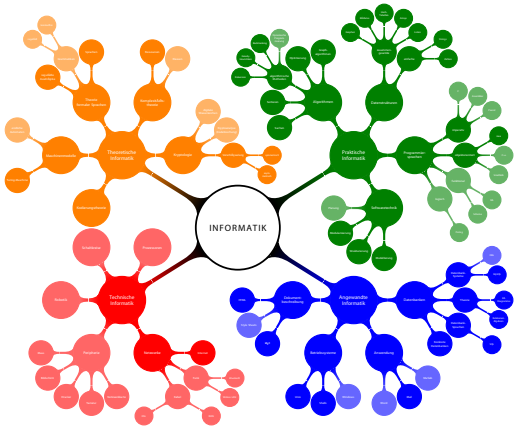
UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR
THEORETISCHE INFORMATIK

Kapitel 29
Hashing
Wirklich schnelles Suchen

Vorlesung Einführung in die Informatik 1 vom 13. Februar 2014 von Till Tantau



Lernziele von Kapitel 29

1. Konzept des Hashwerts verstehen
2. Einfache Hashfunktionen kennen und implementieren können
3. Einfache Hashverfahren kennen und implementieren können

Gliederung von Kapitel 29

Schon nicht schlecht.

Es gibt viele Problemstellungen, bei denen wir Objekte verwalten müssen.

- Verwaltung von Fachschaftslisten
- Verwaltung von Dateisystemen
- Verwaltung von Molekül- und Gendaten

Dazu kann man *Arrays*, *Listen* oder *Suchbäume* verwenden.
Der Zeitverbrauch war (mit h irgendwo zwischen $\log n$ und n):

Implementation	Suchen	Einfügen	Löschen
sortierter Array	$O(\log n)$	$O(n)$	$O(n)$
unsortierte Liste	$O(n)$	$O(1)$	$O(n)$
Suchbaum	$O(h)$	$O(h)$	$O(h)$

Das Versprechen der Hashtabellen

In *Hashtabellen* kann man

- in Zeit $O(1)$ etwas einfügen,
- in Zeit $O(1)$ etwas löschen,
- in Zeit $O(1)$ etwas suchen.

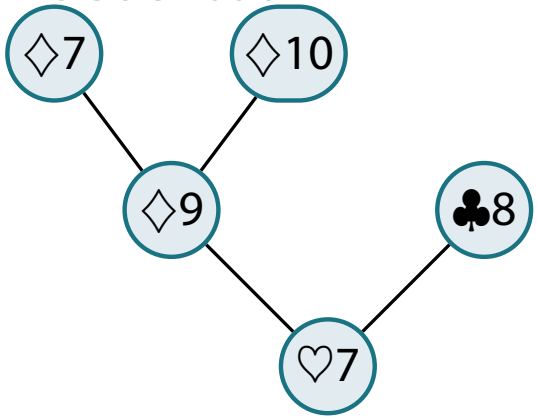
Man sollte allerdings das Kleingedruckte lesen.

Die Verwaltung von Spielkarten.

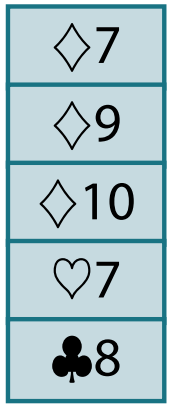
Problemstellung

Wir wollen ein Skatprogramm schreiben. Dazu wird eine Datenstruktur benötigt, die die Hand eines Spielers speichert.

Als Suchbaum



Als sortierter Array



Die erste Idee beim Hashing.

- Es gibt nur 32 Spielkarten.
- Wir könnten also einfach einen Array der Größe 32 anlegen.
- An Stelle 0 kommt dann (falls vorhanden) das Objekt für die ♢7.
- An Stelle 1 kommt dann (falls vorhanden) das Objekt für die ♢8.
- Und so weiter.

Die zweite Idee beim Hashing.

- In einer Hand kommen nur 10 Karten vor.
- Wir reservieren deshalb nur 10 Positionen im Array.
- Dann benutzen wir dieselbe Position für verschiedene Karten.
  Beispiel: Alle Siebenerkarten an Stelle 0, alle Achterkarten an Stelle 1, usw.
- Wenn dann zwei Karten an dieselbe Stelle sollen, so tut man etwas Schlaues.

Die allgemeine Problemstellung beim Hashing.

