Datenstrukturen und Algorithmen
Übung 6 – Hashing

Musterlösung

Theoretische Aufgaben

1. Betrachten Sie eine Hashtabelle der Grösse $m = 512$ und eine zugehörige Hashfunktion

$$h(k) := \lfloor m(kA \bmod 1) \rfloor,$$

wobei

$$A := (\sqrt{5} - 1)/2.$$

(a) Berechnen Sie die Plätze, auf die die Schlüssel 2019, 2020, 2021, 2022, 2023 abgebildet werden.

k	kA	kA **mod** 1	$m\,(kA$ **mod** $1)$	$h(k)$
2019	1247.811	0.811	415.039	**415**
2020	1248.429	0.429	219.473	**219**
2021	1249.047	0.047	23.906	**23**
2022	1249.665	0.665	340.339	**340**
2023	1250.283	0.283	144.773	**144**

(b) Finden Sie einen Schlüssel, der eine Kollision mit dem Schlüssel 2019 auslöst

Lösung per Ausprobieren (*brute force*) Die 20 ersten Kollisionen: 45, 655, 1032, 1642, 2629, 3239, 3616, 4226, 4836, 5213, 5823, 6200, 6810, 7420, 7797, 8407, 8784, 9017, 9394, 10004.

Systematische Lösung (Bonus-Inhalt, nicht Teil des Vorlesungsstoffes) Wir versuchen ein $\delta \in \mathbb{N}$ zu finden, so dass $h(k) = h(k+\delta)$ (meistens) gilt. Eine hinreichende (aber nicht notwendige) Bedingung für δ finden wir wie folgt:

$$\begin{aligned} & h(k+\delta) \stackrel{!}{=} h(k) \\ \iff & \lfloor m((k+\delta)A \bmod 1) \rfloor = \lfloor m(kA \bmod 1) \rfloor \\ \impliedby & (k+\delta)A \bmod 1 = m(kA \bmod 1) \\ \iff & \delta A = n \text{ für ein } n \in \mathbb{N} \end{aligned}$$

Durch den Rundungsoperator klappt das oft auch abhängig von m und k, wenn diese Gleichung nicht exakt erfüllt ist.

Unser Ziel ist also, ein δ so zu wählen, dass δA möglichst nah an einer ganzen Zahl liegt. Wir können das auch so ausdrücken, dass $\delta \bar{A} \in \mathbb{N}$ exakt erfüllt ist für ein $\bar{A} \approx A$.

Wir approximieren nun A mit einer rationalen Zahl [a].
Eine Folge solcher Approximation können wir per binärer Suche auf einem Stern-Brocat-Baum berechnen.
Gehen wir also nun davon aus, dass wir $p, q \in \mathbb{N}$ kennen, so dass $p/q = \bar{A} \approx A$ gilt. Dann soll gelten $\delta\bar{A} = \delta p/q = n$ für ein $n \in \mathbb{N}$. Wir können das umstellen zu $\delta = nq/p$. Da unser δ auch eine ganze Zahl sein soll, müssen wir also ein n so wählen, dass nq/p ganz ist. Eine einfache Wahl dafür ist $n = p$, damit wäre dann $\delta = pq/p = q$ (und $\bar{A}\delta = p/q * q = p$).
Mit immer genauer werdenen rationalen Approximationen finden wir also Werte für δ, und wenn die Approximation genau genug ist, funktioniert unser δ wie gewünscht.
Rationale Approximation von A: $\bar{A} = p/q = 2584/4181$
(Approximationsfehler: $|p/q - A| \approx 2.558319e - 08$)
$\delta := q = 4181$
$\bar{A} * \delta = 2584.0$
$A * \delta \approx 2584.000107$
$h(2019 + \delta) = h(6200) = 415$

