

Übung zur Vorlesung *Einführung in die Informatik 2 für Ingenieure (MSE)*

Alexander van Renen (renen@in.tum.de)

<http://db.in.tum.de/teaching/ss20/ei2/>

Lösungen zu Blatt 5

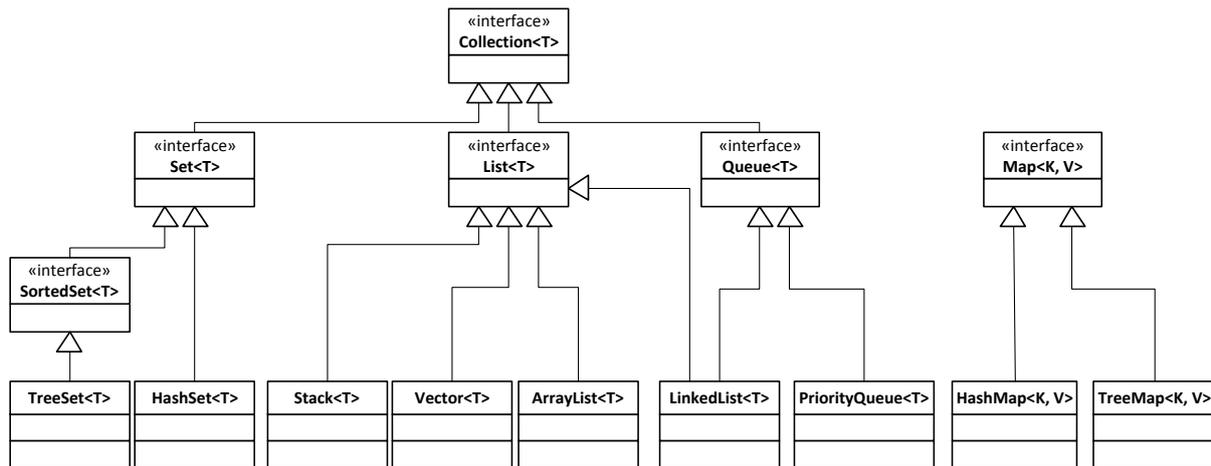


Abbildung 1: Ausschnitt aus dem Java Collections Framework (JCF)

Aufgabe 1: Java Collection Framework (JCF)

Das Java Collections Framework stellt viele häufig benötigte Datenstrukturen zur Verfügung, so dass Sie das Rad nicht neu erfinden müssen. Sie finden eine gute Einführung zu den Collections unter <http://docs.oracle.com/javase/tutorial/collections/>. In dieser Aufgabe geht es darum für verschiedene Anwendungsbeispiele jeweils die richtige Datenstruktur auszuwählen.

In dieser Aufgabe geht es um die Verwaltung von Filmen.¹ Auf der Webseite zur Vorlesung finden Sie vorbereitete Dateien. Sie sollen jeweils die effizienteste Datenstruktur wählen, wobei wir uns auf `PriorityQueue`, `HashMap`, `TreeMap` und `ArrayList` beschränken. Begründen Sie, welche Datenstruktur Sie gewählt haben und welche Nachteile die anderen gehabt hätten. Vervollständigen Sie den Java-Code von der Website entsprechend.

Tipp: Datenstrukturen wie die `PriorityQueue` und die `TreeMap` müssen ihre Elemente vergleichen können. Wenn also eigene Klassen in diesen Datenstrukturen verwendet werden (wie zum Beispiel `Movie`) dann muss angegeben werden wie Objekte dieser Klasse verglichen werden können. Dafür werden in Java sogenannte Comparatoren verwendet: <https://docs.oracle.com/javase/7/docs/api/java/util/Comparator.html>.

- a) Die zu implementierende Klasse `MovieRangeSearch` soll Filme nach ihrem Erscheinungsjahr verwalten. Die Methode `getMoviesBetweenYears(int fromYear, int toYear)` soll alle Filme zurückgeben, die in einem bestimmten Zeitraum erschienen sind.

¹Basierend auf der Liste der Top 250 Filme von [imdb.com](http://www.imdb.com/chart/top): <http://www.imdb.com/chart/top>

- b) Die Klasse `MovieTitleSearch` verwaltet Filme nach ihrem Namen. Dabei gibt die Methode `getMovieWithTitle(String title)` den Film mit dem angegebenen Titel zurück.
- c) Die Klasse `MovieVoteSearch` verwaltet Filme nach abgegebenen Bewertungen. Dabei soll die Methode `getMovieWithFewestVotes()` den Film mit den wenigsten Stimmen zurückgeben.
- d) Die Klasse `MovieRankSearch` verwaltet Filme anhand ihrer Platzierung in der Top 250 Liste von IMDB. Die Methode `getMovieForRank(int rank)` bestimmt für eine Position den Film.

