Karlsruher Institut für Technologie

Tutorium 3

Algorithmen I SS 14

Institut für Theoretische Informatik

Übungsblatt 2 und 3

→ bei Fragen bitte am Ende nach vorne kommen

Hashing

- direkte Adressierung (Array) teilweise unpraktikabel
  - bei großen Mengen M $\Rightarrow$ Array der Größe $|M|$
  - wenn tatsächlich gespeicherte Schlüsselmengen $|K| << |M| \Rightarrow$ Hashtabellen
- $\Theta(|K|)$ statt $\Theta(|M|)$
- Laufzeit im Average-Case nur $\mathcal{O}(1)$
  - direkte Adressierung $\mathcal{O}(1)$ auch im Worst-Case
- Slot eines Elements k wird durch die Hashfunktion h(k) bestimmt $\Rightarrow$ „gute“ Hashfunktion entscheidend für die Laufzeit

Laufzeiten

- Worstcase:
  - alle Schlüssel in einem Slot
  - $\Theta(n)$
- Best-Case
  - jeweils nur einen Schlüssel pro Slot
  - $\mathcal{O}(1)$
- Average-Case, mit
  - gleichmäßigem Hashing
  - doppelt verkette Listen zur Kollisionsauflösung
  - Anzahl der Slots proportional zur Anzahl der einzufügenden Elemente
  - $\mathcal{O}(1)$

Kollisionsauflösung

- verkettete Listen:
  - hinter jedem Tabellenslot steht eine verkettete Liste
  - die Liste muss nach dem gesuchten Element abgesucht werden
- lineare Suche
  - $h(k,i) = (h'(k) + i) mod\ m$
  - die Liste muss nach dem gesuchten Element abgesucht werden

Hashing Aufgabe 1

Gegeben sei die Hashfunktion:

$$H(k) = (k + 3) mod\ 11$$

Mit einer zugehörigen 11 Einträge großen Hashtabelle

Einfügen der Elemente < 70, 71, 66, 97, 82, 15, 93, 40, 19, 76 >

1. Für Hashing mit verketteten Listen
2. Für Hashing mit Linearer Suche

Hashing Aufgabe 1 Lösung

Mit Verketteten Listen:

0	1	2	3	4	5	6	7	8	9	10
19	97	76	66	nil	nil	nil	15	93	nil	40
nil	nil	nil	nil				70	82		nil
							nil	71		
								nil		

Mit Linearer Suche:

0	1	2	3	4	5	6	7	8	9	10
93	97	40	66	19	76		70	71	82	15

Hashing Aufgabe 1 b)

1. Bestimme die Anzahl von Hashtabellenzugriffen für beide Hashing Varianten
2. Bestimme die zu erwartenden Kosten für eine erfolgreiche Suche wenn die Wahrscheinlichkeiten der Suche eines Datums gleich verteilt sind
3. Bestimme den Speicherverbrauch für die beiden Hashing Varianten
   - ein Maschinenwort pro Zeiger, Element und Nullzeiger (Ende einer Liste)

Hashing Aufgabe 1 b) Lösung

Hashtabellenzugriffe

- mit Listen: $|\text{Einfügeoperationen}| = 10$
- mit linearer Suche:
  $|\text{Einfügeoperationen}| + |\text{Hashings auf bereits belegte Werte}| = 10 + 17 = 27$

Erwartete Kosten

- mit Listen: $\frac{\text{Traversierung aller Listen in der Tabelle}}{|\text{eingefügte Elemente}|} = \frac{14}{10}$
- mit linearer Suche: $\frac{\text{Hashtabellenzugriffe}}{|\text{eingefügte Elemente}|} = \frac{27}{10}$

Hashing Aufgabe 1 b) Lösung

Speicherbedarf

- mit Listen: $|\text{Zeiger}| + |\text{Elemente}| + |\text{Nullzeiger}| = 10 + 10 + 11 = 31$
- mit linearer Suche: $m = 11$

Hashing Aufgabe 2

Gegeben sei ein Array A = A[1], . . . , A[n] mit n Zahlen in beliebiger Reihenfolge. Für eine gegebenen Zahl x soll ein Paar (A[i], A[j]), $1 \leq$ i, j $\leq$ n gefunden werden, für das gilt: A[i] + A[j] = x.

1. Geben Sie eine Lösung für x = 33 und A = (7, 15, 21, 14, 18, 3, 9) an.
2. Geben Sie einen effizienten Algorithmus an, der das Problem in erwarteter Zeit $\mathcal{O}(n)$ löst, und bei Erfolg ein Paar (A[i], A[j]) ausgibt, ansonsten Nil.

