

Algorithmen und Datenstrukturen

Vorlesung #11– Hashing

Benjamin Blankertz

Lehrstuhl für Neurotechnologie, TU Berlin

benjamin.blankertz@tu-berlin.de

27 · Jun · 2023



Themen der heutigen Vorlesung

- ▶ Symboltabellen: Implementation über Arrays und Listen – Vor- und Nachteile
- ▶ **Hashtabellen**
- ▶ Perfektes Hashing und Kollisionen
- ▶ Hashfunktionen:
 - ▶ Multiplikationsmethode
 - ▶ Divisions-Rest-Methode
 - ▶ Universelles Hashing
- ▶ Kollisionsauflösung:
 - ▶ mit Verkettung (*separate chaining*)
 - ▶ mit Sondierung (*open addressing*), lineare und quadratische Sondierung; *Double Hashing*
 - ▶ Primäre und sekundäre Häufungen
- ▶ **Randomisierte Algorithmen**: Las-Vegas und Monte-Carlo Methoden

- ▶ Für viele Anwendungen werden sogenannte Wörterbücher oder **Symboltabellen** (*dictionary, symbol table*; auch *associative array, map*) verwendet:
 - ▶ Der Datensatz besteht aus **Schlüssel-Wert** (*key-value*) Paaren.
 - ▶ Es gibt die Operationen: Paar **einfügen** (*put*), Wert zu gegebenem Schlüssel **suchen** (*get*), und Paar mit gegebenem Schlüssel **löschen** (*delete*).
- ▶ Für ein normales Lexikon wäre das Stichwort der Schlüssel und die Definition der Wert.
- ▶ Um in einem Lexikon die Definition für ein Stichwort nachzuschauen, muss man in der Datenbank zu gegebenem Schlüssel den Wert des entsprechenden Datensatzes herausuchen. Weitere Beispiele:

Anwendung	Schlüssel	Wert
Bankverbindung	IBAN, BIC	Kontoinformation, Transaktionen
Compiler	Variablennamen	Speicheradresse
Internet DNS	Hostname	IP Adresse
Spielsolver	Spielstellung	Evaluationswert

Ein interface und ein API für Symboltabellen

Als (minimale) Anforderung haben wir also das folgende Interface:

```
interface SymbolTable<K,V> {  
    V get(K key);  
    void put(K key, V value);  
    void delete(K key);  
}
```

Diese Anforderungen können wir auch als API formulieren:

API einer Symboltabelle

```
public class SymbolTable<K,V>
```

SymbolTable()	Erzeugt eine leere Symboltabelle.
---------------	-----------------------------------

void	put(K key, V val)	Fügt der Symboltabelle das Schlüssel-Wert-Paar (key, val) hinzu. Falls schon ein Eintrag mit key existiert, ersetze val.
-------------	-------------------	--

void	delete(K key)	Löscht den Eintrag mit Schlüssel key.
-------------	---------------	---------------------------------------

V	get(K key)	Gibt den zum Schlüssel key gehörigen Wert zurück.
---	------------	---

Implementationsvorschlag 1: Array

- ▶ Die Frage ist nun, welche Implementation am besten ist.
- ▶ Falls die Schlüssel nicht-negative, ganzzahlige Werte in einem relativ engen Bereich sind, kann ein Array benutzt werden:

```
public class ArrayST<V> implements SymbolTable<Integer, V> {  
    private V[] valuearray;  
  
    public ArrayST(Integer maxKey) {           // Konstruktor: Schlüssel 0 ... maxKey-1  
        valuearray= (V[]) new Object[maxKey];  
    }  
  
    public void put(Integer key, V value) { valuearray[key]= value; }  
    public void delete(Integer key)       { valuearray[key]= null; }  
    public V get(Integer key)              { return valuearray[key]; }  
}
```

Vorteile:

- ▶ Einfache Implementation
- ▶ **sehr schnell**: alle Operationen $O(1)$

Einschränkungen und Nachteile:

- ▶ Nur möglich, wenn Schlüssel vom Typ Integer sind.
- ▶ Der maximal mögliche Schlüssel muss bekannt sein. Andernfalls müsste das Array dynamisch vergrößert werden.
- ▶ Wenn die Verteilung der Schlüssel größere **Lücken** aufweist, wird sehr viel **Speicher verschwendet**.

Implementationsvorschlag 2: Verkettete Liste

- In jedem Fall (unabhängig vom Typ des Schlüssels) kann eine verkettete Liste benutzt werden:

```
public class LinkedListST<K,V> implements SymbolTable<K,V> {  
    private Node head;  
  
    private class Node {  
        K key;  
        V val;  
        Node next;  
  
        Node(K key, V val, Node next) {  
            this.key = key;  
            this.val = val;  
            this.next = next;  
        }  
    }  
}  
// Fortsetzung auf der nächsten Seite
```

Implementierung mit verketteter Liste (Teil 2)

```
// Fortsetzung der Klasse LinkedListST<K,V>
public V get(K key) {
    for (Node node = head ; node != null ; node = node.next)
        if (node.key == key)
            return node.val;
    return null;
}

public void put(K key, V val) {
    for (Node node = head ; node != null ; node = node.next)
        if (node.key == key) {
            node.val = val;                // erfolgreiche Suche: überschreiben
            return;
        }
    head = new Node(key, val, head); // erfolglose Suche: hinzufügen
}

public void delete(K key) { put(key, null); } // lazy delete :)
}
```


Nachtrag: Implementation von *eager deletion*

- ▶ Das obige Programmbeispiel implementiert `delete` als verzögertes Löschen (*lazy deletion*).
- ▶ Das Listenelement wird nicht entfernt, sondern der zugehörige Wert wird lediglich auf `null` gesetzt (und eventuell später gelöscht).
- ▶ Für die Funktionalität ist das ausreichend, aber es ist nicht effizient. Daher wird hier die Variante des sofortigen Löschens (*eager deletion*) nachgeliefert.

```
public void delete(K key) {  
    Node prev = null;  
    Node node = head;  
    while (node != null && node.key != key) {  
        prev = node;  
        node = node.next;  
    }  
    if (node != null) {  
        if (prev == null) head = node.next; // erfolgreiche Suche  
        else prev.next = node.next; // ersten Knoten löschen  
    }  
}
```

Vor- und Nachteile bei verketteter Liste

Vorteile:

- ▶ sehr effiziente Speicherlösung
- ▶ für alle Datentypen von Schlüsseln geeignet

Nachteile:

- ▶ keine gute Performanz: Laufzeit aller Methoden ist in $O(N)$

Bemerkung:

- ▶ Bei Verwendung von binären Suchbäumen lässt sich die Performanz im durchschnittlichen Fall auf $O(\lg N)$ verbessern, falls auf den Schlüsseln eine Ordnung existiert.
- ▶ Balancierte Bäume, wie 2-3-Bäume bzw. Rot-Schwarz-Bäume schaffen es auch im *worst case* in $O(\lg N)$.

Siehe dazu [Sedgewick & Wayne, Kapitel 3.2 und 3.3].

Memory-Performance Trade-off

Die Frage nach der besten Implementation einer Symboltabelle betrifft das Abwägen zwischen Speicherbedarf und Rechenzeit.

- ▶ In dem kommenden Hauptteil dieser Vorlesung werden wir uns mit **Hashing** befassen.
- ▶ Dies erlaubt einen Mittelweg zwischen den Implementationen mit Arrays und verketteten Listen.
- ▶ Innerhalb von Hashing kann durch die Wahl von Parametern der Speicher/Rechenzeit Trade-off beeinflusst werden.

