Algorithmen und Datenstrukturen II: Hashverfahren

Prof. Dr. Oliver Braun

Letzte Änderung: 10.05.2017 16:21

Hashverfahren

- bisher
  - jeder Datensatz durch eindeutigen Schlüssel kK gekennzeichnet
  - zur Suche mehrere Schlüsselvergleiche
- statt Schlüsselvergleiche **Berechnung** wo der Datensatz mit Schlüssel k gespeichert ist
- Datensätze werden in linearem Feld mit Indizes $0, ..., m-1$ gespeichert
  - **Hashtabelle**
  - m ist **Größe** der Hashtabelle
- **Hashfunktion** $h : \mathcal{K} \to \{0, ..., m-1\}$ ordnet jedem Schlüssel k einen Index $h(k)$ mit $0 \leq h(k) \leq m-1$ zu, die **Hashadresse**
- für Hashtabelle der Größe m, die gerade n Schlüssel speichert, heißt $\alpha = n/m$ **Belegungsdichte**

Kollisionen

- Hashfunktion ist im Allgemeinen nicht injektiv
  - zwei Schlüssel k, k' mit $h(k) = h(k')$ heißen **Synonyme**
  - ergeben **Adresskollision**
- Forderung an Hashverfahren
  1. möglichst wenig Kollisionen $\Rightarrow$ Wahl einer "guten" Hashfunktion
  2. Adresskollisionen sollen möglichst effizient aufgelöst werden
- d.h. **Hashverfahren = Hashfunktion + Überlaufstrategie**
- im schlimmsten Fall sind Hashverfahren sehr ineffizient
- im Durchschnitt aber weitaus effizienter als Schlüsselvergleiche
  - nicht abhängig von Anzahl gespeicherter Schlüssel

Wahl der Hashfunktion

- gute Hashfunktion
  - leicht und schnell berechenbar
  - gleichmäßige Verteilung der zu speichernden Datensätze im Adressbereich
- gut verteilt, selbst wenn Schlüssel nicht gut verteilt sind
- Adresskollisionen auch bei optimal gewählter Hashfunktion
  - *Geburtstags-Paradoxon*: Wenn mindesten 23 Personen in einem Raum sind, haben wahrscheinlich 2 davon am gleichen Tag des Jahres Geburtstag.
  - Allgemein: wenn eine Hashfunktion $\sqrt{\pi m/2}$ Schlüssel auf eine Hashtabelle der Größe m abbildet, gibt es wahrscheinlich eine Adresskollisionen
  - für $m = 365$ gilt $\sqrt{\pi m/2} = 23, ...$
- im Folgenden werden nur nicht negative ganzahlige Schlüssel betrachtet

Divisions-Rest-Methode

- nahe liegendes Verfahren zur Erzeugung Hashadresse
  $h(k), 0 \leq h(k) \leq m-1$ zu $k \in \mathbb{N}_0$ ist
  $h(k) = k \bmod m$
- entscheidend: gute Wahl von m
- schlecht:
  - gerade $m \Rightarrow h(k)$ gerade wenn k gerade und ungerade wenn k ungerade
  - Potenz der Basis des Zahlensystems $\Rightarrow$ nur letzte Stellen zählen, Rest ist weg
- praktisch bewährt: Primzahl die keine der Zahlen $r^n \pm j$ teilt, mit r Basis des Zahlensystems der Schlüssel und n und j kleine nicht negative ganze Zahlen

Die multiplikative Methode

- der gegebene Schlüssel wird mit einer irrationalen Zahl multipliziert, der ganzzahlige Anteil des Resultats wird abgeschnitten
  - verschiedene Werte zwischen 0 und 1 für verschiedene Schlüssel
  - für Schlüssel $1, 2, 3, \ldots$ ziemlich gleichmäßig verteilt im Interval $[0, 1)$
- für alle Zahlen $, 0 \leq \Theta \leq 1$, führt der goldene Schnitt $\Phi^{-1} = \frac{\sqrt{5}-1}{2} \approx 0,6180339887$ zur gleichmäßigsten Verteilung. Wir erhalten als Hashfunktion $h(k) = \lfloor m(k\Phi^{-1} - \lfloor k\Phi^{-1} \rfloor) \rfloor$

Weitere Methoden

- Ziffernanalyse
- Mid-square-Methode
- Faltung
- algebraische Verschlüsselung
- ...
- Studie hat festgestellt, dass nach Divisions-Rest-Methode arbeitende Hashfunktion im Durchschnitt die besten Resultate liefert

Perfektes Hashing

- Anzahl der zu speichernden Schlüssel nicht größer als Anzahl Speicherplätze
  - kollisionsfreie Speicherung möglich
- wir ordnen die Schlüssel lexikografisch und bilden jeden auf seine Ordnungsnummer ab
  - Menge möglicher Schlüssel muss vorab bekannt sein
  - perfekte Hashfunktion
- Anwendungsfälle nur selten, z.B. Zuordnung von Schlüsselwörtern einer Programmiersprache zu festen Plätzen einer Symboltabelle