Wir können anhand dieser Methode auch einen Zusammenhang zur Wahl von A schaffen: A ist nämlich keine zufällige Konstante, sondern als die Inverse von ϕ, dem Goldenem Schnitt, gewählt. Diese Zahl ist (für eine geeignete Definition) als „most irrational number“ bekannt, d.h., sie lässt sich sehr schlecht rational approximieren, was natürlich für ihre Inverse genauso gilt.
Dies ist ein Grund, warum es eine gute Wahl für multiplikatives Hashing ist.
Weiterführendes Material für Interessierte:

- Fibonacci Hashing: The Optimization that the World Forgot (or: a Better Alternative to Integer Modulo)
- Going beyond the Golden Ratio
- The Golden Ratio (why it is so irrational) - Numberphile
- The Stern-Brocot tree, Hurwitz's theorem, and the Markoff uniqueness conjecture

[a] Da die Maschinengenauigkeit sowieso begrenzt ist, wird bei der normalen Berechnung (üblicherweise) genaugenommen sowieso auch eine rationale Approximation verwendet.

(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)**

Die Zeit für Löschen und erfolgreiches Suchen bleiben unverändert. Eine erfolglose Suche kann in den meisten Fällen früher abgebrochen werden, bleibt asymptotisch aber gleich. Beim Einfügen sehen wir die größte Änderung: zusätzlich zur regulären konstanten ($\mathcal{O}(1)$) Einfüge-Komplexität kommt die Suche nach der richtigen Einfügeposition in der Liste, was eine Laufzeit von $\mathcal{O}(n)$ (erwartete Laufzeit $\Theta(1 + \alpha)$) hat; hier wird es also asymptotisch schlechter.

Die Idee könnte sich also höchstens dann lohnen, wenn wir sehr oft erfolglos Elemente suchen, und selten Elemente einfügen. Wenn wir allerdings nie „blind“ einfügen, sondern immer erst suchen und dann einfügen, könnten wir die zusätzliche Komplexität

allerdings vermeiden (dafür müssten wir uns den Vorgänger des gefundenen Elements (bei erfolgloser Suche: das letzte Elements) merken).

Üblicherweise vermeidet man allerdings sowieso hohe Füllgrade von Hashtabellen, so dass sich diese Komplexität eher weniger lohnt.

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

Hinweis: Die Spalte i gibt an, wie viele zusätzliche Sondierungsschritte nötig waren, bzw. den Wert von i beim erfolgreichen Einfügen. Blau markierte Felder zeigen den gefundenen Slot, grau markierte Felder zeigen beim Sondieren übersprungene Slots.

Schritt	h'	i	0	1	2	3	4	5	6	7	8	9	10
INSERT(24)	2	0			24								
INSERT(18)	7	0			24					18			
INSERT(13)	2	1			24	13				18			
INSERT(56)	1	0		56	24	13				18			
INSERT(44)	0	0	44	56	24	13				18			
INSERT(7)	7	1	44	56	24	13				18	7		
INSERT(19)	8	1	44	56	24	13				18	7	19	
INSERT(23)	1	3	44	56	24	13	23			18	7	19	
INSERT(33)	0	5	44	56	24	13	23	33		18	7	19	

   (b) quadratischem Sondieren mit $c_1 = 1$ und $c_2 = 3$

Schritt	h'	i	0	1	2	3	4	5	6	7	8	9	10
INSERT(24)	2	0			24								
INSERT(18)	7	0			24					18			
INSERT(13)	2	1			24				13	18			
INSERT(56)	1	0		56	24				13	18			
INSERT(44)	0	0	44	56	24				13	18			
INSERT(7)	7	2	44	56	24				13	18			7
INSERT(19)	8	0	44	56	24				13	18	19		7
INSERT(23)	1	1	44	56	24			23	13	18	19		7
INSERT(33)	0	1	44	56	24		33	23	13	18	19		7

