FAU, Informatik 2, AUD-Team
aud@i2.cs.fau.de

10. Übung

Abgabe bis 22.01.2018, 10:00 Uhr

Einzelaufgabe 10.1: Sortierte verkettete Listen — 4 EP

Betrachten Sie folgende Datenstruktur:

- einfach verkettete Liste mit Wächterelement (analog zu den Folien der Vorlesung),
- eingeschränkt auf den Typ `Double` (Wächterelement enthält den Wert `Double.NaN`),
- die Elemente der Liste werden aufsteigend sortiert verwaltet.

a) Fügen Sie die Elemente 0.815, -42.24, 4711.42 und 42.666 in eine zuvor leere Liste ein und notieren Sie das Ergebnis. Bitte orientieren Sie sich bei der Darstellung der Liste, ihrer Einträge sowie den Referenzen an den Vorlesungsfolien.

b) Fügen Sie in das Ergebnis der Teilaufgabe a) das Element 666.42 an der korrekten Stelle ein und notieren Sie das Ergebnis.

c) Fügen Sie in das Ergebnis der Teilaufgabe a) das Element `Double.MIN_VALUE` an der korrekten Stelle ein und notieren Sie das Ergebnis.

d) Welchen Aufwand hat das Einfügen eines Elements in eine sortierte, einfach verkettete Liste mit n Elementen (unter Beibehaltung der Sortierung)?

Geben Sie Ihre Lösung als `SortierteVerketteteListe.pdf` über EST ab.

Einzelaufgabe 10.2: Streuspeicherung — 10 EP

Fügen Sie die folgenden Werte in der gegebenen Reihenfolge in eine Streutabelle der Größe 8 (mit den Indizes $0 \ldots 7$) und der Streufunktion $h(x) = x \,\%\, 8$ ein, indem Sie im Falle von Kollisionen das jeweils genannte Verfahren verwenden:

`24, 10, 20, 26, 15, 7, 18, 29.`

a) *Offenes Hashing*: Kollisionsauflösung durch *verkettete Listen*.
Beispiel: Für die beiden Werte **8** und **16** würde die Lösung wie folgt aussehen:

Bucket	Inhalt
0	**8 - 16**
1	
2	
⋮	⋮

b) *Geschlossenes Hashing*: Kollisionsauflösung durch *lineares Sondieren* mit Schrittweite $+5$. Treten beim Einfügen Kollisionen auf, dann notieren Sie die Anzahl der Versuche zum Ablegen des Wertes im Subskript (z.B. das Einfügen des Wertes 8 gelingt im 5. Versuch: „8_5“).
Beispiel: Für die beiden Werte **8** und **16** würde die Lösung wie folgt aussehen:

Bucket	Inhalt
0	**8**
1	
2	
3	
4	
5	$\mathbf{16_1}$
⋮	⋮

c) Welches Problem tritt auf, wenn zur Kollisionsauflösung lineares Sondieren mit Schrittweite $+4$ verwendet wird? Warum ist $+5$ eine bessere Wahl?

d) Geben Sie für Ihre Lösungen zu a) und b) jeweils den Lastfaktor an.

e) Gegeben sei eine Streutabelle, in der die gleichen Schlüssel mehrfach enthalten sein können. Welches der beiden Verfahren aus den Teilaufgaben a) und b) ist in der *Worst-Case*-Laufzeit performanter beim Einfügen eines neuen Eintrags? Welches Verfahren ist performanter beim Abrufen des Wertes zu einem Schlüssel? Begründen Sie kurz Ihre Aussage!

Geben Sie Ihre Lösung als `Streuspeicherung.pdf` über EST ab.

Einzelaufgabe 10.3: Binäre Suchbäume — 23 EP

In dieser Aufgabe sollen Sie eine Unterklasse von `AbstractBinTreeNode` namens `BinTreeNode` implementieren, die die Datenstruktur *binärer Suchbaum* (*ohne* Balancierung und *ohne* Duplikate) darstellen soll. Dabei repräsentiert Ihre Klasse gleichzeitig sowohl einen einzelnen Knoten, als auch den gesamten Unterbaum, der diesen Knoten als Wurzel hat. Die *Ordnungsrelation* müssen Sie dadurch zusichern, dass alle zulässigen Werte die Schnittstelle *Comparable* bereitstellen müssen.
Implementieren Sie die von der Oberklasse vorgegebenen Methoden gemäß Java-Doc-Kommentare.
ACHTUNG: Es sind **sämtliche** Aufrufe in die Java-API **untersagt**, abgesehen von den durch die Superklasse erzwungene Verwendung bestimmter Klassen, Schnittstellen und Ausnahmen! Ihre Klasse darf **keine** Attribute und **keine** zusätzlichen non-`private` Methoden deklarieren!
Geben Sie Ihre Lösung als `BinTreeNode.java` über EST ab.