Vergleich des Rechenbedarfs

Rechenzeit nachdem N Elemente in die Datenstruktur eingefügt wurden:

Algorithmus / Datenstruktur	durchschnittlicher Fall		schlimmster Fall	
	get	put	get	put
Schlüssel-indiziertes Array	$O(1)$	$O(1)$	$O(1)$	$O(1)$
balancierter Suchbaum	$O(\lg N)$	$O(\lg N)$	$O(\lg N)$	$O(\lg N)$
Binärer Suchbaum	$O(\lg N)$	$O(\lg N)$	$O(N)$	$O(N)$
verkettete Liste	$O(N)$	$O(N)$	$O(N)$	$O(N)$

Wir wollen eine speichereffiziente Lösung, die in den meisten Fällen eine Performanz nah an $O(1)$ hat.

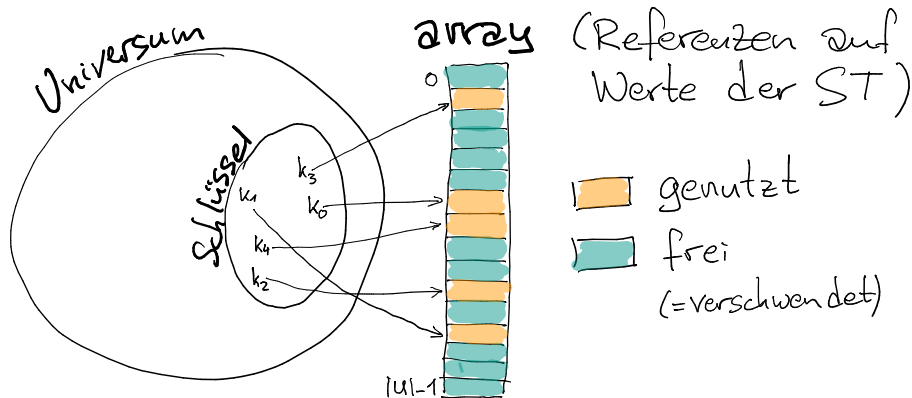
Hashtabellen für Schlüsselmengen oder Schlüssel-Werte Paare

- ▶ Die Werte einer Symboltabelle spielen algorithmisch keine Rolle.
- ▶ Relevant ist die effiziente Organisation von Schlüsseln.
- ▶ Der Aspekt der Werte im Folgenden daher meist ausgespart.
- ▶ Bei einigen Anwendungen hat man **gar keine Werte** und es geht nur um die **Verwaltung einer Schlüsselmenge**.
- ▶ In den Java Collections gibt es hierfür die Klasse `HashSet<K>`.
 - und für Schlüssel-Werte Paare `HashMap<K, V>`.
 - Siehe auch die praktischen Hinweise auf Seite 51.

Von Symboltabellen mit Arrays zu Hashtabellen

- ▶ Die Menge aller **möglichen** Schlüssel wird als **Universum**, U bezeichnet. Sie kann **sehr groß** sein.
- ▶ Die Menge der *tatsächlich verwendeten* Schlüssel K ist im Vergleich zu U **klein**: $|K| \ll |U|$.

Situation bei Symboltabellen mit Arrays

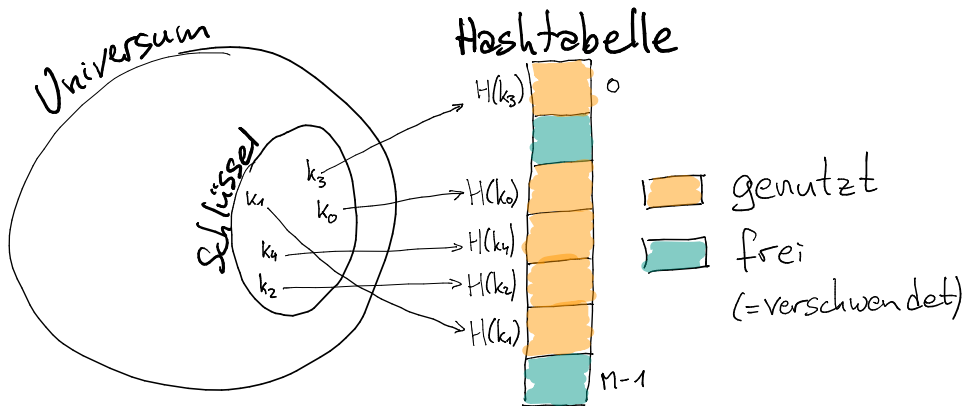


- ▶ Unter der Annahme $|K| \ll |U|$ wird hier viel Speicher verschwendet.
- ▶ In vielen Anwendungen ist eine direkte Indizierung des Arrays mit Schlüsseln unmöglich. (Zeichenketten als Integer ergeben schnell Werte, die viel größer als Speicheradressen sind.)

Der Ansatz mit Hashtabellen

- **Hashtabellen** beruhen darauf, dass die Schlüssel auf Werte $< M$ mit $M \ll |U|$ abbildet werden (*hashing*).

$$H : U \rightarrow \{0, \dots, M-1\}; \quad k \mapsto H(k)$$



Vom Objekt zum Index in einer Hashtabelle

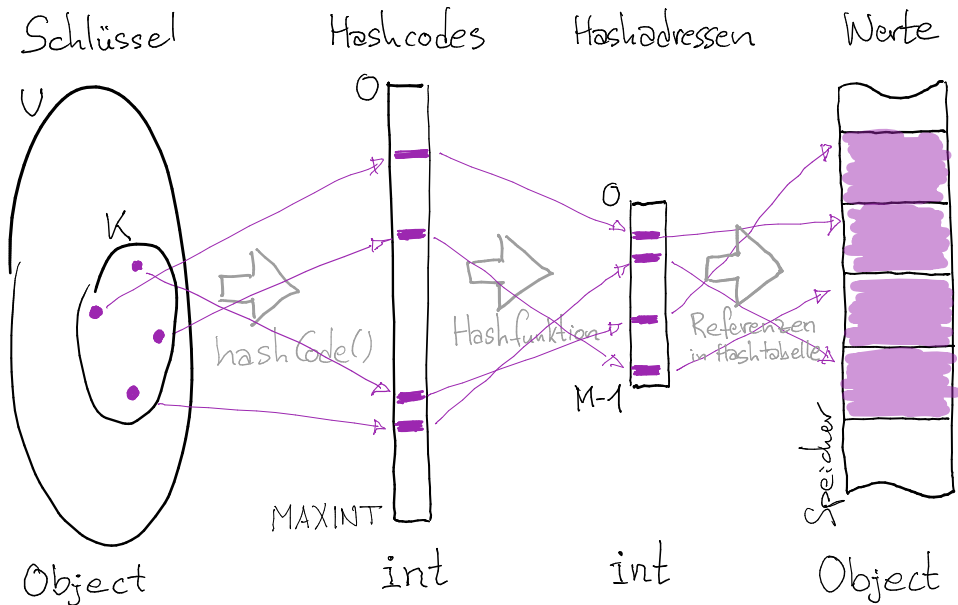
- ▶ Schlüssel werden in zwei Schritten auf Indizes in der Hashtabelle abgebildet:

- 1 Die Schlüssel (beliebige Objekte) auf **Hashcodes** (`int`)

- 2 und diese per **Hashfunktion** h auf einen kleinen Wertebereich $\{0, \dots, M - 1\}$.

- ▶ Diese sogenannten **Hashadressen** dienen als Index in einer **Hashtabelle**.
- ▶ So kann der schnelle Zugriff eines Array mit einer sparsameren Speicherung erreicht und zudem beliebige Objekte als Schlüssel benutzt werden.

