Hashing und Hashmaps

Dozent: Prof. Dr. Michael Eichberg

Kontakt: michael.eichberg@dhbw.de, Raum 149B

Version: 1.0

Quelle: Die Folien sind teilweise inspiriert von oder basierend auf Robert Sedgewick und Kevin Wayne, "Algorithms", Addison-Wesley, 4th Edition, 2011 sowie auf Lehrmaterial von Prof. Dr. Ritterbusch

Folien: https://delors.github.io/theo-algo-hashing/folien.de.rst.html

https://delors.github.io/theo-algo-hashing/folien.de.rst.html.pdf

Fehler melden: https://github.com/Delors/delors.github.io/issues

1. Einführung

Suchen in einer Liste

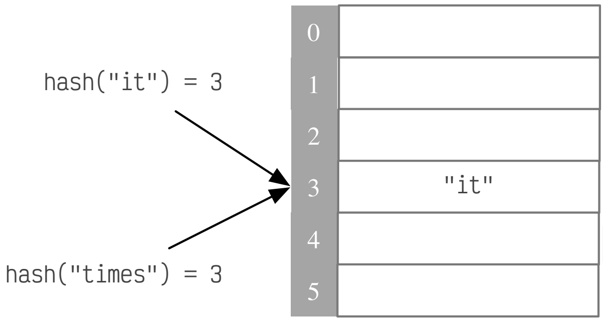
Implementation	Garantie			Durchschnittlicher Fall			Operationen auf den Schlüsseln
	Suchen	Einfügen	Löschen	Suchen	Einfügen	Löschen	
sequentielle Suche (unsortierte Liste)	N	N	N	½ N	N	½ N	equals()
binäre Suche (geordnetes Array)	lg N	N	N	lg N	½ N	½ N	compareTo()
BST [1]	N	N	N	1.39 lg N	1.39 lg N	√ N	compareTo()

Frage

Können wir effizienter suchen?

[1] Binary Search Tree

Hashing - Grundidee



- Die Elemente werden über den Schlüssel indexiert in einer Tabelle gespeichert.

  Der Index ist eine Funktion des Schlüssels.
- Hash-Funktion: Methode zur Berechnung des Array-Index aus dem Schlüssel.

Herausforderungen

1. Berechnung der Hash-Funktion.
2. Gleichheitstest: Methode zur Überprüfung, ob zwei Schlüssel gleich sind.
3. Kollisionsauflösung: Algorithmus und Datenstruktur zur Behandlung von zwei Schlüsseln, die auf denselben Array-Index hindeuten.

Hinweis

Klassischer Kompromiss zwischen Raum und Zeit!

- Keine Platzbeschränkung: triviale Hash-Funktion mit Schlüssel als Index.
- Keine Zeitbeschränkung: triviale Kollisionsauflösung mit sequentieller Suche.
- Raum- und Zeitbeschränkung: Hashing (die reale Welt).

In dem Beispiel ist der Schlüssel das Wort `it`.

Berechnung der Hash-Funktion

Idealistisches Ziel: Die Schlüssel gleichmäßig verwürfeln, um einen Tabellenindex zu erzeugen.

- Effizient berechenbar.
- Jeder Tabellenindex ist für jeden Schlüssel gleich wahrscheinlich.

Die Frage, wie man gute Schlüssel berechnet, ist ein gründlich erforschtes Problem, dass in der Praxis immer noch problematisch ist.

Beispiel 1. Telefonnummern.

Schlecht: die ersten drei bis fünf Ziffern.

Besser: die letzten vier Ziffern.

Beispiel 2. Sozialversicherungsnummer

Schlecht: die ersten beiden Ziffern.

Besser: die letzten Ziffern.

Die ersten beiden Stellen bei der Sozialversicherungsnummer identifizieren den Rentenversicherungsträger.

Praktische Herausforderung: für jeden Schlüsseltyp ist ein anderer Ansatz erforderlich.

Hashfunktionen

Definition

Eine Hashfunktion $h : M \to \mathbb{Z}_n$ bildet eine Menge M mit $|M| \geq |\mathbb{Z}_n|$ auf die Zahlen $0, \ldots, n-1$ ab.

