Humboldt-Universität zu Berlin
Institut für Informatik
Prof. Dr. Marius Kloft
Bux, Grußien, Schäfer, Tschorsch, Völlinger

Berlin, den 27.06.2016

Übungen zur Vorlesung
Algorithmen und Datenstrukturen

Übungsblatt 6

Abgabe: Montag den 11.07.2016 bis 10:55 Uhr vor der Vorlesung im Hörsaal oder bis 10:45 Uhr im Briefkasten (RUD 25, Raum 3.321). Die Übungsblätter sind in Gruppen von zwei (in Ausnahmefällen auch drei) Personen zu bearbeiten. **Die Lösungen sind auf nach Aufgaben getrennten Blättern abzugeben. Heften Sie bitte die zu einer Aufgabe gehörenden Blätter vor der Abgabe zusammen.** Vermerken Sie auf allen Abgaben Ihre Namen, Ihre Matrikelnummern, den Namen Ihrer Goya-Gruppe und an welchem Übungstermin (Wochentag, Uhrzeit, Dozent) Sie die korrigierten Abgaben abholen möchten. Beachten Sie auch die aktuellen Hinweise auf der Übungswebsite unter https://hu.berlin/algodat16.

Aufgabe 1 (Hashing-Schreibtischtest) — 2 + 3 + 3 + 2 = 10 Punkte

Beim Hashing bestimmt die Sondierungsreihenfolge für jeden Schlüssel, in welcher Reihenfolge die Einträge der Hashtabelle auf einen freien Platz durchsucht werden. Da beim Hashing überwiegend mit modulo-Werten gerechnet wird, beginnt die Indizierung aller Arrays A beim Hashing mit 0 und endet bei $|A| - 1$.

Gegeben sei eine Hashtabelle mit 11 Feldern für Einträge (Index 0 bis 10) und eine Hashfunktion $h(k) = k \bmod 11$. Führen Sie für die folgenden Hashverfahren einen Schreibtischtest durch, indem Sie jeweils die Werte 54, 15, 27, 76, 22, 5 und 14 in dieser Reihenfolge in eine leere Tabelle einfügen und diese nach jeder Einfügeoperation ausgeben.

a) **Hashing mit direkter Listenverkettung**
Hierbei ist die Hashtabelle als Array von einfach verketteten Listen realisiert.

b) **Offenes Hashing mit linearem Sondieren**
Bei Kollisionen wird hier das einzufügende Element an der nächsten freien Stelle links vom berechneten Hashwert eingefügt. Das heißt, die Sondierungsreihenfolge (die Reihenfolge, in der die Plätze im Array durchgegangen werden, bis erstmals ein freier Platz angetroffen wird) ist gegeben durch $s(k, i) = (h(k) - i) \bmod 11$ für $i = 0, \ldots, 10$.

c) **Doppeltes Hashing**
Das doppelte Hashing ist ein offenes Hashing, bei dem die Sondierungsreihenfolge von einer zweiten Hashfunktion $h'(k) = 1 + (k \bmod 7)$ abhängt. Die Position für das i-te Sondieren ist bestimmt durch die Funktion $s(k, i) = (h(k) - i \cdot h'(k)) \bmod 11$ für $i = 0, \ldots, 10$.

d) **Uniformes offenes Hashing**
Hier erhält jeder Schlüssel mit gleicher Wahrscheinlichkeit eine der 11! Permutationen von $\{0, 1, \ldots, 10\}$ als Sondierungsreihenfolge. Sei L eine Funktion, die jedem Schlüssel $k \in U$ uniform zufällig eine der 11! Permutationen zuweist. Weiterhin sei $L(k)[i]$ das i-te Element der Permutation $L(k)$ beginnend bei $i = 0$. Die Sondierungsreihenfolge für einen

Schlüsselwert $k \in U$ ist durch die Funktion $s(k,i) = L(k)[i]$ gegeben, d.h. beim i-ten Sondieren wird das i-te Element in $L(k)$ als Position in der Hashtabelle ausgewählt.

