Algorithmen und Datenstrukturen
Musterlösung 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

1. Der Beweis kann auf der Betrachtung eines Entscheidungsbaums basiert werden, der die Vergleichsoperationen repräsentiert, die ein Sortieralgorithmus auf n-elementigen Eingabearrays durchführt (Illustration siehe Cormen et al., Introduction to Algorithms, Abschnitt 8.1). Ein innerer Knoten repräsentiert eine Vergleichsoperation $a_i < a_j$ und ein Blatt des Baums eine bestimmte Permutation des Eingabearrays. Die Ausführung eines Sortieralgorithmus, der auf Vergleichsoperationen basiert, entspricht dann dem Folgen eines Pfads von der Wurzel des Baums bis zu einem Blatt, je nach den Ergebnissen der Vergleichsoperationen: Beim Ergebnis $a_i < a_j$ eines Vergleichs an einem der inneren Knoten wird im Folgenden der linke Unterbaum betrachtet, beim Ergebnis $a_i \geq a_j$ der rechte Unterbaum. An einem Blatt angelangt, kann mit den Ergebnissen der Vergleiche bestimmt werden, welche Permutation des Eingabearrays vorliegt und damit das Array sortiert werden.

   Damit ein Algorithmus alle möglichen Eingabearrays sortieren kann, müssen im Baum alle $n!$ möglichen Permutationen eines n-elementigen Arrays als Blatt vorhanden sein. Ein Baum der Höhe h hat maximal 2^h Blätter, es muss also gelten:

   $$\begin{aligned}
   & 2^h \geq n! \\
   \Rightarrow\ & h \geq \log(n!) && \text{(logarithmieren)} \\
   \Rightarrow\ & h \geq \log\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\right) && \text{(Einsetzen von Stirling's Approximation)} \\
   \Rightarrow\ & h \geq \log(\sqrt{2\pi n}) + n\,(\log n - \log e) \\
   \Rightarrow\ & h = \Omega(n \log n)
   \end{aligned}$$

   Da ein Pfad, den ein Sortieralgorithmus von der Wurzel des Entscheidungsbaums bis zu einem Blatt verfolgen muss, im schlechtesten Fall h Schritte lang ist, ist $h = \Omega(n \log n)$ eine untere Schranke für die benötigte Anzahl an Vergleichsoperationen im Worst Case.

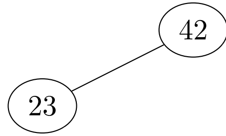
2. Bucket-Sort basiert nicht allein auf Vergleichen, sondern verwendet die Werte selbst zur Sortierung. Die Voraussetzung des Beweises ist somit nicht gegeben. Zudem gibt es bei

endlicher Schlüsselanzahl eben nicht $n!$ verschiedene Permutation eines Arrays der Länge n.

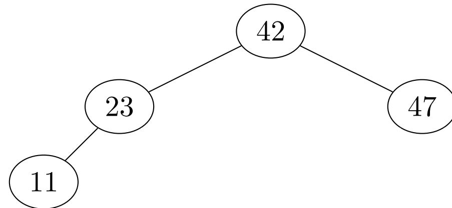
Zwar wächst die Laufzeit von Bucket-Sort $\mathcal{O}(n + m)$ für $m = \mathcal{O}(n)$ asymptotisch linear, also langsamer als die von vergleichsbasierten Sortieralgorithmen, jedoch haben Letztere z.B. den Vorteil, dass sie von schwächeren Annahmen über die Eingabedaten ausgehen. Bucket-Sort liegt die Annahme zu Grunde, dass die Eingabedaten nach einer endlichen Menge von Schlüsseln zu sortieren sind. Bei vergleichsbasierten Sortieralgorithmen wird dagegen lediglich vorausgesetzt, dass eine Ordnungsrelation für die zu sortierenden Elemente existiert und ein entsprechender Vergleichsoperator vorhanden ist oder implementiert werden kann.

