Effiziente Algorithmen

Sommersemester 2019

Prof. Dr. Martin Hoefer
Dr. Daniel Schmand
Martin Ludwig, Conrad Schecker

GOETHE UNIVERSITÄT
FRANKFURT AM MAIN

Institut für Informatik
Algorithmen & Komplexität

Übung 9

Ausgabe: 18.06.2019
Abgabe: 26.06.2019

Dieses Hausaufgabenblatt besteht aus einer Programmieraufgabe und hat spezielle Abgabemodalitäten. Die Abgabe erfolgt durch das Vorführen und Erklären des selbst geschriebenen Codes am Mittwoch, den 26.06.2019 in der Zeit von 10-12 Uhr, 13-15 Uhr oder 16-18 Uhr in Raum 104a, 114 oder 117, Robert-Mayer Str. 11-15. Sollte die Abgabe an diesen Terminen nicht möglich sein, kann ein früherer Abgabetermin individuell mit Daniel Schmand vereinbart werden.

Ziel dieser Übung ist es, die Datenstruktur `BinomialHeap` aus der Vorlesung zu implementieren. Wir stellen hierzu Templates und eine Testklasse für die Programmiersprachen Java und Python auf der Homepage bereit. Die Aufgabe kann auch in anderen Programmiersprachen gelöst werden, die Templates und die Testklasse sind dann allerdings selbst entsprechend anzupassen.

Hinweis: Jede Teilaufgabe gilt als erfolgreich absolviert, wenn die Implementierung korrekt vorgenommen wurde und erklärt werden kann. Dabei ist es notwendig (aber nicht unbedingt hinreichend), dass die Testklasse `BinomialHeapTestClass` für die entsprechende Aufgabe `OK` ausgibt. Es kann in jeder Teilaufgabe entweder die volle Punktzahl oder 0 Punkte erreicht werden.

Aufgabe 9.1. (6 Punkte)

Implementiere eine Klasse `BinHeapNode`, die einen Knoten in einem Binomischen Heap repräsentiert. Ein Knoten benötigt folgende Datenfelder:

- `data`: der Wert des Knotens (hier: Integer)
- `parent`: eine Referenz auf den Vater des aktuellen Knotens
- `index`: Jeder Knoten ist immer auch die Wurzel eines eigenen binomischen Baums. Der `index` des Knoten ist k, wenn er selbst Wurzelknoten eines binomischen Baums B_k ist.
- `children_sorted`: Eine Liste der Kinder des aktuellen Knotens. Aus Laufzeitgründen halten wir diese Liste immer nicht-fallend sortiert nach index vor. Das heißt, das erste Element der Liste ist ein Kind mit kleinstem Knotenindex.

In der zur Verfügung gestellten Vorgabe `BinHeapNode.py` oder `BinHeapNode.java` sind bereits die Datenfelder und Getter/Setter-Methoden implementiert. Es fehlt noch die Implementierung der Methode `add_child(self, child)` beziehungsweise `public void addChild(BinHeapNode child)`. Diese Methode soll in linearer Zeit ein neues Kind an die richtige Stelle in der Liste der Kinder dieses Knotens einsortieren und auch den Vater des neuen Kindes entsprechend aktualisieren.

Die Hauptklasse dieser Programmieraufgabe ist `BinomialHeap`. Die Klasse enhält lediglich ein Datenfeld `_root_nodes_sorted` bzw. `rootNodesSorted`, eine Liste von Wurzelknoten von Binomischen Bäumen im Heap. Wir möchten auch diese Liste aus Laufzeitgründen nach nicht-fallendem `index` der Wurzelknoten sortiert vorhalten. Die erste Aufgabe für diese Klasse ist es, die Methode

`insert(self, value)` bzw. `public BinHeapNode insert(int value)` zu implementieren. Dazu benötigen wir mehrere Zwischenschritte. Zunächst beginnen wir mit der Methode `_clean_up(self)` beziehungsweise `private void cleanUp()`. Beim CleanUp-Prozess soll erreicht werden, dass die Liste der sortierten Wurzelknoten keine zwei Knoten mit gleichem Index enthält. Zwei solche Bäume sollen zu einem neuen Binomischen Baum mit nächstgrößerem Index verschmolzen werden. Beachte dabei, dass beim Verschmelzen die Heap-Eigenschaft nicht verletzt werden darf. Es bietet sich an, als Hilfsmethode die statische Methode `_merge_trees(first, second)` oder `private static BinHeapNode mergeTrees(BinHeapNode first, BinHeapNode second)` zu implementieren und benutzen. (Die Implementierung vom CleanUp-Prozess in linearer Laufzeit ist nicht zwingend vorgeschrieben, aber durchaus möglich. Wie?)

Als letztes können wir nun die Methode `insert` implementieren. Hierbei soll, wie in der Volesung besprochen, ein neuer BinHeapNode erstellt, an die korrekte Position eingefügt, und anschließend ein CleanUp durchgeführt werden. Die Methode gibt dann eine Referenz auf den neu erstellten BinHeapNode zurück.

Aufgabe 9.2. (3 Punkte)

Als nächstes soll die Methode `delete_min(self)` beziehungsweise `public int deleteMin() throws NoSuchElementException` implementiert werden. Die Methode soll ein aktuell kleinstes Element im `BinomialHeap` finden, ausgeben und löschen. Beachte, dass nach dem Löschen die neu entstanden Binomischen Bäume korrekt eingefügt und anschließend ein `_clean_up` durchgeführt werden muss. Falls der Heap leer ist, soll diese Methode einen `IndexError` bzw. eine `NoSuchElementException` werfen.

Aufgabe 9.3. (3 Punkte)

In dieser Aufgabe soll die Methode `decrease_key (self, node, new_value)` bzw. `public void decreaseKey (BinHeapNode node, int newValue) throws IllegalArgumentException` implementiert werden. Die Methode soll dazu dienen, den Wert eines übergebenen Knoten auf den Wert `new_value` bzw. `newValue` zu setzen. Falls der übergebene Wert größer als der aktuelle Wert ist, solle ein `ValueError` bzw. eine `IllegalArgumentException` geworfen werden. Beim Verringern des Schlüsselwertes kann es zu einer Verletzung der Heapeigenschaft kommen, die entsprechend wieder repariert werden muss. In dieser Aufgabe muss dies in logarithmischer Laufzeit in der Anzahl der Elemente im `BinomialHeap` implementiert werden.

Aufgabe 9.4. (3 Punkte)

Als letztes soll über die Methoden `decrease_key` und `delete_min` eine Methode `delete(self, node)` realisiert werden.

E-Mail: `{mhoefer,schmand}@em.uni-frankfurt.de`