

Datenstrukturen und Algorithmen

Cedric Aehi 17-103-235

Nicolas Müller 17-122-094

Datenstrukturen und Algorithmen

Übung 6, Frühling 2018

29. März 2018

Abgabe: Diese Übung muss zu Beginn der Übungsstunde bis spätestens um 16 Uhr 15 am 12. April abgegeben werden. Die Abgabe der DA Übungen erfolgt immer in schriftlicher Form auf Papier. Programme müssen zusammen mit der von Ihnen erzeugten Ausgabe abgegeben werden. Drucken Sie wenn möglich platzsparend 2 Seiten auf eine A4-Seite aus. Falls Sie mehrere Blätter abgeben, heften Sie diese bitte zusammen (Büroklammer, Bostitch, Mäppchen). *Der gesamte Sourcecode muss außerdem elektronisch über Ilias abgegeben werden.*

Die Übung sollte vorzugsweise in Zweiergruppen bearbeitet werden, kann aber auch einzeln abgegeben werden. Vergessen Sie nicht, Ihren Namen und Ihre Matrikelnummer auf Ihrer Abgabe zu vermerken. Jede Übungsserie gibt 10 Punkte. Im Durchschnitt müssen Sie 7 von 10 Punkten erreichen, um die Testatbedingungen zu erfüllen.

Theoretische Aufgaben

1. Betrachten Sie eine Hashtabelle der Grösse $m = 500$ und eine zugehörige Hashfunktion $h(k) = \lfloor m(kA \bmod 1) \rfloor$ mit $A = (\sqrt{5} - 1)/2$. Berechnen Sie die Plätze, auf die die Schlüssel 41, 42, 43, 44 und 45 abgebildet werden. (**1 Punkt**)
2. Professor Joe behauptet, dass eine erhebliche Performanzsteigerung erzielt werden kann, wenn wir das Verkettungsschema so modifizieren, dass jede Liste in sortierter Ordnung gehalten wird. Wie beeinflusst die durch den Professor vorgeschlagene Änderung die Laufzeit für erfolgreiches Suchen, erfolgloses Suchen, Einfügen und Löschen? (**1 Punkt**)
3. Betrachten Sie das Einfügen der Schlüssel 24, 18, 13, 56, 44, 7, 19, 23, 33 in eine Hashtabelle der Länge $m = 11$ durch offene Adressierung mit der Hilfshashfunktion $h'(k) = k \bmod m$. Illustrieren Sie das Ergebnis des Einfügens dieser Schlüssel mithilfe von:
 - (a) linearem Sondieren
 - (b) quadratischem Sondieren mit $c_1 = 1$ und $c_2 = 3$
 - (c) doppeltem Hashing mit $h_2(k) = 1 + (k \bmod (m - 1))$**(1 Punkt)**
4. Zeigen Sie, dass die mittlere Anzahl von Hashtabellenplätzen, die bei einer erfolgreichen Suche (mit gleicher Wahrscheinlichkeit für alle Schlüssel) inspiziert werden, bei Hashing mit linearem Sondieren nicht von der Reihenfolge abhängt, in der die Schlüssel in die

anfangs leere Hashtabelle eingefügt worden sind. Gilt die entsprechende Aussage auch für quadratisches Sondieren? (**1 Punkt**)

5. Bei der Kollisionsauflösung nach dem Verkettungsschema muss für kollidierende Element neuer Speicher alloziert werden, bevor sie an den Kopf der Liste an ihrem Tabellenplatz gesetzt werden können. Machen Sie einen Vorschlag, wie dazu Speicherplatz innerhalb der Tabelle genutzt, also zugewiesen und freigegeben werden kann. Alle unbenutzten Tabellenplätze sollen in einer doppelt verlinkten Liste geführt werden (einer Freiliste). Diese Freiliste soll direkt innerhalb der Tabelle implementiert sein. Nehmen Sie dazu an, dass auf jedem Tabellenplatz eine Markierung (frei/benutzt) und entweder ein Element und ein Zeiger oder zwei Zeiger abgespeichert werden können:

$$[flag, element, ptr] \text{ oder } [flag, ptr1, ptr2].$$

Alle Operationen (einfügen, löschen, suchen) auf dem Wörterbuch unter Benutzung der Freiliste sollten in einer erwarteten Zeit von $O(1)$ laufen. Muss die Freiliste doppelt verkettet sein oder genügt eine einfach verkettete Liste? (**1 Punkt**)