Gruppenaufgabe 10.4: SkipList — 23 GP

Der lineare Aufwand beim Zugriff auf die Elemente einer sortierten verketteten Liste im Falle einer naiven Implementierung kann durch die geschickte Nutzung zusätzlicher Referenzen verbessert werden. Diese verbesserte Laufzeit wird dabei durch einen erhöhten Speicherverbrauch *„erkauft“*.
Die Datenstruktur *Skip-List* ermöglicht das schnelle Auffinden von Daten in einer derartigen Liste. Anstatt eine einfache Kette aus Elementen zu erzeugen, wie es bei klassischen verketteten Listen mit Sortierung üblich ist, wird eine ganze Hierarchie an verketteten Elementen aufgebaut. Jede einzelne Stufe der Hierarchie entspricht dem bekannten Prinzip – allerdings werden auf jeder höheren Ebenen mehr Elemente ausgelassen („skipped“, vgl. Abbildung 1). Sämtliche Operationen auf einer Skip-List starten auf der obersten Ebene und steigen bei Bedarf schrittweise auf tiefere Ebenen ab.
In dieser Aufgabe sollen Sie das Prinzip der *Skip-List* in einer Unterklasse von `AbstractSkipList` umsetzen. Diese soll *exakt* die Schnittstelle `Set<E>` gemäß Java-API-Dokumentation bereitstellen.
ACHTUNG: Es sind **sämtliche** Aufrufe in die Java-API **untersagt**, abgesehen von den durch die Superklasse bzw. Schnittstelle erzwungene Verwendung bestimmter Klassen und Ausnahmen! Ihre Klasse darf **höchstens ein** `int`-Attribut, aber **keine** zusätzlichen Methoden deklarieren!

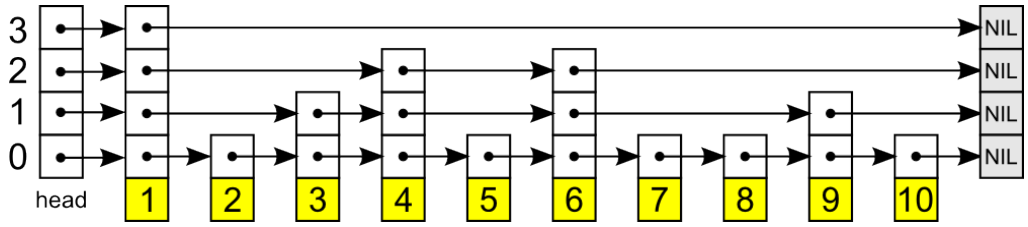

Abbildung 1: Beispiel einer Skip-List mit 4 Ebenen. (Quelle: *Wikipedia*)

- Ihre Skip-List soll Elemente des generischen Typs `E extends Comparable<? super E>` verwalten, d.h. alle Elemente sind mittels `compareTo()` vergleichbar (in dieser Aufgabe jedoch *ausnahmsweise nicht* auch noch durch `equals()` – unbedingt öffentliche Testfälle beachten!).
- Der „Wert `null`“ soll *nicht* verwaltet und daher in *keiner* Methode verarbeitet werden können. Werfen Sie in diesem Fall eine `NullPointerException` in den relevanten Methoden. Die weiteren (in der API) dokumentierten Ausnahmen dürfen Sie vernachlässigen.
- Eine Menge darf *keine* zwei gemäß `compareTo()` gleichen Elemente enthalten (*Menge!*).
- Zur hierarchischen Verkettung ist die Klasse `SkipListNode<E>` vorgegeben. Verwenden Sie diese zur internen Repräsentation. Im Array `next` kann für jede Ebene der jeweilige Nachfolger gespeichert werden.
- Die höchste Ebene, die ein einzufügendes Element erreicht, wird beim Einfügen zufällig bestimmt. Implementieren Sie hierfür die Methode `getRandomLevel()` und nutzen Sie die Methode `nextBoolean()` der schon vorhandenen Instanz des `Random`-Objekts. Liefert `nextBoolean()` den Wert `true` zurück, erreicht ein Element die nächste Ebene. Die Zufallsverteilung soll die im Kommentar angegebenen Eigenschaften erfüllen. Beachten Sie, dass ein Element auch in **allen** tieferen Ebenen in die Verkettung aufgenommen werden muss.
- Achten Sie beim Einfügen (und Löschen) darauf, dass Sie pro besetzter Ebene den richtigen Platz in der Verkettung suchen müssen. Der Suchraum wird dabei jeweils durch den bereits gefunden Platz in der Ebene darüber eingegrenzt.
- Die Methoden `size()` und `isEmpty()` *müssen* einen Laufzeitaufwand in $\mathcal{O}(1)$ haben.
- `add()`, `remove()` und `contains()` *müssen* den mittleren Aufwand $\mathcal{O}(\log(n))$ haben.
- Die Hilfsmethoden `addAll()`, `containsAll()` und `removeAll()`, jeweils angewandt auf eine Skip-List mit n Elementen und einer `Collection` mit m Einträgen, dürfen Aufwand $\mathcal{O}(m \cdot \log(n))$ haben und müssen nicht weiter optimiert werden. Außerdem ist in diesen Methoden keine gesonderte Ausnahmebehandlung nötig.

Geben Sie Ihre Lösung als `SkipList.java` über EST ab.

37 EP + 23 GP = 60 Punkte