- Es ist eine Datenstruktur zur Verwaltung von Objekten gesucht.
- Die Objekte entstammen einem bekannten *Universum*.
- Die Objekte werden in einem *Array fester Größe* gespeichert.
- Jedem Objekt wird durch eine *Hashfunktion* eine Position in diesem Array zugeordnet.
- Hashverfahren unterscheiden sich danach
  - wie Hashfunktion berechnet wird und
  - was passiert, wenn zwei Objekten dieselbe Stelle im Array zugeordnet wird (Kollisionsauflösung).

Wie können wir einen geeignete Hashwert bestimmen?

- Bei Karten ist es einfach, Kartenwerte auf Arraypositionen abzubilden.
- Bei Strings ist dies schon wesentlich schwieriger.
- Ebenfalls schwierig erscheint dies bei einem Objekt vom Typ `Atom`, bestehend aus einer Ordnungszahl und 3D-Koordinaten.

Wünschenswerte Eigenschaften von Hashfunktionen

1. Die Hashfunktion sollte *einfach und schnell* zu berechnen sein.
2. Sie sollte die Werte möglichst *gleichmäßig* auf die möglichen Hashwerte verteilen.
3. Sie sollte die Werte möglichst *zufällig* auf die möglichen Hashwerte verteilen.

5-Minuten-Aufgabe

Zur Übung

Eine Hashtabelle habe die Größe 100.
Entwerfen Sie Hashfunktionen für folgende Arten von Objekten:

1. `double` Zahlen
2. Strings

Standardmethodik bei Hashfunktionen

Hashing geschieht oft in zwei Schritten:

1. Eine Funktion wandelt Objekte in (eventuell sehr große) natürliche Zahlen um.
2. Eine zweite Funktion bildet beliebige Zahlen auf Werte im Intervall $[0, T-1]$ ab, wobei T die Größe der Hashtabelle ist.

Beispiel

Ein String bestehe aus den Zeichen $a_0a_1a_2$. Die ASCII-Werte der Zeichen seien 43, 68 und 100. Dann kann der String auf die Zahl

$$43 + 256 \cdot 68 + 256 \cdot 256 \cdot 100 = 6571051$$

abgebildet werden.

Von der Zahl zum Tabelleneintrag

Wie sollte man nun eine Zahl n auf das Intervall $[0, T-1]$ abbilden?

Die Modulo-Methode

Man bildet n auf $n \bmod T$ ab.

Die Multiplikations-Methode

Man wählt eine »krumme« Zahl φ und bildet n auf $\lfloor T \cdot (\varphi n - \lfloor \varphi n \rfloor) \rfloor$ ab.

Probleme bei Hashfunktionen

Zur Diskussion

Welche Probleme können bei der Modulo-Methode und welche bei der Multiplikations-Methode entstehen?
Zur Erinnerung:

Wünschenswerte Eigenschaften von Hashfunktionen

1. Die Hashfunktion sollte *einfach und schnell* zu berechnen sein.
2. Sie sollte die Werte möglichst *gleichmäßig* auf die möglichen Hashwerte verteilen.
3. Sie sollte die Werte möglichst *zufällig* auf die möglichen Hashwerte verteilen.

Erfahrungen aus der Praxis.

- Bei der Modulo-Methode sollte T eine Primzahl sein und keinesfalls eine Zweier- oder Zehnerpotenz.
- Bei der Multiplikations-Methode sollte φ gleich dem goldenen Schnitt sein.
- Die Berechnung eines guten Hashwertes ist eine schwarze Kunst und wird in der Regel falsch gemacht.

Zurück zum Ausgangsproblem

- Wir haben nun eine Hashtabelle (also einen Array) mit T Stellen angelegt.
- Unsere Objekte werden mittels einer Hashfunktion h auf Zahlen zwischen 0 und $T - 1$ gleichmäßig abgebildet.
- Wir wollen also Objekt x an Stelle $h(x)$ speichern.
- Was aber sollen wir tun, wenn zwei unterschiedliche Objekte x und y an derselben Stelle $h(x) = h(y)$ gespeichert werden sollen?

Erstes Verfahren zur Kollisionsauflösung: Verkettete Listen

- In den Arrayfeldern speichern wir *nicht direkt die Werte*.
- Vielmehr ist jedes Arrayfeld der Anfang einer *verketteten Liste*.
- Alle Objekte, die von der Hashfunktion auf dasselbe Feld abgebildet werden, werden in der Liste des Feldes gespeichert.
- *Suchen in dieser Liste* ist zwar *langsam*, aber da Kollisionen selten sind, ist die Liste sehr *kurz*.