Vom Objekt zum Index in einer Hashtabelle



- ▶ Die Java Klasse `Object`, von der alle anderen Klasse abgeleitet sind, deklariert die Methode **`hashCode()`**, die zu jedem Objekt einen `int` Wert zurückgibt.
- ▶ Für alle Klassen in den Java Bibliotheken ist die Methode `hashCode` implementiert.
- ▶ Für jede eigene Klasse muss eine **eigene Methode** implementiert werden, falls sie die Hashing Funktionalität unterstützen soll.
- ▶ Wird `hashCode()` nicht überschrieben, erbt die Klasse die Standardimplementation, die die Speicheradresse des Objektes zurückgibt (also nicht sinnvoll ist).
- ▶ Mehr zu praktischen Aspekten der Implementation von Hashcodes später.
- ▶ Wir gehen davon aus, dass Hashcodes **nicht negativ** sind. Zur Umwandlung der `int` Rückgabe von `hashCode()` in eine nicht-negative Zahl, siehe Seite 37.

Beispiel Hashcodes für String

- ▶ In Java 8 wird z.B. die Methode `hashCode()` für `String` Objekte im Prinzip folgendermaßen implementiert, allerdings als Objektmethode:

```
public static int hashCode(String str)
{
    int hash = 0;
    for (int i = 0; i < str.length(); i++)
        hash = 31 * hash + str.charAt(i);
    return hash;
}
```

- ▶ Die tatsächliche Implementierung speichert berechnete Hashcodes in einer Variable, um sie bei erneutem Aufruf nicht neu berechnen zu müssen.

Hashcodes für eigene Datentypen

- ▶ Für Datentypen aus den offiziellen Java Bibliotheken sind jeweils `hashCode()` Methoden implementiert.
- ▶ Für primitive Datentypen kann über den entsprechenden Wrappertyp eine vorhandene `hashCode()` Methode verwendet werden.
- ▶ Für eigene Datentypen läuft man über alle relevanten Datenfelder und verbindet deren Hashcodes wie im `String` Beispiel (einfacher Vorschlag):

```
public class FlowEdge {  
    private int v, w;  
    private double flow, capacity;  
    ...  
    public int hashCode() {  
        int hash = v;  
        hash = 17 * hash + w;  
        hash = 17 * hash + ((Double) flow).hashCode();  
        hash = 17 * hash + ((Double) capacity).hashCode();  
        return hash;  
    }  
}
```

Wesentliche Elemente der Hashverfahren

- ▶ Wir wollen die Schlüssel in einen kleinen Wertebereich bis $M \ll |U|$ abbilden.
- ▶ **Kollision**: Zwei unterschiedliche Schlüssel werden $k_0, k_1 \in U$ ($k_0 \neq k_1$) auf gleiche Hashadressen abgebildet: $H(k_0) = H(k_1)$.
- ▶ Kollisionen verursachen ein Problem bei der Speicherung in einer Hashtabelle. Sie müssen also **aufgelöst** werden.
- ▶ Ein gutes Hashverfahren zeichnet sich durch eine schnelle Hashfunktion und ein effizientes Verfahren zur **Auflösung von Kollisionen** aus.
- ▶ Durch die Wahl des Hashverfahrens (und seiner Parameter) kann der **Trade-off** zwischen Speicher- und Performanz-Effizienz gewählt werden.

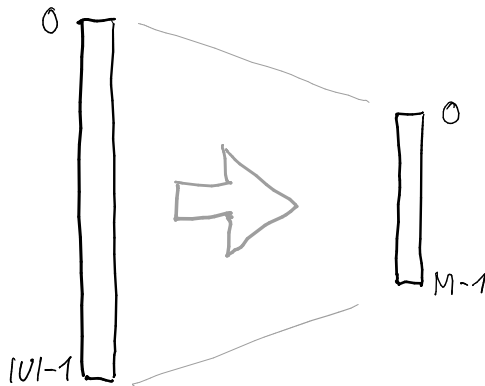
Hashfunktionen und perfektes Hashing

- ▶ Kollisionen treten eher auf, wenn die Schlüssel nicht gleichmäßig verteilt sind. Dies ist meistens der Fall. Compilerbeispiel: Variablennamen sind nicht zufällig, sondern oft ähnlich.
- ▶ Ist die Menge der Schlüssel im voraus bekannt, so lässt sich eine kollisionsfreie Hashfunktion definieren: **perfektes Hashing**.
 - Z.B. Schlüssel sortieren und jeden Schlüssel auf die Ordnungszahl in der Sortierung abbilden. Siehe [Mehlhorn 1986] für weitere Verfahren zur Konstruktion perfekter Hashfunktionen.
- ▶ Dieser Fall (Schlüsselmenge im voraus bekannt) ist in der Anwendung eher untypisch.

- ▶ Wir gehen davon aus, dass die Voraussetzung für perfektes Hashing **nicht** erfüllt ist.
- ▶ Kollisionen lassen sich dann nicht systematisch vermeiden.
- ▶ Die Wahrscheinlichkeit für Kollisionen ist selbst dann hoch, wenn M groß gewählt ist, also der **Belegungsfaktor** (*load factor*) $\alpha = \frac{N}{M}$ klein ist:
 - Bei einer Hashtabelle der Größe $M = 10.000$ tritt im Durchschnitt eine Kollision bei dem 125. Schlüssel auf. Allgemein gilt dies bei $\sim \sqrt{\pi M/2}$ vielen Schlüsseln.
- ▶ Im folgenden werden einige Hashfunktionen vorgestellt.
- ▶ Alle haben das Ziel, die Hashadressen möglichst **gleichmäßig** zu verteilen, auch wenn die Hashcodes es nicht sind (Clustervermeidung).

Ansätze für Hashfunktionen

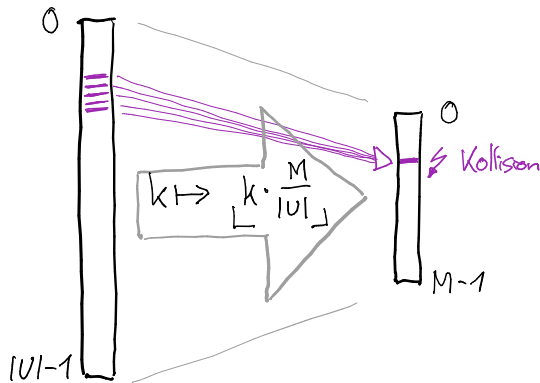
- ▶ Um den großen Zahlenbereich $\{0, \dots, |U| - 1\}$ auf einen kleinen Zahlenbereich $\{0, \dots, M - 1\}$ abzubilden, gibt es zwei einfache Methoden:
 - ▶ Multiplikation mit dem Faktor $M/|U|$ plus Abrunden.
 - ▶ Rest bei der Division durch M bestimmen.
- ▶ Mit einer geschickten Modifikation, die insbesondere bei der multiplikativen Methode notwendig ist, führen beide Ansätze zu guten Hashfunktionen.



Hashfunktion: Naive Multiplikation ist ungünstig

- ▶ Betrachten wir zunächst den einfachen Versuch einer multiplikativen Hashfunktion

$$h : k \mapsto \lfloor k \frac{M}{|U|} \rfloor$$



- ▶ Diese Funktion bildet zwar die Bereiche korrekt ab, erfüllt aber nicht die gewünschte Eigenschaft der Clusterauflösung:
- ▶ Aufeinander folgende Schlüssel bzw. Hashcodes werden auf dieselbe Hashadresse abgebildet und verursachen Kollisionen.
- ▶ Ein weiterer Nachteil ist, dass $|U|$ bekannt sein muss.



Hashfunktion: Multiplikationsmethode

- ▶ Bei der **Multiplikationsmethode** wird der Schlüssel k zunächst mit einer irrationalen Zahl multipliziert. Von dem Ergebnis wird nur der Nachkommateil genommen, mit M multipliziert und abgerundet.
- ▶ Die Methode ist durch den folgenden Satz motiviert.