Hashing Aufgabe 2 Lösung

- **Lösung mit Hashing:**
  1. alle Elemente in die Hash-Tabelle einfügen
  2. zu jeder Zahl das „Gegenstück“ suchen („Gegenstück“ von A[i] ist x - A[i])
- **Wichtig:**
  Die Hashtabelle muss in $\Omega(n)$ Slots haben, damit der Satz aus der Vorlesung gilt. Aus diesem folgt das die erwartete Laufzeit von Einfügen und Tabellenlookup in $\mathcal{O}(n)$ ist.

Kreativaufgabe

a) Entwerfen Sie eine Realisierung eines *SparseArray* (auf deutsch soviel wie „spärlich besetztes Array“). Dabei handelt es sich um eine Datenstruktur mit den Eigenschaften eines beschränkten Arrays, die zusätzlich schnelle Erzeugung und schnellen Reset ermöglicht. Nehmen Sie dabei an, dass **allocate** beliebig viel *uninitialisierten* Speicher in konstanter Zeit liefert. Im Detail habe das *SparseArray* folgende Eigenschaften:

- Ein *SparseArray* mit n Slots braucht $\mathcal{O}(n)$ Speicher.
- Erzeugen eines leeren *SparseArray* mit *n* Slots braucht $\mathcal{O}(1)$ Zeit.
- Das *SparseArray* unterstützt eine Operation *reset*, die es in $\mathcal{O}(1)$ Zeit in leeren Zustand versetzt.
- Das *SparseArray* unterstützt die Operation *get(i)* und *set(i, x)*. Dabei liefert *A.get(i)* den Wert, der sich im *i*-ten Slot des *SparseArray A* befindet; *A.set(i, x)* setzt das Element im i-ten Slot auf den Wert x. Wurde der i-te Slot seit der Erzeugung bzw. dem letzten reset noch nicht mit auf einen bestimmten Wert gesetzt, so liefert A.get(i) einen speziellen Wert $\perp$. Die Operationen get und set dürfen zudem beide nicht mehr als $\mathcal{O}(1)$ Zeit verbrauchen (man nennt so etwas *wahlfreien Zugriff*, engl. *random access*)

Kreativaufgabe Lösung a)

- Benutzung von zwei uninitialisierten Arrays D und H mit jeweils n Slots
- in D werden die Nutzdaten gespeichert und zusätzlich in jedem Slot eine nicht negative Zahl
- in H werden Indizes abgespeichert
- zusätzlich Zähler c mit 0 initialisiert
- **set(i,x)**: D[i].daten := x ; D[i].zahl := c; H[c] := i; c++
- **get(i)**: Überprüft ob der i-te Slot gesetzt wurde:
  - D[i].zahl < c und H[D[i].zahl] = i $\Rightarrow$ D[i] wurde gesetzt
- **reset**: Es wird c:= 0 gesetzt und somit werden alle Einträge invalidiert

Kreativaufgabe

b) Nehmen Sie an, dass die Datenelemente, die Sie im *SparseArray* ablegen wollen, recht groß sind (also z.B. nicht nur einzelne Zahlen, sondern Records mit 10, 20 oder mehr Einträgen). Geht Ihre Realisierung unter dieser Annahme sparsam oder verschwenderisch mit dem Speicherplatz um? Wenn Sie Ihre Realisierung für verschwenderisch halten, überlegen Sie ob und wie Sie es besser machen können.

Kreativaufgabe Lösung a)

- Lösung aus a) ist verschwenderisch
  - n Slots für Nutzdaten allokiert: zwar uninitialisiert aber reserviert
- Stattdessen:
  - Nutzdaten mit ins Array H speichern
  - für H ein *unbounded Array* verwenden, Anfangs 1 Slot
- Sei m die Anzahl der tatsächlich gesetzten Elemente. Dann belegt D den Platz für n nichtnegative ganze Zahlen. Das unbeschränkter Array H belegt maximal den Platz für 3m Nutzdatenelemente und Indizes (während des Kopiervorganges, wenn das unbeschränkte Array kopiert reallokiert wird)

Kreativaufgabe

c) Vergleichen Sie Ihr *SparseArray* mit bounded Arrays. Welche Vorteile und Nachteile sehen Sie im Hinblick auf den Speicherverbrauch und auf das Iterieren über alle Elemente mit set eingefügten Elemente?

Kreativaufgabe Lösung b)

- Iterieren dauert nur O(m) statt O(n) Zeit. Der Speicherverbrauch ist schlechter, wenn die Nutzdatenelemente klein sind. Für große Nutzdatenelemente kann der Speicherverbrauch sogar erheblich besser sein, sofern man eine sparsame Realisierung wie in b) verwendet und nicht zuviele Elemente eingefügt werden.