Eine Hashfunktion ist *surjektiv* [2]: für jedes $y \in Z_n$ gibt es ein $x \in M$ mit $h(x) = y$.

Eine Hashfunktion ist *gleichverteilt*, wenn zwei Bilder $y1, y2 \in \mathbb{Z}_n$ immer ungefähr gleich viele Urbilder haben $|h^{-1}(y1)| \approx |h^{-1}(y2)|$.

Hashes für unterschiedliche Anwendungen

- **Hashes für Datenstrukturen** *müssen sehr effizient* sein.
- Für **Hashes, welche verwendet werden im Rahmen von Verschlüsselung und Signaturen,** muss es schwer sein:
  - ein Urbild zu finden (d. h. von Y auf X zu schließen)
  - zwei kollidierende Werte zu finden.

  MD5 ist seit 2008 und SHA1 seit 2017 „geknackt".

  - kryptographische Hashes *sollten effizient berechenbar* sein.
- **Hashes für Passwortspeicherung** müssen die selben Anf. erfüllen wie Hashes für Signaturen und Verschlüsselungszwecke, dürfen aber *nicht effizient berechenbar* sein.

Wichtig

Im Folgenden konzentrieren wir uns auf Hashes für Datenstrukturen.

Wenn das Ziel ist, Hash-Werte mit einer bestimmten Länge (zum Beispiel 32Bit) zu berechnen, dann wären folgende Hashfunktionen denkbar:

Exemplarische Hashfunktionen

Ganze Zahlen

```
hash(x: u32) : u32 { return x; } // u32 == 32-Bit unsigned integer
```

Gleitkommazahlen

```
hash(x: f64) : u32 { // f64 == 64-Bit (IEEE) floating point number
    bits : u64 = f64ToBits(x); // u64 = 64-Bit (signed) integer
    return (u32) (bits ^ (bits >>> 32));
}
```

`>>>` ist der *unsigned right shift* Operator.

Zeichenketten

Horners Methode für Zeichenketten der Länge L:

$$h = s[0] \cdot 31^{L-1} + \ldots + s[L-2] \cdot 31^1 + s[L-1] \cdot 31^0.$$

```
fn hash(s: [char,4]) : u32 {
    hash: u32 = 0
    for i in 0..4 { hash = s[i] + hash * 31; }
    return hash;
}
// hash(['c','a','l','l']) = // ≙ hash("call")
//    99 · 31·31·31 + 97 · 31·31 + 108 · 31 + 108 =
//    108 + 31 · ( 108 + 31 · (97 + 31 · (99)))
```

Bemerkung

char	unicode
'a'	97
'b'	98
'c'	99
⋮	⋮
'l'	108

[2] In machen Fällen ist der Nachweis nicht möglich, aber es wird vermutet.

2. Hashing in Python

Verwendung von Hashes in Python

- Bei der Speicherung von Objekten in Sets und Dictionaries verwendet Python Hashes.
- Sobald ein Objekt in einem Set oder Dictionary gespeichert wird, darf der Objektzustand (zumindest im Hinblick auf die Hashfunktion) nicht mehr verändert werden!
- Der Hashwert eines (nicht veränderlichen) Objekts kann mit der Funktion `hash()` berechnet werden.
- Eigene Objekte in Sets und Dictionaries speichern:
  - Um benutzerdefinierte Objekte in einer Hashmap zu speichern, müssen wir die Methoden `__hash__` und `__eq__` implementieren.
  - Zu beachten:
    - Hashwerte *müssen für gleiche Objekte gleich sein*.
    - Hashwerte *für unterschiedliche Objekte sollten unterschiedlich sein*.