Satz von Vera Turán Sós (*three-gap theorem*)

Sei γ eine irrationale Zahl. Wir betrachten die Zahlenfolge $c(n) = n\gamma - \lfloor n\gamma \rfloor \in [0, 1]$. Die von den ersten n Zahlen gebildeten n Intervalle (bei Identifizierung der Intervallgrenzen) haben höchstens drei unterschiedliche Längen. Der nächste Punkt c_{n+1} fällt immer in ein Intervall der größten Länge.

- ▶ Bei der Wahl der irrationalen Zahl führt der goldene Schnitt $\phi = (\sqrt{5} - 1)/2$ zu einer besonders gleichmäßigen Verteilung:

$$h(k) = \lfloor M(k\phi - \lfloor k\phi \rfloor) \rfloor \quad (\text{Fibonacci-Hash})$$

Für Details sowie Tricks bei der Implementation siehe [Knuth 1973].

Hashfunktion: Divisions-Rest-Methode

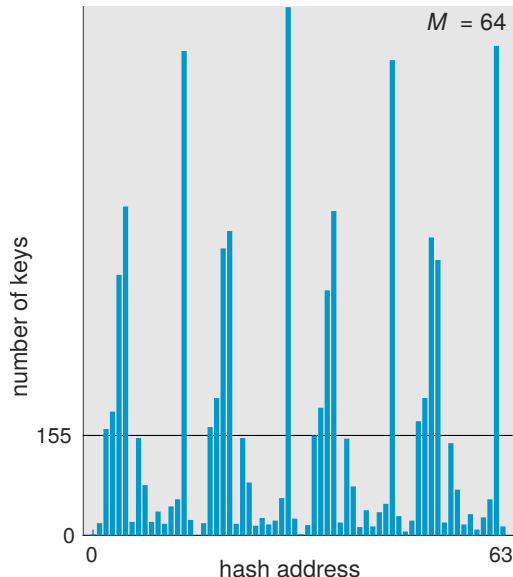
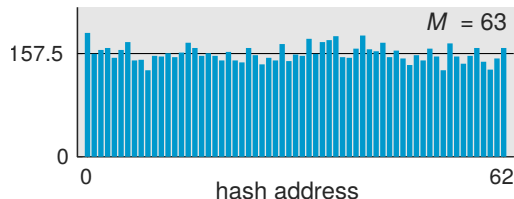
- ▶ Eine einfache Methode, um beliebige Schlüssel $k \in \mathbb{N}_0$ (Hashcodes) auf Zahlen $< M$ (Hashadressen) abzubilden, ist es, den **Rest bei der Division durch M** zu berechnen:

$$h : \mathbb{N}_0 \rightarrow \{0, \dots, M - 1\}; \quad k \mapsto k \bmod M$$

- ▶ Dabei ist die genaue Wahl von M wichtig (nicht nur die Größenordnung):
- ▶ Für Zweierpotenzen M , werden nur die niedrigwertigsten Bits des Schlüssels berücksichtigt > Kollisionen bei ungleichmäßiger Schlüsselverteilung
- ▶ Gute Werte für M sind **Primzahlen**, besonders solche, die **nicht nah an einer Zweierpotenz** liegen. Weitere Tipps siehe [Knuth 1973]
- ▶ Die **Divisions-Rest-Methode** lieferte in der empirischen Vergleichsstudie [Lum et al, 1971] bei guter Wahl von M durchschnittlich die **besten** Ergebnisse.

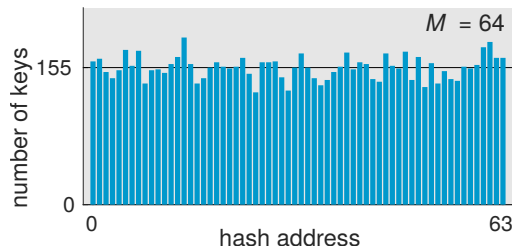
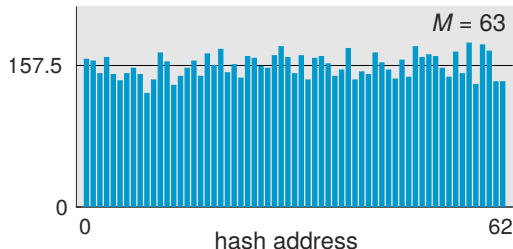
Hashfunktion: Divisions-Rest-Methode – Illustration

- ▶ Zur Illustration nehmen wir absichtlich **ungleichmäßig** verteilte Schlüssel:
- ▶ In `hashCode()` von `String` werden die vier niedrigwertigsten Bits durch die entsprechenden Bits des letzten Buchstaben ersetzt.
- ▶ Histogramm der resultierenden Hashadressen aller Wörter des Werkes *Kritik der reinen Vernunft* bei Anwendung der Div-Rest-Methode:



Hashfunktion: Divisions-Rest-Methode – Illustration

- ▶ Bei der originalen `hashCode()` Funktion von `String` sind die Hashcodes bei normalen Texten meist in allen Bits recht **gleichmäßig** verteilt.
- ▶ In diesem Fall spielt die exakte Wahl von M keine große Rolle.
- ▶ Hier die Histogramme für diesen Fall mit demselben Text wie auf der vorherigen Seite:



- ▶ Auch wenn man eine Hashfunktion aus den beiden vorgestellten Ansätzen auswählt, kann es bei einer **ungünstigen Schlüsselverteilung** zu überproportional vielen Kollisionen kommen.
- ▶ Eine fest gewählte Hashfunktion kann nicht für alle Situationen geeignet sein.
- ▶ Ein Ansatz, diesem Dilemma zu entkommen, ist **Universelles Hashing**.
- ▶ Dabei wird für jede neue Anwendung die Hashfunktion **zufällig** aus einer geeigneten Familie von Funktionen ausgewählt. Bis zum Ende der Anwendung wird die Hashfunktion natürlich konstant gelassen.
- ▶ Durch die Randomisierung kann man im Erwartungswert die kleinstmögliche Anzahl an Kollisionen, unabhängig von der Verteilung der Schlüssel, erzielen.
- ▶ Mehr Informationen zu universellem Hashing sind im Anhang.

Universelles Hashing - Eigenschaften

- ▶ Unabhängig von der Folge von Schlüsseln, die in eine Hashtabelle eingetragen werden, ist die Anzahl der zu erwartenden Kollisionen jeweils gleich dem Füllgrad der Tabelle.
- ▶ Dies entspricht dem Anteil bei einer zufälligen Wahl von Hashadressen, also der bestmöglichen Durchmischung.
- ▶ Bemerkenswert ist hier, dass es für eine beliebig (gemein) gewählte Folge von Schlüsseln gilt.
- ▶ Zu beachten ist, dass sich die Aussage auf den Erwartungswert bezüglich der universellen Menge \mathcal{H} bezieht.
- ▶ Für eine einzelne Anwendung, bei der die gewählte Hashfunktion fix bleibt, erhält man keine Garantie.
- ▶ Praktischer Nutzen? Mit universellem Hashing kann denial-of-service Angriffen vorgebeugt werden, siehe [Crosby & Wallach, 2003].

- ▶ Ein Schlüssel k , der beim Hashing nicht unter seiner Hashadresse $H(k)$ in die Tabelle eingetragen werden kann, da besetzt ist, wird **Überläufer** genannt.
- ▶ Es gibt zwei typische Arten mit Überläufern umzugehen:

1 Hashverfahren mit Verkettung (*separate chaining*):

Überläufer werden **außerhalb** der Hashtabelle in einer zusätzlichen Datenstruktur gespeichert.

2 Hashverfahren mit Sondierung (Offene Hashverfahren, *open addressing*):

Überläufer werden **innerhalb** der Hashtabelle an einem anderen, freien Ort gespeichert.

