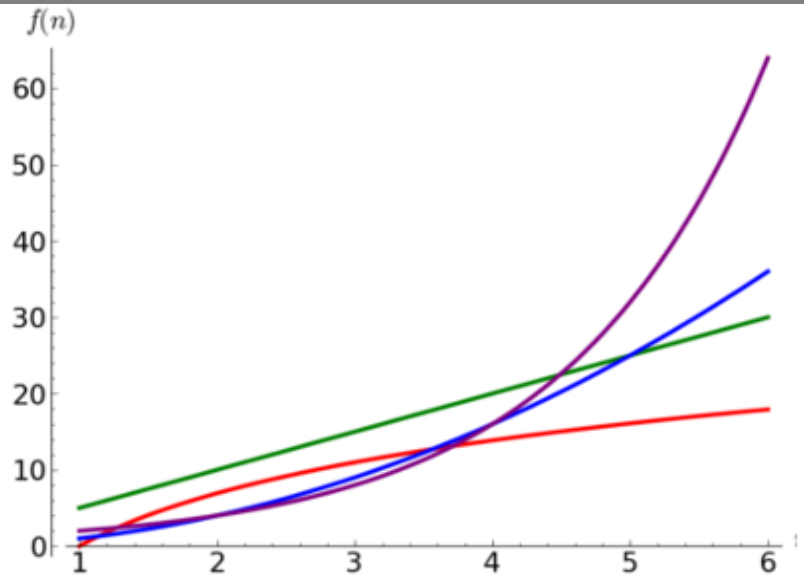
Karlsruher Institut für Technologie

Tutorium Algorithmen 1

Simon Bischof (simon.bischof2@student.kit.edu) | 14. Mai 2013

INSTITUT FÜR THEORETISCHE INFORMATIK, PROF. SANDERS




```
if (num1 < 10) or (num2 < 10)
    return num1*num2
/* calculates the size of the numbers */
m = max(size(num1), size(num2))
low1, low2 = lower half of num1, num2
high1, high2 = higher half of num1, num2
/* 3 calls made to numbers approximately half the size */
z0 = karatsuba(low1,low2)
z1 = karatsuba((low1+high1),(low2+high2))
z2 = karatsuba(high1,high2)
return (z2*10^(m))+( (z1-z2-z0) *10^(m/2))+(z0)
```

Zum Übungsblatt

- Wenn eine Datenstruktur verlangt ist, immer die komplette Datenstruktur beschreiben (inklusive Art der Verkettung, first-/last-Pointer, Wächterelement?, . . .)
- Spezialfälle (z.B. leere Liste) beachten

Hashing

Ziel des Hashings ist, für eine Menge M folgende Operationen erwartet in $O(1)$ zu unterstützen:

- insert(e): fügt ein Element ein
- lookup(k): suche ein Element mit Schlüssel k und gib es zurück (oder $\perp$, falls keines vorhanden)
- delete(k): lösche ein Element mit Schlüssel k

Perfekte Hashfunktionen

- Eine perfekte Hashfunktion h bildet Elemente von M injektiv auf eindeutige Einträge der Tabelle $t[0..m-1]$ ab
- Problem: Wie finde ich eine solche Funktion?
- Sonst: Kollisionsauflösung notwendig

Perfekte Hashfunktionen

- Eine perfekte Hashfunktion h bildet Elemente von M injektiv auf eindeutige Einträge der Tabelle $t[0..m-1]$ ab
- Problem: Wie finde ich eine solche Funktion?
- Sonst: Kollisionsauflösung notwendig

Perfekte Hashfunktionen

- Eine perfekte Hashfunktion h bildet Elemente von M injektiv auf eindeutige Einträge der Tabelle $t[0..m-1]$ ab
- Problem: Wie finde ich eine solche Funktion?
- Sonst: Kollisionsauflösung notwendig

Hashing mit verketteten Listen

Verwende eine Tabelle $t[0..m-1]$ mit einfach verketteten Listen als Einträgen. In der Liste von $t[i]$ stehen alle Elemente $e \in M$ mit $h(e) = i$.

- insert(e): hänge Element an den Anfang der entsprechenden Liste. Laufzeit:
- lookup(k): durchsuche die Liste bei $t[h(k)]$. Laufzeit:
- lookup(k): lösche das entsprechende Element aus der Liste bei $t[h(k)]$. Laufzeit:

Hashing mit verketteten Listen

Verwende eine Tabelle $t[0..m-1]$ mit einfach verketteten Listen als Einträgen. In der Liste von $t[i]$ stehen alle Elemente $e \in M$ mit $h(e) = i$.

- insert(e): hänge Element an den Anfang der entsprechenden Liste. Laufzeit:
- lookup(k): durchsuche die Liste bei $t[h(k)]$. Laufzeit:
- lookup(k): lösche das entsprechende Element aus der Liste bei $t[h(k)]$. Laufzeit:

Hashing mit verketteten Listen

Verwende eine Tabelle $t[0..m-1]$ mit einfach verketteten Listen als Einträgen. In der Liste von $t[i]$ stehen alle Elemente $e \in M$ mit $h(e) = i$.