Beispielklasse Person

```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __eq__(self, other):
        if isinstance(other, Person):
            return  self.name == other.name and \
                    self.age == other.age
        return False

    def __hash__(self):
        return hash((self.name, self.age))
```

Verwendung der Klasse Person

```python
person1 = Person("Alice", 30)
person2 = Person("Bob", 25)
person3 = Person("Alice", 30) # gleiche Werte wie "person1"
```

Beispielausgabe

```
>>> person1
<__main__.Person object at 0x101474c20>
>>> person2
<__main__.Person object at 0x1013daad0>
>>> person3
<__main__.Person object at 0x1013db110>
```

Speicherung von Person-Objekten in einem Set

```python
people = {person1, person2, person3}
```

Ausgabe des Sets

```python
for p in people: print(p.name)
```

Ausgabe

```
Bob
Alice
```

Verwendung der hash-Funktion

```python
print(hash(person1))
```

```
print(hash(person2))
print(hash(person3))
```

Beispielausgabe

```
3529483511948588452
-9048922068811934735
3529483511948588452
```

In Python ist die Ausgabe der Funktion `hash()` nach jedem Neustart der Pythonumgebung unterschiedlich, da die Hashfunktion einen Zufallswert enthält, der bei jedem Neustart neu generiert wird.

Beispielklasse Person mit konstantem Hashwert

```
class PersonWithBadHash:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __eq__(self, other):
        if isinstance(other, Person):
            return  self.name == other.name and \
                    self.age == other.age
        return False

    def __hash__(self):
        return 1 # immer der gleiche Hashwert
```

Die Verwendung des Alters der Person als Hashwert wäre in den allermeisten Fällen auch keine gute Idee, da es (vermutlich) viele Hashkollisionen geben würde.

Verwendung einer Klasse mit einer konstanten Hashfunktion

```
person1 = Person("Alice", 30)
person2 = Person("Bob", 25)
person3 = Person("Alice", 30)
people = {person1, person2, person3}
print(hash(person1))
print(hash(person2))
print(hash(person3))
print(" ".join(map(lambda p: p.name, people)))
```

Beispielausgabe

```
1
1
1
Alice Bob
```

Warnung

Die Verwendung einer konstanten Hashfunktion ist in der Regel keine gute Idee, da sie die Effizienz von Hashmaps ganz erheblich beeinträchtigen kann.

Übung

2.1. Eine Klasse zur Repräsentation von Studierenden.

Die Klasse `Student` soll:

- die Attribute/Properties `name` und `matriculation_number` haben.
- die Methoden `__eq__` und `__hash__` sinnvoll/korrekt definieren

Aufgaben:

1. Erzeugen Sie drei `Student`-Objekte und speichern Sie diese in einem Set.
2. *Fragen Sie sich wie sie effizient den Hashwert berechnen können.*
3. Geben Sie die Namen der Studierenden aus.
4. Was passiert, wenn Sie — *nachdem Sie ein Student Objekt dem Dictionary hinzugefügt haben* — den Namen des Studenten ändern?

   Schreiben Sie entsprechenden Code, um Ihre Annahme zu überprüfen!

Rumpfimplementierung

```python
class Student:
    def __init__(self, ... ):
        raise NotImplementedError("TODO")

    def __eq__(self, other):
        raise NotImplementedError("TODO")

    def __hash__(self):
        raise NotImplementedError("TODO")
```

3. Hashfunktionen

Gängige Ansätze für Hashfunktionen

Modulo-Hashfunktion:

Sei n möglichst eine Primzahl:

$h_n^{mod}(x) = x \; mod \; n$

Bewertung

- einfach zu berechnen/sehr effizient
- surjektiv
- gleichverteilt
- wenn n keine Primzahl ist, dann kann es (leicht) passieren, dass bestimmte (Teil-)daten weniger oder keinen Einfluss auf den Hashwert haben:
  - $x \cdot 10^3 \mod 40 = 0$
  - $x \cdot 10^3 \mod 42 \in \{0, 2, 4, \ldots, 40\}$ Anm.: $ggt(42, 1000) = 2$
  - $x \cdot 10^3 \mod 41 \in \{0, 1, 2, 3, \ldots, 40\}$ Anm.: $ggt(41, 1000) = 1$

Multiplikations-Hashfunktion:

Sei c fest, oft $c = \frac{\sqrt{5}-1}{2} \approx 0,6180339887498949$:

$h_n^{mul}(x) = \lfloor n \cdot (c \cdot x - \lfloor c \cdot x \rfloor) \rfloor$

Bewertung

- nicht beweisbar surjektiv
- nur asymptotisch gleichverteilt
- Das verwendete c sollte eine gute Durchmischung der Key-Bits fördern.

