

252-0027-00: Einführung in die Programmierung

Übungsblatt 9

Abgabe: 26. November 2024, 23:59

Checken Sie mit Eclipse wie bisher die neue Übungsvorlage aus. Importieren Sie beide Eclipse-Projekte (das Projekt für den Bonus und das Projekt für die restlichen Aufgaben).

Aufgabe 1: Square Grid

In dieser Aufgabe betrachten wir gerichtete Graphen, wobei es für jeden Knoten g höchstens zwei gerichtete Kanten von g zu anderen Knoten f, h geben kann (f, g, h können gleich sein). Wir unterscheiden dabei zwischen der rechten und der unteren Kante (und damit dem rechten und dem unteren Knoten).

Die Klasse `Node` repräsentiert einen Knoten in einem solchen Graphen. Die Methode `Node.getRight()` (bzw. `Node.getDown()`) gibt den rechten Knoten (bzw. unteren Knoten) zurück (als `Node`-Objekt). Wenn der rechte Knoten von n_0 nicht existiert, dann gibt `Node.getRight()` `null` zurück (analog für den unteren Knoten). Die Methode `Node.setRight(Node r)` (bzw. `Node.setDown(Node d)`) setzt den rechten (bzw. unteren) Knoten.

Das Ziel der Aufgabe ist, einen von einem `Node`-Objekt definierten Graphen zu analysieren. Konkret geht es darum, die Grösse des grössten quadratischen Gitters in dem Graphen zu bestimmen. Der Graph ist gegeben als eine Referenz auf das `Node`-Objekt des Ursprungsknotens des Graphs.

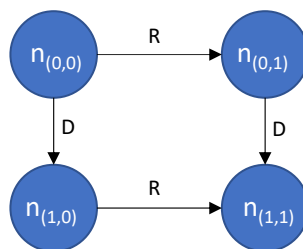


Abbildung 1: Graph als perfektes quadratisches Gitter

Abbildung 1 zeigt ein Beispiel für einen Graphen mit Ursprungsknoten $n_{(0,0)}$ und Koordinaten $\{(0,0), (0,1), (1,0), (1,1)\}$, wobei jeweils R der rechten Kante und D der unteren Kante eines Knotens entspricht. Abbildung 2 zeigt zwei andere Graphen.

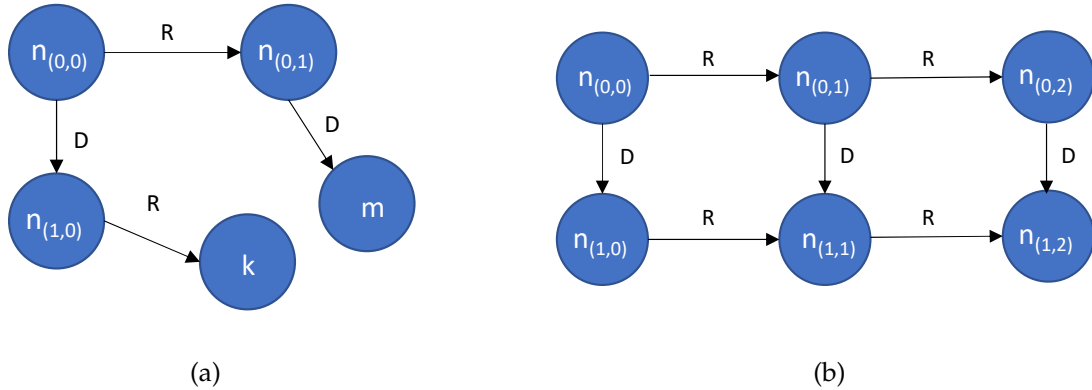


Abbildung 2: Graphen mit quadratischen Gittern als Teilgraphen

Ein Teilgraph G von einem Graphen G' (wie oben definiert) mit Ursprungsknoten u definiert ein quadratisches Gitter mit Ursprungsknoten u und Koordinaten K (wobei $K \subseteq \mathbb{N}_{\geq 0} \times \mathbb{N}_{\geq 0}$; das heisst, Koordinaten haben keine negativen Komponenten), so dass folgende Bedingungen gelten:

- Jeder Knoten in G ist auch in G' und jede Kante in G ist auch in G' .
- Jeder Knoten in G ist über die gerichteten Kanten vom Ursprungsknoten u erreichbar.
- Es gibt gleich viele Knoten in G wie Koordinaten in K . Ausserdem wird jede Koordinate $(i, j) \in K$ durch genau einen einzigartigen Knoten $n_{(i,j)}$ in G repräsentiert. Das heisst, wenn $\{(i, j), (i', j')\} \subseteq K$ und $(i, j) \neq (i', j')$, dann gilt $n_{(i,j)} \neq n_{(i',j')}$.
- Sei $(i, j) \in K$. Wenn der untere Knoten von $n_{(i,j)}$ existiert, dann gilt $(i + 1, j) \in K$ und der untere Knoten von $n_{(i,j)}$ ist gegeben durch $n_{(i+1,j)}$. Wenn der rechte Knoten von $n_{(i,j)}$ existiert, dann gilt $(i, j + 1) \in K$ und der rechte Knoten von $n_{(i,j)}$ ist gegeben durch $n_{(i,j+1)}$.
- Sei n die Grösse des Quadrats ($n \geq 1$). Dann ist K gegeben durch $K = \{(i, j) \mid 0 \leq i < n, 0 \leq j < n\}$.

Implementieren Sie die Methode `SquareGrid.analyzeSquareGrid(Node origin)`, welche die Grösse des *grössten* quadratischen Gitters in dem Graphen mit Ursprungsknoten `origin` (welches `origin` als Ursprungsknoten hat) zurückgibt. Sie können davon ausgehen, dass `origin` nicht `null` ist (das bedeutet im Umkehrschluss, dass das kleinste Quadrat immer Grösse 1 hat).

In Abbildung 1 hat für den Ursprungsknoten $n_{(0,0)}$ das grösste Quadrat Grösse 2. In Abbildung 2a hat für den Ursprungsknoten $n_{(0,0)}$ das grösste Quadrat Grösse 1, und in Abbildung 2b hat für den Ursprungsknoten $n_{(0,0)}$ das grösste Quadrat ebenfalls Grösse 2 (das grösste Quadrat für den Ursprungsknoten $n_{(0,1)}$ hat hier ebenfalls Grösse 2; für alle anderen Ursprungsknoten in dem Graphen hat das grösste Quadrat Grösse 1).

In der Klasse `Main` finden Sie eine `main`-Methode, welche Sie verwenden können, um Ihre Implementierung zu testen. In der Datei `"SquareGridTest.java"` finden Sie ausserdem einige Tests.

Aufgabe 2: Umkehrung

In einem vorherigen Übungsblatt haben Sie eine Linked List für Integers implementiert. In dieser Aufgabe fügen Sie dieser `LinkedList` eine weitere Methode hinzu, welche die Liste umkehrt. Eine Liste gilt als umgekehrt, wenn für jedes Paar von Nodes *a* und *b*, für welche zuvor *a* der Nachfolger von *b* war, *b* nun der Nachfolger von *a* ist. Zusätzlich entspricht nach der Umkehrung der erste Node der neuen Liste dem letzten Node der ursprünglichen Liste (und umgekehrt).

Vervollständigen Sie die Methode `reverse()` in der Klasse `LinkedList`. Die Methode soll, wie oben definiert, die Liste umkehren. Achten Sie darauf, dass Sie wirklich die Reihenfolge der Nodes selbst umkehren. Es reicht nicht aus, die Reihenfolge der enthaltenen `int`-Werte umzukehren. Es müssen auch in der umgekehrten Liste dieselben Instanzen von `IntNodes` wie in der ursprünglichen Liste verwendet werden. Erstellen Sie also *keine* neuen `IntNodes` mit `new IntNode()`. In der Datei `UmkehrungTest.java` finden Sie einen einfachen Test.

