Hashing

Hash-Tabellen und -Funktionen
Hash-Tabellen in Java

Einleitung

- Anwendung von Hashing: Verfahren für dynamisch veränderliche Menge von Objekten mit effizienten Grundoperationen (aka. Wörterbuchoperationen)
  - Suchen
  - Einfügen
  - Löschen
- Probleme bei Implementierung durch schon bekannte Datenstrukturen
  - Listen Suche ist aufwändig (O(*N*)) und vor jedem Löschen und Einfügen (an sich effizient) ist eine Suche notwendig
  - Bäume: Suche ist bei balancierten Such-Bäumen effizient (O(log(N)), aber nach jedem Löschen und Einfügen ist Ausbalancieren notwendig
  - Reihungen: Suche schnell aber dynamisches Wachstum teuer, viel Speicher notwendig bei dünn besetzten Reihungen

Komponenten von Hashverfahren (Streuspeicherverfahren)

- Hash-Funktion (*hash function*)
  - Bildet Objekte ab auf ganze Zahlen (int, hash codes)
  - Hash codes dienen als Schlüssel (*key*) zur Identifikation der Objekte
  - Es ist nicht ausgeschlossen, dass zwei verschiedene Objekte den selben Schlüssel haben
- Hash-Tabelle (*hash table*)
  - Speichert Objekte unter ihrem hash code ab
  - Implementierung: setze array index = hash code

Verallgemeinerung von Hashing

- Nicht-numerische Schlüssel liefern
  - Abbildungen von Objekten („Schlüsselobjekte“) auf Objekte („Wertobjekte“)
  - Aka. „Übersetzungstabellen“, „Wörterbücher“ (*dictionaries*)
- Prinzip: Wertobjekt unter Schlüsselobjekt (z.B. *string*) wird als Paar
  <Schlüsselobjekt, Wertobjekt>
  in Hash-Tabelle unter Hash Code des Schlüsselobjekts gespeichert
- Java Implementierung ist so organisiert

Einfachstes Beispiel Hash-Tabelle

- Numerische Schlüssel dienen als Index in ein Array
- Sinnvoll wenn:
  - Hash-Funktion injektiv (verschiedene Objekte haben verschiedene Schlüssel)
  - genügend Platz vorhanden, um alle möglichen Schlüssel als Index unterzubringen
    - Das ist oft nicht der Fall: Schlüsselbereich typischerweise dünn besetzt
- In Praxis aber nicht injektive Hash-Funktionen
  - Kollision: zwei Objekte haben denselben Schlüssel (spezielle Überlaufverfahren erforderlich)

Beispiel: Mitarbeiterdatenbank

- Objekte sind durch Namen der Mitarbeiter gegeben (*strings*)
- Hash-Funktion: $f(s) = i$ g.d.w. der erste Buchstabe des Nachnamens der i-te Buchstabe im Alphabet ist

Anton Wagner	W → 23
Doris Bach	B → 2
Doris May	M → 13
Friedrich Dörig	D → 4

- Array der Länge 26 reicht aus für Hash-Tabelle
- Achtung: Keine Behandlung von Kollisionen!
- Schneller Zugriff in konstanter Zeit O(1)

Hash-Funktionen

- Hash-Funktion allgemein:
  h : Menge der Objekte → N (oder N^k)
- Objekt s wird Speicherplatz HT[h(s)] zugewiesen
- Beispiel
  - h (´´Doris Bach´´) = 2 führt zu HT[2]= ´´Doris Bach´´
- Zur Vermeidung von Kollisionen:
  - Tabelle hinreichend groß wählen
  - Hash-Funktion sollte möglichst vom ganzen Objekt abhängen
    - z.B. bei Zeichenketten abhängig von allen Zeichen

Hash-Funktionen für Strings

- $h(s)$ = Konkatenation des ASCII-Codes der Zeichen in s,
- h (´´INFO´´) = $495E465F_{16}$