Praktische Aufgaben

In dieser Übung werden Sie eine Variante von Hashing entwickeln, um räumliche Daten zu verwalten. Als Ausgangslage dazu stellen wir auf ILIAS Code zur Verfügung, der eine einfache Partikelsimulation implementiert. Nehmen Sie sich ein paar Minuten Zeit, um den Code zu studieren. Der Code simuliert eine Menge von Partikeln, die sich frei im zwei-dimensionalen Raum bewegen, bis sie mit einem anderen Partikel oder mit der Umgebung kollidieren. Bei einer Kollision werden die Partikel gemäss einem einfachen Modell abgelenkt. Die Simulation erfolgt über diskrete Zeitschritte. In jedem Schritt werden die Partikel gemäss ihrer Geschwindigkeit weiterbewegt, und bei Kollisionen wird zusätzlich die neue Geschwindigkeit und Richtung berechnet.

Eine naive Implementation der Kollisionsdetektion überprüft jedes Paar von Partikeln auf eine mögliche Kollision. Der Aufwand für die Kollisionsdetektion ist somit $O(n^2)$, wobei n die Anzahl Partikel ist. Bei einer grösseren Anzahl Partikel wird dadurch der Aufwand zur Berechnung jedes Schritts in der Simulation schnell von der Kollisionsdetektion dominiert.

Ihre Aufgabe ist es, einen effizienteren Algorithmus zur Kollisionsdetektion zu entwickeln. Die Idee ist es, eine Datenstruktur zu verwenden, welche die Partikel gemäss ihrer räumlichen Position verwaltet. Weil die Partikel zufällig im Raum verteilt sind, kann die Position eines Partikels verwendet werden, um einen Hashwert zu berechnen ähnlich wie zu Beginn von Kapitel 11.3 im Buch beschrieben. Gegeben die x Koordinate des Partikels im Bereich $0 \leq x < w$ ist der Hashwert $h(x) = \lfloor xm/w \rfloor$, wobei m die Grösse der Hashtabelle ist. Ebenso kann für die y Koordinate ein Hashwert berechnet werden. Die beiden Hashwerte können nun als Indizes in eine zweidimensionale Hashtabelle verwendet werden.

Die zweidimensionale Hashtabelle entspricht einem zweidimensionalen Gitter: jeder Partikel wird entsprechend seiner Position einer Gitterzelle zugeordnet. Damit kann die Kollisionsdetektion effizienter gemacht werden, indem jeder Partikel nur auf Kollisionen in seiner eigenen Gitterzelle und den unmittelbaren Nachbarzellen untersucht wird. Die Nachbarzellen

müssen getestet werden, weil Partikel eine gewisse Ausdehnung haben und mit den Nachbarzellen überlappen können. Beachten Sie auch, dass die Partikel nach jedem Simulationsschritt entsprechend ihrer neuen Position aus der Hashtabelle entfernt und in die richtige Zelle wieder eingetragen werden müssen.

1. Modifizieren Sie die Klasse *BouncingBallsSimulation*, um den skizzierten Algorithmus zu implementieren. Verwenden Sie ein zweidimensionales Feld von verketteten Listen als Hashtabelle. Für die Listen können Sie die Java *LinkedList* Klasse verwenden. Ihre Hashtabelle ist dann vom Typ *LinkedList<Ball>[][]*. Verwenden Sie einen Iterator vom Typ *ListIterator<Ball>* um die Listen zu traversieren. Dieser Iterator erlaubt Ihnen mit seiner Methode *remove()* effizient das zuletzt besuchte Element aus der Liste zu entfernen.