Aufgabe 3: Künstliche Intelligenz für das Ratespiel

In Übung 5 implementierten Sie ein Spiel, in welchem der Computer ein Wort auswählt und der Spieler dieses erraten muss. Dort war der Spieler der Benutzer des Programms. In dieser Aufgabe sollen Sie verschiedene "künstliche" Spieler entwickeln. Das heisst, anstelle des Menschen, der über die Konsole Tipps eingibt, werden die Tipps von (mehr oder weniger "intelligenten") Programmen abgegeben. Ihr Ziel ist es, einen künstlichen Spieler zu entwickeln, der über mehrere Spiele hinweg die Wörter in so wenig Versuchen wie möglich errät.

Die Übungsvorlage enthält bereits den Code für das Ratespiel. Gegenüber Übung 5 ist dieser nun in verschiedene Klassen aufgeteilt. Die drei Hauptklassen sind `RateSpiel`, `Computer` und `Spieler`. Die Klasse `RateSpielApp` enthält eine `main`-Methode, welche das Spiel aufsetzt und durchführt. Durch die Aufteilung ist es möglich, mittels Vererbung Spieler mit unterschiedlichem Verhalten zu schreiben. Die Klasse `Spieler` enthält nämlich nur die Deklarationen der benötigten Methoden, aber keine (sinnvolle) Funktionalität. Subklassen von `Spieler` überschreiben diese Methoden und definieren damit das Verhalten eines Spielers.

Ein konkreter Spieler ist ebenfalls schon in der Vorlage vorhanden: der `KonsolenSpieler`. Dieser besitzt allerdings keine eigene "Intelligenz", sondern holt sich die Tipps über die Konsole vom Benutzer. Ein `RateSpiel` mit einem `KonsolenSpieler` verhält sich also so wie das Spiel in Übung 5. Starten Sie die `RateSpielApp` und überzeugen Sie sich selbst¹.

- a) Schreiben Sie als erstes eine Klasse `ZufallsWortSpieler`, welche einen Spieler implementiert, der in jeder Runde zufällig ein Wort aus der Liste der verwendeten Wörter tippt. Die Klasse soll von `Spieler` erben und die benötigten Methoden überschreiben.

In der `neuesSpiel`-Methode, welche immer zu Beginn eines Spiels vom `RateSpiel` aufgerufen wird, soll sich der Spieler das Array der im Spiel verwendeten Wörter merken. Speichern Sie eine Referenz dazu in ein `woerter`-Feld. Mit der `gibTipp`-Methode gibt der Spieler seinen nächsten Tipp ab. Überschreiben Sie diese Methode, so dass sie einen zufälligen Index für das `woerter`-Array (siehe `Random.nextInt(int)`) erzeugt und das entsprechende Wort zurück gibt. Das Generieren des Index sollen Sie in einer separaten Methode `zufallsWortIndex()` implementieren. (Sie sehen später, weshalb.) Überschreiben Sie ausserdem `name()`.

¹Beachten Sie, dass die Wörter-Datei jetzt 500 Wörter enthält!

Ändern Sie jetzt das RateSpielApp-Programm so ab, dass statt einem KonsolenSpieler ein ZufallsWortSpieler am Spiel teilnimmt. Erhöhen Sie ausserdem die Anzahl der Spiele, die durchgeführt werden, indem Sie das Argument der nSpiele-Methode von 1 z.B. auf 1000 ändern. Sie sollten ungefähr folgende Ausgabe erhalten:

```
...
Spiel 999
Spiel 1000
Zufalls-Wort-Spieler hat durchschnittlich 495.138 Versuche benötigt.
```

- b) Erstellen Sie einen zweiten Spieler, ZufallsWortSpielerMitGedaechtnis, der sich in jedem Spiel merkt, welche Wörter er schon ausprobiert hat. Da dieser einige Ähnlichkeit zum ZufallsWortSpieler hat, sollen Sie ihn als Subklasse von ZufallsWortSpieler entwerfen.