  Andere irrationale Zahlen sind ggf. auch sinnvoll/möglich.
- Berechnung benötigt eine effiziente Fließkomma-Verarbeitung

Übung

3.1. Hashwerte berechnen I

Berechnen Sie:

1. $h_{257}^{mod}(1\,000)$
2. $h_{257}^{mul}(1\,000)$

3.2. Hashwerte berechnen II

Berechnen Sie:

1. $h_{263}^{mod}(10\,000)$
2. $h_{263}^{mul}(10\,000)$

4. Hashtabellen (🇺🇸 *Hashmaps* oder 🇺🇸 *Dictionaries*)

Grundlagen von Hashtabellen

Das Grundprinzip von Hashtabellen ist einfach:

- Im Vorfeld wird ein Array `A` einer Größe `n` angelegt,

  Die Größe des Arrays übersteigt die erwartete Belegung deutlich.
- Daten mit einem Schlüssel `k` werden dann an der Position `A[h(k)]` gespeichert - oder an einer Ersatzposition.
- Sollte die Belegung zu groß werden, wird das Array vergrößert und die Elemente werden (ggf.) neu bzw. wieder gehasht.
- Sollten zwei Schlüssel den gleichen Hash haben (d. h. $h(x_1) = h(x_2)$), dann wird eine Kollisionsauflösung benötigt.

Belegung von Hashtabellen

Die Belegung von Hashtabllen ist für die Effizienz entscheidend.

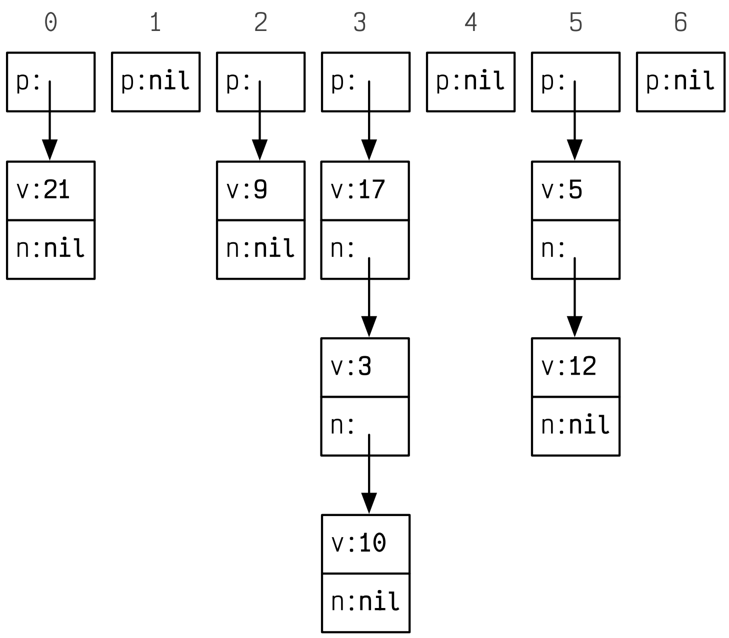
Definition

Ein Array A der Kapazität n mit einer Hashfunktion h_n wird $Hashtabelle(A, h_n)$ genannt.

Sind zu einem Zeitpunkt m (erste) Felder belegt, so hat die $Hashtabelle(A, h_n)$ eine Belegung von $\alpha = \frac{m}{n}$.

