Programmierung 2 (SS 2021)
Universität des Saarlandes
Fakultät MI
Compiler Design Lab

Musterlösung 11
Hashing und Dijkstra

Prof. Dr. Sebastian Hack
Julian Rosemann, B. Sc.
Thorsten Klößner, M. Sc.
Julia Wichlacz, M. Sc.
Maximilian Fickert, M. Sc.

Im Vorlesungskalender finden Sie Informationen über die Kapitel des Skripts, die parallel zur Vorlesung bearbeitet werden sollen bzw. dort besprochen werden. Die Übungsaufgaben dienen der Vertiefung des Wissens, das in der Vorlesung vermittelt wird und als Vorbereitung auf Minitests und Klausur.
Weitere Aufgaben zu den Themen finden Sie jeweils am Ende der Skriptkapitel.
Die Schwierigkeitsgrade sind durch Steine des 2048-Spiels gekennzeichnet, von 512 „leicht“ bis 2048 „schwer“. 4096 steht für Knobelaufgaben.

Hashing

Aufgabe 11.0: hashCode und equals

Im Folgenden sind zwei Klassen `Fraction` und `Str` gegeben, die jeweils Brüche und Zeichenketten darstellen sollen. Implementieren Sie für beide sinnvolle `hashCode()` und `equals()` Methoden, die die entsprechenden Java-Konventionen einhalten.

a)

```java
public class Fraction {

    private final int numerator, denominator;

    public Fraction(int numerator, int denominator) {
        if (denominator == 0) throw new IllegalArgumentException();
        this.numerator = numerator;
        this.denominator = denominator;
    }

    public int getNumerator() {
        return numerator;
    }

    public int getDenominator() {
        return denominator;
    }
}
```

b)

```java
public class Str {

    private final byte[] values;

    public Str(byte... values) {
        this.values = values;
    }

    @Override
    public String toString() {
        return String.valueOf(values);
    }
}
```

Lösung

a)

```java
public class Fraction {

    private final int numerator, denominator;

    public Fraction(int numerator, int denominator) {
        if (denominator == 0)
            throw new IllegalArgumentException();
        this.numerator = numerator;
        this.denominator = denominator;
    }

    public int getNumerator() {
        return numerator;
    }

    public int getDenominator() {
        return denominator;
    }

    @Override
    public int hashCode() {
        // make numerator and denominator unique by computing their
        // greatest common divisor and dividing by it
        int gcd = computeGcd(numerator, denominator);
        return numerator / gcd + (denominator / gcd) * 31;

        // Alternative way to do it:
        // It may suffer errors introduced by floating point arithmetic
        //return new Double((double) numerator / denominator).hashCode();
    }

    private static int computeGcd(int a, int b) {
        return b == 0 ? a : computeGcd(b, a % b);
    }

    @Override
    public boolean equals(Object obj) {
        if (!(obj instanceof Fraction))
            return false;
        Fraction other = (Fraction) obj;
        // this ensures that 3 / 4 and 6 / 8 are equal
        return this.numerator * other.denominator ==
         this.denominator * other.numerator;
    }
}
```

b)

```java
public final class Str {

    private final byte[] values;

    public Str(byte... values) {
        this.values = values;
    }

    @Override
    public int hashCode() {
        final int prime = 29;
        int hash = 1;
        // uses the Horner schema
        for (byte b : values) {
            hash = prime * hash + b;
        }
        return hash;
    }

    @Overrided
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Str other = (Str) obj;
        if (values.length != other.values.length)
            return false;
        for (int i = 0; i < values.length; i++) {
            if (values[i] != other.values[i])
                return false;
        }
        return true;
    }

    @Override
    public String toString() {
        return String.valueOf(values);
    }
}
```

Die Klasse `String` in `java.lang.String` ist `final`. Das bedeutet, von dieser Klasse kann nicht geerbt werden. Diese Eigenschaft wurde hier übernommen. Deshalb wird in der Methode `equals` aus Effizienz `getClass` an Stelle von `instanceof` verwendet - letzteres ist jedoch nicht falsch. Bedenken Sie, dass bei der Verwendung von `getClass` separat auf `null` getestet werden muss.

