

Übungsserie 6

Datenstrukturen & Algorithmen

Universität Bern
Frühling 2018

Übungsserie 6

- > **Hashtables**
 - > 5 theoretische Aufgaben
 - > 2 praktische Aufgaben
 - Partikelsimulation mit Hashtable beschleunigen
 - > Abgabe am 12. April
 - > Poolstunde 9. April 1700 – 1800
-

Theoretische Aufgaben

- > **Aufgabe 1** Hashfunktionen
 - Schlüssel Hashtable einfügen

 - > **Aufgabe 2** Lohnt es sich, verkettete Listen für Kollisionsauflösung zu sortieren?
 - Zeitaufwand für:
 - Einfügen (erfolgreich und nicht erfolgreich)?
 - Suchen?
 - Löschen?
 - Sortiereigenschaft aufrechterhalten!
 - Zugriff auf Element in Linkedlist: $O(n)$

 - **Kein Pseudocode schreiben!**
-

Theoretische Aufgaben

- > **Aufgabe 3** Sondieren mit offener Adressierung
 - Hashfunktion $\text{hash}(\text{key}, i)$, Schlüssel key , Sondierzahl i
 - Strategien durchspielen und illustrieren für:
 - Lineares Sondieren
 - Quadratisches Sondieren
 - Doppeltes Hashing
 - Buch Kapitel 11.4
-

Theoretische Aufgaben

- > **Aufgabe 4** Reihenfolge vs. Suchzeit
 - Schlüsselmenge (k_1, \dots, k_n) wird bei Initialisierung in Hashtable eingefügt: $insert(k_1), insert(k_2), \dots, insert(k_n)$
 - **Lineares Sondieren** Hat Reihenfolge Einfluss auf (durchschnittliche) Suchzeit bei **erfolgreichem** Suchen?
 - Einfluss der Reihenfolge auf Hashplätze?
-

Theoretische Aufgaben

> Aufgabe 4 Lösungsansatz

— Durchschnittliche Suchzeit (erfolgreiche Suche):

$$\frac{1}{n} \sum_{i=1}^n \text{Suchzeit}(k_i)$$

— Es reicht, zu zeigen:

– $(k_1, \dots, k_j, k_{j+1}, \dots, k_n)$ und $(k_1, \dots, \underbrace{k_{j+1}, k_j}_{\text{vertauscht}}, \dots, k_n)$ führen zu derselben Suchzeitsumme

— **Begründung** Jede Permutation kann mit endlicher Anzahl Nachbarvertauschungen beschrieben werden

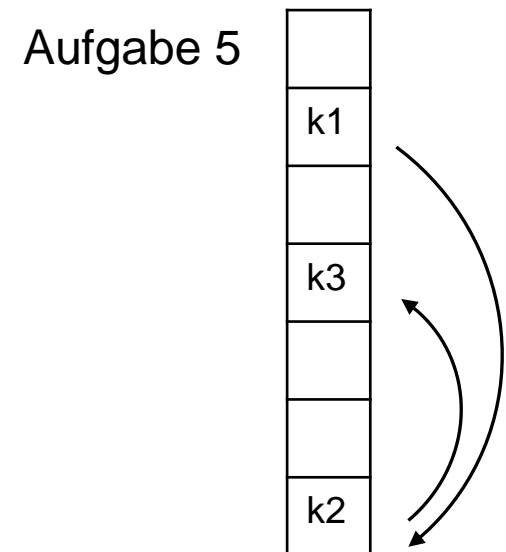
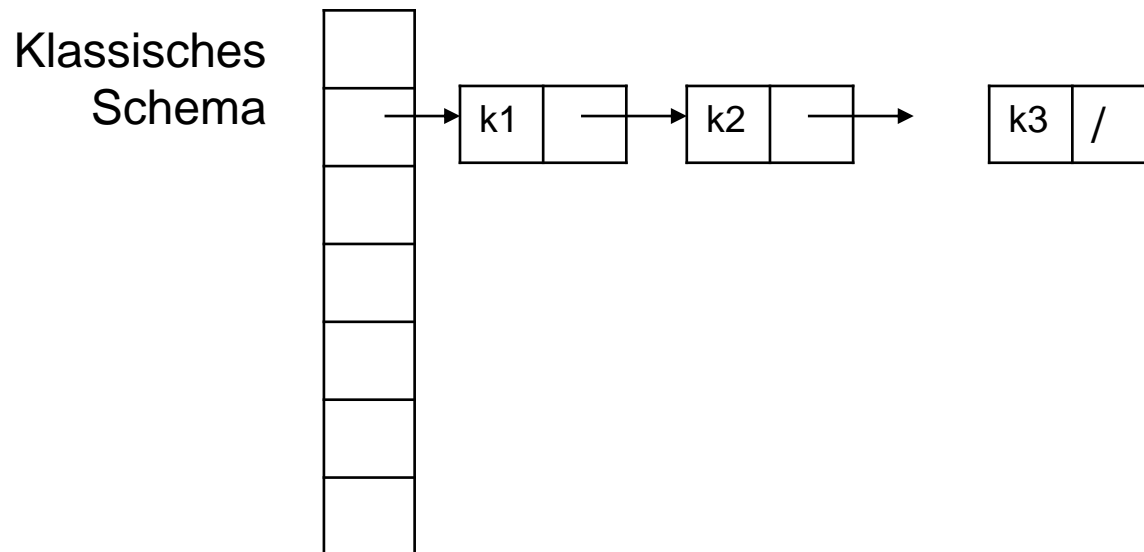
Theoretische Aufgaben

