Datenstrukturen und Algorithmen: Übungsblatt #8

Abgabe am 21. Juni 2018

Finn Hess (378104), Jan Knichel (377779), Paul Orschau (381085)

15.06.2018

Finn Hess (378104), Jan Knichel (377779), Paul Orschau (381085)

Aufgabe 1

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

3	6	2	4	5	5	0	*
1	0	2	1	1	2	1	
1	1	2	4	5	7	8	
		2					

3	6	2	4	5	5	*	*
1	0	2	1	1	2	1	
0	1	2	4	5	7	8	
0		2					

3	6	2	4	5	*	*	*
1	0	2	1	1	2	1	
0	1	2	4	5	6	8	
0		2				5	

3	6	2	4	*	*	*	*
1	0	2	1	1	2	1	
0	1	2	4	5	5	8	
0		2			5	5	

3	6	2	*	*	*	*	*
1	0	2	1	1	2	1	
0	1	2	4	4	5	8	
0		2		4	5	5	

3	6	*	*	*	*	*	*
1	0	2	1	1	2	1	
0	1	1	4	4	5	8	
0	2	2		4	5	5	

3	*	*	*	*	*	*	*
1	0	2	1	1	2	1	
0	1	1	4	4	5	7	
0	2	2		4	5	5	6

*	*	*	*	*	*	*	*
1	0	2	1	1	2	1	
0	1	1	3	4	5	7	
0	2	2	3	4	5	5	6

Aufgabe 2

Teil c)
Eine erfolglose Suche wird nie terminieren, da immer wieder eine zufällige Slotnummer ermittelt wird, der gesuchte Schlüssel aber nie gefunden wird. Daher ist die Average-Case-Komplexität nicht ermittelbar bzw. in $\Theta(\infty)$.

Teil d)

- **Einfache Berechenbarkeit:** Eine zufällige Slotnummer lässt sich effizient ermitteln, daher hat randomisiertes Hashing eine einfache Berechenbarkeit.
- **Surjektivität:** Früher oder später wird jeder Slot gefüllt sein, denn bei jedem Einfügevorgang hat jeder Slot die Chance gefüllt zu werden. Je voller das Array, desto größer die Chance, bis schließlich nur noch ein freies Feld übrig ist, welches dann sicher beim nächsten Einfügen gefüllt werden muss.
- **Gleichverteilung auf alle Indizes:** Durch die zufällige Wahl der Slotnummer werden die Schlüssel optimal verteilt.
- **Möglichst breite Verteilung ähnlicher Schlüssel auf die Hashtabelle:** Da die gewählte Slotnummer in keiner Weise vom Schlüssel abhängt, ist sichergestellt, dass ähnliche Schlüssel genau so zufällig verteilt werden, wie völlig verschiedene Schlüssel. Die Verteilung ist also gut.

Zusätzliches Kriterium: **Terminierung**
Der Algorithmus braucht mit zunehmendem Füllgrad immer länger, um neue Elemente einzufügen, weil er zum Beispiel nur eine Chance von $\frac{1}{m}$ hat, den letzten freien Slot auf Anhieb zu füllen. Abhängig von der Größe der Hashtabelle kann der letzte Einfügevorgang also ewig dauern! Ist die Tabelle voll, so terminiert der Algorithmus beim nächsten Einfügen gar nicht, was sehr schlecht ist.

Aufgabe 3

Funktion `f`:

- **Einfache Berechenbarkeit:** Vergleichsweise schlecht, da Divisionen "teuerer" sind als z.B. Modulorechnung oder Multiplikationen.
- **Surjektivität:** Sehr schlecht, denn es werden sehr viele Werte nie getroffen. Die Wertemenge ist $\{0, 1, 2, 3, 5, 10\}$, also werden die Felder $\{4, 6, 7, 8, 9\}$ nie belegt!
- **Gleichverteilung auf alle Indizes:** Sehr schlecht, nicht nur aufgrund der Argumente zur Sujektivität, sondern auch weil alle Zahlen $k > 10$ auf die 0 abgebildet werden.
- **Möglichst breite Verteilung ähnlicher Schlüssel auf die Hashtabelle:** Diese Eigenschaft mag für die ersten beiden Zahlen $k = 1$ und $k = 2$ noch gegeben sein, aber schon ab $k = 3$ liegen alle Werte entweder direkt nebeneinander oder sogar auf dem selben Feld, siehe oben.

Funktion `g`:

- **Einfache Berechenbarkeit:** Die diskrete Exponentialfunktion ist auch für große Exponenten effizient berechenbar.

