Karlsruhe Institute of Technology

Tutorium 4: Hashing

Matthias Schimek | 3. Juni 2017

TUTORIUM ZUR VORLESUNG ALGORITHMEN I IM SS17

Gliederung

Letztes Blatt

Vollständige Induktion:

- Im Ind. Schritt muss die komplette IV für $n + 1$ gezeigt werden

- Manchmal ist folgendes nützlich:
  - Um Aussage A zu beweisen, per Induktion zeigen, dass $B(n)$ für alle $n \in \mathbb{N}$ gilt
  - Aus der (damit bewiesenen) Aussage "$\forall n \in \mathbb{N}\ B(n)$" die Aussage A folgern

Letztes Blatt

Vollständige Induktion:

- Im Ind. Schritt muss die komplette IV für $n + 1$ gezeigt werden

- Manchmal ist folgendes nützlich:
  - Um Aussage A zu beweisen, per Induktion zeigen, dass $B(n)$ für alle $n \in \mathbb{N}$ gilt
  - Aus der (damit bewiesenen) Aussage "$\forall n \in \mathbb{N}\ B(n)$" die Aussage A folgern

Hashing - Grundlagen

Wörterbuchoperationen

- Wollen folgende Operation auf Menge M in (erwartet) $\mathcal{O}(1)$:
  - $M.insert(e)$: $M := M \cup \{e\}$
  - $M.remove(k : key)$: $M := M \setminus \{e\},\ key(e) = k$
  - $M.find(k : key)$: return $e \in M$ with $key(e) = k$; $\perp$ falls $e \notin M$

Hashing - Grundlagen

Wörterbuchoperationen

- Wie können wir Wörterbuchoperationen effizient umsetzen?
- Liste $\rightarrow$ zu langsam
- Array $\rightarrow$ wo speichern? - hier kommt Hashfunktion ins Spiel

Element - Key

- Elemente: beliebig, z.B. Autos, Bücher, Studenten, ...
- Key: repräsentiert Element eindeutig, hiermit wird ”gerechnet”
- Funktion: $key : M \rightarrow KEYS$
  - Es gilt: Wenn $key(e_1) = key(e_2) \Rightarrow e_1 = e_2$
  - z.B. Nummernschild, ISBN-Nr., Matrikelnummer, ...

Hashing - Grundlagen

Wörterbuchoperationen

- Wie können wir Wörterbuchoperationen effizient umsetzen?
- Liste $\rightarrow$ zu langsam
- Array $\rightarrow$ wo speichern? - hier kommt Hashfunktion ins Spiel

Element - Key

- Elemente: beliebig, z.B. Autos, Bücher, Studenten, ...
- Key: repräsentiert Element eindeutig, hiermit wird ”gerechnet”
- Funktion: $key : M \rightarrow KEYS$
  - Es gilt: Wenn $key(e_1) = key(e_2) \Rightarrow e_1 = e_2$
  - z.B. Nummernschild, ISBN-Nr., Matrikelnummer, ...

Hashing - Grundlagen

Hashfunktion:

- Haben Array mit m Plätzen
- h : KEYS $\rightarrow \{1, ..., m\}$
- D.h. für Element e:
  $h(key(e))$ ist Index, an dem e gespeichert wird
- h sollte einfach und effizient zu berechnen sein
- Speicherung von h sollte wenig Platz verbrauchen

Hashing - Perfekte Hashfunktion

- Perfekte Hashfunktionen schwer zu finden
- Manchmal unmöglich
- Lösung?

