# Programmiertechnik II

## Hash-Tabellen

# Überblick

- Hashfunktionen: Abbildung von Schlüsseln auf Zahlen
  - Hashwert: Wert der Hashfunktion
- Hashtabelle: Symboltabelle, die mit Hashwerten indiziert ist
- Kollision: Paar von Schlüsseln mit gleichem Index in Hashtabelle
  - Kollisionsauflösung: Suche in kollidierenden Schlüsseln
- Kompromiss zwischen Speicherverbrauch und Rechenzeit
  - unbegrenzter Speicher: Schlüssel ist Index; Rechenzeit ist konstant
  - unbegrenzte Zeit: lineare Suche in Schlüsseln, kein zusätzlicher Speicher

# Hashfunktionen

- Abbildung von Schlüsseln auf Werte von 0..M-1 (M: Größe der Tabelle)
  - Idealfall: Hashwerte sind gleichverteilt
- Hashfunktion hängt vom Schlüsseltyp ab
  - üblich: Berechnung des Hashwerts auf Basis der internen Darstellung des Schlüsselwerts
  - Java: int java.lang.Object.hashCode();
- Beispiel: Gleitkommazahlen f von B bis E
  - Abbildung auf 0..1, dann Multiplikation mit M
  - hash(f) = $\lfloor (f-B)/(E-B)*M \rfloor$
- Beispiel: Natürliche Zahlen z von 0 bis $2^w$

1. Abbildung auf Gleitkommazahlen von 0..1, dann wie Gleitkommazahlen
   - Gleichverteilung nur dann erreicht, wenn Zahlen gleichverteilt sind
2. Modulo-Hashing: hash(z) = z % M
   - in vielen Fällen erscheinen die Reste modulo M wie zufällig

# Hashfunktionen (2)

- Gleichverteilung: Alle Bits des Schlüssels sollten berücksichtigt werden
- Modulo-Hashing: falls M eine Zweierpotenz, werden nur die unteren log M Bits berücksichtigt
- üblich: M sollte eine Primzahl sein
  - mögliche Werte für M sind oft in Programm kodiert
    - etwa Mersenne-Primzahlen: $2^t$-1 für t = 2, 3, 5, 7, 13, 17, 19, 31, ...
- Schnelles Hashing: Design von Hashfunktionen beachtet nicht nur Gleichverteilung, sondern auch effiziente Implementierbarkeit
- Sicheres Hashing (Kryptologie): Hash-Funktion sollte nicht umkehrbar sein
  - MD-5 (RFC 1321): Message Digest #5
  - SHA-224 (RFC 3874): Secure Hash Algorithm

# String-Hashing

- Hashfunktionen für Strings: Berücksichtigung aller Zeichen
- Betrachtung des Strings als ganze Zahl, etwa zur Basis 256 (8-bit Zeichen) oder 65536 (16-Bit-Zeichen)
  - Integer-Arithmetik auf großen Zahlen ist aber ineffizient
- Modulo-Hashing mit Hilfe des Hornerschemas
  - Beispiel: String a,b,c,d, 8-bit-Zeichen (0..255)
  - $(a*256^3+b*256^2+c*256+d) \% M =$
    $(((a*256 + b)*256 + c)*256+d) \% M =$
    $(((a*256 + b) \% M * 256 + c) \% M *256 + d) \% M$
  - Zwischenergebnisse verlassen niemals den Wertebereich für int
- Verwendung von 256 als Faktor nicht formal erforderlich
  - Verteilung wird u.U. besser, wenn Faktor keine Zweierpotenz

# String-Hashing (2)

```
static int hash(String s, int M)
{
  int h = 0, a = 31; // 31 ist Faktor im JDK
  for (int i = 0; i < s.length(); i++)
      h = (a*h + s.charAt(i)) % M;
  return h;
}
```

# String-Hashing (3)

- java.lang.String.hashCode:
  - Faktor ist 31
  - hashCode wird nur beim ersten Zugriff berechnet, danach gespeichert (cached)
- Wiederverwendung von Hash-Funktionen:
  - Objekte implementieren nur .hashCode()
    - Rückgabetyp ist z.B. int
  - Rufer von .hashCode() führen Modulo-Operation selbst aus

# Perfektes Hashing

- Annahme: Menge M der Schlüssel ist vorab bekannt
  - Beispiel: Namen von Methoden eines Objekts
  - Beispiel: Liste von Schlüsselwörtern einer Programmiersprache
- Perfekte Hashfunktion: Funktionswerte kollidieren nicht
  - minimale perfekte Hashfunktion: Funktionswerte sind von 0..#M-1
- Fox, Heath, Chen, Daoud. Practical minimal perfect hash functions for large databases
  - CACM, 35(1):105-121, Januar 1992
- Schmidt. GPERF – A Perfect Hash Function Generator