- > **Aufgabe 4** Begründe, warum bei linearem Sondieren:
 - nach Vertauschen von k_j und k_{j+1} dieselben Einträge in der Hashtable belegt sind
 - Der Suchpfad nach den Elementen $k_1, \dots, k_{j-1}, k_{j+2}, \dots, k_n$ gleich bleibt
 - Die Suchpfade für k_j und k_{j+1} **in der Summe** gleich bleiben
 - **Kein Induktionsbeweis, kein Pseudocode, Begründung!**

 - > Quadratisches Sondieren? (Gegenbeispiel)
-


Theoretische Aufgaben

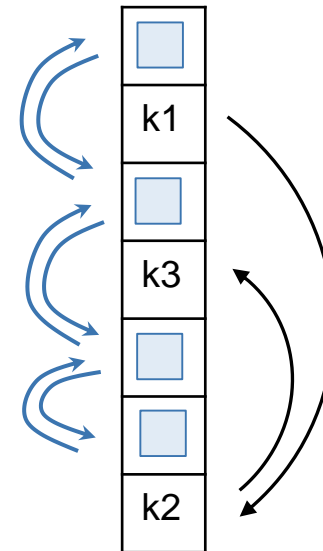
- > **Aufgabe 5** Alternative zur offenen Adressierung:
- Kollisionsauflösung nach Verkettungsschema mit einfacher Verkettung
 - **Achtung** LinkedLists sollen **innerhalb der Tabelle** geführt werden



Theoretische Aufgaben

> Aufgabe 5 Alternative zur offenen Adressierung:

- **Problem** Freien Platz finden
 - Tabelle durchsuchen ineffizient!
- **Lösung** Doppelt verkettete Liste mit freien Plätzen: ()