Aufgabe 11.1: Kollisionslisten

Konrad Klug möchte seine Lieblingsinformatiker in einem `HashSet` speichern. Dazu hat er folgende Klasse angelegt und eine Hashfunktion implementiert.
`quersumme(int x)` berechnet dabei die Quersumme von `x`.


```
1    public class Wissenschaftler {
2      private String name;
3      private short tag, monat, jahr;
4
5      public int hashCode() {
6        return quersumme(tag) + quersumme(monat) + quersumme(jahr);
7      }
8    }
```

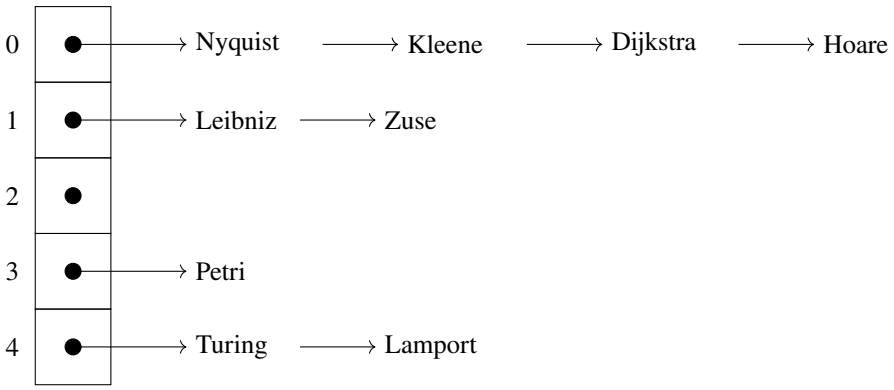
Name	Geburtstag
Alan Turing	23.6.1912
Gottfried Wilhelm Leibniz	21.6.1646
Harry Nyquist	07.2.1889
Leslie Lamport	07.2.1941
Stephen Cole Kleene	05.1.1909
Edsger Wybe Dijkstra	11.5.1930
Konrad Ernst Otto Zuse	22.6.1910
Carl Adam Petri	12.7.1926
Charles Antony Richard Hoare	11.1.1934

Tabelle 1: Wissenschaftler

Hashen Sie die Wissenschaftler aus Tabelle 1 in ein HashSet der Größe 5 ein. Verwenden Sie zur Kollisionsauflösung Kollisionslisten.

Lösung

Name	Geburtstag	Quersumme	Platz
Alan Turing	23.6.1912	24	4
Gottfried Wilhelm Leibniz	21.6.1646	26	1
Harry Nyquist	07.2.1889	35	0
Leslie Lamport	07.2.1941	24	4
Stephen Cole Kleene	05.1.1909	25	0
Edsger Wybe Dijkstra	11.5.1930	20	0
Konrad Ernst Otto Zuse	22.6.1910	21	1
Carl Adam Petri	12.7.1926	28	3
Charles Antony Richard Hoare	11.1.1934	20	0



Aufgabe 11.2: Lineares Sondieren

Konrad Klug hat seine Meinung geändert. Er möchte die Kollisionen nun doch mittels linearem Sondieren auflösen.

1. Hashen Sie die Wissenschaftler aus Tabelle 1 in ein HashSet der Größe 9 ein. Verwenden Sie zur Kollisionsauflösung lineares Sondieren.
2. Löschen Sie Leibniz wieder aus dem HashSet. Danach möchte Konrad überprüfen, ob Nyquist in seinem HashSet enthalten ist. Welches Problem tritt auf und wie können Sie es beheben?

Lösung

1.

0	Nyquist
1	Kleene
2	Dijkstra
3	Zuse
4	Petri
5	Hoare
6	Turing
7	Lamport
8	Leibniz

Name	Geburtstag	Quersumme	Platz
Alan Turing	23.6.1912	24	6
Gottfried Wilhelm Leibniz	21.6.1646	26	8
Harry Nyquist	07.2.1889	35	8
Leslie Lamport	07.2.1941	24	6
Stephen Cole Kleene	05.1.1909	25	7
Edsger Wybe Dijkstra	11.5.1930	20	2
Konrad Ernst Otto Zuse	22.6.1910	21	3
Carl Adam Petri	12.7.1926	28	1
Charles Antony Richard Hoare	11.1.1934	20	2

2. Wenn nur getestet wird, ob an Position 8 ein Element enthalten ist, wird Nyquist nicht gefunden. Es muss markiert werden, ob an der entsprechenden Position mal ein Element enthalten war um gegebenenfalls weiter zu suchen.

Aufgabe 11.3: Quadratisches Sondieren

Konrad Klug hat seine Meinung nochmal geändert. Er möchte die Kollisionen nun doch mittels quadratischem Sondieren auflösen.