Beispiel einer Hashtabelle mit verketteten Listen

Eckdaten der verketteten Hashtabelle

Tabellengröße $T = 5$

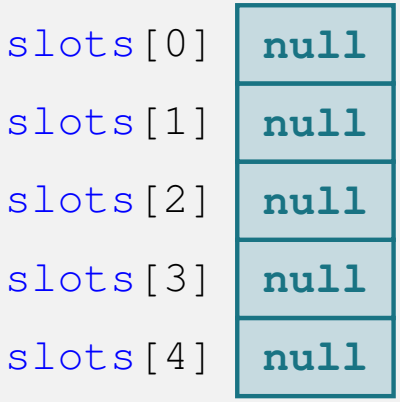
Objekte Spielkarten

Karte als Zahl Kartenwert (Joker = 0) plus Farbe mal 14.

Hashfunktion Karte als Zahl modulo T.

Einfügen von ♢2 ($h = 2$), ♠A ($h = 4$), ♣10 ($h = 2$).

1

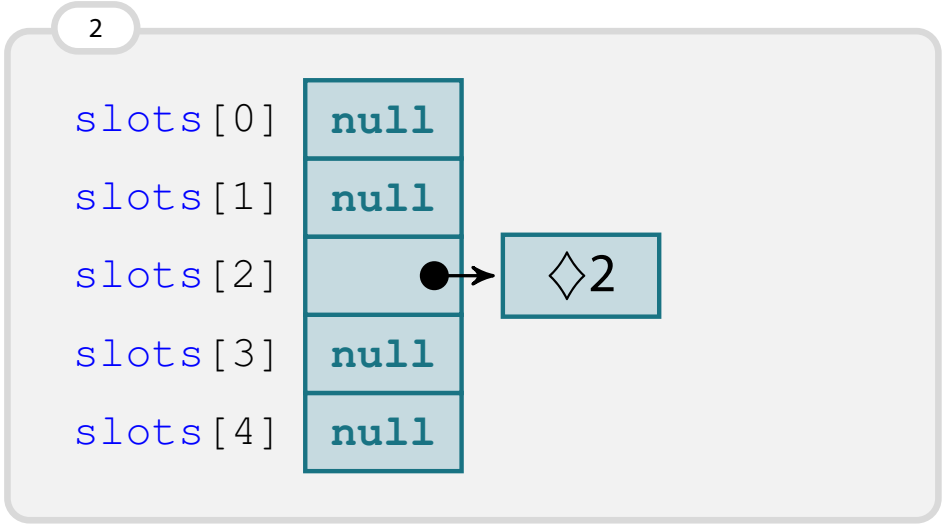


Beispiel einer Hashtabelle mit verketteten Listen

Eckdaten der verketteten Hashtabelle

Tabellengröße	$T = 5$
Objekte	Spielkarten
Karte als Zahl	Kartenwert (Joker $= 0$) plus Farbe mal 14.
Hashfunktion	Karte als Zahl modulo T.

Einfügen von ♢2 ($h = 2$), ♠A ($h = 4$), ♣10 ($h = 2$).

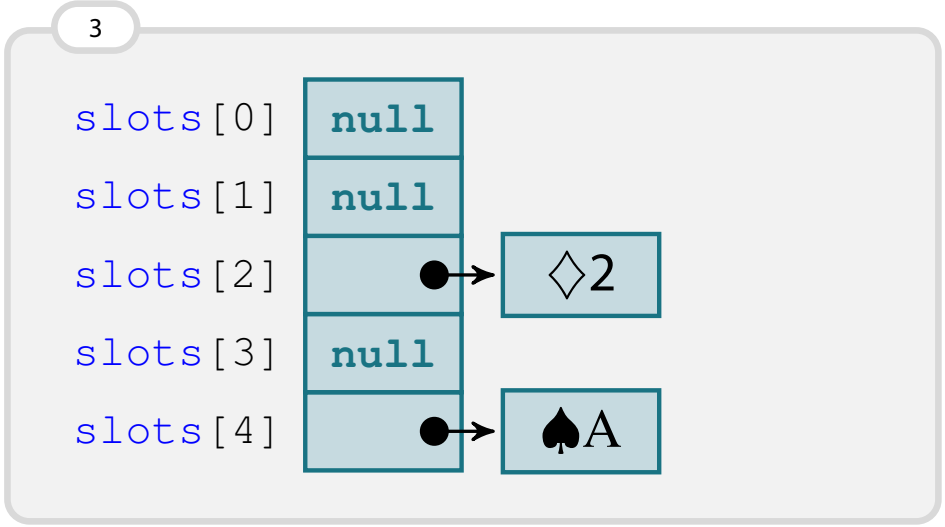


Beispiel einer Hashtabelle mit verketteten Listen

Eckdaten der verketteten Hashtabelle

Tabellengröße	$T = 5$
Objekte	Spielkarten
Karte als Zahl	Kartenwert (Joker $= 0$) plus Farbe mal 14.
Hashfunktion	Karte als Zahl modulo T.