Universelles Hashing

- sobald Hashfunktion fest, kann man viele Schlüssel finden, die auf die selbe Hashadresse abgebildet werden
- Lösung: Hashfunktion aus einer sorgfältig gewählten Menge H aus Hashfunktionen auswählen
- sei
  - H endliche Kollektion von Hashfunktionen, so dass
  - jede Funktion $h \in H$ jeden Schlüssel im Universum K aller möglichen Schlüssel
  - auf eine Hashadresse aus $\{0, ..., m-1\}$ abbildet
- H heißt **universell**, wenn für je zwei verschiedene Schlüssel x, y K gilt:

$$\frac{|\{h \in H : h(x) = h(y)\}|}{|H|} \leq \frac{1}{m}$$

Konstruktion einer universellen Klasse von Hashfunktionen

- Annahmen
  - alle Schlüssel sind nicht negative ganze Zahlen
  - $|\mathcal{K}| = p$ eine Primzahl
  - zur Vereinfachung: $\mathcal{K} = \{0, ..., p-1\}$
- für zwei beliebige Zahlen $a \in \{1, ..., p-1\}$ und $b \in \{0, ..., p-1\}$ sei die Funktion $h_{a,b} : \mathcal{K} \to \{0, ..., m-1\}$ wie folgt definiert:
  $h_{a,b}(x) = ((ax + b) \bmod p) \bmod m$
- Dann gilt: Die Klasse $\mathcal{H} = \{h_{a,b} | 1 \leq a < p \text{ und } 0 \leq b < p\}$ ist eine universelle Klasse von Hashfunktionen
- Beweis in Ottmann & Widmayer, Algorithmen und Datenstrukturen

Hashverfahren mit Verkettung der Überläufer

Aufgabe

- Lesen Sie den Abschnitt 4.2 aus Ottmann & Widmayer
- Überlegen Sie sich eine Implementierung in C++:
  - TG1: Separate Verkettung der Überläufer
  - TG2: Direkte Verkettung der Überläufer
- Sie können sich auf das auf Seite 198 angegebene Beispiel beschränken:
  - Größe der Hashtabelle $m = 7$
  - $\mathcal{K} = 0, 1, ..., 500$
  - $h(k) = k \bmod m$

Offene Hashverfahren

- vorher: Verkettung der Überläufer *außerhalb* der Hashtabelle
  - heißen geschlossene Hashverfahren
- bei offenen Hashverfahren versucht man diese *in* der Hashtabelle unterzubringen
- wenn also Position $h(k)$ in der Hashtabelle t beim Einfügen von k bereits belegt ist, muss eine andere *offene* Stelle nach festen Regeln gesucht werden
  - Folge der zu betrachtenden Speicherplätze für einen Schlüssel heißt **Sondierungsfolge** zu diesem Schlüssel
- problematisch ist das Entfernen von Schlüsseln
  - Warum? Lösung?

Entfernen von Schlüsseln bei offenen Hashverfahren

- wenn ein Schlüssel entfernt wird, kann der in der Sondierungsfolge *dahinter* liegende nicht mehr gefunden werden
- Lösung: Schlüssel werden nicht gelöscht, sondern nur als gelöscht makiert
- nicht sehr effizient
- offene Hashverfahren wenn möglichst wenig entfernt wird

Schema für offene Hashverfahren

- für die meisten geeignet
- sei $s(j, k)$ eine Funktion, so dass
  $(h(k) - s(j, k)) \bmod m$ für $j = 0, 1, ..., m - 1$
  eine Sondierungsfolge bildet, d.h. Permutation aller Hashadressen
  Es sei stets noch mindestens ein Platz in der Hashtabelle frei

Suchen nach Schlüssel k

- beginne mit Hashadresse $i = h(k)$
- solange k nicht in $t[i]$ gespeichert ist und $t[i]$ nicht frei ist,
- suche weiter bei $i = (h(k) - s(j, k)) \bmod m$, für aufsteigende Werte von j
- falls $t[i]$ belegt ist, wurde k gefunden

Einfügen eines Schlüssels k

- wir nehmen an, dass k nicht in t vorkommt
- beginne mit Hashadresse $i = h(k)$
- solange $t[i]$ belegt ist
- mache weiter bei $i = (h(k) - s(j, k)) \bmod m$, für aufsteigende Werte von j
- trage k bei $t[i]$ ein

Entfernen eines Schlüssels k

- suche k
- war Suche erfolgreich, ist i die Adresse an der k gefunden wurde
- markiere $t[i]$ als entfernt

Lineare Sondieren

- Sondierungsfolge
  $h(k), h(k)-1, h(k)-2, ..., 0, m-1, ..., h(k)+1$
  also Sondierungsfunktion $s(j,k) = j$
- Beispiel, Buch Seite 206

Quadratisches Sondieren

- um die primäre Häufung des linearen Sondierens zu vermeiden
- es wird mit quadratisch wachsendem Abstand sondiert
- Sondierungsfolge
  $h(k), h(k)+1, h(k)-1, h(k)+4, h(k)-4, \ldots$
  also Sondierungsfunktion $s(j,k) = (\lceil j/2 \rceil)^2(-1)^j$
- Beispiel, Buch Seite 207