Zeichen	I	N	F	O
ASCII	49_{16}	$5E_{16}$	46_{16}	$5F_{16}$

- Problem dabei: $h(s)$ ergibt zu große Indizes
  - (Adressraum hat nur 32 oder 64 Bit)
- Lösung: Reduktion von $h(s)$ durch Rechnung *modulo m*
  - Hash-Tabelle hat Länge von maximal m Einträgen
  - Kollisionen nicht ausgeschlossen
- Bsp. Mit $m = 83$ ist h(´´INFO´´) = 18_{10}, denn $495E465F_{16}$ *modulo* $83_{10} = 1230915167_{10}$ *modulo* $83_{10} = 18_{10}$

Wiederholung

- ‚I‘ * 256^3 + ‚N‘ * 256^2 + ‚F‘ * 256^1 + ‚O‘ * 256^0

Zeichen	I	N	F	O
ASCII	49_{16}	$5E_{16}$	46_{16}	$5F_{16}$

- $49_{16} = 4*16^1 + 9 * 16^0 = 73_{10}$

Horner Schema

Beispiel: ‚INFO‘

- Z := ((‚I‘ * 256 +‘N‘)*256+‘F‘)*256+‘O‘
- Z%5 = (((((‚I‘ * 256 +‘N‘)%5)*256+‘F‘)%5) *256+‘O‘)%5

Horner Schema zu Modulo Rechnung

- Berechnung von $h(s)$ mit modulo Rechnung nicht in CPU-Arithmetik (in long) durchführbar wenn s.length > 8
- Zu berechnen ist mit Horner-Schema:
  - sei $z(i,s)$, $i = 0, ..., n-1$, der Integer ASCII Code des i-ten Zeichens des Strings s der Länge n (0-tes Zeichen steht vorne)
  - $Z(s) = Z_{n-1}(s)$ ergibt sich aus Iteration
    - $Z_0(s) = z(0,s)$
    - $Z_k(s) = 256\ Z_{k-1}(s) + z(k,s)$ für $k = 1, 2, ..., n-1$
  - $h(s) = Z(s) \bmod m$
  - $Z(s)$ ist i.d.R. zu groß für Zahlen im Format long

Horner-Schema in Rechnerarithmetik

- Modulo Rechnung ist aber Komponentenweise machbar:
  - $(a + b)$ modulo m = ((a modulo m) + (b modulo m)) modulo m,
  - $(a * b)$ modulo m = ((a modulo m) * (b modulo m)) modulo m.
- Damit erhalten wir:
  - $h(s) = Z(s)$ modulo $m = h_{n-1}(s)$ ergibt sich aus Iteration
    - $h_0(s) = z(0,s)$
    - $h_k(s) = (256 \cdot h_{k-1}(s) + z(k,s))$ mod m für $k = 1, 2, ..., n-1$
  - JAVA Code Fragment dazu

```
String s = "INFORMATIK";
int z = 0;
for (int i=0; i<s.length(); i++)
z=(z*256 + s.charAt(i)) % m;
```

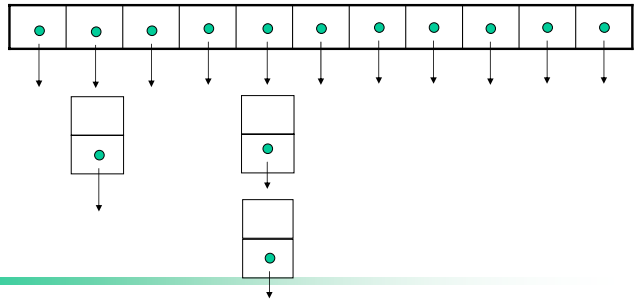
Wahl der modulo Basis

- Es gibt schlechte Wahlen für *m*
  - *m* = 256 führt dazu, daß *h*(*s*) nur vom letzten Zeichen abhängt.
- Primzahlen *m* führen i.d.R. zu guten Hash-Funktionen

Kollisionsbehandlung I: Listenlösung

- Kollision besteht beim Einfügen zwei verschiedener Objekte mit demselben Schlüssel, $h(s_1) = h(s_2)$, $s_1 \neq s_2$
- Verkettung von Überläufen: Objekte werden in Listen gehalten