(c) doppeltem Hashing mit $h_2(k) = 1 + (k \bmod (m-1))$

Schritt	h'	h_2	i	0	1	2	3	4	5	6	7	8	9	10
INSERT(24)	2	5	0			24								
INSERT(18)	7	9	0			24					18			
INSERT(13)	2	4	1			24				13	18			
INSERT(56)	1	7	0		56	24				13	18			
INSERT(44)	0	5	0	44	56	24				13	18			
INSERT(7)	7	8	1	44	56	24		7		13	18			
INSERT(19)	8	10	0	44	56	24		7		13	18	19		
INSERT(23)	1	4	1	44	56	24		7	23	13	18	19		
INSERT(33)	0	4	5	44	56	24		7	23	13	18	19	33	

(1 Punkt)

4. Bei der Kollisionsauflösung nach dem Verkettungsschema muss für kollidierende Elemente 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 **a)** ein Element und ein Zeiger oder **b)** zwei Zeiger abgespeichert werden können:

$$\langle flag, element, ptr \rangle \text{ oder } \langle flag, ptr_1, ptr_2 \rangle.$$

Alle Operationen (Einfügen, Löschen, erfolgreiches sowie erfolgloses Suchen) sollten in einer erwarteten Zeit von $\mathcal{O}(1+\alpha)$ ablaufen (keine Analyse notwendig).

Muss die Freiliste dafür doppelt verkettet sein oder genügt eine einfach verkettete Liste? **(2 Punkte)**

Unser Schema besteht aus der Tabelle $A[0 \ldots m-1]$ und der Freiliste $free$ (entweder ausgeführt als Zeiger $free.kopf$ auf das erste Element bzw. auf NIL, oder als Wächterelement $free$.NIL für eine zyklische Liste).

Ein freier Eintrag in der Tabelle hat die Form $\langle 0, prev, next \rangle$, ein besetzter Eintrag hingegen die Form $\langle 1, k, next \rangle$.

Einen Eintrag in die Freiliste einzutragen, bedeutet, das Flag auf 0 zu setzen und mit ihm eine INSERT-Operation auf der Freiliste auszuführen.

Der Einfachheit halber identifizieren wir hier Pointer mit Indizes, d.h. ein Eintrag $next = 3$ entspricht dem Tabelleneintrag $A[3]$.

Wir werden bei den Operationen die folgende Invariante sicherstellen: Das Element in Bucket x ist entweder

a) frei ($\Longrightarrow$ kein Eintrag zu Hash x, x ist in Freiliste enthalten),
b) besetzt ist mit einem Element k mit $h(k) = x$ ($\Longrightarrow$ mindestens ein Eintrag mit Hash x vorhanden), oder
c) besetzt ist mit einem fremden Element $h(k) \neq x$ ($\Longrightarrow$ ebenfalls kein Eintrag zu Hash x vorhanden).

Definieren wir nun die Operationen:

SEARCH(k) Wir prüfen $A[h(k)]$ darauf, welcher der oben genannten Fälle zutrifft und geben in Fall (a) und (c) einen Fehlschlag zurück, und in (b) verfahren wir wie bei der regulären Hashtabelle mit verlinkten Listen.

INSERT(k) Wir unterscheiden die 3 Fälle:

a) $A[h(k)] = \langle 0, prev, next \rangle$ (frei):
Entferne den Eintrag aus der Freiliste (UNLINK), dann setze
$A[h(k)] := \langle 1, k, \text{NIL} \rangle$

b) $A[h(k)] = \langle 1, k', next \rangle$ mit $h(k') = h(k)$:
Wir allokieren ein freies Element, indem wir das erste Element aus der Freiliste unlinken, sei dies Index j. Setze dann $A[j] = \langle 1, k, A[h(k)].next \rangle$ und $A[h(k)].next := A[j]$.

c) $A[h(k)] = \langle 1, k', next \rangle$ mit $h(k') \neq h(k)$:
Wir wollen diesen Fremdeintrag verschieben: Dafür allokieren wir wie zuvor ein neues Element j und setzen $A[j] = A[h(k)]$. Nun muss der Vorgänger in der einfach verlinkten Bucket-Liste auf das neue Element $A[j]$ umgebogen werden; um dieses zu finden, folgen wir der Liste ab $A[h(elem)]$ und setzen den *next*-Eintrag des Vorgängers entsprechen auf $A[j]$. Nun ist $A[h(k)]$ unbenutzt, und wir verfahren wie in Fall a) (ohne den Eintrag zu unlinken, er ist ja nicht in der Freiliste enthalten).