- ▶ Achtung: Offene Hashverfahren heißen *open addressing* und *closed hashing*, während Hashverfahren mit Verkettung zu *closed addressing* bzw. *open hashing* zählen.

Hashverfahren mit verketteten Listen

- ▶ Die Hashtabelle wird als Feld von [Referenzen auf verkettete Listen](#) angelegt.
- ▶ Wenn Kollisionen auftreten, werden die Überläufer an die jeweilige Liste angehängt.
- ▶ In jedem Element der verketteten Liste ist der abgelegte Schlüssel, sowie eine Referenz auf den zugehörigen Wert gespeichert.
 - Und, wie immer bei einer verketteten Liste, gibt es eine Referenz auf den nächsten Listeneintrag, bzw. null am Listenende.

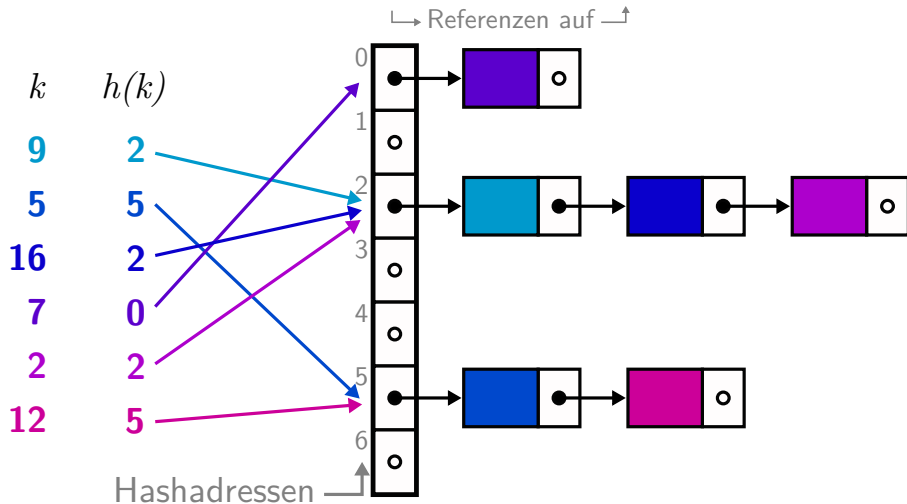
Hashverfahren mit verketteten Listen illustriert

Hashfunktion: $h(k) = k \bmod 7$

Schlüssel

Hashtabelle

verkettete Listen



Einfügen, Suchen und Löschen bei Hashing mit verketteten Listen

- ▶ Bei allen Operationen (`put`, `get`, `delete`) wird zunächst die Hashfunktion angewendet, um die Hashadresse festzustellen.
- ▶ Unter dieser Hashadresse ist in der Hashtabelle eine Referenz auf eine verkettete Liste gespeichert.
- ▶ Dann wird die Operation auf die verkettete Liste angewendet.
- ▶ Je länger die Listen sind, desto länger dauern die Operationen. (Erfolgreiche Suche kann auch schnell abbrechen.)
- ▶ Hört sich einfach an? Ist es auch!

Implementation von Hashing mit verketteten Listen

```
1 public class SeparateChainingHash<K, V> implements SymbolTable<K, V> {
2     private int M;
3     private LinkedListST<K, V>[] st;
4
5     public SeparateChainingHash(int M) {
6         this.M = M;
7         st = (LinkedListST<K, V>[]) new LinkedListST[M];
8         for (int m=0; m<M; m++) {
9             st[m] = (LinkedListST<K, V>) new LinkedListST();
10        }
11    }
12
13    private int hash(K key) {
14        return (key.hashCode() & 0x7fffffff) % M;
15    }
16
17    public V get(K key)          { return st[hash(key)].get(key); }
18    public void put(K key, V val) { st[hash(key)].put(key, val); }
19    public void delete(K key)    { st[hash(key)].delete(key); }
20 }
```

Implementation von Hashing mit verketteten Listen

```
1 public class SeparateChainingHash<K, V> implements SymbolTable<K, V> {  
2     private int M;  
3     private LinkedListST<K, V>[] st;  
4  
5     public SeparateChainingHash(int M) {  
6         this.M = M;  
7         st = (LinkedListST<K, V>[]) new LinkedListST[M];  
8         for (int m=0; m<M; m++) {  
9             st[m] = (LinkedListST<K, V>) new LinkedListST();  
10        }  
11    }  
12  
13    private int hash(K key) {  
14        return (key.hashCode() & 0x7fffffff) % M;  
15    }  
16  
17    public V get(K key)          { return st[hash(key)].get(key); }  
18    public void put(K key, V val) { st[hash(key)].put(key, val); }  
19    public void delete(K key)    { st[hash(key)].delete(key); }  
20 }
```

Implementation von Hashing mit verketteten Listen

```
1 public class SeparateChainingHash<K, V> implements SymbolTable<K, V> {
2     private int M;
3     private LinkedListST<K, V>[] st;
4
5     public SeparateChainingHash(int M) {
6         this.M = M;
7         st = (LinkedListST<K, V>[]) new LinkedListST[M];
8         for (int m=0; m<M; m++) {
9             st[m] = (LinkedListST<K, V>) new LinkedListST();
10        }
11    }
12
13    private int hash(K key) {
14        return (key.hashCode() & 0x7fffffff) % M;
15    }
16
17    public V get(K key)          { return st[hash(key)].get(key); }
18    public void put(K key, V val) { st[hash(key)].put(key, val); }
19    public void delete(K key)    { st[hash(key)].delete(key); }
20 }
```

Implementation von Hashing mit verketteten Listen

```
1 public class SeparateChainingHash<K, V> implements SymbolTable<K, V> {
2     private int M;
3     private LinkedListST<K, V>[] st;
4
5     public SeparateChainingHash(int M) {
6         this.M = M;
7         st = (LinkedListST<K, V>[]) new LinkedListST[M];
8         for (int m=0; m<M; m++) {
9             st[m] = (LinkedListST<K, V>) new LinkedListST();
10        }
11    }
12
13    private int hash(K key) {
14        return (key.hashCode() & 0x7fffffff) % M;
15    }
16
17    public V get(K key) { return st[hash(key)].get(key); }
18    public void put(K key, V val) { st[hash(key)].put(key, val); }
19    public void delete(K key) { st[hash(key)].delete(key); }
20 }
```


Randbemerkung zu Math.abs()

Die Implementation von hash() ist etwas wunderlich:

```
private int hash(K key) {  
    return (key.hashCode() & 0x7fffffff) % M;  
}
```

- ▶ **F:** Warum wird das oberste Bit maskiert, um ein negative Vorzeichen loszuwerden? Es wäre doch natürlicher, Math.abs() zu benutzen.
- ▶ **A:** Math.abs(MIN_VALUE) liefert MIN_VALUE, also eine **negative** Zahl zurück. Dies ist ein Problem der Zweierkomplementdarstellung, bei der es eine negative Zahl mehr als positive Zahlen gibt.
 - Der Wert von Math.abs("polygenelubricants".hashCode()) ist negativ!
- ▶ Durch die Modulo Operation bleibt der Wert negativ, d.h. eine Implementation von hash() unter Verwendung von Math.abs() könnte eine ungültige Hashadresse liefern, was zu einem Fehler führen würde.

Effizienz von Hashing mit Verkettung

- ▶ Die **Effizienz** von Hashing mit Verkettung hängt von der allgemein schwer abzuschätzenden **Verteilung der Hashadressen** ab.
- ▶ Die hängt wiederum von der Verteilung der Schlüssel bzw. Hashcodes und der verwendeten Hashfunktion ab.
- ▶ Die praktische Erfahrung zeigt allerdings, dass bei einer sinnvoll gewählten Hashfunktion die Verteilung der Listenlängen ähnlich zu einer Poisson-Verteilung ist.
- ▶ Dies ist nur eine geringe Abweichung vom Idealfall mit gleichmäßig verteilten Schlüsseln, bei dem sich eine Binomialverteilung ergibt.