HT:



Kollisionsbehandlung II: open hashing

- Offenes Adressieren (open hashing): Wenn HT[h(s)] schon besetzt ist, suche erste freie Stelle *h(s)+k, k=1,2,3, ...*
- Bei Suche nach Objekt s prüfe zuerst HT[h(s)]. Wenn dort ein Objekt ungleich s steht, suche weiter bei *h(s)+k, k=1,2,3, ...*
  Bis entweder s oder eine unbelegte Stelle gefunden wurde
- Es können in Tabelle Häufungen von belegten Positionen auftreten, die die Effizienz mindern
- Dazu gibt es komplexere *open hashing* Strategien

Hash-Funktionen in JAVA

- Klasse Object ist Basisklasse jeder Klasse und enthält virtuelle Methode int hashCode()
- Durch Überschreiben der Methode in abgeleiteter Klasse ist Anpassung an Anwendung möglich
- *Achtung:* Objekte werden durch Referenz gespeichert, hashCode() benutzt nur diese Referenz (eine Kopie mit anderer Referenz ergibt anderen hashCode!)

Hashing Konzept in JAVA

- Paket java.util enthält Klasse Hashtable
- Klasse Hashtable greift auf die Methode hashCode zu und ist durch sie implizit parametrisiert
- Implementiert ist allgemeiner eine Abbildung (Interface map) einer Menge von Schlüssel-Objekten in eine Menge von Wertobjekten
- Entspricht einer endlichen, partiellen Funktion

Das Interface Map I

Die Klasse Hashtable implementiert Interface Map

Spezifikationen für zu implementierende Methoden im Interface Map:

Object put (Object key, Object value)
Legt value in der Map unter key ab. Falls schon ein Element dem key in der Map zugeordnet war, so wird dieses überschrieben. Das Ergebnis ist dieses bisher dort abgelegte Element, oder null, falls es kein solches gab.

Object get (Object key)
Das Ergebnis ist das Element, das in der Map unter key gespeichert ist.

Object remove(Object key)
Löscht das unter key abgelegte Element und gibt es zurück. Falls es kein zu key gehörendes Element gab, wird null zurückgegeben.

Das Interface Map II

- Realisierung
  - Zum Schlüssel-Objekt wird ein numerischer hashCode berechnet
  - In der Hash-Tabelle wird beim Index hashCode das Schlüsselobjekt und das Wertobjekt abgelegt
- Kollision:
  - Open hashing wird verwendet zur Behandlung von Kollisionen
  - Schlüssel werden verglichen mit virtueller Methode equals, die in Klasse Object definiert ist
  - Achtung: nur die Referenz des Schlüsselobjektes wird hier verwendet

Das Interface Map III

- Schlag nach bei www.sun.com
  - Interface `Map`
  - Klasse `Hashtable`

Beispiel 1: Mitarbeiterliste

- Wert-Objekte: Namen der Personen
- Schlüssel-Objekte: Hash-codes der Strings der Namen
- Table.put benötigt als Parameter Klassenobjekte, daher ist nicht int sondern Integer (Hüllklasse) für Schlüssel-Objekt zu verwenden

```
import java.util.*;
public class HashtableTestSimple {
  public static void main(String[] args) {
    Hashtable table = new Hashtable();

    // Trage Mitarbeiter in
    // Hashtable table ein.
    String s;
    Integer k;
```


Beispiel 1: Mitarbeiterliste (cont´d)

```
s = "Anton Wagner";
k = new Integer(s.hashCode());
table.put(k, s);

s = "Doris Bach";
k = new Integer(s.hashCode());
table.put(k, s);

s = "Doris May";
k = new Integer(s.hashCode());
table.put(k, s);

s = "Friedrich Dörig";
k = new Integer(s.hashCode());
table.put(k, s);
```

- Achtung: Interner Index in Hash-Tabelle ergibt sich aus hashCode des Schlüsselobjektes k
- Also werden für ein s insgesamt zweimal Hash-Codes berechnet:
  - s -> s.hashCode() -> (s.hashCode()).hashCode()