Drucken Sie nur Ihre modifizierte Klasse *BouncingBallsSimulation* aus und geben Sie sie ab. **(3 Punkte)**

2. Testen Sie Ihren Algorithmus für verschiedene Größen der Hashtabelle und verschiedene Anzahl Partikel. Stellen Sie die Zeitmessungen in einer Tabelle zusammen. Was ist jeweils die optimale Größe der Hashtabelle? Was könnte der Grund sein, dass die Geschwindigkeit bei grösseren Tabellen wieder abnimmt? **(2 Punkte)**

Vergessen Sie nicht Ihren Sourcecode innerhalb der Deadline über die Ilias Aufgabenseite einzureichen.

1. Betrachten Sie eine Hashtabelle der Grösse $m = 500$ und eine zugehörige Hashfunktion $h(k) = \lfloor m(kA \bmod 1) \rfloor$ mit $A = (\sqrt{5} - 1)/2$. Berechnen Sie die Plätze, auf die die Schlüssel 41, 42, 43, 44 und 45 abgebildet werden. (1 Punkt)

$$h(41) = 169 \quad h(44) = 96$$

$$h(42) = 478 \quad h(45) = 405$$

$$h(43) = 287$$

2. Professor Joe behauptet, dass eine erhebliche Performanzsteigerung erzielt werden kann, wenn wir das Verkettungsschema so modifizieren, dass jede Liste in sortierter Ordnung gehalten wird. Wie beeinflusst die durch den Professor vorgeschlagene Änderung die Laufzeit für erfolgreiches Suchen, erfolgloses Suchen, Einfügen und Löschen? (1 Punkt)

Löschen: ändert sich nichts $\Rightarrow O(1)$

Einfügen: Unter der gleichen Annahme, dass der Schlüssel in der Tabelle nicht schon vor kommt. Dann muss das Element an der richtigen Stelle eingefügt werden, und nicht nur am Head. Worst-case $O(n)$ statt $O(1)$.

Erfolgreiches: ändert sich nichts.

Suchen

Erfolglos: Änderung nur in der Praxis. Die Suche innerhalb einer Liste wird abgebrochen, weil alle anderen Elemente nachher grösser sind.

3. Betrachten Sie das Einfügen der Schlüssel 24, 18, 13, 56, 44, 7, 19, 23, 33 in eine Hashtabelle der Länge $m = 11$ durch offene Adressierung mit der Hilfshashfunktion $h'(k) = k \bmod m$. Illustrieren Sie das Ergebnis des Einfügens dieser Schlüssel mithilfe von:

- (a) linearem Sondieren
- (b) quadratischem Sondieren mit $c_1 = 1$ und $c_2 = 3$
- (c) doppeltem Hashing mit $h_2(k) = 1 + (k \bmod (m - 1))$

a) $h(k, i) = ((k \bmod m) + i) \bmod m$

$k=24$ insert in 2

$k=18$ insert in 7

$k=13$ occupied 2, insert in 3

$k=56$ insert in 1

$k=44$ insert in 0

$k=7$ occupied 7, insert in 8

$k=19$ occupied 8, insert in 9

$k=23$ occupied 1, occupied 2, occupied 3, insert in 4

$k=33$ occupied 0, 1, 2, 3, 4, insert in 5

0	1	2	3	4	5	6	7	8	9	10
44	56	24	13	23	33	NULL	18	7	19	NULL

b) $h(k, i) = ((k \bmod m) + 1 \cdot i + 3 \cdot i^2) \bmod m$

$k=24$ (In Zukunft ist grün erfolgreich eingefügt und rot besetzt) 2

$k=18$ 7

$k=13$ 2, 6

$k=56$ 1

$k=44$ 0

$k=7$ 7, 0, 10

$k=19$ 8

$k=23$ 1, 5

$k=33$ 0, 4

0 1 2 3 4 5 6 7 8 9 10
44 56 24 NULL 33 23 13 18 19 NULL 7

$$c) h(k, i) = ((k \bmod m) + i \cdot (1 + (k \bmod (m - 1)))) \bmod m$$

$k=24$ 2

$k=18$ 7

$k=13$ 2, 6

$k=56$ 1

$k=44$ 0

$k=7$ 7, 4

$k=19$ 8

$k=23$ 1, 5

$k=33$ 0, 4, 8, 1, 5, 9

0 1 2 3 4 5 6 7 8 9 10
44 56 24 NULL 7 23 13 18 19 33 NULL

4) Die Summe der Suchzeiten von k_i ($\sum_{i=1}^n (k_i)$) ist gleich die Summe einer beliebigen Permutation von k_i ($\sum_{i=1}^n \text{perm.}(k_i)$)

Sehen wir uns einen Konflikt an, welcher bei einem bereits belegten Platz entsteht.