DELETE(x) Wenn $A[x].next = \text{NIL}$, fügen wir $A[x]$ einfach in die Freiliste ein, und setzen den *next*-Eintrag des Vorgängers auf NIL (falls x nicht das erste Listenelement ist)

Ansonsten sei $y = A[x].next$, wir setzen $A[x] := A[y]$ und fügen $A[y]$ in die Freiliste ein — wir ersetzen also x durch seinen Nachfolger y und löschen den Eintrag an Stelle $A[y]$ stattdessen.

Anmerkung: Wir könnten Löschen auch in konstanter Zeit implementieren, indem wir im NIL-Fall nicht den Vorgänger aktualisieren; dann müssen wir an anderer Stelle beim Traversieren von Listen allerdings immer darauf achten, dass der Hash des aktuellen Keys weiter korrekt ist; eine Änderung des Keys wäre also ein implizites terminieren der Liste.

Einfache oder doppelte Verkettung der Freiliste Wenn die Freiliste nur einfach verkettet ist, können wir beim Einfügen (z.B. Fall (a)) ein Element nicht einfach aus der Freiliste entfernen, da wir erst den Vorgänger finden müssten, was nur in $\Omega(m)$ möglich wäre. Für die gewünschte Laufzeit-Komplexität ist also eine doppelt verlinkte Liste nötig.

Praktische Aufgaben

In dieser Übung werden Sie eine Variante von Hashing entwickeln, um räumliche Daten zu verwalten. Als Ausganslage 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 $\mathcal{O}\left(n^2\right)$, 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 eine zweidimensionale ArrayList von verketteten Listen als Hashtabelle. Für die Listen können Sie die Java `LinkedList` Klasse verwenden. Ihre Hashtabelle ist dann vom Typ `ArrayList<ArrayList<LinkedList<Ball>>>` [1] . 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 (Falls sie anderen Code verändert haben, dann geben sie den bitte auch ab — das sollte aber nicht nötig sein). **(3 Punkte)**

```
1  public class BouncingBallsSimulation extends Component
    ↪  implements Runnable
2  {
3  ArrayList<ArrayList<LinkedList<Ball>>> balls;
4  ArrayList<Ball> balls_drawlist;
5
6  static int hashtable_m = 60;
7  // [...]
8
9  public int hashX(Ball ball) {
10   int hash = (int)Math.floor(
11       (hashtable_m - 1) * ball.x / w);
12   return Math.max(Math.min(hashtable_m-1, hash), 0);
13 }
14 public int hashY(Ball ball) {
15   int hash = (int)Math.floor(
16       (hashtable_m - 1) * ball.y / h);
17   return Math.max(Math.min(hashtable_m-1, hash), 0);
18 }
19 public LinkedList<Ball> getBucket(int hx, int hy) {
```

