Algorithmen und Datenstrukturen
–Aufgabenblatt 04 (30 Punkte)–

Hinweise

Die Lösungen der Aufgaben sind als PDF-Dokument bzw. Java-Quelltext mit Hilfe des Versionskontrollsystems Subversion (SVN) abzugeben. Platzieren Sie das PDF-Dokument mit ihren Antworten im Ordner `a04` innerhalb Ihres Gruppenverzeichnisses[1]. Platzieren Sie die Java-Quelltexte im Unterordner `a04/src/main/java` entsprechend der Java Package-Hierarchie (siehe Vorlage). Platzieren Sie ihre Tests im Unterordner `a04/src/test/java` entsprechend der Java Package-Hierarchie (siehe Vorlage). Abweichungen führungen möglicherweise zum Punktabzug. Alle Klassen und Interfaces sind in eigenen Dateien abzulegen. Schreiben Sie bitte noch in die abgegebenen Dateien die Namen und Matrikelnummern aller Gruppenmitglieder. Verspätete Abgaben werden nicht gewertet! Nutzen Sie keine vorgefertigten Datenstrukturklassen in Java, sondern implementieren Sie selbst die Algorithmen.

Ausgabe: 20.06.2024
Abgabe: 06.07.2024 bis 23:59 Uhr

Aufgabe 1 - Hashing (20 Punkte)

In der Vorlesung wurde *Hashing* als ein Verfahren zur Speicherung von Daten eingeführt, das eine effiziente Suche in unsortierten Datensätzen ermöglicht. Gegeben sei die Schlüsselfolge $\{25, 30, 18, 24, 40, 27\}$, die in einer *Hash-Tabelle* der Größe M gespeichert werden soll.

Anmerkung: Die Nutzung der Java `HashMap`/`HashSet` ist nachfolgend nicht gestattet.

(a) Wenden Sie die einfache multiplikative Methode $h(k) = \frac{k-min}{max-min} \times (M-1)$ aus der Vorlesung mit Rundung der Werte und *Separate Chaining* zur Kollisionsauflösung an! Es sei $M = 5$ und der Wertebereich der Schlüssel liege zwischen 0 und 40. Skizzieren Sie die Verteilung der Elemente in der Hashtabelle! Wie ändert sich die Tabelle, wenn Sie wissen, dass die Werte der Schlüssel nur zwischen 18 und 40 liegen können? Geben Sie auch die neue Verteilung an! (4 Punkte)

(b) Wenden Sie nun die modulare Methode bei einer Tabellengröße von $M = 7$ an und nutzen Sie *Open Addressing* für die Kollisionsbehandlung! Skizzieren Sie die Verteilung der Elemente in der Hashtabelle (3 Punkte)

1. bei linearem Sondieren mit einem Inkrement von $b = 1$
2. bei linearem Sondieren mit einem Inkrement von $b = 2$
3. bei quadratischem Sondieren!

(c) Programmieren Sie eine Hashtabelle für Strings in der Klasse *StringTable*. Die Kapazität der Hashtabelle M wird dem Konstruktor der Klasse übergeben. Schreiben Sie eine Methode *hash*, die den Hashwert eines Strings berechnet, der sich aus der Summe der ASCII-Codes der Zeichen modulo M ergibt. Der Rückgabewert ist die Position in der Hashtabelle (ohne Sondierung). Implementieren Sie mit Hilfe von *hash* die Methoden (9 Punkte)

1. *put* zum Einfügen in die Tabelle. Die Methode gibt die Anzahl der Kollisionen zurück, die bei dem Einfügen aufgetreten sind. Bei dem Versuch in eine volle Tabelle einzufügen, soll *put* eine *RuntimeException* werfen.
2. *delete* zum Löschen aus der Tabelle. Die Methode gibt die Anzahl der Kollisionen zurück, die beim Löschen aufgetreten sind.
3. *contains* zum Prüfen, ob ein Wert in der Tabelle enthalten ist.

Verwenden Sie *Open Addressing* und lineares Sondieren.

[1] Ihr Gruppenverzeichnis ist unter `https://svn.informatik.uni-rostock.de/lehre/ads2025/groups/<group>/` mit <group> $\in \{01, ..., 42\}$ zu finden.