Lösung 1

Die am besten passende Datenstruktur ist jeweils in **rot** gekennzeichnet.

- a) **TreeMap**. Die passende Datenstruktur für die `MovieRangeSearch` ist die `TreeMap`. Diese erlaubt effiziente Bereichszugriffe über die Methode `submap(from, to)`. Um eine Kollision bei Filmen die im gleichen Jahr produziert wurden zu vermeiden, sollte die `TreeMap` nicht auf einzelne Filme abbilden sondern auf eine Liste mit Filmen.

ArrayList, HashMap, PriorityQueue. Die anderen drei Datenstrukturen sind ungeeignet, da man bei ihnen für eine Bereichssuche jeden Film einzeln überprüfen muss. Da man jeden Film betrachten muss, benötigt dies bei n Filmen ungefähr n Operationen. In der Landau-Notation schreibt man dies als $\mathcal{O}(n)$.

- b) **HashMap**. Zur `MovieTitleSearch` passt am besten die Datenstruktur `HashMap`. Der Punktzugriff über die Methode `get()` benötigt nur konstante Zeit und ist somit unabhängig von der Anzahl der Filme. In Landau-Notation schreibt man dies als $\mathcal{O}(1)$.

TreeMap. Die `TreeMap` benötigt aufgrund der Baumeigenschaft bei n Filmen ungefähr $\log(n)$ Operationen für den Punktzugriff (Landau-Notation: $\mathcal{O}(\log(n))$). Der Grund dafür: Es muss von der Wurzel ein Pfad bis zum entsprechenden Blatt traversiert werden und der Baum hat eine Höhe von ungefähr $\log(n)$.

ArrayList, PriorityQueue. Bei der `ArrayList` und der `PriorityQueue` muss man über alle Elemente iterieren, um den passenden Film zu finden. Entsprechend ist die Laufzeit linear in der Anzahl der Filme (Landau-Notation: $\mathcal{O}(n)$).

- c) **PriorityQueue**. Die `PriorityQueue` verwaltet ihre Elemente nach Priorität. Dabei ist das Element mit der höchsten Priorität in der Wurzel verfügbar. Dies können wir ausnutzen, indem wir „weniger Stimmen“ als höhere Priorität definieren. Folglich ist somit in der Wurzel der `PriorityQueue` der Film mit den wenigsten Stimmen gespeichert und kann in konstanter Zeit abgefragt werden ($\mathcal{O}(1)$).

ArrayList, HashMap. Bei einer `HashMap` und einer `ArrayList` muss man über alle Filme iterieren, um manuell den mit den wenigsten Stimmen zu finden. Dies benötigt daher lineare Zeit ($\mathcal{O}(n)$).

TreeMap. Bei der `TreeMap` könnte man zumindest auf eine logarithmische Laufzeit hoffen, indem man die Anzahl an Stimmen als Schlüssel verwendet. Dies ist aber nicht möglich, da man keine zwei Elemente mit dem gleichen Schlüssel einfügen kann und es sehr wohl mehrere Filme mit der gleichen Anzahl Stimmen geben kann. Entsprechend müsste man ein anderes (eindeutiges) Attribut der Filmklasse als Schlüssel nehmen (z.B. den Titel wie bei Aufgabenteil a), das einem bei der Suche nicht hilft. Die Laufzeit ist dann wie bei `ArrayList` und `HashMap` linear in der Anzahl der Filme ($\mathcal{O}(n)$).

d) **ArrayList**. Die `ArrayList` ist hier am besten geeignet, da die Schlüssel dicht gepackt sind und sie die Operationen `get()` und `set()` mit konstanter Laufzeit ermöglicht ($\mathcal{O}(1)$). Im Gegensatz zu einem einfachen Array, erlaubt die `ArrayList` eine dynamische Größenveränderung. Man könnte Sie also problemlos vergrößern, wenn man statt den Top 250 die Top 500 Filme speichern wollte.

HashMap. Die `HashMap` bietet ebenfalls Punktzugriffe in $\mathcal{O}(1)$ und ist somit auch geeignet.

PriorityQueue. Die `PriorityQueue` erlaubt keinen Punktzugriff auf einen Schlüssel.

TreeMap. Die `TreeMap` benötigt logarithmische Zeit für einen Punktzugriff ($\mathcal{O}(\log(n))$).