[1] Sie dürfen auch stattdessen eine einzige lineare `ArrayList<LinkedList<Ball>>` verwenden, bspw. mit dem Index $h_y \cdot m + h_x$

```
20    assert hx >= 0 && hx < hashtable_m;
21    assert hy >= 0 && hy < hashtable_m;
22    return balls.get(hy).get(hx);
23  }
24  public LinkedList<Ball> getBucket(Ball ball) {
25    return getBucket(hashX(ball), hashY(ball));
26  }
27
28  // [...]
29
30
31  public BouncingBallsSimulation(int w, int h, int n, float r,
     ↪  float v)
32  {
33    // [...]
34
35    balls = new ArrayList<>();
36    for(int row=0; row<hashtable_m; row++) {
37      var rowlist = new ArrayList<LinkedList<Ball>>();
38      balls.add(rowlist);
39      for(int col=0; col<hashtable_m; col++) {
40        rowlist.add(new LinkedList<>());
41      }
42    }
43
44    // Initialize balls by distributing them randomly.
45    balls_drawlist = new ArrayList<>(n);
46    for(int i=0; i<n; i++)
47    {
48      float vx = 2*(float)Math.random()-1;
49      float vy = 2*(float)Math.random()-1;
50      float tmp = (float)Math.sqrt((double)vx*vx+vy*vy);
51      vx = vx/tmp*v;
52      vy = vy/tmp*v;
53      int color = (i < n / 100 ) ? 1 : 0;
54      var ball = new Ball(
55          r+(float)Math.random()*(w-2*r),
56          r+(float)Math.random()*(h-2*r),
57          vx, vy,
58          r, color);
59      getBucket(ball).add(ball);
60      balls_drawlist.add(ball);
61    }
62
63  }
64
65  public void paint(Graphics g)
66  {
67    // [...]
68
69    for (var ball: balls_drawlist) {
70      gi.setColor(ball.color == 1 ? Color.red : Color.black);
```

```
      gi.fill(new Ellipse2D.Float(ball.x-r, ball.y-r, 2*r, 2*r));
    }
  }

  public void run()
  {
    while(true)
    {
      // move balls to correct buckets according to the current
       ↪  positions

      for(int row=0; row<hashtable_m; row++) {
        var rowlist = balls.get(row);
        for(int col=0; col<hashtable_m; col++) {
          var bucket = rowlist.get(col);
          var it = bucket.listIterator();
          while(it.hasNext())
          {
            Ball ball = it.next();
            int hx = hashX(ball);
            int hy = hashY(ball);
            if (hx != col || hy != row) {
              it.remove();
              getBucket(hx, hy).add(ball);
            }
          }
        }
      }

      // Move balls and handle collisions
      for(int row=0; row<hashtable_m; row++) {
        var rowlist = balls.get(row);
        for(int col=0; col<hashtable_m; col++) {
          var bucket = rowlist.get(col);
          var it = bucket.listIterator();
          while(it.hasNext())
          {
            Ball ball = it.next();

            // Move the ball.
            ball.move();

            // Handle collisions with boundaries.
            if(ball.doesCollide((float)w,0.f,-1.f,0.f))
              ball.resolveCollision((float)w,0.f,-1.f,0.f);
            if(ball.doesCollide(0.f,0.f,1.f,0.f))
              ball.resolveCollision(0.f,0.f,1.f,0.f);
            if(ball.doesCollide(0.f,(float)h,0.f,-1.f))
              ball.resolveCollision(0.f,(float)h,0.f,-1.f);
            if(ball.doesCollide(0.f,0.f,0.f,1.f))
              ball.resolveCollision(0.f,0.f,0.f,1.f);

```

```
122  // Handle collisions with other balls.
123  // In order to only compare a with b once and not again as
124  // b vs. a, we only test our own bucket
125  // (up to the current `ball`), the bucket to the right,
126  // and the buckets in the row below
127            for (int other_row = row;
128                 other_row < Math.min(row+1, h-1);
129                 other_row++)
130            {
131              var other_rowlist = balls.get(other_row);
132              // avoid bucket to the left, see comment above:
133              int startcol = Math.max(0,
134                      other_row == row ? col : col-1);
135              for (int other_col = startcol;
136                   other_col < Math.min(col+1, w);
137                   other_col++)
138              {
139                var other_bucket = other_rowlist.get(other_col);
140
141                var other_it = other_bucket.listIterator();
142                while(other_it.hasNext())
143                {
144                  Ball other_ball = other_it.next();
145                  if (other_ball == ball) {
146      // we are in the same bucket and the balls are identical
147      // skip this and the remaining balls in this bucket:
148                    break;
149                  }
150                  if(ball.doesCollide(other_ball))
151                    ball.resolveCollision(other_ball);
152                }
153              }
154            }
155
156          }
157        }
158      }
159      repaint();
160      // [...]
161    }
162  }
163  }
```

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)**