
- Der Knoten hat **zwei Kinder** und das größte Kind im linken Baum ist **nicht sein Kind**.

Wie auch schon bei der Implementierung der doppelt verketteten Liste, können wir versuchen Hilfsmethoden zu implementieren, welche die immer wieder benötigten Logiken nur einmal implementieren. Wir benutzen dazu hier die Hilfsklasse `NodePointer`, welche einen **Knoten** (`node`) im Baum repräsentiert, zusammen mit dessen **Vater** Knoten (`parent`) und der Information, ob er das **linke Kind** (`left`) des Vaters ist.

Hat man alle diese Informationen (Knoten, Vater, Angabe links/rechts) zusammen, kann man das Ersetzen des Knotens (siehe Methode `replaceWith`), bzw. das Einfügen (oder Update) eines Knoten (siehe Methode `updateOrAdd`) genau **einmal implementieren** und überall verwenden. Schauen Sie sich die Implementierung von `put`, `get` und `containsKey` and, die alle schon basierend auf den Hilfskonstrukten definiert sind.

- Bringen Sie Ihre Interface Definitionen im Package `exercise.adt` auf den neusten Stand und öffnen Sie die Vorlage `TreeMap.java` (für Package `exercise.adt.impl`).
- Implementieren Sie die Hilfemethode `search`, welche gegeben einem `key` den Knoten mit diesem Schlüssel im Baum sucht. Ergebnis der Methode ist ein `NodePointer`:
  - `node` ist die Referenz auf den Knoten oder `null`, falls es den Knoten nicht gibt.
  - `parent` ist die Referenz auf den Vater des Knoten oder `null`, falls der Knoten die Wurzel ist. Für Knoten, die nicht im Baum sind, soll `parent` der letze Knoten sein, der bei der Suche durchlaufen wurde, sprich ein Knoten mit freiem Kind, wo der gesuchte Knoten (korrekt sortiert) eingefügt werden kann.
  - `left` ist wahr, wenn `parent` nicht `null` ist und der Knoten das linke Kind von `parent` ist (oder wäre).
- Implementieren Sie die Hilfsmethode `largest`, welche gegeben einem Startknoten (und dessen Vater) zum Knoten mit dem größten Schlüssel navigiert. Dieser Knoten soll dann, zusammen mit seinem Vater und der Information `left` zurück gegeben werden.
- Vervollständigen Sie nun mit Hilfe von `search` und `largest` die `remove` Methode!

Aufgabe 10.3 (`TreeMap` Split - Praxis)
Wir wollen auf der `TreeMap` Klasse eine weitere Methode anbieten, welche aus der `Map` alle Elemente entfernt, die einen echt kleineren Schlüssel haben als ein gegebener. Die entfernten Elemente sollen dabei wiederum als neue `Map` zurück gegeben werden.

Um dies zu implementieren, kann man sich zunutze machen, dass man beim Suchen des Schlüssels den Binärbaum einmal so durchläuft, dass bei jedem Knoten der Knoten und einer seiner Teilbäume definitiv kleiner oder größer gleich dem gesuchten Schlüssel sind. Damit teilt der Durchlauf von Wurzel bis Blatt den Baum genau so wie gewünscht! Implementieren Sie die `removeSmaller` Methode also mit folgender Strategie:

- Sie verwalten **zwei Bäume**, einen mit allen *kleineren* Elementen, einen mit allen *größeren* (und evtl. dem gleich großen Element).
- Für beide Bäume verwalten Sie die **Wurzel Referenz**, sowie die **Referenz** auf den Knoten mit dem **größten bzw. kleinsten** Schlüssel. Im Baum mit den *kleineren* Elementen hat der Knoten mit dem *größten* Schlüssel kein *rechtes* Kind (und umgekehrt im anderen).
- Beim Durchlauf durch den Baum fügt man den Knoten und seinen linken (oder rechten) Teilbaum in den Baum mit den kleineren (oder größeren) Elementen ein, **löst den anderen Teilbaum ab** und fährt dann mit dem Durchlauf fort.

Am Ende gibt man eine neue `TreeMap` mit der Wurzel der kleineren Elemente zurück und setzt die `root` Referenz im aktuelle Baum auf die Wurzel des Baums mit den größeren Elementen.