Aufgabe 2: Generics

In dieser Aufgabe wollen wir einen generischen binären Suchbaum implementieren. Falls Sie sich noch unsicher mit Generics fühlen, können sie zunächst einen binären Suchbaum implementieren der lediglich Integer unterstützt und diesen dann in einem zweiten Schritt erweitern. Testen Sie Ihre Implementierung in jedem Fall mit geeigneten Beispielen.

1. Implementieren Sie einen binären Suchbaum,² der beliebige Elemente enthalten kann. Jeder Knoten sollte je einen Zeiger auf das linke und das rechte Kind haben. Der Baum sollte Methoden zum Einfügen und Suchen von Elementen anbieten. Wenn Sie eine Herausforderung suchen, können Sie auch noch Löschen implementieren.
2. Welches Problem gibt es, wenn man nacheinander die Zahlen $1, 2, 3, \dots, 1.000.000$ in aufsteigender Reihenfolge einfügt?

Lösung 2

Die folgende Klasse implementiert einen binären Suchbaum.

```
1 import java.util.Arrays;
2 import java.util.Collection;
3
4 public class MyBinaryTree<T extends Comparable<T>> {
5
6     private TreeNode<T> root = null;
7
8     public void insert(T element) {
9         if (root == null) {
10             root = new TreeNode<>(element);
11         } else {
12             root.insert(element);
13         }
14     }
15
16     public T search(T element) {
17         if (root == null) {
18             return null;
19         } else {
```

²Falls Ihnen entfallen ist, was ein Binärbaum ist: http://de.wikipedia.org/wiki/Binärer_Suchbaum

```

20     return root.search(element);
21 }
22 }
23
24 public void delete(T element) {
25     // Empty tree
26     if (root == null) {
27         return;
28     }
29
30     // If root itself contains element
31     if (root.key.equals(element)) {
32         // Case 1: no children
33         if ((root.leftChild == null) && (root.rightChild == null)) {
34             root = null;
35         }
36         // Case 2+3: 1 child
37         else if ((root.leftChild != null) && (root.rightChild == null))
38             {
39             root = root.leftChild;
40             root.parent = null;
41         } else if ((root.leftChild == null) && (root.rightChild != null
42             )) {
43             root = root.rightChild;
44             root.parent = null;
45         }
46         // Case 4: two children
47         else {
48             root.leftChild.insert(root.rightChild);
49             root = root.leftChild;
50             root.parent = null;
51         }
52     }
53     return;
54 }
55
56 // At this point ensured that root does not contain key, delegate
57 // to root node
58 root.delete(element);
59 }
60
61 private static class TreeNode<T extends Comparable<T>> {
62     private final T key;
63     private TreeNode<T> leftChild;
64     private TreeNode<T> rightChild;
65     private TreeNode<T> parent;
66
67     TreeNode(T key) {
68         this(key, null);

```

```

65     }
66
67     TreeNode(T key, TreeNode<T> parent) {
68         this.key = key;
69         this.parent = parent;
70         this.leftChild = null;
71         this.rightChild = null;
72     }
73
74     public void insert(T element) {
75         insert(new TreeNode<T>(element, null));
76     }
77
78     public void insert(TreeNode<T> node) {
79         if (key.compareTo(node.key) > 0) {
80             if (leftChild != null) {
81                 leftChild.insert(node);
82             } else {
83                 node.parent = this;
84                 leftChild = node;
85             }
86         }
87         if (key.compareTo(node.key) < 0) {
88             if (rightChild != null) {
89                 rightChild.insert(node);
90             } else {
91                 node.parent = this;
92                 rightChild = node;
93             }
94         }
95     }
96
97     public T search(T element) {
98         if (key.equals(element)) {
99             return element;
100         }
101
102         if (key.compareTo(element) > 0 && leftChild != null) {
103             return leftChild.search(element);
104         } else if (key.compareTo(element) < 0 && rightChild != null) {
105             return rightChild.search(element);
106         } else {
107             return null;
108         }
109     }
110
111     private void replaceChild(TreeNode<T> child, TreeNode<T>
        replacement) {