Einfügen von ♢2 ($h = 2$), ♠A ($h = 4$),
♣10 ($h = 2$).

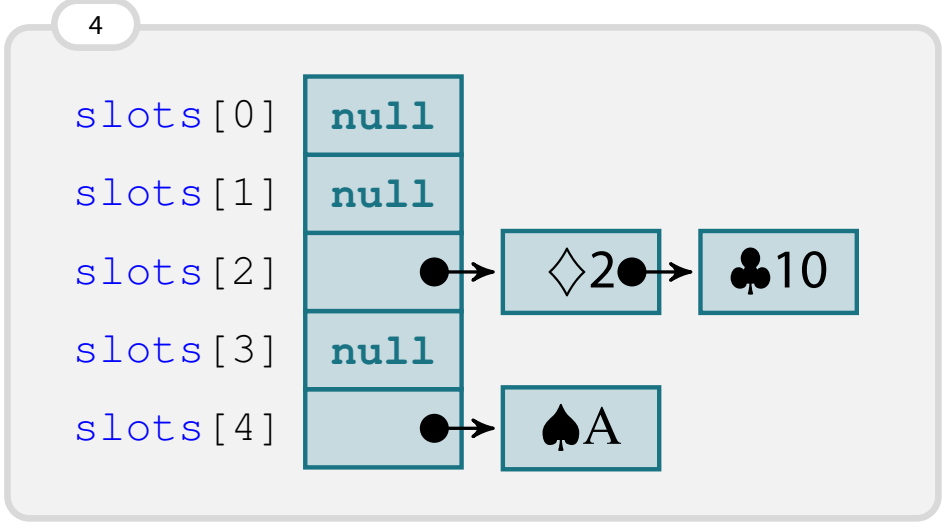


Beispiel einer Hashtabelle mit verketteten Listen

Eckdaten der verketteten Hashtabelle

Tabellengröße	$T = 5$
Objekte	Spielkarten
Karte als Zahl	Kartenwert (Joker $= 0$) plus Farbe mal 14.
Hashfunktion	Karte als Zahl modulo T.

Einfügen von ♢2 ($h = 2$), ♠A ($h = 4$), ♣10 ($h = 2$).



Java-Code einer Hashtabelle für die Fachschaft

```
class Cell {
  Studie studie;
  Cell   next;

  Cell (Studie s) { this.studie = s; }
}

class HashTable {
  private int    T;
  private Cell[] slots;

  HashTable (int size) {
    this.T = size;
    this.slots = new Cell[this.T];
  }

  Studie search (String name) {...}
  void   add    (Studie s)    {...}
  void   remove (String name) {...}
}
```

Java-Code der `search`-Methode

```
class HashTable {
  ...

  Studie search (String name)
  {
    Studie return_me = null;
    Cell cursor      = this.slots[name.hashValue ()
                                  % this.T];
    while (cursor != null)
    {
      if (cursor.studie.name.equals(name)) {
        return_me = cursor;
      }
      cursor = cursor.next;
    }
    return return_me;
  }
}
```

5-Minuten-Aufgabe

Zur Übung

1. Entwerfen Sie den Code der Methode `void add (Studie s)`.
2. Wie würde das Löschen funktionieren?

Zusammenfassung

Hash-Funktion

Eine *Hash-Funktion h* bildet beliebige Objekte auf Zahlen in einem Intervall $[0, T-1]$ ab, wobei T die Größe der Hashtabelle ist.
Eine gute Hash-Funktionen ist, eine »Binärdarstellung« des Objektes zu betrachten, diese als Zahl aufzufassen und dann modulo einer Primzahl T zu rechnen.

Hash-Tabelle

Eine Hash-Tabelle ist ein Array der Größe T. In ihm wird ein Objekt x an der Stelle $h(x)$ gespeichert. Kommt es zu einer *Kollision*, so kann man die kollidierenden Elemente in einer verketteten Liste speichern.
»Wenn alles gut geht«, so kann man in einer Hash-Tabelle in Zeit $O(1)$ sowohl Suchen, Einfügen wie Löschen.

Zusammenfassung

T. H. Cormen, C. E. Leiserson, R. L. Rivest.
Introduction to Algorithms, The MIT Press, 1992.
Kapitel 12 (Hash Tables)

R. Pagh und F. F. Rodler.
Cuckoo Hashing,
Proceedings of ESA 2001,
Lecture Notes in Computer Science, vol. 2161, pages 121–, 2001.