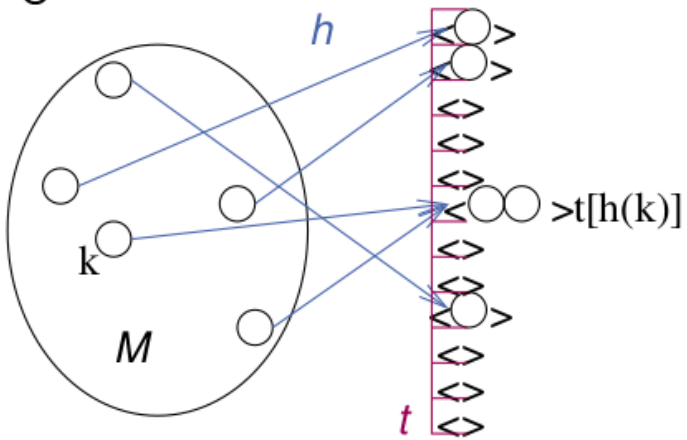
Hashing - verkettete Listen

Implementiere die Folgen in den Tabelleneinträgen
durch einfach verkettete Listen

insert(e): Füge e am Anfang von $t[h(\text{key}(e))]$ ein.

remove(k): Durchlaufe $t[h(k)]$.
Element e mit $\text{key}(e) = k$ gefunden?
$\rightsquigarrow$ löschen und zurückliefern.

find(k) : Durchlaufe $t[h(k)]$.
Element e mit $\text{key}(e) = k$ gefunden?
$\rightsquigarrow$ zurückliefern.
Sonst: $\perp$ zurückgeben.



1

[2] Folien 'Algorithmen I', KIT

Analyse für zufällige Hashfunktionen

Satz Für eine zufällige Hashfunktion $h : \text{KEYS} \to \{1, ..., m\}$ gilt $\forall k$: die erwartete Anzahl kollidierender Elemente ist in $\mathcal{O}(1)$ falls $|\text{KEYS}| \in \mathcal{O}(m)$ (m Größe der Hashtabelle)

Damit:

- Wörterbuchoperationen in *erwartet* konstanter Zeit

Analyse für zufällige Hashfunktionen

Satz Für eine zufällige Hashfunktion $h : \text{KEYS} \to \{1, ..., m\}$ gilt $\forall k$: die erwartete Anzahl kollidierender Elemente ist in $\mathcal{O}(1)$ falls $|\text{KEYS}| \in \mathcal{O}(m)$ (m Größe der Hashtabelle)

Damit:

- Wörterbuchoperationen in *erwartet* konstanter Zeit

Hashing - Beispiele

Hier: Element $e = key(e)$
$M = \{36, 78, 50, 1, 92, 15, 43, 99, 64\}$

Hashfunktionen:

- $a \rightarrow 0$
- $a \rightarrow a \bmod 5$
- $a \rightarrow a \bmod 7$
- $a \rightarrow a \bmod 9$

Worst-Case-Laufzeit

- Jedes Element bekommt Schlüssel aus Universum U zugewiesen ($U = \text{KEYS}$)
- n Elemente werden in Hashtabelle der Größe m eingefügt
- zufällige Funktion h aus $\{1, ..., m\}^U$ als Hashfunktion

Zeige, dass für $|U| > nm$ eine Teilmenge von U der Größe n existiert, sodass alle Schlüssel zum gleichen Slot gehasht werden. D.h. insbesondere ist die Worst-Case-Zeit für eine Suchoperation in $\Theta(n)$.

Wichtig: Für $n \in \mathcal{O}(m)$ Suchoperation trotzdem in *erwartet* $\mathcal{O}(1)$

Worst-Case-Laufzeit

- Jedes Element bekommt Schlüssel aus Universum U zugewiesen ($U = \text{KEYS}$)
- n Elemente werden in Hashtabelle der Größe m eingefügt
- zufällige Funktion h aus $\{1, ..., m\}^U$ als Hashfunktion

Zeige, dass für $|U| > nm$ eine Teilmenge von U der Größe n existiert, sodass alle Schlüssel zum gleichen Slot gehasht werden. D.h. insbesondere ist die Worst-Case-Zeit für eine Suchoperation in $\Theta(n)$.

