Algorithmen 1 SS 2013 – Tutorium 7

4. Tutorium

Sarah Lutteropp

14. Mai 2013

Übersicht

2. Übungsblatt

Aufgabe 1

1.a): $1 = 2^0$ is eine Zweierpotenz.

1.b): c passend wählen!

Aufgabe 2

Irgendwie erkenntlich machen, dass das Mastertheorem angewendet wurde $\rightsquigarrow$ Fälle angeben!

Aufgabe 4

Bei Fallunterscheidungen alle Fälle abdecken!

2. Übungsblatt

Möchte jemand vorrechnen?

Hashtabelle

Eine Hashtabelle unterstützt folgende Operationen:

- *insert*(e : Element)
- *remove*(k : Key): Lösche Element e mit key(e)$= k$
- *find*(k : Key): Gebe das Element e mit key(e)$= k$ zurück, falls dieses in der Tabelle ist, ansonsten $\perp$

Dafür ist eine bijektive Schlüssel-Funktion key : Element $\rightarrow$ Key gegeben. D.h. verschiedene Elemente haben verschiedene Schlüssel.

Hashfunktion

Die Hashfunktion bildet die Menge der Schlüssel auf einen endlichen Indexbereich ab.

Definition Hashfunktion

$h\colon \text{ Key} \to \{0, \ldots, m-1\}, k \mapsto h(k)$

Beispiel

$\text{Key} = \mathbb{Z},\ m = 7,\ h\colon \mathbb{Z} \to \mathbb{Z}/7\mathbb{Z}, h(k) = k \mod 7$

$h(12) = ?$

$h(27) = ?$

$h(0) = ?$

Universelle Hashfunktion

- Eine zufällige Hashfunktion aus einer Familie universeller Hashfunktionen verursacht mit einer Warscheinlichkeit von $\frac{1}{m}$ eine Kollision.
- *insert*, *remove* und *find* in erwartet konstanter Zeit $\mathcal{O}(1)$

Exkurs: Gute Hashfunktionen

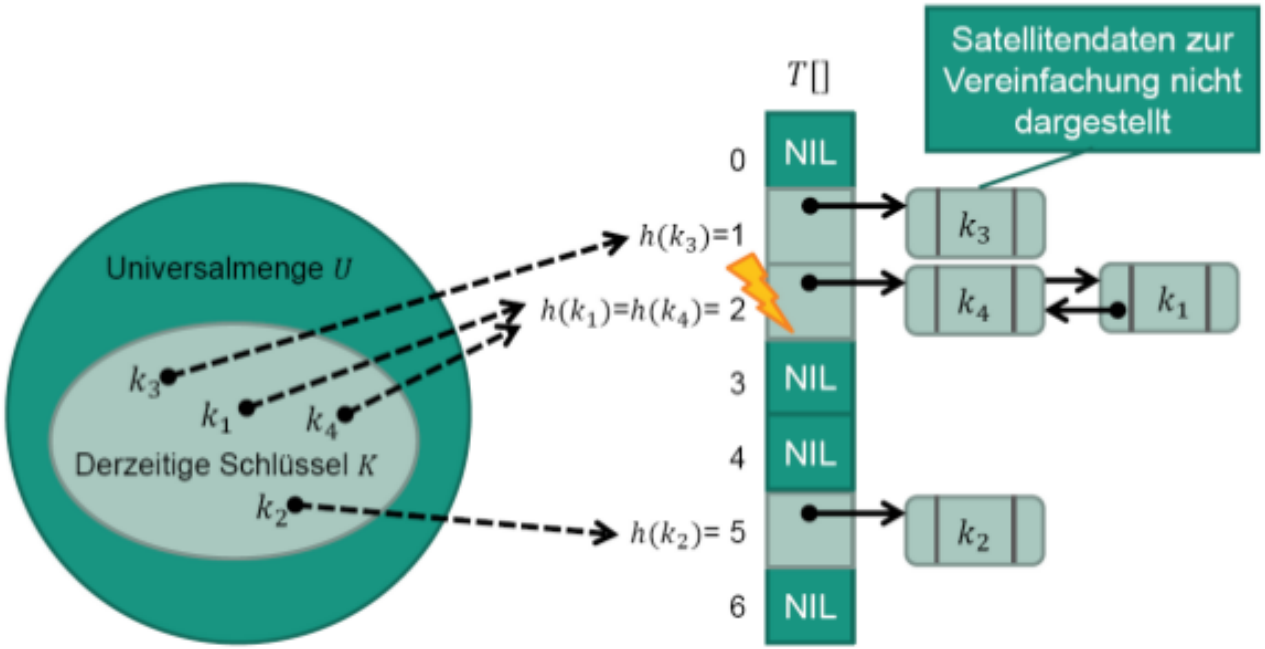
Divisionsmethode

- Hashfunktion: $h(k) = k \; mod \; m$
  - Wähle für m eine Primzahl
  - Wähle ein m welches fern von einer Zweierpotenz liegt
  - Wähle ausreichend großes m unter Berücksichtigung von α

Multiplikationsmethode

- Hashfunktion: $h(k) = \lfloor m(k * A \; mod \; 1) \rfloor$
  - Wahl von m unkritisch
  - $0 < A < 1$
  - Für bestimmte Werte von A besseres Verhalten