Als „zufällige“ Permutationen nutzen Sie:

$$L(54) = 3,7,9,2,0,6,5,1,4,10,8 \qquad L(22) = 9,2,7,10,4,1,0,5,3,6,8$$
$$L(15) = 8,0,10,7,4,3,1,2,6,9,5 \qquad L(5) = 3,6,5,9,0,2,7,1,4,8,10$$
$$L(27) = 4,10,0,9,6,5,8,2,1,3,7 \qquad L(14) = 4,1,9,3,7,6,8,10,0,2,5$$
$$L(76) = 8,1,6,0,4,5,2,10,3,7,9$$

Aufgabe 2 (Binäre Suchbäume) **1 + 2 + 2 + 3 + 3 + 3 = 14 Punkte**

In der Vorlesung haben Sie binäre Suchbäume kennengelernt, die ganze Zahlen als Schlüssel speichern und die Operationen Einfügen, Suchen und Löschen von Schlüsseln unterstützen. Wie in der Vorlesung betrachten wir hier nur Suchbäume, die keine Duplikate enthalten, d. h., alle Schlüssel des Baumes sind paarweise verschieden.

a) Beweisen oder widerlegen Sie folgende Aussage: Wenn man in einen binären Suchbaum zwei verschiedene Schlüssel x und y einfügt, so erhält man unabhängig von der Einfügereihenfolge stets den gleichen Suchbaum.

b) Beweisen oder widerlegen Sie folgende Aussage: Wenn man aus einem binären Suchbaum zwei verschiedene Schlüssel x und y löscht, so erhält man unabhängig von der Löschreihenfolge stets den gleichen Suchbaum. Wenden Sie dabei die *Symmetrischer Vorgänger*-Methode an, d.h. beim Löschen eines Schlüssels v, der in einem Knoten mit zwei Kindern gespeichert ist, wird der Schlüssel v durch den größten Schlüssel im linken Teilbaum ersetzt (siehe Vorlesung).

c) Zeigen Sie: Wenn ein Knoten eines binären Suchbaumes zwei Kinder hat, dann hat sein symmetrischer Nachfolger, d. h. der Knoten im Baum mit dem nächstgrößeren Schlüssel, kein linkes Kind.

d) Beweisen oder widerlegen Sie, dass es zu jedem binären Suchbaum T eine Reihenfolge seiner Schlüssel gibt, so dass man durch Einfügen der Schlüssel in dieser Reihenfolge in einen anfangs leeren Baum den Baum T erhält.

e) Angenommen Sie suchen einen Schlüssel k in einem binären Suchbaum und finden ihn schließlich. Sei P die Menge der Schlüssel auf dem Suchpfad zu k und L bzw. R die Menge aller Schlüssel in Teilbäumen, die links bzw. rechts vom Suchpfad liegen (d. h., L, R und P sind paarweise disjunkt und die Vereinigung von L, R und P ist die gesamte Schlüsselmenge).

   Beweisen oder widerlegen Sie, dass dann für jede Wahl von $x \in L$, $y \in P$ und $z \in R$ die Aussage $x \leq y \leq z$ gilt.

f) Gegeben sei ein beliebiges Array A mit n paarweise vergleichbaren Elementen. Beweisen Sie, dass es keinen Algorithmus gibt, welcher in $o(n \log n)$ Schritten aus A einen (beliebigen) Suchbaum konstruiert.

Aufgabe 3 (AVL Bäume) **7 + 7 = 14 Punkte**

Führen Sie einen Schriebtischtest für die folgenden zwei Aufgaben zu AVL-Bäumen aus.

(a) Sei T ein leerer AVL-Baum. Fügen Sie nacheinander die Schlüssel

$$4, 15, 42, 20, 12, 2, 28, 22, 69, 16, 18, 80$$

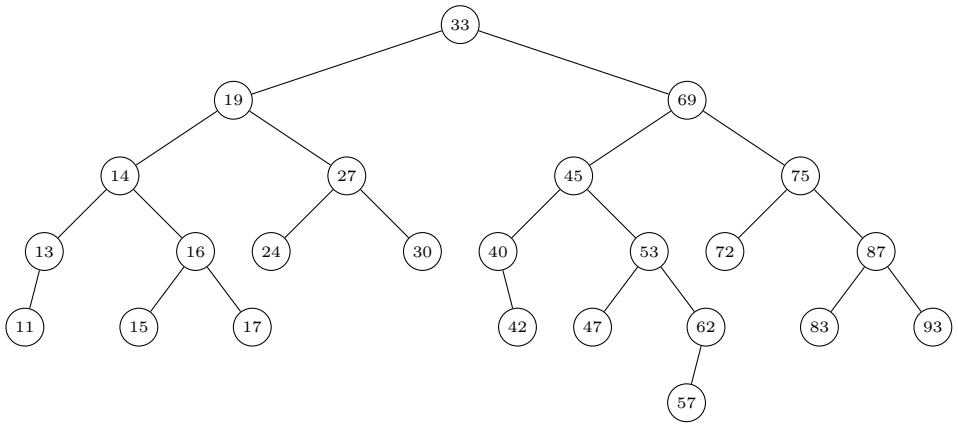
in T ein. Zeichnen Sie den jeweiligen AVL-Baum nach dem Einfügen jedes einzelnen Schlüssels. Notieren Sie außerdem die Balance-Faktoren der Knoten und machen Sie kenntlich, ob Rotationen notwendig waren, um die Balance wieder herzustellen.