Das Gedächtnis des Spielers können Sie als boolean[]-Feld ausdrücken, welches für jedes mögliche Wort angibt, ob dieses Wort schon ausprobiert wurde. Dieses Array sollte zu Beginn jedes Spiels neu initialisiert werden. Überschreiben Sie dazu die neuesSpiel-Methode vom ZufallsWortSpieler. **Vorsicht:** der Code in der neuesSpiel-Methode von ZufallsWortSpieler sollte trotzdem ausgeführt werden, denn da merkt er sich ja die Liste der Wörter! Fügen Sie deshalb einen entsprechenden super-Methodenaufruf hinzu.

Da Sie in a)) das Erzeugen des Wort-Index in einer separaten zufallsWortIndex-Methode implementierten, können Sie nun diese Methode überschreiben, um das Verhalten von gibTipp() dieses Spielers zu ändern. (Dafür darf die Sichtbarkeit von zufallsWortIndex() in

ZufallsWortSpieler nicht private sein). Überschreiben Sie sie so, dass sie das Gedächtnis des Spielers einbezieht. Sie müssen dafür einen zufälligen Index generieren, welcher nicht schon verwendet wurde.

Überschreiben Sie auch name() und ändern Sie dann RateSpielApp erneut, so dass beide Zufalls-Spieler spielen. Die Ausgabe sollte jetzt etwa so aussehen:

```
...
Zufalls-Wort-Spieler hat durchschnittlich 492.513 Versuche benötigt.
Zufalls-Wort-Spieler mit Gedächtnis hat durchschnittlich 242.493 Versuche benötigt.
```

- c) Die beiden Zufalls-Spieler sind noch nicht wirklich "intelligent". Der Grund ist, dass sie gar keinen Nutzen aus den Hinweisen des Computers ("enthält", "enthält nicht", usw.) ziehen. Schreiben Sie deshalb einen (oder mehrere) Spieler, welche bessere Tipps abgeben und die Hinweise nutzen, um die Menge der noch möglichen Wörter einzuschränken. Dazu müssen Sie die bekommeHinweis-Methode von Spieler überschreiben.

Wenn Sie verschiedene Ideen ausprobieren wollen, schreiben Sie verschiedene Spieler-Subklassen und vergleichen Sie sie mithilfe der RateSpielApp. Schauen Sie sich auch den RateSpiel-Konstruktor an, welcher zwei hilfreiche Parameter zur Verfügung stellt.

Wie schlägt sich Ihr bester Spieler im Vergleich zu den Spielern Ihrer Mitstudierenden?

Tipps:

- Beginnen Sie einfach – z.B. mit einem Spieler, der zuerst alle Buchstaben des Alphabets tippt und dann die noch möglichen Wörter durchprobiert.
- Heuristiken (z.B. 'e' kommt öfter vor als 'x') sind hilfreich.
- Überlegen Sie sich, ob es so etwas wie eine "optimale" Strategie gibt.

Aufgabe 4: Klassenrätsel

In dieser Aufgabe sollen Sie zeigen, dass Sie mit Klassen und Vererbung umgehen können. Im Anhang [A](#) finden Sie ein Programm, welches Instanzen von Klassen erstellt und Methoden aufruft. Das Programm macht nichts Sinnvolles und dient nur dem Testen Ihrer Fähigkeiten. In Anhang [B](#) befinden sich die verwendeten Klassen, jedoch sind die Klassen noch nicht vollständig. Bei manchen der Klassen fehlt noch die `extends`-Klausel, welche angibt, dass eine Klasse von einer anderen Klasse erbt. Ihre Aufgabe ist es, die nötigen `extends`-Klauseln hinzuzufügen, so dass alles kompiliert und so dass die Ausgabe des Programms von Anhang [A](#) am Ende so aussieht wie im Anhang [C](#) gezeigt.