0 1 2 3 4 5 6 7 8 9 10
1 2

$k=1$ bekommt Platz 5

$k=2$ Platz 5 besetzt, bekommt Platz 6

oder

0 1 2 3 4 5 6 7 8 9 10
2 1

$k=2$ bekommt Platz 5

$k=1$ 5, bekommt 6

Man sieht die Belegungsstruktur bleibt gleich. Es werden nur 1 und 2 vertauscht. Auch wenn beliebig viele Elemente dazwischen liegen.

Berechnung beim linearen sondieren:

Wenn der ursprüngliche Platz besetzt ist, wird der nächste Platz mit $(\text{mod } n)$ berechnet.

Also ist die Struktur unabhängig von der Reihenfolge

Quadrat! sch:

0 1 2 3 4 5 6 7 8 9 10
1 2

k=1 bekommt Platz 5

k=2 5 besetzt, bekommt Platz 9

hier sieht man bereits den Unterschied. 1 ist bei
 $i=5$ und 2 bei $i=9$

Jetzt können auch Elemente dazwischen liegen, was zur Folge hat, dass die Struktur abhängig von der Reihenfolge ist.

5)

Vorschlag für Speichernutzung innerhalb der Tabelle:

-Man könnte die Überläufer in einem freien Tabellenplatz abspeichern und dann

mit einem Zeiger darauf verweisen. Somit müsste kein Speicher ausserhalb der Tabelle genutzt werden.

Einfach/Doppelte Verkettung

-Um die Laufzeit O(1) zu erhalten, muss die Liste doppelt verkettet sein. So kann bei der Suche direkt auf das vorangehende Element zugegriffen werden und

so in Laufzeit O(1) realisiert werden. Beim Löschen eines Elements muss die Liste auch doppelt verkettet sein, um die Operation in O(1) Laufzeit ausführen zu können.

Praktischer Teil

1)

```

1 import java.awt.*;
2 import java.awt.geom.Ellipse2D;
3 import java.util.Iterator;
4 import java.util.LinkedList;
5
6 /**
7  * Implements a bouncing ball simulation.
8 */
9 public class BouncingBallsSimulationImproved extends Component implements Runnable {
10
11     LinkedList<Ball>[][] hashtable; // List of balls.
12     LinkedList<Ball> balls; // List that helps to iterate
13     Image img; // Image to display balls.
14     int w, h; // Width an height of image.
15     Graphics2D gi; // Graphics object to draw balls.
16     float r; // Radius of balls.
17     int n; // Number of balls.
18     Thread thread; // Thread that runs simulation loop.
19     int m; // Size of the hashtable
20
21 /**
22  * Initializes the simulation.
23  *
24  * @param w width of simulation window.
25  * @param h height of simulation window.
26  * @param n number of balls.
27  * @param r radius of balls.
28  * @param v initial velocity of balls.
29  */
30     public BouncingBallsSimulationImproved(int w, int h, int n, float r, float v, int
m) {
31         this.r = r;
32         this.n = n;
33         this.w = w;
34         this.h = h;
35         this.m = m;
36
37         // Initialize balls by distributing them randomly.
38         balls = new LinkedList<Ball>();
39         hashtable = (LinkedList<Ball>[][])) new LinkedList<?>[m][m];
40         for (int i = 0; i < hashtable.length; i++) {
41             for (int j = 0; j < hashtable[i].length; j++) {
42                 hashtable[i][j] = new LinkedList<Ball>();
43             }
44         }
45         for (int i = 0; i < n; i++) {
46             float vx = 2 * (float) Math.random() - 1;
47             float vy = 2 * (float) Math.random() - 1;
48             float tmp = (float) Math.sqrt((double) vx * vx + vy * vy);
49             vx = vx / tmp * v;
50             vy = vy / tmp * v;
51             Ball b = new Ball(r + (float) Math.random() * (w - 2 * r), r + (float)
Math.random() * (h - 2 * r), vx, vy, r);
52             balls.add(b);
53             hashtable[0][0].add(b);
54         }
55     }
56
57     public Dimension getPreferredSize() {
58         return new Dimension(w, h);
59     }
60
61 /**
62  * Paint the window that displays the simulation. This method is called
63  * automatically by the Java window system as a response to the call to
64  * repaint() in the run() method below.
65  */
66     public void paint(Graphics g) {
67         gi.clearRect(0, 0, w, h);
68
69         Iterator<Ball> it = balls.iterator();
70         while (it.hasNext()) {
71             Ball ball = it.next();
72             gi.fill(new Ellipse2D.Float(ball.x - r, ball.y - r, 2 * r, 2 * r));
73         }
74     }