Effizienz von Hashing mit Separate Chaining

Bei einer sinnvollen Wahl von Hashcodes und Hashfunktion ist die Anzahl der Schlüsselvergleiche beim Einfügen und (erfolglosen) Suchen in einer Hashtabelle mit M verketteten Listen und N Schlüsseln $\sim N/M$.

Effizienz von Hashing mit Verkettung

- ▶ Das Hashverfahren *separate chaining* funktioniert auch dann noch, wenn der Belegungsfaktor größer als 1 ist. Dabei werden bewusst viele Kollisionen in Kauf genommen.
- ▶ Sei eine Hashtabelle der Größe $M = 1.000$ mit $N = 10.000$ Schlüssel gleichmäßig belegt > Listen haben Länge 10 (Belegungsfaktor N/M).
 - Dann ist `get()` im Schnitt 5 und im schlimmsten Fall 10 mal langsamer als bei einem Schlüssel-indizierten Array, siehe Seite 11;
- ▶ andererseits 1.000 mal schneller als bei einer verketteten Liste (ohne Hashtabelle).
- ▶ Eine typische Wahl für die Größe der Hashtabelle ist $M \sim \frac{N}{5}$. Dann ist also die Anzahl der notwendigen Schlüsselvergleiche im Schnitt 5, siehe vorige Seite.

Hashverfahren mit Sondierung

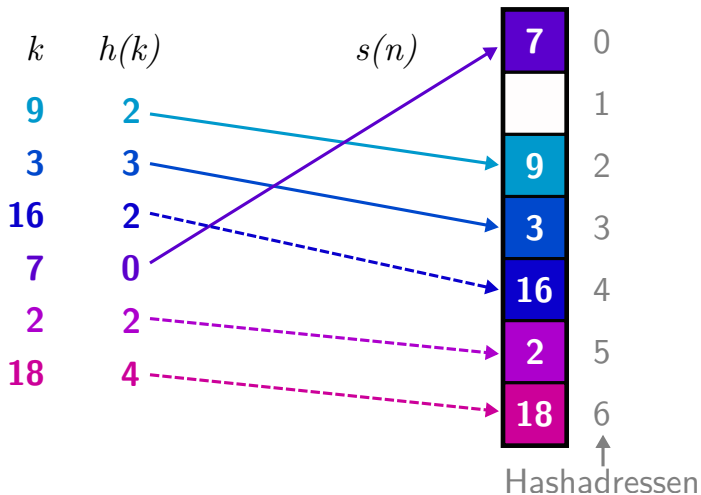
- ▶ Beim **offenen Hashing** (*open addressing*) wird bei einer Kollision für den Überläufer ein freier Platz **innerhalb** der Hashtabelle gesucht.
- ▶ Diese Suche zum Einfügen eines Schlüssels muss nach einer **festen Regel** erfolgen, die entsprechend auch bei `get()` verwendet wird.
- ▶ Dafür verwendet man eine **Sondierungsfunktion**.
- ▶ Einfachster Fall – **linearen Sondierung**: $s(n) = n$
- ▶ Die Suche nach einem freien Platz für Schlüssel k prüft der Reihe nach die Plätze entlang der **Sondierungsfolge**

$$(h(k) + s(n)) \bmod M \quad \text{für } n = 0, \dots, M - 1$$

Hashing mit linearer Sondierung illustriert

Hashfunktion: $h(k) = k \bmod 7$

Schlüssel **Sondierung** **Hashtabelle**



Suchen, Einfügen und Löschen beim Hashing mit Sondierung

- ▶ Das **Suchen** geschieht entlang der Sondierungsfolge. Der Schlüssel in dem jeweiligen Eintrag in der Hashtabelle wird untersucht:
 - ▶ Ist er gleich dem gesuchten Schlüssel: **erfolgreiche Suche**
 - ▶ Ist er gleich null: **erfolglose Suche**
 - ▶ Andernfalls geht es in der Sondierungsfolge weiter.
- ▶ Beim **Einfügen** folgt man gleichermaßen der Sondierungsfolge, bis ein freier Platz (Schlüssel null) gefunden wird.
- ▶ Das **Löschen** benötigt eine besondere Aufmerksamkeit, siehe Seite 47.
- ▶ Falls ein Element gelöscht wird, welches in der Sondierungsfolge eines später eingefügten Schlüssels k' übersprungen werden musste, dann könnte jener Schlüssel k' nicht mehr wiedergefunden werden. Die Suche entlang der Sondierungsfolge würde an dem frei gewordenen Eintrag abbrechen.
- ▶ Bei freien Einträgen die Suche *nicht* zu stoppen, ist keine Alternative, da sonst u.U. die komplette Hashtabelle durchsucht wird, was äußerst ineffizient wäre.

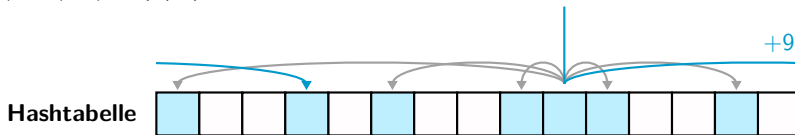
Primäre und sekundäre Häufungen

- ▶ Bei Kollisionsauflösung durch Sondierung entstehen oft **Häufungen** (*clusters*) in der Belegung der Hashtabellen.
- ▶ Häufungen führen zu langen Sondierungsfolgen (beim Einfügen und bei der Suche) und sind daher schlecht für die Performanz.
- ▶ Sobald eine Häufung entstanden ist, wächst die Wahrscheinlichkeit, dass weitere Kollisionen irgendwo mit dem Cluster passieren und er dadurch immer weiter wächst. Dies nennt man **primäre Häufung** (*primary clustering*).
- ▶ **Sekundäre Häufungen** bezeichnen speziell die Häufungen, die dadurch entstehen, dass viele Schlüssel auf dieselbe Hashadresse abgebildet werden und daher derselben Hashsequenz folgen.

Quadratisches Sondieren

- ▶ Mit **quadratischen Sondierungsfunktionen** können primäre Häufungen vermieden werden.
- ▶ Durch Alternieren des Vorzeichens wird die Streuung noch weiter verbessert.
- ▶ Somit ist **Quadratisches Sondieren** (*quadratic probing*) ein offenes Hashing mit der Sondierungsfunktion

$$s(n) = (-1)^n \cdot \lceil n/2 \rceil^2$$



- ▶ Wenn M als eine Primzahl der Form $4k + 3$ gewählt wird, dann durchläuft diese Sondierungsfolge alle M Hashadressen [Radke 1970].

- ▶ Quadratisches Sondieren vermeidet im Gegensatz zum linearen Sondieren die **primären** Häufungen.
- ▶ Das Problem der **sekundären** Häufungen bleibt bestehen, da die Sondierungsfolge für alle Schlüssel mit gleichem Hashcode dieselbe ist.
- ▶ Sekundäre Häufungen lassen sich nur dann vermeiden, wenn die Sondierungsfolge vom Schlüssel selbst abhängt und nicht nur von seinem Hashcode.