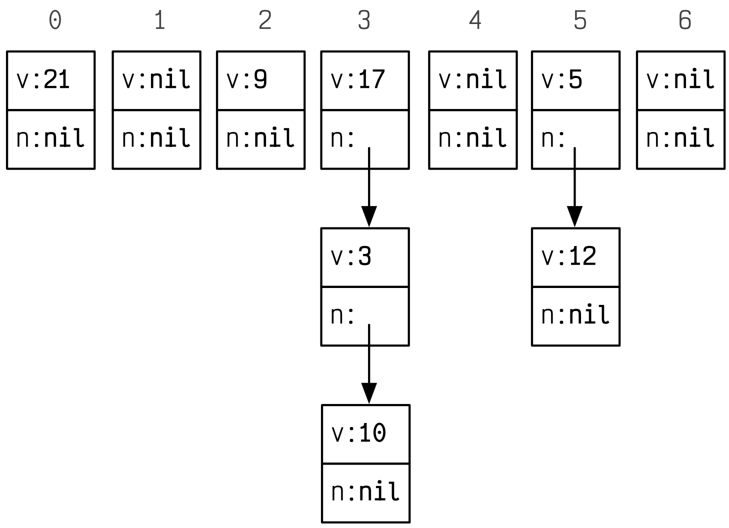
Verkettete Hashtabellen

Direkte Verkettung



Die *direkte Verkettung* von Überläufern verwendet eine $Hashtabelle(A, h_n)$, mit einem Array A, das aus Zeigern auf einfach verkettete Listen besteht, dessen Schlüssel der Einträge alle den gleichen Hashwert besitzen, oder die `nil` sind, wenn kein Eintrag bisher mit dem jeweiligen Hashwert vorhanden ist.

Separate Verkettung



Die *separate Verkettung* von Überläufern verwendet eine $Hashtabelle(A, h_n)$, bei der das Array A aus Knoten einer einfach verketteten Liste besteht, dessen Wert nil ist, wenn unter dem Hashwert noch nichts gespeichert wurde.

Ein Eintrag mit Schlüssel k wird der verketteten Liste zugeordnet, die in $A[h_n(k)]$ verlinkt ist oder startet, und kann entsprechend hinzugefügt, gelöscht und gefunden werden.

Offene Adressierung

Definition

Soll der $Hashtabelle(A, h_n)$ mit einem Array A ein Datensatz mit Schlüssel k hinzugefügt werden soll, so erfolgt dies in $A[h_n(k)]$, wenn dieser Eintrag noch nicht belegt ist. Ansonsten werden $i = 1, \ldots, n-1$ weitere Positionen $A[g_n(k, i)]$ geprüft.

Strategien

Lineares-Sondieren:

Das Array wird linear durchsucht.

$$g_n^{lin}(k, i) = (h_n(k) + i)\ mod\ n$$

Quadratisches-Sondieren:

Das Array wird quadratisch steigend durchsucht.

$$g_n^{quad}(k, i) = (h_n(k) + i^2)\ mod\ n$$

Doppeltes-Hashing:

Das Array wird mit Hilfe einer zweiten Hashfunktion:

$$h_n^{'}(k) = (k\ mod\ (n-2)) + 1$$

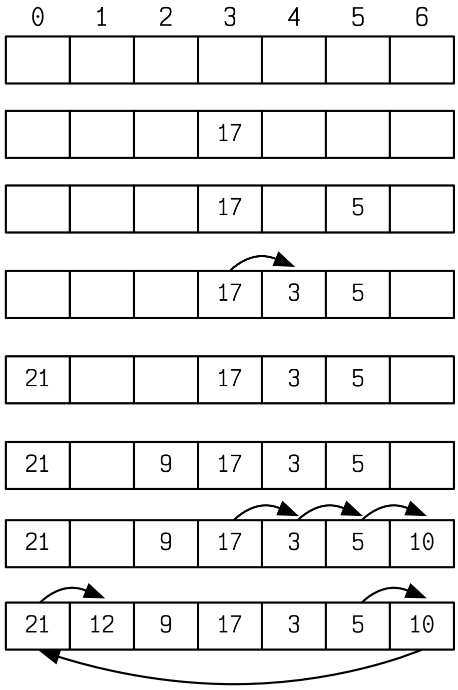
durchsucht.

$$g_n^{doppel}(k, i) = (h_n(k) + i \cdot h_n^{'}(k))\ mod\ n$$

Beispiel Offene Adressierung (Hashing: $x \ mod \ 7$)

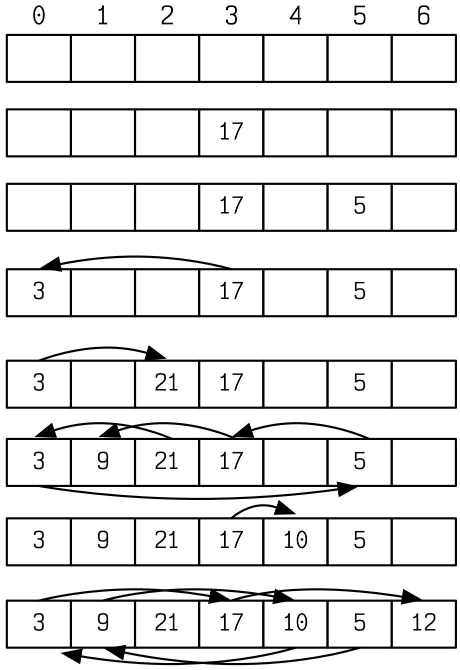
Lineare Sondierung

Hinzufügen von (17, 5, 3, 21, 9, 10, 12)