(b) Entfernen Sie nacheinander die Schlüssel

$$14, 16, 11$$

aus dem unten gegebenen AVL-Baum. Zeichnen Sie den jeweiligen AVL-Baum nach dem Entfernen jedes einzelnen Schlüssels. Notieren Sie außerdem die Balance-Faktoren der Knoten und machen Sie kenntlich ob Rotationen notwendig waren, um die Balance wieder herzustellen.

Falls ein Knoten mit zwei Kindern gelöscht wird, dann soll mit dem symmetrischen Vorgänger getauscht werden.

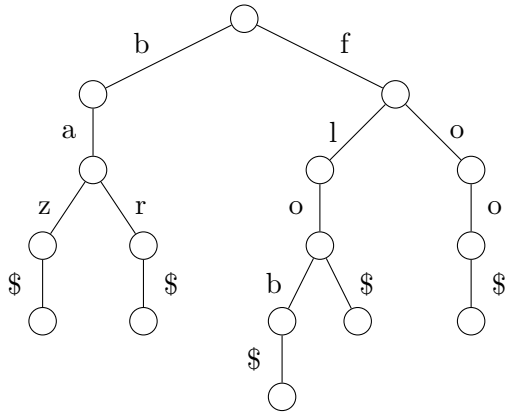


Aufgabe 4 (Kantenbeschriftete Bäume) **3 · 4 = 12 Punkte**

Sogenannte *Tries* eignen sich besonders um eine Menge von Wörtern zu speichern und effizient darauf zuzugreifen. Tries sind Bäume, deren Kanten mit Buchstaben beschriftet sind. Jedes im Trie gespeicherte Wort entspricht einem Pfad von der Wurzel zu einem Blatt. Gemeinsame Präfixe unterschiedlicher Wörter werden dadurch nur einmal gespeichert. Tries werden daher auch als Präfixbäume bezeichnet.

Wir kennzeichnen das Ende eines Wortes, indem wir an jedes zu speichernde Wort ein $-Zeichen anhängen. Dadurch kann auch das Ende eines Wortes, welches zugleich ein Präfix eines anderen Wortes ist, erkannt werden.

Im Folgenden finden Sie ein Beispiel für einen Trie, in den die Wörter `foo`, `bar`, `baz`, `flo` und `flob` eingefügt wurden:



Ihre Aufgabe ist es, die Implementierung eines Tries zu vervollständigen. Nutzen Sie hierzu die auf der Übungswebseite zur Verfügung gestellte Datei `Trie.java` und ergänzen Sie die unten aufgeführten Methoden um fehlenden Code. An bestehendem Code soll nichts verändert werden. Sie dürfen aber die `main()`-Methode zum Testen verwenden und auch um Ihre eigenen Testfälle erweitern.

(a) Implementieren Sie die Methode `add(word)`, die ein gegebenes Wort in den bestehenden Baum einfügt.

(b) Implementieren Sie die Methode `contains(word)`, die überprüft, ob ein gegebenes Wort im Baum vorhanden ist und entsprechend `true` oder `false` zurückgibt.

(c) Die Methode `printWords()` gibt alle im Trie enthaltenen Wörter aus. Sie verwendet dazu die Methode `printWordsRecursive(node, word)`, welcher die Wurzel und der leere String als Parameter übergeben werden.

Implementieren Sie die Methode `printWordsRecursive(node, word)`, die rekursiv alle Wörter w ausgibt, für die folgendes gilt: Wort w ist die Konkatenation des Präfixes `word` mit einem im Teilbaum des Knotens `node` enthaltenen Wort.

Hinweis zur Abgabe: Ihr Java-Programm muss unter *Java 1.7* auf dem Rechner *gruenau2* laufen. Kommentieren Sie Ihren Code, sodass die einzelnen Schritte nachvollziehbar sind. Die Abgabe der von Ihnen modifizierten Datei `Trie.java` erfolgt über Goya.