- insert(e): hänge Element an den Anfang der entsprechenden Liste. Laufzeit: O(1)
- lookup(k): durchsuche die Liste bei $t[h(k)]$. Laufzeit: O(Listenlänge)
- lookup(k): lösche das entsprechende Element aus der Liste bei $t[h(k)]$. Laufzeit: O(Listenlänge)

zufällige Hashfunktionen

- Satz 1: Gegeben sei eine Tabelle $t[0..m-1]$ und es sei $|M| \in O(m)$. Dann gilt für jedes $i \in \{0, \ldots, m-1\}$: In der Liste $t[i]$ ist die Anzahl erwarteter Kollisionen in $O(1)$.
- Problem: wie findet und implementiert man zufällige Hashfunktionen?

zufällige Hashfunktionen

- Satz 1: Gegeben sei eine Tabelle $t[0..m-1]$ und es sei $|M| \in O(m)$. Dann gilt für jedes $i \in \{0, \ldots, m-1\}$: In der Liste $t[i]$ ist die Anzahl erwarteter Kollisionen in $O(1)$.
- Problem: wie findet und implementiert man zufällige Hashfunktionen?

Universelles Hashing

- $\mathcal{H} \subseteq \{0, \ldots, m-1\}^{Key}$ ist universell falls für alle $x, y \in Key$ mit $x \neq y$ und zufälligem $h \in \mathcal{H}$ gilt: $\mathbb{P}[h(x) = h(y)] = \frac{1}{m}$.
- Satz 1 gilt auch für universelle Familien von Hashfunktionen.
- Es sei m eine Primzahl, $Key \subseteq \{0, \ldots, m-1\}^k$ und für $a \in \{0, \ldots, m-1\}^k$ sei $h_a(x) := a \cdot x \mod m$. Dann ist $H := \{h_a \mid a \in \{0, \ldots, m-1\}^k\}$ eine universelle Familie von Hashfunktionen.

Universelles Hashing

- $\mathcal{H} \subseteq \{0, \ldots, m-1\}^{Key}$ ist universell falls für alle $x, y \in Key$ mit $x \neq y$ und zufälligem $h \in \mathcal{H}$ gilt: $\mathbb{P}[h(x) = h(y)] = \frac{1}{m}$.
- Satz 1 gilt auch für universelle Familien von Hashfunktionen.
- Es sei m eine Primzahl, $Key \subseteq \{0, \ldots, m-1\}^k$ und für $a \in \{0, \ldots, m-1\}^k$ sei $h_a(x) := a \cdot x \mod m$. Dann ist $H := \{h_a \mid a \in \{0, \ldots, m-1\}^k\}$ eine universelle Familie von Hashfunktionen.

Universelles Hashing

- $\mathcal{H} \subseteq \{0, \ldots, m-1\}^{Key}$ ist universell falls für alle $x, y \in Key$ mit $x \neq y$ und zufälligem $h \in \mathcal{H}$ gilt: $\mathbb{P}[h(x) = h(y)] = \frac{1}{m}$.
- Satz 1 gilt auch für universelle Familien von Hashfunktionen.
- Es sei m eine Primzahl, $Key \subseteq \{0, \ldots, m-1\}^k$ und für $a \in \{0, \ldots, m-1\}^k$ sei $h_a(x) := a \cdot x \mod m$. Dann ist $H := \{h_a \mid a \in \{0, \ldots, m-1\}^k\}$ eine universelle Familie von Hashfunktionen.

Aufgabe zu Hashing

- Gegeben sei eine Menge M von Paaren von ganzen Zahlen im Bereich $1, \ldots, |M|$. M definiert eine binäre Relation R_M. Skizzieren Sie einen Algorithmus, der in erwarteter Zeit $O(|M|)$ überprüft, ob R_M symmetrisch ist.
- Begründen Sie die erreichte Laufzeit.

Hashing mit linear probing

- Array, in dem Einträge direkt drinstehen
- zusätzliche Slots (der letzte Slot ist immer $\perp$)
- insert: wenn Platz belegt, teste den nächsten Platz
- lookup: gehe Liste bis zum Ende durch
- delete: Elemente rutschen evtl. nach vorne

Hashing mit linear probing

- Array, in dem Einträge direkt drinstehen
- zusätzliche Slots (der letzte Slot ist immer $\perp$)
- insert: wenn Platz belegt, teste den nächsten Platz
- lookup: gehe Liste bis zum Ende durch
- delete: Elemente rutschen evtl. nach vorne

Kreativaufgabe

Entwerfen Sie eine Realisierung eines SparseArray. Dabei handelt es sich um eine Datenstruktur mit den Eigenschaften eines beschränkten Arrays mit schneller Erzeugung und schnellem Reset. Nehmen Sie dabei an, dass allocate beliebig viel uninitialisierten Speicher in konstanter Zeit liefert. Im Detail habe das SparseArray folgende Eigenschaften:

- Ein SparseArrays mit n Slots braucht O(n) Speicher.
- Erzeugen eines leeren SparseArrays mit n Slots braucht O(1) Zeit.
- Das SparseArray unterstützt eine Operation reset, die es in O(1) Zeit in leeren Zustand versetzt.
- Das SparseArray unterstützt die Operation get(i) und set(i, x) in O(1). 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 $\perp$.

Kreativaufgabe (Fortsetzung)

- 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.
- 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?

Feedback

1. Wie gefällt euch die Vorlesung?
2. Wie ist die Abstimmung zwischen Vorlesung und Übung?
3. Wie gefällt euch mein Tut?
4. Habt ihr Verbesserungsvorschläge?
5. Womit habt ihr noch Schwierigkeiten?
6. Welches Thema hat euch bisher am meisten gefallen?