```

```

75
76     g.drawImage(img, 0, 0, null);
77 }
78
79 /**
80 * Starts the simulation loop.
81 */
82 public void start() {
83     img = createImage(w, h);
84     gi = (Graphics2D) img.getGraphics();
85     gi.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.
86     VALUE_ANTIALIAS_ON);
87     thread = new Thread(this);
88     thread.run();
89 }
90
91 /**
92 * The simulation loop.
93 */
94 public void run() {
95     // Set up timer.
96     int c = 0;
97     Timer timer = new Timer();
98     timer.reset();
99
100    // Loop forever (or until the user closes the main window).
101    while (true) {
102        for (int i = 0; i < hashtable.length; i++) {
103            for (int j = 0; j < hashtable[i].length; j++) {
104                Iterator<Ball> it1 = hashtable[i][j].iterator();
105                while (it1.hasNext()) {
106                    Ball bl = it1.next();
107                    int a = (int) Math.floor(bl.x / w * (m - 1));
108                    int b = (int) Math.floor(bl.y / h * (m - 1));
109                    if (a != i || b != j) {
110                        it1.remove();
111                        hashtable[a][b].add(bl);
112                    }
113                }
114            }
115        }
116
117        // Move the ball.
118        for (int i = 0; i < hashtable.length; i++) {
119            for (int j = 0; j < hashtable[i].length; j++) {
120                Iterator<Ball> it3 = hashtable[i][j].iterator();
121
122                while (it3.hasNext()) {
123                    Ball ball = it3.next();
124
125                    // Handle collisions with boundaries.
126                    if (ball.doesCollide((float) w, 0.f, -1.f, 0.f))
127                        ball.resolveCollision((float) w, 0.f, -1.f, 0.f);
128                    if (ball.doesCollide(0.f, 0.f, 1.f, 0.f))
129                        ball.resolveCollision(0.f, 0.f, 1.f, 0.f);
130                    if (ball.doesCollide(0.f, (float) h, 0.f, -1.f))
131                        ball.resolveCollision(0.f, (float) h, 0.f, -1.f);
132                    if (ball.doesCollide(0.f, 0.f, 0.f, 1.f))
133                        ball.resolveCollision(0.f, 0.f, 0.f, 1.f);
134
135                    //Handle collisions with other balls.
136                    for (int q = i - 1; q < i + 2; q++) {
137                        for (int w = j - 1; w < j + 2; w++) {
138                            if (q >= 0 && w >= 0 && q < hashtable.length && w <
hashtable[q].length) {
139                                Iterator<Ball> it2 = hashtable[q][w].iterator();
140                                while (it2.hasNext()) {
141                                    Ball ball2 = it2.next();
142
143                                    //check for collision
144                                    if (ball != ball2 && ball.doesCollide(ball2)
145
146                                }
147                            }
148                        }
149                    }
150                }
151            }
152        }
153    }
154 }

```

```
148             }
149         }
150         //Move the ball
151         ball.move();
152     }
153 }
154 // Trigger update of display.
155 repaint();
156
157 // Print time per simulation step.
158 c++;
159 if (c == 10) {
160     System.out.printf("Timer per simulation step: %fms\n", (float) timer
161 .timeElapsed() / (float) c);
162     timer.reset();
163     c = 0;
164 }
165 }
166 }
167 }
168 }
```

2)