Algorithmen und Datenstrukturen
Übungszettel 4

Martin Avanzini <martin.avanzini@uibk.ac.at>
Thomas Bauereiß <thomas.bauereiss@uibk.ac.at>
Herbert Jordan <herbert@dps.uibk.ac.at>
René Thiemann <rene.thiemann@uibk.ac.at>

5. April, zur Besprechung am 12. April

Aufgabe 1) Komplexität von Sortieralgorithmen Gegeben sei ein Sortieralgorithmus A für ein Eingabe-Array $[a_1, a_2, \ldots, a_n]$, der bei der Sortierung ausschließlich Vergleichsoperationen verwendet, um Informationen über die Elemente des Eingabe-Arrays zu erhalten. Das bedeutet, dass A auf die Werte der Elemente nur im Rahmen von Vergleichsoperationen der Form $a_i < a_j$ zugreifen darf. Beispiele aus der Vorlesung für solche Algorithmen sind Merge-Sort oder Quick-Sort.

1. Beweisen Sie, dass $\Omega(n \log n)$ eine untere Schranke für die Wort-Case-Komplexität eines solchen Algorithmus ist. Hinweise:
   (a) Sie können annehmen, dass alle a_i verschieden sind.
   (b) Gehen Sie davon aus, dass nur Vergleichsoperationen der Form $a_i < a_j$ durchgeführt werden. Andere Vergleichsoperatoren wie $\leq$, $\geq$ oder $>$ werden nicht benötigt.
   (c) Sie können Ihren Beweis auf der Anzahl möglicher Permutationen $n!$ einer n-elementigen Sequenz basieren. Dabei können Sie die Stirling-Approximation $n! \geq \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$ verwenden.
2. In der Vorlesung wurde gezeigt, dass Bucket-Sort eine Komplexität von $\mathcal{O}(n + m)$ für n Eingabe-Elemente und m Schlüssel hat. Begründen Sie, warum dies nicht im Widerspruch zu der in Teilaufgabe 1 bewiesenen Aussage steht.

Aufgabe 2) Einfügen und Löschen in binären Suchbäumen Führen Sie die folgenden Operationen auf einem anfangs leeren binären Suchbaum aus:

E42, E23, E47, E11, E37, L11, L23, E11, E45, E40, E39, L42, E64, E48, E46, L47

(E = Einfügen, L = Löschen)

Zeichnen Sie den entstehenden Baum nach je zwei Operationen.

Aufgabe 3) Baum-Iteratoren Die in der Vorlesung vorgestellte Java-Implementierung eines Baum-Iterators unterstützt nur das Iterieren durch den Baum, nicht aber das Verändern des Baums. Erweitern Sie die Implementierung des Baum-Iterators, so dass die `remove()`-Methode zum Entfernen des aktuellen Elements unterstützt wird. Dabei sollen für `remove()` folgende Regeln gelten:

- Wenn `remove` aufgerufen wird, wird das Element aus dem Baum entfernt, das zuletzt von `next` zurückgeliefert wurde.
- `remove` darf nur nach einem Aufruf von `next` aufgerufen werden. Es ist nicht erlaubt, `remove` mehr als einmal ohne dazwischenliegenden Aufruf von `next` aufzurufen.
- Bei einem unerlaubten Aufruf von `remove` soll eine `IllegalStateException` geworfen werden.
- Nach einem erlaubten Aufruf von `remove` soll der Iterator normal weiterlaufen, d.h. am Ende der Iteration müssen alle Elemente durchlaufen und kein Element doppelt zurückgeliefert worden sein.

In den Java-Sourcen dieser Übung finden Sie eine leicht vereinfachte Implementierung eines Binärbaums und eines Baumiterators aus der Vorlesung, der den Baum in Inorder-Reihenfolge durchläuft.

1. Erweitern Sie die gegebene Implementierung, indem Sie die `remove`-Methode des Baum-Iterators implementieren (und andere Methoden anpassen, falls nötig). Sie können Ihre Implementierung mit dem beiliegenden Programm `BinTreeTest` testen. Es iteriert durch den Baum, gibt die enthaltenen Knoten aus, und enthält drei legale Aufrufe von `remove` und einen illegalen am Ende. Es sollte also alle Knoten des Baums jeweils genau einmal ausgeben, gefolgt von einer `IllegalStateException`.
2. Würde diese Implementierung auch für Preoder- oder Postorder-Traversierung korrekt funktionieren? Falls nicht, beschreiben Sie das auftretende Problem.