Aufgabe 3 wird auf nächster Seite fortgesetzt...

- **Surjektivität:** Nicht surjektiv, da die Funktion niemals auf die 0 abbilden wird. Beweis: Alle Zahlen, die 2^x für steigende x erzeugen kann, haben offensichtlich stets eine Primfaktorzerlegung in der nur 2er vorkommen, insbesondere ist keine der Zahlen durch 101 teilbar. Dies ist aber Vorraussetzung dafür, dass $a \mod 101 = 0$ für ein $a := 2^x$ gelten kann.
- **Gleichverteilung auf alle Indizes:** Wir haben keine Begründung gefunden, gehen aber davon aus, dass die Indizes nicht gleichmäßig verteilt werden.
- **Möglichst breite Verteilung ähnlicher Schlüssel auf die Hashtabelle:** Diese Eigenschaft ist gegeben, denn wenn man x um eins erhöht, ändert sich die Zahl, die modulo genommen wird, dramatisch.

Funktion `h`: Entspricht Divisionsmethode aus der Vorlesung mit $m = 11$.

- **Einfache Berechenbarkeit:** Modulo ist sehr effizient berechenbar.
- **Surjektivität:** Surjektiv, da bereits die ersten 10 Werte alle Indizes belegen.
- **Gleichverteilung auf alle Indizes:** Wenn jeder zulässige Wert einmal abgespeichert werden würde, enhielte jeder Index im Wertebereich etwa 9 Schlüssel, und auch für größere Ausgangsmengen bleiben die Schlüssel gleichmäßig verteilt, da immer von einem Wert zum nächsten gesprungen wird.
- **Möglichst breite Verteilung ähnlicher Schlüssel auf die Hashtabelle:** Sehr schlecht, da zwei benachbarte Schlüssel stets in benachbarten Indizes landen, außer wenn der letzte Listenplatz getroffen wurde.

Funktion `i`:

- **Einfache Berechenbarkeit:** Modulo ist zwar sehr effizient berechenbar, aber durch die Division durch 2 ist die Funktion etwas schwerer zu berechnen.
- **Surjektivität:** Surjektiv, da bereits die ersten 100 Werte alle Indizes belegen.
- **Gleichverteilung auf alle Indizes:** Die Schlüssel sind gleichmäßig verteilt, da immer von einem Wert zum nächsten gesprungen wird. Auch wenn eine zufällige Folge von natürlichen Zahlen abgespeichert wird, sind die resultierenden Indizes zufällig verteilt.
- **Möglichst breite Verteilung ähnlicher Schlüssel auf die Hashtabelle:** Sehr schlecht, da zwei benachbarte Schlüssel stets in benachbarten Indizes landen, außer wenn der letzte Listenplatz getroffen wurde. Bei dieser Funktion landen sogar immer zwei Zahlen hintereinander auf dem selben Index, da durch 2 geteilt und dann abgerundet wird!

Aufgabe 4

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]
5										
5		21								
5		21	23							
5	17	21	23							
5	17	21	23	11						
5	17	21	23	11	7					
5	17	21	23	11	7	1				

Aufgabe 5

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]
5										
5		21								
5		21			23					
5	17	21			23					
5	17	21		11	23					
5	17	21	7	11	23					
5	17	21	7	11	23			1		

Aufgabe 6

Teil a)

```
1  void function(int n) {
2    k = 2/3 * n + 1;
3
4    for (int i=0; i<k; i++) {
5      print();
6    }
7  }
```

Teil b)

```
1  void function(int n) {
2    for (int i=0;i<n;i++) {
3      for (int j=0;j<n;j++) {
4        print();
5        for (int k=0;k<n;k++) {
6          print();
7        }
8      }
9    }
10 }
```

Aufgabe 6 wird auf nächster Seite fortgesetzt...

Teil c)

```
1  void function(int n) {
2    a = 1;
3    b = 1;
4    for (int i=0;i<n;i++) {
5      a = a*3;
6      b = b*n;
7    }
8    k = a + b;
9
10   for (int i=0;i<k;i++) {
11     print();
12   }
13 }
```

Teil d)

```
1  void function(int n) {
2    k = 0;
3    while (n>1) {
4      n = n/2;
5      k++;
6    }
7
8    while (k>1) {
9      k = k/2;
10     print();
11   }
12 }
```

Teil e)

```
1  void function(int n) {
2    k = 0;
3    while (n > 1) {
4      n = n/2;
5      k++;
6    }
7
8    for (int i=0;i*i*i<k;i++) {
9      print();
10   }
11 }
```