$$\text{hash}(k1) = \text{hash}(k2) = \text{hash}(k3)$$

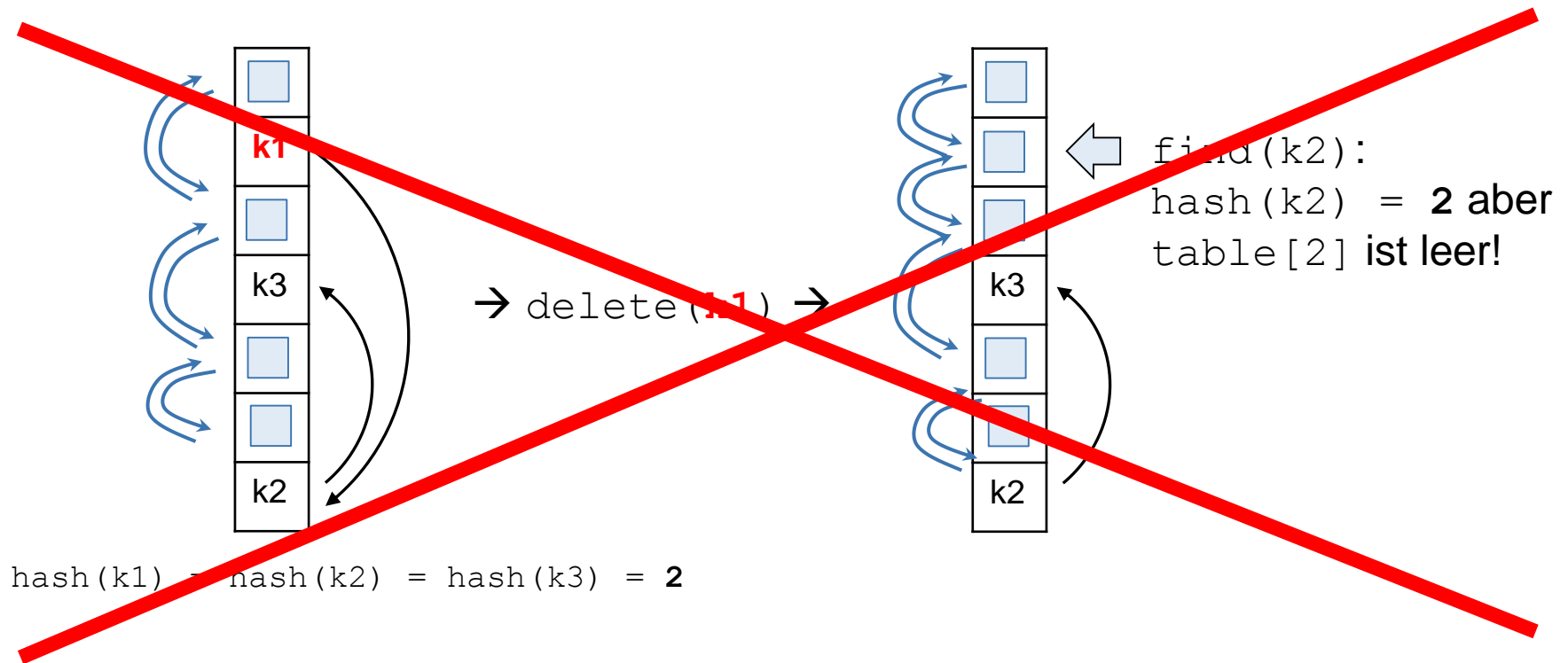
Theoretische Aufgaben

- > **Aufgabe 5** Alternative zur offenen Adressierung:
 - Beschreibe die Operationen:
 - `insert`
 - `delete`
 - `search`
 - Operationen sollen in $O(1)$ laufen!
 - Fallunterscheidungen (z.B. `insert`) :
 - Platz an Hashwert frei vs. Belegt
 - Falls belegt: Gehört das gespeicherte Element wirklich dorthin?
 - **Kein Pseudocode**, genaue Beschreibung
-

Theoretische Aufgaben

> Aufgabe 5 Alternative zur offenen Adressierung:

- Vorsicht bei `delete`:
- Immer ein Element mit korrektem Hash an korrekter Position!



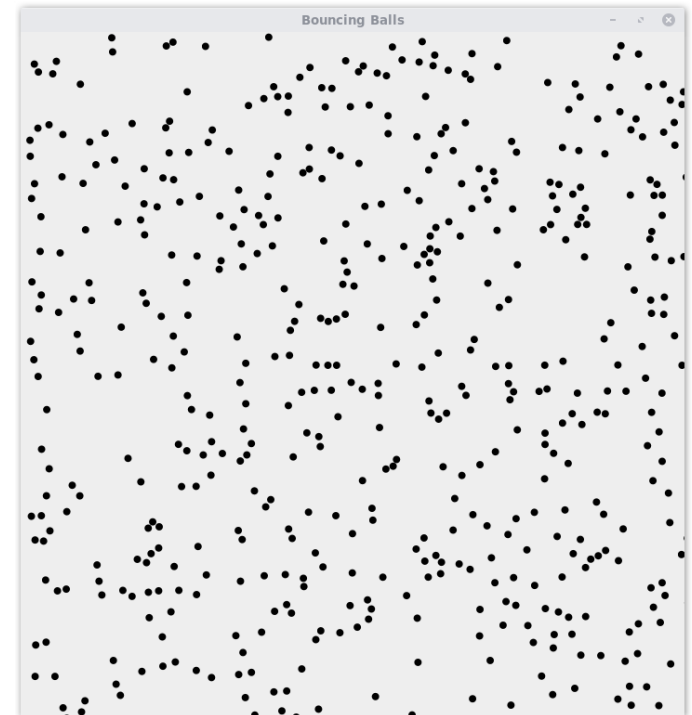
Theoretische Aufgaben

- > **Aufgabe 5** Alternative zur offenen Adressierung:
 - > **Datenstruktur** Freiliste ist doppelt verkettete Liste
 - Freier Platz enthält zwei Pointer (ist in Freiliste)
 - Besetzter Platz enthält einen Schlüssel und einen Pointer auf das nächste Element (evtl. `null`)
-