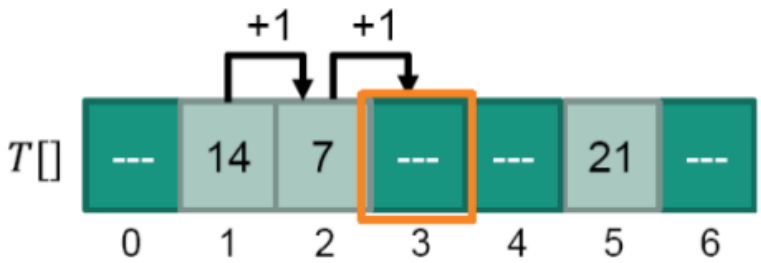
Geschlossenes Hashing



Offenes Hashing

Lineare Sondierung

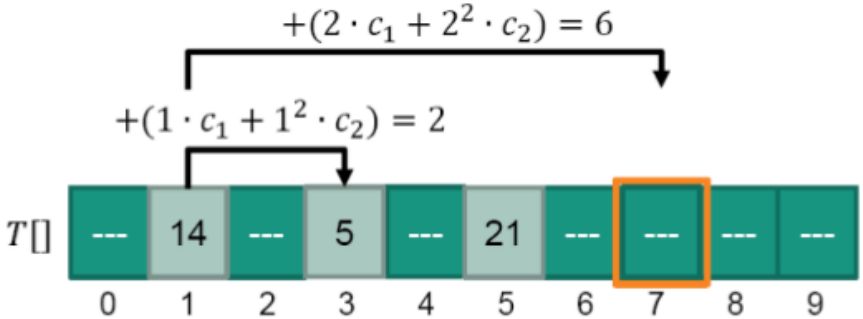
- Hashfunktion: $h(k,i) = (h'(k) + i)\ mod\ m$
  - $h'(k)$ ist eine beliebige Hilfshashfunktion die U auf $\{0,1,\ldots,m-1\}$ abbildet



Exkurs: Offenes Hashing

Quadratische Sondierung

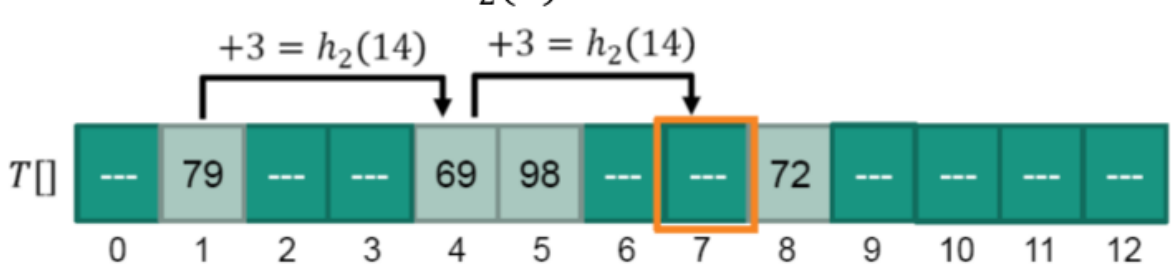
- Hashfunktion: $h(k,i) = (h'(k) + c_1 i + c_2 i^2)\ mod\ m$
  - $h'(k)$ ist eine beliebige Hilfshashfunktion die U auf $\{0,1,\ldots,m-1\}$ abbildet



Exkurs: Offenes Hashing

Doppeltes Hashing

- Hashfunktion: $h(k,i) = (h_1(k) + i * h_2(k)) \; mod \; m$
  - $h_1(k)$ und $h_2(k)$ sind beliebige Hilfshashfunktionen
  - Wert von $h_2(k)$ muss Teilerfremd zu m sein



- Hilfshashfunktion $h_1(k) = k \; mod \; 13$
- Hilfshashfunktion $h_2(k) = 1 + (k \; mod \; 12)$
- Einfügen von Schlüssel 14

Geschlossenes und offenes Hashing

Aufgabe

Key $= \mathbb{Z}, m = 11, h\colon \mathbb{Z} \to \mathbb{Z}/11\mathbb{Z}, h(k) = k \mod 11$
Führt folgendes mit geschlossenem Hashing und offenem Hashing (lineare Sondierung) aus:
insert 12,24,42,16,81,27
delete 24,16

Kreativaufgabe 

Aufgabe

Gegeben sei eine Menge M von Paaren von ganzen Zahlen im Bereich $1, \ldots, |M|$, M definiert eine binäre Relation R_M . Skizziert einen Algorithmus, der in erwarteter Zeit $\mathcal{O}(|M|)$ überprüft, ob R_M symmetrisch ist und begründet die erreichte Laufzeit.

Lösungsidee

Skizzierter Algorithmus

Speichere alle Paare $(a, b) \in M$ in der Hashtabelle H mit (a, b) als Schlüssel. Gehe alle Paare erneut durch und überprüfe für jedes (a, b), ob auch (b, a) in der Hashtabelle enthalten ist. Ist dieser Test für alle Elemente erfolgreich, dann ist R_M symmetrisch.

Kreativaufgabe (SparseArray)

Entwurf einer Datenstruktur

Entwerft 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. Nehmt dabei an, dass **allocate** beliebig viel uninitialisierten Speicher in konstanter Zeit liefert.

Kreativaufgabe (SparseArray)

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 (SparseArray)

Entwurf einer Datenstruktur

Nehmt nun an, dass die Datenelemente, die im SparseArray abgelegt werden sollen, recht groß sind (also z.B. nicht nur einzelne Zahlen, sondern Records mit 10,20 oder mehr Einträgen). Geht eure Realisierung unter dieser Annahme sparsam oder verschwenderisch mit dem Speicherplatz um? Wenn ihr eure Realisierung für verschwenderisch haltet, überlegt, ob und wie ihr es besser machen könnt.

Kreativaufgabe (SparseArray)

Entwurf einer Datenstruktur

Vergleicht nun euer SparseArray mit unbounded Arrays. Welche Vorteile und Nachteile seht ihr im Hinblick auf den Speicherverbrauch und auf das Iterieren über alle mit set eingefügten Elemente?

Bis zum nächsten Mal! ☺