TU München, Fakultät für Informatik
Lehrstuhl III: Datenbanksysteme
Prof. Alfons Kemper, Ph.D.

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

Christoph Anneser (anneser@in.tum.de)
http://db.in.tum.de/teaching/ss23/ei2/

Lösungen zu Blatt Nr. 5

Dieses Blatt wird am Montag, den 05.06.2023 besprochen.

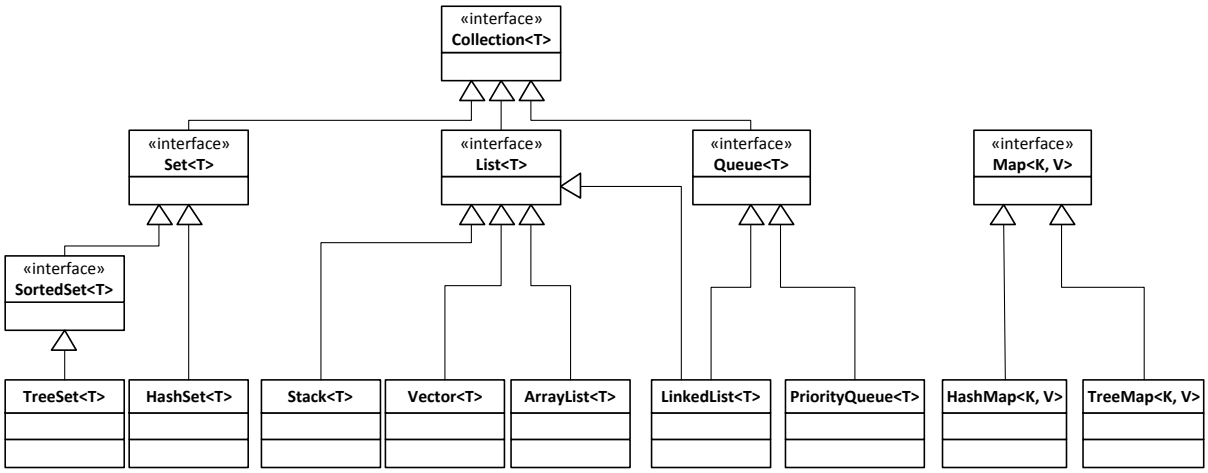

Abbildung 1: Ausschnitt aus dem Java Collections Framework (JCF)

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. Vorvollständigen 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 angeben 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`.

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

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.

b) Die Klasse `MovieTitleSearch` verwaltet Filme nach ihrem Namen. Dabei gibt die Methode `getMovieWithTitle(String title)` den Film mit dem angegebenem 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)` gibt für eine Position den Film zurück.

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.

ArraList, 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 ($\mathcal{O}(1)$).

ArraList, 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: Telefonbuch ⌨

Laden Sie sich die Datei telefonbuch.java von der Vorlesungswebsite herunter. In der `main`-Methode wird ein Telefonbuch mit 10M Einträgen angelegt.

Sie sollen nun auf zwei Arten für eine gesuchte Nummer den Namen herausfinden.

1. Iterieren Sie durch das ganze Telefonbuch, bis Sie den passenden Eintrag finden.
2. Erstellen Sie vor der Abfrage eine HashMap oder TreeMap, die Telefonnummern als Schlüssel speichert und die Namen als Werte.

Vergleichen Sie die Laufzeiten beider Ansätze miteinander. Sind TreeMaps oder HashMaps schneller? Wie lange dauert die Erstellung der Maps?

Aufgabe 3: 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 zweitern Schritt erweitern. Testen Sie Ihre Implmentierung in jedem Fall mit geeigeneten 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 $1, 2, 3, ..., 1.000.000$ in aufsteigender Reihenfolge einfügt?

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;
```

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

```
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 {
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        root = root.leftChild;
39        root.parent = null;
40      } else if ((root.leftChild == null) && (root.rightChild != null)) {
41        root = root.rightChild;
42        root.parent = null;
43      }
44      // Case 4: two children
45      else {
46        root.leftChild.insert(root.rightChild);
47        root = root.leftChild;
48        root.parent = null;
49      }
50      return;
51    }
52
53    // At this point ensured that root does not contain key, delegate to root node
54    root.delete(element);
```

```
55    }
56
57    private static class TreeNode<T extends Comparable<T>> {
58      private final T key;
59      private TreeNode<T> leftChild;
60      private TreeNode<T> rightChild;
61      private TreeNode<T> parent;
62
63      TreeNode(T key) {
64        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 left child to parent; inse
135             leftChild.insert(rightChild);// eventually results in strong unbalance
136           }
137           return;
138         }
139
140         // Traverse downwards the tree
141         if (key.compareTo(element) > 0 && leftChild != null) {
142           leftChild.delete(element);
143         } else if (key.compareTo(element) < 0 && rightChild != null) {
144           rightChild.delete(element);
145         }
146       }
147     }
148   }
```

Aufgabe 4: 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
3  public class Eratosthenes {
4
5     public static void main(String[] args) {
6        // Zahlen 2..10000 einfuegen
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 pruefende Teiler schon groesser 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 }
```

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 gelöschtes 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] Basierend auf dem Sieb des Eratosthenes: `http://de.wikipedia.org/wiki/Sieb_des_Eratosthenes`

```java

public class EratosthenesLoesung {
  public static void main(String[] args) {
    // Zahlen 2..10000 einfuegen
    ArrayList<Integer> list = new ArrayList<Integer>();
    for (int i = 2; i <= 10000; i++) {
      list.add(i);
    }

    ListIterator<Integer> iterator = list.listIterator();
    while (iterator.hasNext()) {
      Integer zahl = iterator.next();
      // Probiere alle vorherigen Primzahlen als Teiler
      for (Integer teiler : list) {
        // Wenn der zu pruefende Teiler schon groesser als
        // die Quadratwurzel ist, muss es eine Primzahl sein
        if (teiler > Math.sqrt(zahl))
          break;
        // Zahlen mit Teiler sind keine Primzahlen und
        // werden daher hier entfernt
        if (zahl % teiler == 0) {
          iterator.remove();
          break;
        }
      }
    }

    // Alle verbliebenen Zahlen sind Primzahlen
    for (Integer primzahl : list)
      System.out.println(primzahl);
  }
}
```

Aufgabe 5: Arraylist ⌨

Implementieren Sie eine generische Liste, die ihre Elemente in dynamischen Arrays speichert. Nutzen Sie hierfür die folgende Vorlage, die Sie auf gitlab.db.in.tum.de finden. Klicken Sie hierfür auf das Download-Symbol und laden Sie sich das Projekt als Zip-Datei herunter. Entpacken Sie die Datei und öffnen Sie sie in Ihrer IDE. Machen Sie sich dann mit den Dateien und der Struktur des Projekts vertraut

Weitere Informationen erhalten Sie in der Zentralübung und unter diesem Wikipedia-Eintrag: Dynamic Array.

Implementieren Sie nun die mit entsprechenden Methoden in der Klasse `ArrayList.java`.

Tipp 1: Erarbeiten Sie sich vor der Implementierung der Methoden ein Grundverständnis, wie die Liste funktionieren soll. Erstellen Sie z.B. Skizzen von Listen, wie Sie nach bestimmten

Operationen im Speicher aussehen könnte).

Unter gitlab.db.in.tum.de finden Sie die Lösung.

Tipp 2: Benutzen Sie für Vergleiche stets die `equals`*-Methode. Eine generische Liste kann z.B. auch Punkt-Objekte speichern.*

Unter gitlab.db.in.tum.de finden Sie die Lösung.