Der Code von Anhang [A](#) and Anhang [B](#) befindet sich in Ihrem `src`-Ordner. Zusätzlich enthält `KlassenTest.java` einen Unit-Test, welcher prüft, ob die Ausgabe des Programms dem Output aus Anhang [C](#) entspricht. Beachten Sie, dass Sie für diese Aufgabe **ausschliesslich** `extends`-Klauseln hinzufügen (diese kann es nur an den grauen Boxen aus Anhang [B](#) geben), kein anderer Code darf verändert werden.

Tipp: Lösen Sie die Aufgabe zuerst auf Papier, ohne die Hilfe von Eclipse. Sobald Sie herausgefunden haben, welche Klassen von welchen Klassen erben, testen Sie Ihre Lösung in Eclipse. Dies hilft Ihnen, Ihr Wissen über Vererbung zu testen. In der Vergangenheit wurden ähnliche Aufgaben im schriftlichen Teil der Prüfung gestellt.

Aufgabe 5: Warteschlangen-Server (Bonus!)

Achtung: Diese Aufgabe gibt Bonuspunkte (siehe "Leistungskontrolle" im www.vvz.ethz.ch). Die Aufgabe muss eigenhändig und alleine gelöst werden. Die Abgabe erfolgt wie gewohnt per Push in Ihr Git-Repository auf dem ETH-Server. Verbindlich ist der letzte Push vor dem Abgabetermin. Auch wenn Sie vor der Deadline committen, aber nach der Deadline pushen, gilt dies als eine zu späte Abgabe. Bitte lesen Sie zusätzlich [die allgemeinen Regeln](#).

Damit viele Benutzer gleichzeitig eine Datei lesen oder bearbeiten können, verwendet der Dateiserver eine Warteschlange, um die Dateifreigabe zu koordinieren. Mehrere Benutzer können gleichzeitig dieselbe Datei lesen. Während ein Benutzer in eine Datei schreibt, können jedoch andere Benutzer nicht gleichzeitig die Datei lesen oder bearbeiten. Die Warteschlange ist ein Array fester Größe, das maximal N Warteschlangen-Einträge speichert. Der Server koordiniert die Dateifreigabe für bis zu 100 Dateien. Bei jedem Aufruf der `add()`-Methode und `pop()`-Methode gilt also $0 \leq \text{fileID} < 100$.

Tipp 1: Lesen Sie jeweils die Aufgabenstellung inklusive aller Teilaufgaben sorgfältig durch, bevor sie mit der Implementation der ersten Teilaufgabe beginnen.

Tipp 2: Es kann nützlich sein eine separate Klasse für die Entries zu erstellen, welche alle Eigenschaften eines Elements in der Warteschlange als Attribute enthält.

Teilaufgabe 1:

Implementieren Sie den Konstruktor `WaitQueueServer(int N)`, welcher eine Warteschlange für N Einträge erstellt. Implementieren Sie dann die Methode `Response add(int fileID, char userID, boolean readOnly)`. Sie fügt, wenn möglich, den neuen Eintrag (gegeben durch die drei Methodenargumente) an den ersten freien Platz im Warteschlangen-Array ein. Wenn das Warteschlangen-Array voll ist, wird `null` zurückgegeben. Anderenfalls soll eine Instanz der Klasse `Response` zurückgegeben werden. Die Klasse `Response` hat zwei Felder: `head` und `tail`. Diese sollen wie folgt gesetzt werden:

`head` bezeichnet den Index des ältesten Eintrags zur Datei `fileID`.

`tail` bezeichnet den Index des neu eingefügten Eintrags.

Für Index i jedes Eintrags gilt dabei immer $0 \leq i < N$.