1. Hashen Sie die Wissenschaftler aus Tabelle 1 in folgender Reihenfolge in ein HashSet der festen Größe 9 ein:
   `Turing Kleene Lamport Nyquist Dijkstra Leibniz Petri Zuse Hoare`
   Verwenden Sie zur Kollisionsauflösung quadratisches Sondieren mit der Funktion:

$$h'(x,i) = \left(h(x) + \frac{i(i+1)}{2}\right) \; mod \; 9$$

2. Was fällt Ihnen auf?

Lösung

1.

0	Lamport
1	Petri
2	Dijkstra
3	Zuse
4	
5	Leibniz
6	Turing
7	Kleene
8	Nyquist

2. Hoare kann nicht im freien 4. Platz eingefügt werden, da für kein i gilt: `h'(Hoare, i) = 4`

Aufgabe 11.4: Cuckoo Hashing

Konrad Klug ist begeistert von Hashing und möchte neben linearem und quadratischen Sondieren weitere praxisrelevante Sondierungsmethoden finden. Bei seinen Recherchen stößt Konrad auf **Cuckoo** und **Robin-Hood Hashing**, welche er versucht auf selbst generierte Beispiele anzuwenden. Jedoch muss er feststellen, dass er die Theorie der beiden Methoden noch nicht ganz verstanden hat. Helfen Sie ihm dabei! (Lesen Sie sich hierbei die Theorie hinter Cuckoo und Robin Hood Hashing im Internet an).

Gegeben sei folgende Hashtabelle:

0	1	2	3	4	5	6

1. Gegeben seien folgende Hashfunktionen für die benötigten Hashtabellen: $h_1(x) = (x+1)^2 - 3$, $h_2(x) = 2 * x$. Fügen sie 2,7,11,1,8,15 in die Hashtabelle mit Cuckoo Hashing ein.

2. Gegeben sei folgende Hashfunktion: $h_1(x) = 2x - 1$. Fügen sie 2,7,11,1,8,15 in die Hashtabelle mit Robin-Hood-Hashing ein.

Lösung

1.

0	1	2	3	4	5	6
	8			7		2

0	1	2	3	4	5	6
	11	1				

Will man nun die 15 einhashen, entsteht wieder eine Kollision und man versucht die 8 in das zweite Array einzuhashen. Hier kommt es erneut zu einer Kollision, wodurch die 8 in das zweite Array eingesetzt und respektive die 1 in das erste Array eingesetzt wird. Da wir nun die 15 aus dem ersten Array herausnehmen müssen und in das zweite Array einsetzen, versuchen wir die 8 in das erste Array einzuhashen. Nun muss die 1 wieder in das zweite und verdrängt somit die 15. Jetzt haben wir einen Zyklus gefunden, da wir die Zahl die wir ursprünglich einhashen wollten, erneut in das erste Array einfügen müssten. Wir verdoppeln nun die Größe des Arrays. Anschließend müssen alle Werte neu eingehasht werden:

0	1	2	3	4	5	6	7	8	9	10	11	12	13
	15				7	2		8					

0	1	2	3	4	5	6	7	8	9	10	11	12	13
		1						11					

2. Nach dem Einhashen der Werte und dem Lösen der Kollisionen sieht die Hashtabelle wie folgt aus:

0	1	2	3	4	5	6
11	1	8	15	2		7
0	1	1	1	3		6

Aufgabe 11.5: Veränderter Inhalt

Betrachten Sie folgendes Programm:

```
1 import java.util.HashSet;
2
3 public class Main {
4
5     public static void main(String[] args) {
6         HashSet<Lorex> uhren = new HashSet<Lorex>();
7         Lorex r1 = new Lorex("Explorer", 5600, 500);
8         Lorex r2 = new Lorex("Pre-Daytona", 12050, 700);
9
10        uhren.add(r1);
11        uhren.add(r2);
12        System.out.println(uhren.contains(r1));
13        System.out.println(uhren.contains(r2));
14
15        r1.setPreis(4643);
16        r2.setPreis(19040);
17        System.out.println(uhren.contains(r1));
18        System.out.println(uhren.contains(r2));
19    }
```

```
20 }
```

```
1 public class Lorex {
2  String name;
3  int preis;
4  int goldanteil;
5
6  public Lorex(String name,
7     int preis, int goldanteil) {
8   this.goldanteil = goldanteil;
9   this.preis = preis;
10  this.name = name;
11 }
12
13 public void setPreis(int preis) {
14  this.preis = preis;
15 }
```

```
16 @Override
17 public boolean equals(Object obj) {
18  if (this == obj)
19    return true;
20  if (obj == null)
21    return false;
22  if (getClass() != obj.getClass())
23    return false;
24  Lorex other = (Lorex) obj;
25  if (!name.equals(other.name))
26    return false;
27  return true;
28 }
29
30 @Override
31 public int hashCode() {
32  return (goldanteil + preis) % 10;
33 }
34 }
```