Aufgabe 2) Einfügen und Löschen in binären Suchbäumen

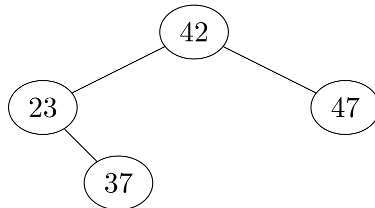
- E42, E23:



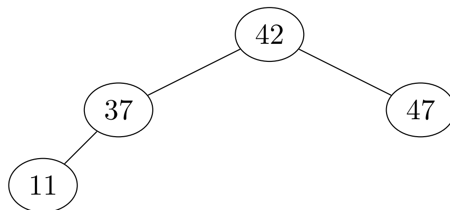
- E47, E11:



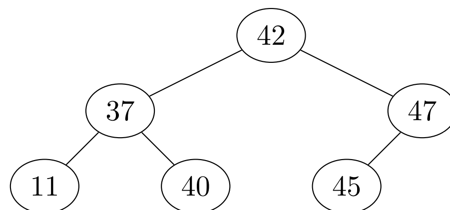
- E37, L11:



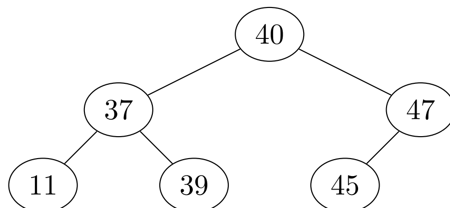
- L23, E11:



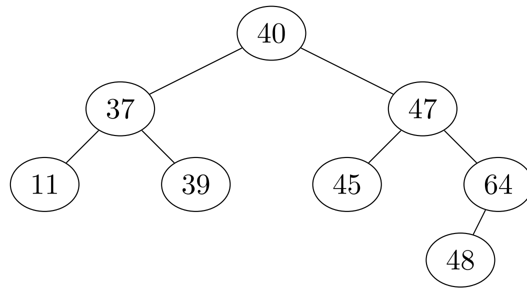
- E45, E40:



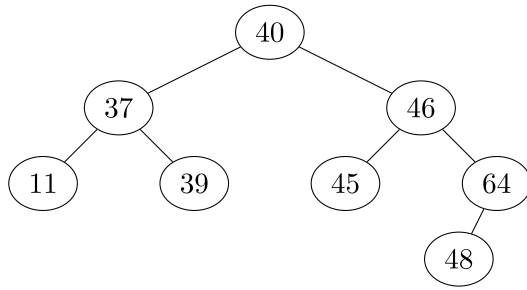
- E39, L42:



- E64, E48:



- E46, L47:



Aufgabe 3) Baum-Iteratoren

1. Änderungen an `TreeIterator` und `BinTree`:

```
1  public class TreeIterator<D> implements Iterator<D> {
2      int direction;   // from which direction ...
3      Node<D> current; // ... did we enter current node
4      Node<D> previous; // the node we previously returned by next()
5      BinTree<D> tree; // the tree we are traversing
6  
7      public TreeIterator(BinTree<D> tree) {
8          this.current = tree.root;
9          this.tree = tree;
10         this.direction = UP;
11         this.previous = null;
12         walkNext(INORDER);
13     }
14     public Entry<D> next() {
15         if (current == null) {
16             throw new NoSuchElementException();
17         } else {
18             Entry<D> entry = new Entry<D>(current.key, current.data);
19             this.previous = this.current;
20             walkNext(NONE);
21             return entry;
22         }
23     }
24 
25     public void remove() {
26         if (this.previous == null) {
27             throw new IllegalStateException();
28         } else {
29             this.tree.removeNode(this.previous);
30             this.previous = null;
31         }
32     }
33 
34 }
```

```
35 public class BinTree<D> implements Dictionary<D> {
36     public Iterator<D> iterator() {
37         return new TreeIterator<D>(this);
38     }
39 
40 }
```

2. Im Fall von Preorder-Traversierung kann es mit der vorliegenden Implementierung Probleme beim Entfernen von Knoten mit zwei Kindern geben. Da beim Entfernen des Knotens an der Wurzel dieses Teilbaums der Knoten mit einem seiner Nachfolger getauscht wird, die ursprüngliche Wurzel aber bereits ausgegeben wurde, kann es passieren dass die neue Wurzel bei der Ausgabe übersprungen wird.

   Bei Postorder-Traversierung kommt dieses Problem nicht vor: Nach Ausgabe eines Knotens sind er und alle seine Nachfolger bereits ausgegeben worden und werden nicht mehr besucht, das Entfernen ist also unproblematisch.