Beispielsweise bei einem Warteschlangen-Array von Länge 8 geben die `add()`-Methodenaufrufe die folgenden Rückgabewerte, wenn sie in dieser Reihenfolge ausgeführt werden (Abbildung 3 zeigt den resultierenden Inhalt des Warteschlangen-Arrays schematisch):

- `add(0, 'A', true)` gibt `Response(0,0)` zurück
- `add(0, 'B', false)` gibt `Response(0,1)` zurück
- `add(1, 'C', false)` gibt `Response(2,2)` zurück
- `add(1, 'A', true)` gibt `Response(2,3)` zurück
- `add(0, 'C', true)` gibt `Response(0,4)` zurück
- `add(1, 'B', false)` gibt `Response(2,5)` zurück

- `add(2, 'D', true)` gibt `Response(6,6)` zurück
- `add(3, 'E', false)` gibt `Response(7,7)` zurück
- `add(3, 'E', false)` gibt `null` zurück, da alle 8 Plätze belegt sind

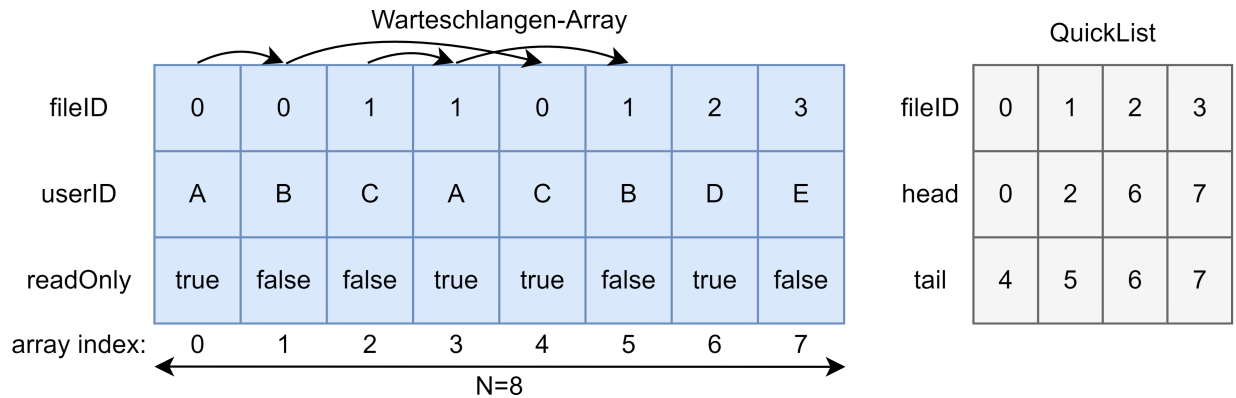


Abbildung 3: Der Zustand der Warteschlange nach den neun add()-Operationen.

Teilaufgabe 2:

Implementieren Sie die Methode `char[] pop(int fileID)` des `WaitQueueArray`. Die `pop()`-Methode entfernt alle Lese-Einträge mit derselben `fileID`, wenn der Eintrag beim `head` von `fileID` ein Lese-Eintrag ist (`readOnly == True`), wie in Abbildung 4 gezeigt. Falls der Eintrag beim `head` von `fileID` ein Schreib-Eintrag ist (`readOnly == False`), entfernt die `pop()`-Methode nur den Eintrag bei `head` von `fileID`, wie in Abbildung 5 gezeigt. Zurückgegeben wird ein `char`-Array mit den `userIDs` der entfernten Einträge in der Reihenfolge, wie sie eingefügt wurden. Das Array kann auch Duplikate enthalten. Wenn es keine Warteschlange mit dieser `fileID` gibt, wird `null` zurückgegeben.

- `pop(0)` gibt {'A', 'C'} zurück, weil bei `head` ein Lese-Eintrag ist. (Abbildung 4)
- `pop(1)` gibt {'C'} zurück, weil bei `head` ein Schreib-Eintrag ist. (Abbildung 5)

Teilaufgabe 3:

Implementieren Sie die Methode `int[][] getQuickList()`, die ein Array mit `fileID`, `head` und `tail` der Warteschlange für alle gespeicherten Dateien zurückgibt – in aufsteigender Reihenfolge der `fileIDs`. Sie gibt `null` zurück, wenn keine Einträge in der Warteschlange vorhanden sind. In den Abbildungen 3-6 finden Sie dieses Array jeweils auf der rechten Seite.