Beispiel 1: Mitarbeiterliste (cont´d)

```
// Einige Tests
Integer key;
String s1, s2;
s1 = "Doris May";
key = new Integer(s1.hashCode());
Object e; // get has return value of type object
e = table.get(key);
if (e == null)
   System.out.println("Kein zu " + key
     + " gehörendes Element vorhanden!");
else {
     s2= (String) e;
     System.out.println("Gefundenes Element zu " + key
                        + ": " + s2);
}
 }
}
```


Beispiel 2: Personen/Geburtstage

- Wert-Objekte: Namen von Personen
- Schlüssel-Objekte: Geburtsdaten
- Annahme: keine zwei Personen am selben Tag geboren
  - Ist das realistisch? Ab wieviel Personen ist die Chance 50:50, daß zwei von ihnen am selben Tag Geburtstag feiern können?
- Darstellung der Schlüsselobjekte in Klasse `Date`

```
public class Date {
    private byte day, month;
    private short year;
    // Konstruktoren
    public Date(){day=1; month=1;}
    public Date(short y) {this(); year=y;}
    public Date(byte m, short y) {this(y); month = m;}
    public Date(byte d, byte m, short y)
            {this(m,y); day=d;}
```

Beispiel 2: Personen/Geburtstage (cont´d)

```
    // Zusätzlicher Konstruktor
    public Date(int d, int m, int y){
       day = (byte) d; month = (byte) m; year = (short) y;
    }
    // Selektoren
    public byte getDay() { return day; }
    public byte getMonth(){ return month; }
    public short getYear(){ return year; }

    public void setDay(byte d) { day=d;}
    public void setMonth(byte m) { month=m;}
    public void setYear(short y) { year=y;}

    // Darsteller
    public String toString() {
      return (day + "." + month + "." + year);
  }
}
```


Beispiel 2: Personen/Geburtstage (cont´d)

- Überschreiben von Methoden `equals` und `hashCode` nötig, da Klasse `Date` verschiedene Instanzierungen haben kann mit demselben Datum

```
public class Date {
//...
// Attribute und Methoden wie zuvor
// Überschreibe Methode equals aus Basisklasse Object
  /**
   * A Date is equal to an Object if it is
   * a Date representing the same date.
   */
  public boolean equals(Object obj) {
    if (!(obj instanceof Date))
      return false;
    return ( ((Date) obj).day == day &&
          ((Date) obj).month==month &&
          ((Date) obj).year==year );
  }
// Überschreibe Methode hashCode aus Basisklasse Object
  public int hashCode() {
   return day+month*32+year*1024;
  }
}
```


Beispiel 2: Personen/Geburtstage (cont´d)

```
public class HashtableTest {
  public static void main(String[] args) {
    java.util.Hashtable table = new java.util.Hashtable();
    String s;
    Date d;
    // Trage Daten in Tabelle ein
    s = "Anton Wagner";
    d = new Date(12,2,1960);
    table.put(d, s);
    s = "Doris Bach";
    d = new Date(27,4,1970);
    table.put(d, s);
    s = "Doris May";
    d = new Date(24,12,1973);
    table.put(d, s);
    s = "Friedrich Dörig";
    d = new Date(1,1,1953);
    table.put(d, s);
```


Beispiel 2: Personen/Geburtstage (cont´d)

```
    // Suche einen Mitarbeiter mit Geburtstag 27.04.1970
    Date sd = new Date(27,4,1970);
    String sr;

    Object e; // get has return value of type object
    e = table.get(sd);
    if (e == null)
      System.out.println("Kein zu " + sd +
                  " gehörendes Element vorhanden!");
    else {
      sr = (String) e;
      System.out.println("Gefundenes Element zu " + sd +
                  ": " + sr);
}}}
```

Die Ausgabe des Hauptprogramms der Klasse HashtableTest ist dann:

```
    Gefundenes Element zu 27.4.1970: Doris Bach
```