Uniformes und zufälliges Sondieren

- auch bei quadratischem Sondieren Häufung, da Sondierungsfolge für alle Synonyme gleich
- beim uniformen Sondieren ist die Folge $s(j, k)$ eine Permutation der Hashadressen die nur von k abhängt
  - alle der $m!$ Permutation sollen mit gleicher Wahrscheinlichkeit verwendet werden
  - es wird vermutet, dass die Anzahl der Kollisionen minimiert wird
  - sehr aufwändig praktisch umzusetzen
- beim zufälligen Sondieren wird in Abhängigkeit von k eine zufällige Hashadresse für $s(j, k)$ gewählt

Double Hashing

- Effizienz des uniformen Hashings wird bereits annähernd erreicht, wenn für Sondierungsfolge eine zweite Hashfunktion verwendet wird
- Sondierungsfolge für k:
  $h(k), h(k) - h'(k), h(k) - 2 \cdot h'(k), ..., h(k) - (m-1)h'(k)$
  also Sondierungsfunktion
  $s(j, k) = j \cdot h'(k)$
- $h'(k)$ muss so gewählt werden, dass für alle Schlüssel k die o.a. Sondierungsfolge eine Permutation der Hashadressen bilden
- $h'(k)$ unabhängig von $h(k)$ wählen, da sonst Synonyme gleiche Sondierungsreihenfolge haben
- Beispiel & Brents Algorithmus & Binärbaumsondieren Buch S. 211-215

Weitere Verfahren

- Ordered Hashing
  - geordnet bzgl. Sondierungsfolge
- Robin-Hood-Hashing
  - Schlüssel werden umgeordnet um die Länge der längsten Sondierungsfolge zu verringern
- ...

Dynamische Hashverfahren

- Lesen Sie den Abschnitt 4.4 bis 4.4.1 ausschließlich

Dynamische Hashverfahren (2)

- Verfahren für stark wachsende oder schrumpfende Datenbestände
- Blöcke fester Größe die mehrere Datensätze speichern können
- statt Hashtabelle Datei aus Blöcken
- Hashfunktion berechnet den Block
- keine Probleme solange es nicht mehr Synonyme gibt als in einen Block passen
- eine Datei mit m Blöcken wächst durch Anhängen eines Blockes und schrumpft durch Abhängen eines Blockes
- besondere Problematik:
  - beim Verändern von m muss die Hashfunktion so geändert werden, das trotzdem alle Schlüssel noch gefunden werden

Lineares Hashing

- Hashfunktion h besteht höchstens aus zwei einfachen Hashfunktionen h_1 und h_2, die jeweils die gesamte Hashdatei adressieren
- für eine anfängliche Dateigröße von m_0 Blöcken und einer aktuellen Größe von m Blöcken gilt:
  - $m_0 \cdot 2^l \leq m < m_0 \cdot 2^{l+1}$ für eine natürliche Zahl l
  - h_1 adressiert den Adressbereich $0 \ldots m_0 \cdot 2^l - 1$ und h_2 den Bereich $0 \ldots m_0 \cdot 2^{l+1} - 1$
- je nach Schlüssel k gilt dann
  - $h_2(k) = h_1(k)$ oder
  - $h_2(k) = h_1(k) + m_0 \cdot 2^l$
- der Dateilevel l gibt dabei die Anzahl der kompletten Dateiverdoppelungen an

Lineares Hashing (2)

- $h_2(k)$ verteilt also die Datensätze des Blocks $h_1(k)$
- nach Einfügen eines leeren Blocks mit der Adresse m ergibt sich die Adresse $i = m - m_0 \cdot 2^l$ als Adresse des Blocks mit den zu verteilenden Datensätzen
- i durchläuft also der Reihe nach die Adressen 0 bis $m_0 \cdot 2^l - 1$
- h_1 ist dann für die Schlüssel anzuwenden mit $i \leq h_1(k) \leq m_o \cdot 2^l - 1$
- für Schlüssel mit $0 \leq h_1(k) < i$, ist h_2 anzuwenden
- geeignete Wahl für h_1 und h_2 beispielsweise
  - $h_1(k) = k \bmod (m_0 \cdot 2^l)$
  - $h_2(k) = k \bmod (m_0 \cdot 2^{l+1})$

Lineares Hashing (3)

- nächste zu splittende Seite ist unabhängig von einzufügendem Datensatz
- daher werden zusätzlich Ketten von Blöcken (in eigenem Speicherbereich) für Überläufer gebildet
- Erweitern der Datei üblicherweise wenn Belegungsfaktor $n/(b \cdot m)$ als Folge einer Einfügeoperation überschritten wird
- wird ein anderer Schwellenwert beim Entfernen unterschritten, wird verkleinert

Virtuelles Hashing

- im Gegensatz zu linearem Hashing werden Überlaufblöcke vollständig vermieden
- sobald Einfügeoperation in einem bereits vollen Block durchgeführt werden soll:
  - Verdoppeln der Größe der Hashdatei
- es sollen aber nur die Datensätze des einen Blockes verteilt werden
- zusätzlich Bittabelle um zu wissen welche Hashfunktion genutzt werden soll