```

```

112     if (leftChild == child) {
113         leftChild = replacement;
114     } else if (rightChild == child) {
115         rightChild = replacement;
116     }
117 }
118
119 public void delete(T element) {
120     // Found node
121     if (key.equals(element)) {
122         // Fall 1: no children
123         if ((leftChild == null) && (rightChild == null)) {
124             parent.replaceChild(this, null);
125         }
126         // Fall 2+3: 1 child
127         else if ((leftChild != null) && (rightChild == null)) {
128             parent.replaceChild(this, leftChild);
129         } else if ((leftChild == null) && (rightChild != null)) {
130             parent.replaceChild(this, rightChild);
131         }
132         // Fall 4: two children
133         else {
134             parent.replaceChild(this, leftChild); // cut out, connect
135             // left child to parent; insert right child later
136             leftChild.insert(rightChild); // eventually results in
137             // strong unbalance
138         }
139         return;
140     }
141     // Traverse downwards the tree
142     if (key.compareTo(element) > 0 && leftChild != null) {
143         leftChild.delete(element);
144     } else if (key.compareTo(element) < 0 && rightChild != null) {
145         rightChild.delete(element);
146     }
147 }
148 }

```

Aufgabe 3: For Each Loops

Das folgende Programm soll alle Primzahlen im Intervall [0, 10.000] bestimmen.³ Leider wirft das Programm bei der Ausführung eine *Exception*. Wo und warum? Was ist eine mögliche Lösung?

```

1 import java.util.ArrayList;
2

```

³Basierend auf dem Sieb des Eratosthenes: http://de.wikipedia.org/wiki/Sieb_des_Eratosthenes

```

3 public class Eratosthenes {
4
5     public static void main(String[] args) {
6         // Zahlen 2..10000 einfüegen
7         ArrayList<Integer> list = new ArrayList<Integer>();
8         for (int i = 2; i <= 10000; i++) {
9             list.add(i);
10        }
11        for (Integer zahl : list) {
12            // Probiere alle vorherigen Primzahlen als Teiler
13            for (Integer teiler : list) {
14                // Wenn der zu prüfende Teiler schon grösser als
15                // die Quadratwurzel ist, muss es eine Primzahl sein
16                if (teiler > Math.sqrt(zahl)) break;
17                // Zahlen mit Teiler sind keine Primzahlen und
18                // werden daher hier entfernt
19                if (zahl % teiler == 0) {
20                    list.remove(zahl);
21                    break;
22                }
23            }
24        }
25        // Alle verbliebenen Zahlen sind Primzahlen
26        for (Integer primzahl : list)
27            System.out.println(primzahl);
28    }
29 }

```

Lösung 3

In Zeile 21 wird ein Element aus der Liste gelöscht, über die wir in diesem Moment mit einer for-each-Schleife in Zeile 11 iterieren. Durch die Veränderung der Datenstruktur über die wir gleichzeitig iterieren, kann es zu Inkonsistenzen kommen, so dass beispielsweise ein gelöscht Element besucht würde. Erfreulicherweise gibt es im Java Collections Framework das Konzept des Iterator-Objekts mit dem wir das Problem leicht lösen können. Der Iterator erlaubt es ähnlich der for-each-Schleife eine Datenstruktur zu durchlaufen. Zusätzlich kann der Iterator durch sein Wissen über die Iteration, die er gerade verwaltet, eine `remove()`-Methode anbieten, die das zuletzt von `next()` zurück gegebenen Elements löscht – ohne dass es dabei zu Inkonsistenzen kommt. Entsprechend verwenden wir in der Lösung in Zeile 12 statt einer for-each-Schleife nun den Iterator und löschen in Zeile 24 die aktuelle Zahl ebenfalls über den Iterator aus der Liste.

```

1 import java.util.ArrayList;
2 import java.util.ListIterator;
3
4 public class EratosthenesLoesung {
5     public static void main(String[] args) {
6         // Zahlen 2..10000 einfüegen
7         ArrayList<Integer> list = new ArrayList<Integer>();

```

```

8   for (int i = 2; i <= 10000; i++) {
9       list.add(i);
10  }
11
12  ListIterator<Integer> iterator = list.listIterator();
13  while (iterator.hasNext()) {
14      Integer zahl = iterator.next();
15      // Probiere alle vorherigen Primzahlen als Teiler
16      for (Integer teiler : list) {
17          // Wenn der zu pruefende Teiler schon groesser als
18          // die Quadratwurzel ist, muss es eine Primzahl sein
19          if (teiler > Math.sqrt(zahl))
20              break;
21          // Zahlen mit Teiler sind keine Primzahlen und
22          // werden daher hier entfernt
23          if (zahl % teiler == 0) {
24              iterator.remove();
25              break;
26          }
27      }
28  }
29
30  // Alle verbliebenen Zahlen sind Primzahlen
31  for (Integer primzahl : list)
32      System.out.println(primzahl);
33  }
34  }

```