(d) Vergleichen Sie Ihre bereits implementierte Hashfunktion mit der Hashfunktion, dessen Hashwerte das Minimum aus der Summe der ASCII-Zeichenwerte und $M - 1$ sind. Der Vergleich soll auf den Anzahlen der Kollisionen, wenn zufällige Werte gehasht werden, basieren. Implementieren Sie die Methoden/Funktionen (4 Punkte)

1. *hash* in der Klasse *StringTable2*. Diese Methode soll die neue Hashfunktion berechnen.
2. *uniformValues* in der Klasse *HashingExperiment*. Sie soll 1.000 Mal ein neues *StringTable*-Objekt mit Kapazität 10 erstellen und diesem Objekt zwei zufällige Werte hinzufügen. Die Ausgabe ist die Anzahl der aufgetretenen Kollisionen. Nutzen Sie die Methode *nextUniform* aus der Klasse *RandomString* für die Zufallswerte.
3. *uniformValues2*, welche denselben Prozess wie *uniformValues* mit Objekten des Typs *StringTable2* durchführt.
4. *main* in der Klasse *HashingExperiment*. Hier sollen die Ergebnisse von *uniformValues* und *uniformValues2* ausgegeben werden. Geben Sie die erwünschte Eigenschaft von Hashingfunktionen aus der Vorlesung an, die bei einer der Hashingfunktionen fehlt, wodurch in einer der Experimentfunktionen besonders viele Kollisionen auftreten.
5. *gaussianValues* in der Klasse *HashingExperiment* analog zu der Funktion *uniformValues*. Benutzen Sie hierbei von dem *RandomString* Objekt die Methode *nextGaussian* anstelle von *nextUniform*, um zufällige Werte zu ziehen. Nutzen Sie die *main* Funktion in der Klasse *HashingExperiment*, um das Ergebnis von *gaussianValues* auszugeben. Geben Sie die erwünschte Eigenschaft von Hashingfunktionen aus der Vorlesung an, die hier maßgeblich für die Anzahl der Kollisionsn in der Experimentfunktion *gaussianValues* ist. Beachten Sie, dass zwei aufeinanderfolgende Zufallswerte von *RandomString* immer verschieden sind.

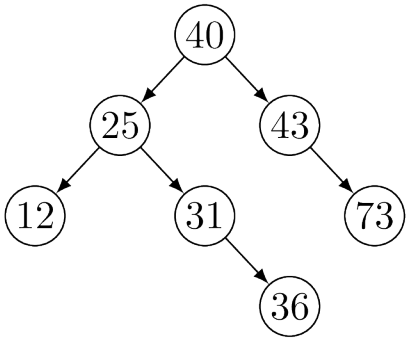
Aufgabe 2 - AVL Bäume (10 Punkte)

Ein Baum ist höhenbalanciert, wenn für jeden seiner Knoten gilt, dass der Unterschied zwischen der Höhe seines linken und der Höhe seines rechten Teilbaumes maximal 1 beträgt. Der Balanceindex ist die Höhendifferenz des linken und rechten Teilbaums eines Knotens. Somit kann der Balanceindex von AVL-Bäumen nur die Werte -1, 0, 1 annehmen.

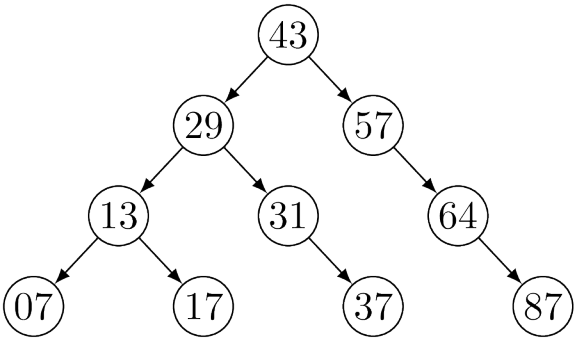
(a) Fügen Sie in einen leeren AVL-Baum nacheinander die Schlüssel 2,3,4,5,6,9,8,7,1 ein! Skizzieren Sie den Baum nach Einfügen des letzten Schlüssels! (4 Punkte)

(b) Geben Sie den Balanceindex für jeden Knoten der Binärbäume (a) bis (d) an! (4 Punkte)

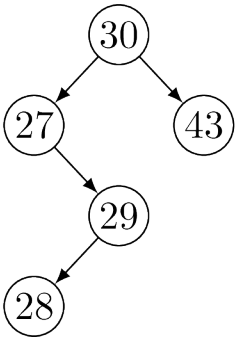
(c) Welche der Binärbäume sind keine AVL-Bäume? Geben Sie die Abfolge von Rotationen an, um diese in AVL-Bäume zu überführen! Skizzieren Sie das Ergebniss der Rotationen! (2 Punkte)


40
25
43
12
31
73
36

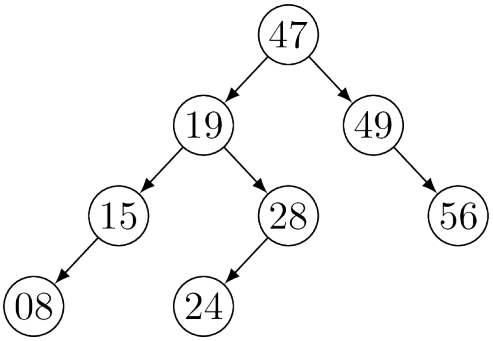
(a)


43
29
57
13
31
64
07
17
37
87

(b)


30
27
43
29
28

(c)


47
19
49
15
28
56
08
24

(d)