# Statisches Hashing

- Hashtabelle T fester Größe (0..M-1)
- Schlüssel werden auf den Bereich 0..M-1 gehasht
- Bei Kollisionen werden Einträge mit gleichem Hashwert in verketteter Liste geführt
  - “overflow table”
- insert: Berechnung des Hashwerts h, Eintrag in T[h]
  - wahlweise am Anfang oder am Ende der verketteten Liste
- lookup: Berechnung des Hashwerts h, Suche nach Schlüssel in T[h]
  - Schlüsselvergleich in jedem Fall erforderlich, außer wenn T[h] leer ist
- Effizienz: lookup benötigt im Mittel N/M Vergleiche
  - Annahme: Hashwerte sind gleichverteilt
  - im Mittel O(1), falls N < M
    - O(1) im worst case, falls Hashfunktion perfekt

# Offene Adressierung: Lineares Sondieren

- Problem des statischen Hashings: Speicherbedarf für *overflow table*
- Lösung: Hashing mit offener Adressierung (open-addressing)
  - Schlüssel hat nicht einen Hashwert, sondern viele
  - Hash-Funktionen $h_0$, $h_1$, $h_2$, ...
- Lineares Sondieren: Weitere Hashfunktionen ergeben sich durch lineare Verschiebung
  - $h_n(x) = (h(x)+n) \% M$
  - Von h(x) beginnend wird der nächste freie Slot gesucht
- Problem: mit steigender Tabellenbelegung werden Kollisionen wahrscheinlicher
  - Füllstand (load factor): N/M (offenes Hashing: N/M < 1)
- Problem: Clusterbildung
  - auch erfolglose Suche muss der Reihe nach alle Schlüssel testen

# Offene Adressierung: Quadratisches Sondieren

- Idee: alternative Adressen mit quadratischem Abstand von h(x)
- Verfeinerung: Abwechselnd positive und negative Offsets
  - h, h+1, h-1, h+4, h-4, h+9, h-9, ...
  - lookup mit anderem Hashwert muss nicht das ganze Cluster durchmustern
    - Etwa: h´ = h+1, durchsucht h+1, h+2, h, h+5, h-3, ...
- M prim, M = 4j+3: alternatives quadratisches Sondieren trifft letztlich alle Tabellenpositionen

# Offene Adressierung: Löschen

- Problem: Suche in Hashtabelle bricht ab, wenn ein Eintrag leer ist
  - löscht man Index $h_1$ aus Folge $h_0$, $h_1$, $h_2$, dann kann $h_2$ nicht mehr gefunden werden
- Lösung für lineares Sondieren: Neuindizierung
  - Lösche Eintrag T[h]
  - danach Neueintragen von T[h+1], T[h+2], ... bis Null-Eintrag gefunden ist
- Quadratisches Sondieren: Liste neu zu indizierender Einträge ist schwer zu ermitteln
  - Lösung: Ersetze Eintrag durch Wächter
    - Suche wird über Wächter-Eintrag “hinwegsuchen”
    - Neueintrag kann Wächter durch gültigen Eintrag ersetzen

# Dynamische Hashtabellen

- Mit wachsendem Füllstand wird Suche ineffizient
  - Übergang zu linearer Suche
  - offene Adressierung: Tabelle ist letztlich voll
- Lösung: Vergrößerung der Tabelle
  - etwa: Verdopplung (oder Vergrößerung um anderen Faktor)
  - Neuindizierung aller Einträge
    - Caching der Hashwerte?
- Wann soll Vergrößerung ausgelöst werden?
  - wenn Füllstand Grenzwert überschreitet
    - etwa: Tabelle ist halbvoll
  - wenn amortisierte Performanz sinkt
    - etwa: wenn mittlere Zahl der Vergleiche pro Operation größer als 4 wird
  - Berücksichtigung von Wächtereinträgen:
    - Verkleinerung der Tabelle, wenn Zahl der Wächtereinträge “groß” wird

# Hashtabellen und Binäre Suchbäume

- Symboltabellen sind üblicherweise als Hashtabellen implementiert
  - für “gutartige” Schlüsselmengen O(1) falls Füllstand klein
  - Schlüssel müssen keiner Ordnungsrelation unterliegen
    - aber: Hashing muss unterstützt sein
    - OO: Hashwert darf sich über die Lebenszeit eines Objekts nicht ändern; gleiche Objekte müssen gleichen Hashwert liefern
- Binäre Bäume garantieren bessere Leistung im schlechten Fall
  - Bäume: O(log N); Hashtabellen können zu linearer Suche entarten
  - Verteilungseigenschaften der Hashfunktion kritisch für Suchzeiten
  - Echtzeitsysteme: Zugriffszeit muss vorhersagbar sein