Praktische Aufgaben

- > **Gegeben** Partikelsimulation
 - Kollidierende Bälle
 - Naive Kollisionsdetektionsimplementation
 - Testet alle Ballpaare auf Kollision: $O(n^2)$

- > **Aufgaben** Beschleunigen
 - Kollisionsdetektion mit Hashtable
 - Verschiedene Parameter testen

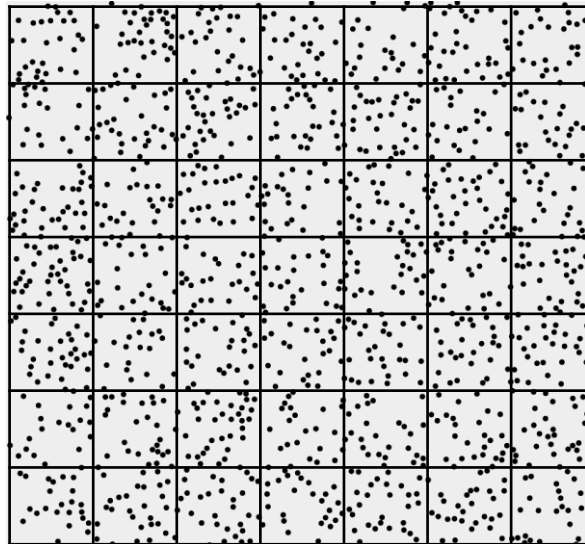


Praktische Aufgaben

> Beschleunigung mittels Hashtable

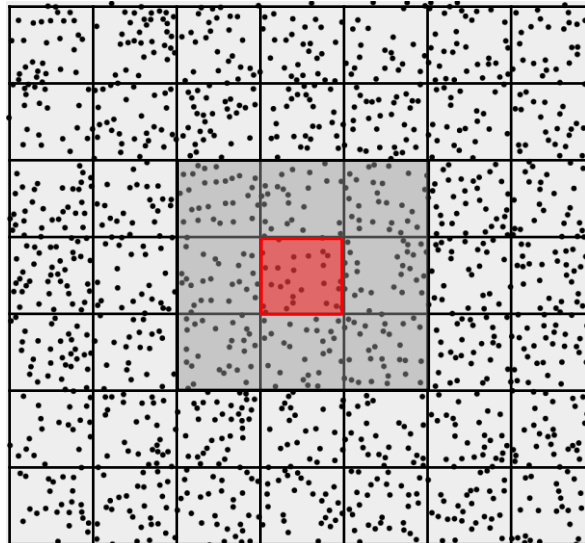
- 2D Hashtabelle `LinkedList[][]`
- Speicherort eines Punkts p ist Funktion seiner Position $(p.x, p.y)$:

$$t[h_x][h_y] = p, \quad h_x = \text{hash}(p.x), \quad h_y = \text{hash}(p.y)$$



Praktische Aufgaben

- > Beschleunigung mittels Hashtable
 - In jeder Iteration
 - Platziere jeden Ball an richtige Hashtableposition
 - Teste nur Punkte derselben Zelle und in direkten Nachbarzellen



Praktische Aufgaben

> Vorgegebener Code

- `BouncingBallsSimulation.java` Zu bearbeitende Klasse
 - Methode `run()` anpassen
 - `BouncingBalls.java` main
 - `Ball.java` Ballklasse, beinhaltet Methoden zur Berechnung von Kollision, Ballbewegung etc.
 - `Timer.java`
-

BouncingBallsSimulation::run()

Schleife über alle Bälle

```
// Iterate over all balls.  
while(it.hasNext())
```

```
{
```

```
    Ball ball = it.next();
```

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

Testet auf Kollision

```
    // 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);
```

Kollision mit Wand

```
    if(ball.doesCollide(0.f,0.f,1.f,0.f))  
        ball.resolveCollision(0.f,0.f,1.f,0.f);
```

Neue Richtung und
Geschwindigkeit

```
    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);
```

Verbessern!

Ball-Ball Kollision

$O(n^2)$

```
    // Handle collisions with other balls
```

```
    Iterator<Ball> it2 = balls.iterator();
```

```
    Ball ball2 = it2.next();
```

```
    while(ball2 != ball)
```

```
    {
```

```
        if(ball.doesCollide(ball2))
```

```
            ball.resolveCollision(ball2);
```

```
            ball2 = it2.next();
```

```
    }
```

```
    // Trigger update of display.  
    repaint();
```