Doppeltes-Hashing

Hinzufügen von (17, 5, 3, 21, 9, 10, 12)



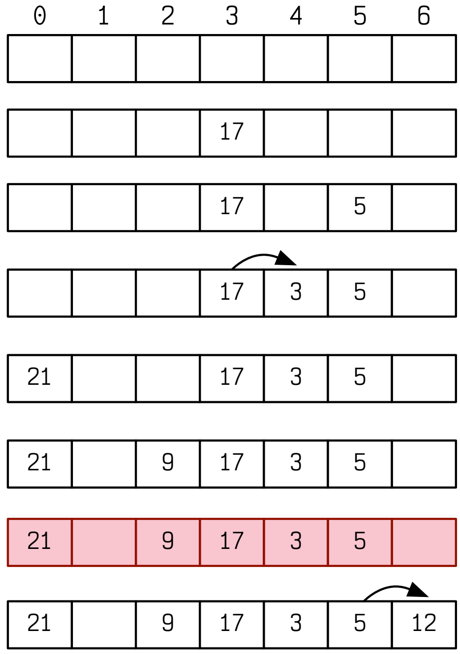
Quadratische Sondierung

Hinzufügen von (17, 5, 3, 21, 9, 10, 12)

Für den Wert 10 wird kein Platz gefunden!

$(10 \ mod \ 7 = 3)$

1. $3 + 0^2 \ mod \ 7 = 3$
2. $3 + 1^2 \ mod \ 7 = 4$
3. $3 + 2^2 \ mod \ 7 = 0$
4. $3 + 3^2 \ mod \ 7 = 5$
5. $3 + 4^2 \ mod \ 7 = 5$
6. $3 + 5^2 \ mod \ 7 = 0$
7. $3 + 6^2 \ mod \ 7 = 4$


0 1 2 3 4 5 6
17
17 5
17 3 5
21 17 3 5
21 9 17 3 5
21 9 17 3 5
21 9 17 3 5 12

Übung

4.1. Werte in kleine Hashtabelle einfügen

Belegen Sie eine Hashtabelle mit $n = 5$ Feldern mit den Werten 37, 18, 32 und 24 auf Basis von $h_5^{mod}(x)$ mit linearer Sondierung, quadratischer Sondierung und doppeltem Hashing mit $h'_5(x) = (x \ mod \ 3) + 1$.

4.2. Werte in größere Hashtabelle einfügen

Belegen Sie eine Hashtabelle mit $n = 11$ Feldern mit den Werten 37, 49, 26 und 39 auf Basis von $h_{11}^{mod}(x)$ mit linearer Sondierung, quadratischer Sondierung und doppeltem Hashing mit $h'_{11}(x) = (x\ mod\ 9) + 1$.

Angriffe auf algorithmische Komplexität

> *Julian Wälde and Alexander Klink reported that the String.hashCode() hash function is not sufficiently collision resistant.*
>
> *hashCode() value is used in the implementations of [Java 6] HashMap and Hashtable classes. A specially-crafted set of keys could trigger hash function collisions, which can degrade performance of HashMap or Hashtable by changing hash table operations complexity from an expected/average O(1) to the worst case O(n). Reporters were able to find colliding strings efficiently using equivalent substrings and meet in the middle techniques. This problem can be used to start a denial of service attack against applications that use untrusted inputs as HashMap or Hashtable keys. An example is a web application server that may fill hash tables with data from HTTP request. A remote attack could use that to make JVM use excessive amount of CPU time by sending a POST request with large amount of parameters which hash to the same value.*
>
> ***—[Abbreviated Version] Jan Lieskovsky 2011-11-01***