1. Was ist die Ausgabe der `main`-Methode?
2. Überrascht Sie diese Ausgabe? Wobei liegt hier das Problem?

Lösung

1.
```
System.out.println(uhren.contains(r1));  =>  true
System.out.println(uhren.contains(r2));  =>  true

System.out.println(uhren.contains(r1));  =>  false
System.out.println(uhren.contains(r2));  =>  true
```

2. Wir wissen, dass 2 Objekte, die sich im Sinne von `equals` gleichen, den gleichen Hashwert besitzen müssen. Das ist hier jedoch nicht der Fall. Mit dem Überschreiben von `equals` definieren wir 2 Objekte der Klasse `Lorex` als gleich, wenn sie denselben Namen besitzen. Unsere Hashfunktion bezieht sich jedoch nur auf die Felder `goldanteil` und `preis`, welche nichts mit `equals` zu tun haben. Deshalb wird in Zeile 17 der `main`-Methode irrtümlicherweise `false` ausgegeben, obwohl sich lediglich der Preis von `r1` und nicht ihr Befinden in `uhren` verändert hat. Eine bessere Hashfunktion würde sich demnach auf `name` beziehen und sieht wie folgt aus:

```
1  @Override
2  public int hashCode() {
3    return name.hashCode();
4    }
```

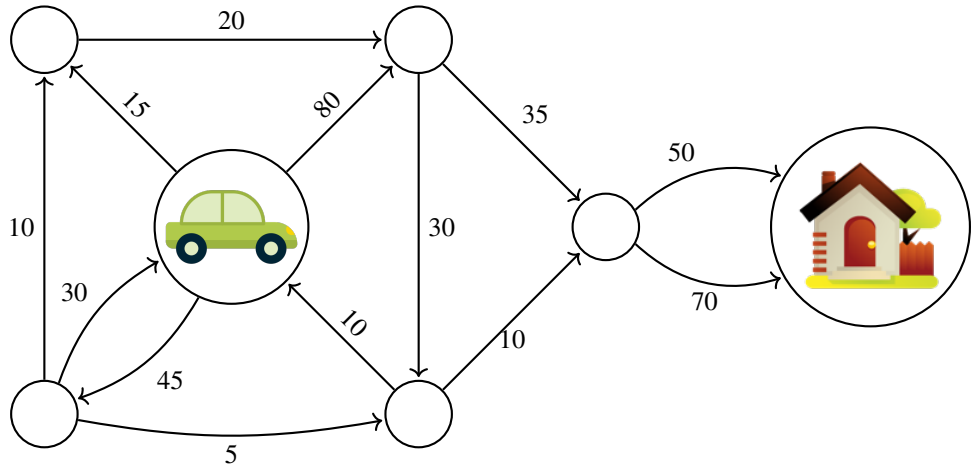
Dijkstra

Aufgabe 11.6: Dijkstra

Konrad Klug braucht mal wieder Hilfe von No Hau. Deswegen schwingt sie sich in ihr No-Hau-Mobil und will schnellstmöglich zu Konrad gelangen. Als sie ihr Navi anschaltet, um den Weg zu Konrad zu erfahren, lädt dieses jedoch Updates herunter. No Hau glaubt, dass sie es schafft den kürzesten Weg zu finden bevor die Updates heruntergeladen sind.

1. No Hau hat gehört dass ihr Navi mit Hilfe des Dijkstra-Algorithmus den kürzesten Weg zu einem gewünschten Ziel findet. Machen Sie sich klar, wie der Algorithmus von Dijkstra funktioniert (siehe z.B. im Skript auf S.200ff.) und fassen Sie ihn im Pseudocode zusammen.