Double Hashing

- ▶ Als ein Verfahren, das sekundäre Häufungen durch eine Schlüssel-abhängige Sondierungsfunktion vermeidet, betrachtet wird das **Double Hashing**.
- ▶ Weitere, noch effizientere, aber auch komplexere Verfahren sind uniformes und randomisiertes Sondieren, siehe [Ottmann & Widmayer S.208ff].
- ▶ Beim Double Hashing hängt die Sondierungsfunktion von dem Schlüssel k ab. Dazu wird eine zweite, von der ersten verschiedene Hashfunktion $h'(k)$ verwendet:

$$s(n, k) = n \cdot h'(k)$$

- ▶ Es sollte $h'(k) \neq 0$ für alle k sein. Wenn M eine Primzahl ist, dann durchläuft die Sondierungsfolge alle M Hashadressen.
- ▶ Um sekundäre Häufungen möglichst effektiv zu vermeiden, sollte $h'(k)$ von $h(k)$ unabhängig sein. Genauer: Für alle Schlüssel k und k' sollte das Ereignis, dass die Schlüssel eine Kollision bei h verursachen ($h(k) = h(k')$) unabhängig von dem Ereignis sein, dass sie eine Kollision bei h' verursachen ($h'(k) = h'(k')$).
- ▶ Diese Bedingung wird von $h(k) = k \bmod M$ und $h'(k) = 1 + k \bmod (M - 2)$ erfüllt.

Die beiden Lösungsverfahren bei offenem Hashing

- ▶ **Verzögertes Löschen** (*lazy deletion*): Der Eintrag wird als gelöscht markiert und:
 - Einfügen (*put*) überschreibt einen als gelöscht markierten Eintrag.
 - Suche (*get*) wird bei als gelöscht markierten Einträgen fortgesetzt (so als ob der Eintrag noch existiert).
- ▶ **Sofortiges Löschen** (*eager deletion*): Schlüssel auf null setzen.
- ▶ Dabei müssen allerdings die folgenden Einträge in dem Cluster verschoben werden.
 - Dies ist komplizierter, als bei einer verketteten Liste, da sich in dem Cluster sowohl Überläufer von unterschiedlichen Hashadressen befinden können (siehe primäre Häufung) als auch reguläre Einträge.
- ▶ Daher werden alle folgenden Einträge in dem Cluster gelöscht und per `put()` wieder neu in die Hashtabelle eingetragen.
- ▶ Bei *double hashing* ist das Löschen noch komplizierter. 

Bemerkung zu Hashverfahren mit Sondierung

- ▶ Die Effizienz der offenen Hashverfahren nimmt sehr stark ab, wenn sich der Belegungsfaktor 1 nähert.
- ▶ Daher versucht man, den Belegungsfaktor **nicht weit über 0.5** steigen zu lassen.
- ▶ Zur Not wird die Hashtabelle vergrößert. Dies erfordert eine Änderung der Hashfunktion und ein Rehashing, ist also sehr rechenintensiv.
- ▶ Bei einem niedrigen Belegungsfaktor sind die offenen Hashverfahren wenig speichereffizient.

Effizienz von Hashverfahren mit linearer Sondierung

- Die Effizienz hängt stark von der Verteilung der Schlüssel ab. Unter der Annahme einer gleichmäßigen Verteilung, die durch eine geeignete Wahl der Hashfunktion halbwegs erreicht wird, gilt folgender Satz.

Effizienz von Hashing mit linearer Sondierung

Die durchschnittliche Anzahl von Sondierungen in einer Hashtabelle der Größe M mit N Schlüsseln und $\alpha = N/M$ ist

$$\begin{aligned} &\sim \frac{1}{2} \left(1 + \frac{1}{1 - \alpha} \right) && \text{für erfolgreiche Suche und} \\ &\sim \frac{1}{2} \left(1 + \frac{1}{(1 - \alpha)^2} \right) && \text{für erfolglose Suche und Einfügungen.} \end{aligned}$$

Bei einem Belegungsfaktor von $\alpha = 1/2$ (der in der Praxis nicht stark überschritten werden sollte) ergeben sich die ungefähren Werte $3/2$ und $5/2$.

Hashing mit Verkettung

- ▶ Nahezu konstante Laufzeit unter guten Bedingungen
- ▶ Laufzeit steigt langsam mit Füllgrad α .
- ▶ Overhead im Speicherbedarf durch externe Listen
- ▶ Weniger sensibel gegenüber ungünstigen Hashfunktionen

Hashing mit Sondierung

- ▶ Nahezu konstante Laufzeit unter guten Bedingungen
- ▶ Laufzeit steigt stark ab $\alpha > 0.5$.
- ▶ Speichereffizient bei guter Wahl der Tabellengröße
- ▶ Aufwändigere Implementation von *eager delete*

- ▶ Die Java Collections stellen die Klassen **HashSet** und **HashMap** zur Verfügung.
- ▶ Zur Benutzung mit eigenen Klassen, müssen die Methoden **hashCode()** und **equals()** **konsistent** überschrieben werden:
- ▶ Die Ausgabe von **hashCode()** muss verträglich mit **equals()** sein:
 - Gilt **a.equals(b)**, dann müssen **a.hashCode()** und **b.hashCode()** die gleichen Werte liefern.
- ▶ Die Berechnung von **hashCode()** sollte möglichst **effizient** sein.
Bei aufwändigeren Berechnungen sollte der Hashcode in einer Objektvariable gespeichert werden, um Mehrfachberechnungen zu vermeiden.
- ▶ Die Schlüssel sollten durch die Hashcodes möglichst **gleichmäßig verteilt** werden.

Randomisierte Algorithmen

- ▶ Algorithmen, die Zufallsentscheidungen einbeziehen, heißen **Randomisierte Algorithmen** (*randomized algorithms*).
- ▶ Das besprochene **universelle Hashing** ist ein randomisierter Algorithmus. Dabei wird die Randomisierung insbesondere eingesetzt, um gezielte Sabotage durch hervorgerufene Kollisionshäufungen zu vermeiden.
- ▶ Bei Optimierungsproblemen unterscheidet man zwischen zwei unterschiedlichen Arten von randomisierten Algorithmen:

- 1 **Las-Vegas-Algorithmen** bestimmen immer die optimale Lösung und im Erwartungswert ist die Laufzeit sehr effizient. Allerdings kann die Laufzeit selbst bei der gleichen Eingabe stark variieren.
 - ▶ Randomisiertes Quicksort hat als Erwartungswert eine Laufzeit in $O(n \log n)$ und im *worst-case* in $O(n^2)$.
 - ▶ Bei einer Variante der Las-Vegas-Algorithmen wird auch erlaubt, dass sie keine Lösung liefern. Aber wenn sie eine Lösung zurückgeben, muss es eine optimale sein.
- 2 **Monte-Carlo-Algorithmen** terminieren immer in effizienter Laufzeit, aber das Ergebnis ist nur mit einer gewissen Wahrscheinlichkeit optimal.
 - ▶ Ein Beispiel ist der *Monte Carlo Tree Search* Algorithmus. Er wird z.B. erfolgreich bei Spielen eingesetzt (Alpha Go u.a.).
 - Ein Las-Vegas-Algorithmus kann in einen Monte-Carlo-Algorithmus umgewandelt werden (Abbruchkriterium nach polynomieller Laufzeit und Ausgabe der bis dahin besten Lösung).

Folgende Fragen sollten Sie auf jeden Fall beantworten können

- ▶ Was sind die Vorteile von Hashing im Gegensatz zu Arrays und Listen?
- ▶ Wie läuft der Weg vom Schlüssel über Hashcodes zu Hashadressen?
- ▶ Warum passieren Kollisionen und was kann man dagegen tun?
- ▶ Wie funktioniert Hashing mit Verkettung?
- ▶ Wie funktioniert Hashing mit linearer Sondierung? Wie löschen?