Wichtig: Für $n \in \mathcal{O}(m)$ Suchoperation trotzdem in *erwartet* $\mathcal{O}(1)$

Hashing mit verketteten Listen

Behauptung: man kann Hashing mit verketteten Listen entscheidend verbessern, indem man die verketteten Listen stets sortiert hält.

- Ist diese Behauptung richtig? Wie ändert sich das worst-case Laufzeitverhalten von *insert*, *remove* und *find* in diesem Fall.
- Verwendet statt sortierter verketteter Listen nun sortierte unbeschränkte Arrays. Welche worst-case Laufzeiten können *insert*, *remove* und *find* nun erreichen?
- Betrachtet nochmals die Hashtabelle aus der 2. Teilaufgabe. Sind die *amortisierten* Laufzeiten von *insert*, *remove* und *find* besser als die worst-case Laufzeiten?

Hashing mit verketteten Listen

Behauptung: man kann Hashing mit verketteten Listen entscheidend verbessern, indem man die verketteten Listen stets sortiert hält.

- Ist diese Behauptung richtig? Wie ändert sich das worst-case Laufzeitverhalten von *insert*, *remove* und *find* in diesem Fall.
- Verwendet statt sortierter verketteter Listen nun sortierte unbeschränkte Arrays. Welche worst-case Laufzeiten können *insert*, *remove* und *find* nun erreichen?
- Betrachtet nochmals die Hashtabelle aus der 2. Teilaufgabe. Sind die *amortisierten* Laufzeiten von *insert*, *remove* und *find* besser als die worst-case Laufzeiten?

Hashing mit verketteten Listen

Behauptung: man kann Hashing mit verketteten Listen entscheidend verbessern, indem man die verketteten Listen stets sortiert hält.

- Ist diese Behauptung richtig? Wie ändert sich das worst-case Laufzeitverhalten von *insert*, *remove* und *find* in diesem Fall.
- Verwendet statt sortierter verketteter Listen nun sortierte unbeschränkte Arrays. Welche worst-case Laufzeiten können *insert*, *remove* und *find* nun erreichen?
- Betrachtet nochmals die Hashtabelle aus der 2. Teilaufgabe. Sind die *amortisierten* Laufzeiten von *insert*, *remove* und *find* besser als die worst-case Laufzeiten?

Hashing mit verketteten Listen

Behauptung: man kann Hashing mit verketteten Listen entscheidend verbessern, indem man die verketteten Listen stets sortiert hält.

- Ist diese Behauptung richtig? Wie ändert sich das worst-case Laufzeitverhalten von *insert*, *remove* und *find* in diesem Fall.
- Verwendet statt sortierter verketteter Listen nun sortierte unbeschränkte Arrays. Welche worst-case Laufzeiten können *insert*, *remove* und *find* nun erreichen?
- Betrachtet nochmals die Hashtabelle aus der 2. Teilaufgabe. Sind die *amortisierten* Laufzeiten von *insert*, *remove* und *find* besser als die worst-case Laufzeiten?

Kreativaufgabe - SparseArray

- 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 $set(i, x)$.
  - $get(i)$: liefert Element an i-ter Stelle oder $\perp$ falls uninitialisiert
  - $set(i, x)$: setzt Feld mit Index i auf Wert x
  - beide Operationen benötigen in konstanter Zeit

allocate liefert beliebig viel *uninitialisierten* Speicher in $\mathcal{O}(1)$.

Aufabe Implementiere ein SparseArray.

Kreativaufgabe - SparseArray

- 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 $set(i, x)$.
  - $get(i)$: liefert Element an i-ter Stelle oder $\perp$ falls uninitialisiert
  - $set(i, x)$: setzt Feld mit Index i auf Wert x
  - beide Operationen benötigen in konstanter Zeit

allocate liefert beliebig viel *uninitialisierten* Speicher in $\mathcal{O}(1)$.

Aufabe Implementiere ein SparseArray.