2. Betrachten Sie den folgenden Graphen, der die Möglichkeiten von No Hau darstellt um zu Konrad zu gelangen. Führen Sie auf ihm den Dijkstra-Algorithmus aus. Markieren Sie dabei jeweils die Knoten mit der Iteration, in der der Algorithmus den Knoten erreicht hat. Beschriften Sie zudem jeden Knoten mit den Kosten des kürzesten Wegs ausgehend vom Startpunkt.



3. *Challenge*: No Hau fand Dijkstra zu nutzen umständlich. Wie immer hat sie aber eine Lösung parat. Sie weiß, dass modernere Navigationssysteme einen anderen Algorithmus namens A* nutzen. Als sie diesen anwendet, findet sie den Weg viel schneller.

   - Recherchieren Sie wie A* funktioniert um den Algorithmus auch selbst auf den in (b) abgebildeten Graphen anwenden zu können.
   - Welche Heuristik würden Sie wählen um den kürzesten Weg zu finden? Welche Varianten fallen Ihnen ein?

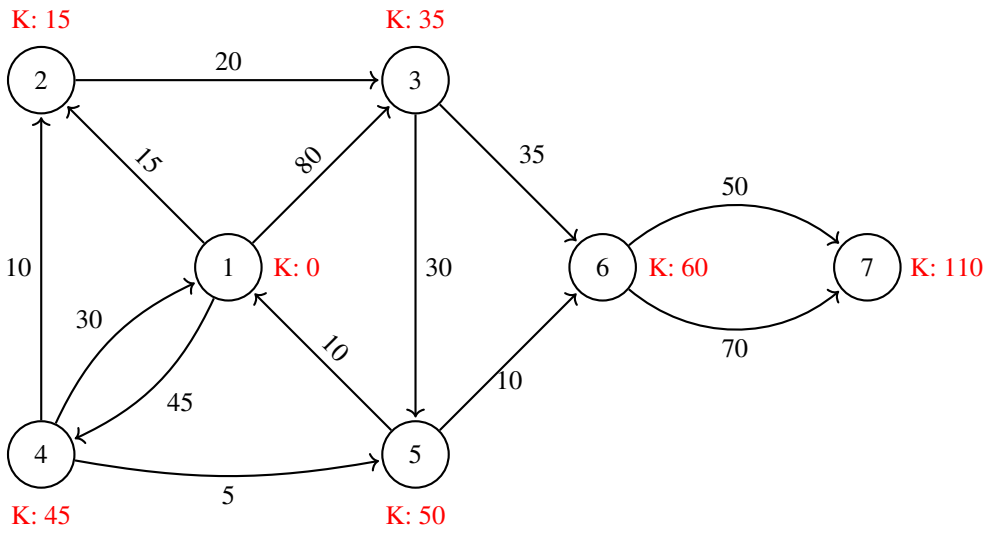
Lösung

1. Der Dijkstra-Algorithmus berechnet, gegeben einen Graphen und einen Startknoten s, den kürzesten Pfad von s zu allen anderen Knoten im Graphen. Hier der Algorithmus als Pseudocode: (als Beispiel für eine konkretere Implementierung in Java siehe im Skript auf S. 201)

```
1 Dijkstra(graph, s):
2     Q := Menge aller Knoten in graph
3     für alle Knoten v in graph:
4         //Länge des kürzesten Pfades zu v
5         v.dist = ∞
6         //Vorgänger des kürzesten Pfades
7         v.pred = none
8     s.dist = 0
9
10    solange Q nicht leer:
11        v := Knoten in Q mit kleinstem v.dist
12        entferne v aus Q
13
14        für alle Kanten e von v:
15            //Zielknoten der Kante e (e geht also von v nach u)
16            u = e.target
17
18            falls u ∈ Q:
```

```
19                  dist2 = v.dist + e.length
20                  //Länge des kürzesten Pfades zu v + Länge der Kante e
21                  falls dist2 < u.dist:
22                      u.dist = dist2
23                      u.pred = v
```

2. Nach der Ausführung des Algorithmus sollte der Graph wie folgt beschriftet sein: (Dabei ist in den Zuständen die Iteration eingetragen in der der Zustand erreicht wurde. Und das annotierte K die Kosten des kürzesten Weges zum jeweiligen Zustand.)



3. Der Algorithmus ist sehr gut auf Wikipedia dokumentiert.

   Beispiel für eine Heuristik, die sofort den kürzesten Weg findet:

$$h(n) = \begin{cases} 105 & n = 2 \\ 85 & n = 3 \\ 0 & \text{sonst} \end{cases}$$