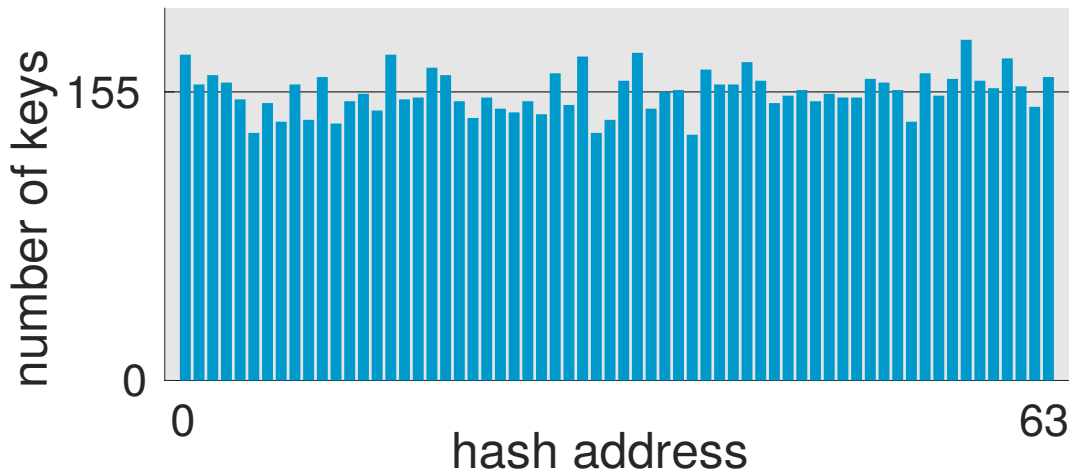
- ▶ Ottmann T, Widmayer P. *Algorithmen und Datenstrukturen*. Springer Verlag, 5. Auflage; 2011. ISBN: 978-3827428042
- ▶ Sedgwick & Wayne, *Algorithmen: Algorithmen und Datenstrukturen*, Pearson Studium, 4. Auflage, 2014. ISBN: 978-3868941845; in Teilen auch auf <http://www.cs.princeton.edu/IntroAlgsDS>
- ▶ TH Cormen, CE Leiserson, R Rivest, C Stein, *Algorithmen - Eine Einführung*. De Gruyter Oldenbourg, 4. Auflage; 2013. ISBN: 978-3486748611
- ▶ Knuth DE. *The art of computer programming, Vol. 3: Sorting and Searching*. Addison-Wesley, Reading, Massachusetts, 1973.
- ▶ Lum VY, Yuen PS, Dodd M. *Key-to-address transform techniques: a fundamental performance study on large existing formatted files*. Communications of the ACM 14(4):228-39, 1971.
- ▶ Radke CE. *The use of quadratic residue research*. Communications of the ACM. 1970; 13(2):103-5.
- ▶ Crosby SA, Wallach DS. *Denial of Service via Algorithmic Complexity Attacks*. In: USENIX Security Symposium 2003 (pp. 29-44). http://static.usenix.org/event/sec03/tech/full_papers/home/staff/alex/export/twycross/crosby/crosby.html
- ▶ Flajolet P, Grabner PJ, Kirschenhofer P, Prodinger H. *On Ramanujan's Q-function*. Journal of Computational and Applied Mathematics. 1995; 58(1):103-16.

Anhang:

- ▶ Histogramm für die Multiplikationsmethode: S. 57
- ▶ Details zum universellen Hashing: S. 58

Histogramm für die Multiplikationsmethode

- Das folgende Histogramm zeigt die Verteilung der Schlüssel auf die Hashadressen analog zu Seite ?? hier mit der Multiplikationsmethode:





- ▶ Wir betrachten ein feste Universum U und ein festes M (Größe der Hashtabelle).
- ▶ Sei \mathcal{H} eine Familie von Hashfunktionen. \mathcal{H} heißt **universell**, falls für zwei beliebige unterschiedliche Schlüssel $k, l \in U$ gilt:

$$\frac{|\{h \in \mathcal{H} \mid h(k) = h(l)\}|}{|\mathcal{H}|} \leq \frac{1}{M}$$

- ▶ Dies bedeutet, dass im Durchschnitt über alle Hashfunktionen aus \mathcal{H} die Wahrscheinlichkeit einer Kollision nicht größer ist, als wenn die zwei Hashadressen zufällig aus $\{0, \dots, M-1\}$ ausgewählt werden.
- ▶ Im Schnitt über \mathcal{H} betrachtet, kann es also keine systematischen Häufungen geben, die bei einzelnen Hashfunktionen durch spezielle Verteilungen der Schlüssel verursacht werden können. (Präzisierung auf der nächsten Seite).



Eigenschaft einer universellen Menge von Hashfunktionen

Sei \mathcal{H} eine universelle Menge von Hashfunktionen. Wir nehmen an, dass in einer Hashtabelle bereits N Schlüssel gespeichert sind. Dann ist für einen gegebenen Schlüssel k der Anteil der Hashfunktionen $h \in \mathcal{H}$, für den es eine Kollision gibt N/M .

Beweis siehe [Ottmann & Widmayer S. 196].



Universelle Familie von Hashfunktionen

Zu einem gegebenen Universum U sei $p \geq |U|$ eine Primzahl. Für ganze Zahlen a, b definieren wir

$$h_{a,b} : U \rightarrow \{0, \dots, M-1\}; \quad k \mapsto ((ak + b) \bmod p) \bmod M$$

Dann ist die Menge $\mathcal{H} = \{h_{a,b} \mid 1 \leq a < p \text{ und } 0 \leq b < p\}$ ist eine universelle Familie von Hashfunktionen.

Beweis siehe [Ottmann & Widmayer S. 197].

Index

associative array, 2

Belegungsfaktor, 23

closed addressing, 32

closed hashing, 32

cluster, 43

clustering

 primary, 43

 secondary, 43

delete

 eager, 8, 47

 lazy, 7, 8, 47

dictionary, 2

Divisions-Rest-Methode, 27

Double Hashing, 46

eager deletion, 8, 47

Effizienz

 lineare Sondierung, 49

 separate chaining, 38

Fibonacci-Hash, 26

Goldener Schnitt, 26

Hashadresse, 16

Hashcode, 16

`hashCode()`, 18

Hashfunktion, 16

 Divisions-Rest-Methode, 27

 Multiplikationsethode, 26

 perfekte, 22

Hashing, 10, 21

 mit Sondierung, 32, 40

 mit Verkettung, 32

 offenes, 40, 41

 perfektes, 22

Hashtabelle, 15, 16

Hashverfahren mit Sondierung,
32

Hashverfahren mit Verkettung,
32, 33

Häufung, 43

 primäre, 43

 sekundäre, 43

Kollision, 21, 33

Las-Vegas-Algorithmus, 53

lazy deletion, 8, 47

linear probing, 40

Lineare Sondierung, 40, 41

Liste

 verkettete, 6

load factor, 23

Löschen

 sofortiges, 8

 verzögertes, 8

map, 2

Monte-Carlo-Algorithmus, 53

Multiplikationsmethode, 26

Offene Hashverfahren, 32

open addressing, 32, 40

open hashing, 32

perfektes Hashing, 22

primary clustering, 43

primäre Häufung, 43

- probing
 - probing*
 - linear*, 40
 - quadratic, 44
- quadratic probing*, 44
- Quadratisches Sondieren, 44
- Randomisierter Algorithmus, 52
- Randomisierung, 30
- Randomized algorithm*, 52
- secondary clustering, 43

- Sekundäre Häufung, 43
- separate chaining, 33
- separate chaining*, 32, 39
- Sofortiges Löschen, 47
- Sofotiges Löschen, 8
- Sondieren
 - randomisiertes, 46
 - uniformes, 46
- Sondierung
 - lineare, 40, 41
 - quadratische, 44

- Sondierungsfolge, 40
- Sondierungsfunktion, 40
- symbol table*, 2
- Symboltabellen, 2
- universell, 58
- Universelles Hashing, 30
- Verzögertes Löschen, 8, 47
- Wörterbuch, 2
- Überläufer, 32, 33