- `getQuickList()` in Abbildung 4 gibt {{0,1,1}, {1,2,5}, {2,6,6}, {3,7,7}} zurück.
- `getQuickList()` in Abbildung 5 gibt {{0,1,1}, {1,3,5}, {2,6,6}, {3,7,7}} zurück.
- `getQuickList()` in Abbildung 6 gibt {{0,1,1}, {1,3,0}, {2,6,6}, {3,7,7}} zurück.

| Warteschlangen-Array nach pop(0) | | | | | | | | QuickList | | | | | |
|----------------------------------|---|-------|-------|------|---|-------|------|-----------|--------|---|---|---|---|
| fileID | - | 0 | 1 | 1 | - | 1 | 2 | 3 | fileID | 0 | 1 | 2 | 3 |
| userID | - | B | C | A | - | B | D | E | head | 1 | 2 | 6 | 7 |
| readOnly | - | false | false | true | - | false | true | false | tail | 1 | 5 | 6 | 7 |

Abbildung 4: Der Zustand der Warteschlange nach einer pop(0)-Operation.

| Warteschlangen-Array nach pop(1) | | | | | | | | | QuickList | | | | |
|----------------------------------|---|-------|---|------|---|-------|------|-------|-----------|---|---|---|---|
| fileID | - | 0 | - | 1 | - | 1 | 2 | 3 | fileID | 0 | 1 | 2 | 3 |
| userID | - | B | - | A | - | B | D | E | head | 1 | 3 | 6 | 7 |
| readOnly | - | false | - | true | - | false | true | false | tail | 1 | 5 | 6 | 7 |

Abbildung 5: Der Zustand der Warteschlange nach einer pop(1)-Operation.

Warteschlangen-Array nach add(1, 'F', true)

| | | | | | | | | |
|----------|------|-------|---|------|---|-------|------|-------|
| fileID | 1 | 0 | - | 1 | - | 1 | 2 | 3 |
| userID | F | B | - | A | - | B | D | E |
| readOnly | true | false | - | true | - | false | true | false |

tail

head

QuickList

| | | | | |
|--------|---|---|---|---|
| fileID | 0 | 1 | 2 | 3 |
| head | 1 | 3 | 6 | 7 |
| tail | 1 | 0 | 6 | 7 |

Abbildung 6: Der Zustand der Warteschlange nach einer add()-Operation. Unter dem Warteschlangen-Array sind head und tail von Datei 1 markiert.


Anhang A: Testprogramm Klassenrätsel

```
class Klassen {
    ...

    public static void klassen(PrintStream output) {
        Z ref1 = new B();
        ref1.bar(output);
        output.println("++");
        Z ref2 = new A();
        ((A) ref2).bar(output);
        output.println("++");


        C c1 = new C();
        output.println("C.foo()");
        c1.foo(output);
        output.println("--");
        D d1 = new D();
        if (d1 instanceof C) {
            ((C)d1).test(output);
        } else {
            d1.foo(output);
        }
    }
}
```

Anhang B: Klassen Klassenrätsel

```
class C  {

    public void foo(PrintStream output) {
        output.println("Here");
    }

    public void test(PrintStream output) {
        output.println("Test");
    }
}

class D  {

    public void foo(PrintStream output) {
        super.foo(output);
    }
}
```

```

    }
}

class Z [REDACTED] {

    public void bar(PrintStream output) {
        output.println("Hello");
    }
}

```

```

class A [REDACTED] {

    int a1 = 0;

    A() {}
    A(int v) {
        a1 = v;
    }

    public void foo(PrintStream output) {
        output.println("Found");
    }
}

```

```

class B [REDACTED] {

    B() { }

    B(int w) {
        super(w);
    }

    public void bar(PrintStream output) {
        super.bar(output);
        output.println("Bingo");
    }
}

```

Anhang C: Ausgabe Klassenrätsel

```
Hello  
Bingo  
++  
Hello  
++  
C.foo():  
Here  
--  
Found
```