Informatik II: Algorithmen und Datenstrukturen SS 2015

Vorlesung 7b, Mittwoch, 10. Juni 2015

(Cache- bzw. IO-Effizienz)

Prof. Dr. Hannah Bast
Lehrstuhl für Algorithmen und Datenstrukturen
Institut für Informatik
Universität Freiburg

UNI FREIBURG

Blick über die Vorlesung heute

UNI FREIBURG

- Inhalt
  - Laufzeit Listen vs. Felder … Vergleich + Analyse
  - Lokalität Speicherzugriffe … Definition + Hintergründe
  - Blockoperationen … ein alternatives Maß für Effizienz
  - Übungsblatt 7, Aufgabe 2: Anzahl Blockoperationen von Cuckoo Hashing analysieren

Laufzeit Listen vs. Felder 1/5

UNI FREIBURG

- Laufzeit doppelt verkettete Liste
  - Einfügen an beliebiger Stelle: $O(1)$
  - Entfernen an beliebiger Stelle: $O(1)$
  - Zugriff auf i-tes Element der Liste: $O(\min\{i, n - i\})$
- Laufzeit dynamisches Feld
  - Einfügen am Ende: amortisiert $O(1)$ … siehe Vorlesung 6b
  - Entfernen am Ende: amortisiert $O(1)$ … siehe Vorlesung 6b
  - Zugriff auf i-tes Element der Liste: $O(1)$
  - Einfügen an i-ter Stelle: $O(n - i)$
  - Entfernen an i-ter Stelle: $O(n - i)$

Laufzeit Listen vs. Felder 2/5

UNI FREIBURG

- Zeitmessung in der Praxis
  - Beim Einfügen / Entfernen am Ende scheinen Liste und dynamisches Feld gleich gut zu sein

    Die Liste sieht sogar besser aus, weil **immer** O(1) und das dynamische Feld nur **im Durchschnitt** O(1)
  - Die Laufzeit wollen wir jetzt mal konkret nachmessen

    Wir fügen dabei der Einfachheit halber nur am Ende ein
  - Beobachtung: LinkedList langsamer als DynamicArray

JAVA: teilweise sehr viel langsamer, teilweise auch fast gleich schnell

C++: Felder ca. 3× schneller

Laufzeit Listen vs. Felder 3/5

UNI FREIBURG

- Laufzeitunterschied, Grund 1
  - Bei der Liste müssen wir für jede Operation **vier** Zeiger umbiegen, beim dynamischen Feld müssen wir ohne Reallokation einfach nur **einen** Eintrag schreiben

    Und die Reallokationen fallen bei geeigneten Parametern (z.B. f = 0.5 im Sinne des Ü6) nicht ins Gewicht

Laufzeit Listen vs. Felder 4/5

UNI FREIBURG

- Laufzeitunterschied, Grund 2
  - Bei der Liste müssen wir für jedes Element **einzeln** Speicher allo**z**ieren, beim dynamischen Feld tun wir das für viele Elemente **auf einmal**

    Jede Speicherallokation hat fixe Kosten, die unabhängig von der Größe des allozierten Speichers sind

    Außerdem benötigen wir durch die Zeiger pro Element auch insgesamt etwas mehr Platz

Laufzeit Listen vs. Felder 5/5

UNI FREIBURG

- Laufzeitunterschied, Grund 3
  - Das dynamische Feld hat eine viel bessere sogenannte **Lokalität** der Speicherzugriffe

    Bei einem Feld stehen ja, wie gesagt, die n Elemente im Speicher hintereinander

    Bei einer verketteten Liste können die Elemente beliebig im Speicher verteilt stehen

    Warum das einen Unterschied macht, schauen wir uns im Rest der Vorlesung an

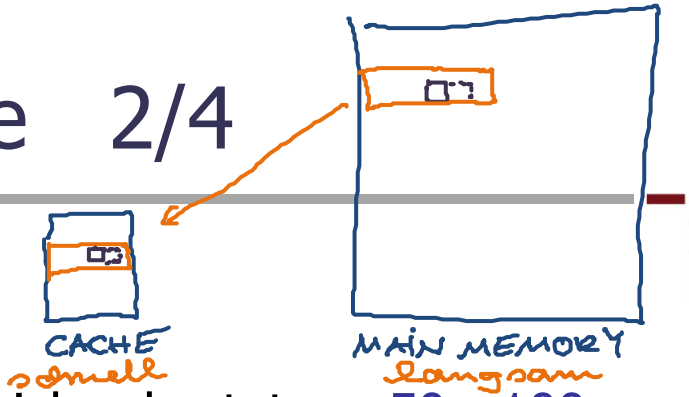
Lokalität Speicherzugriffe 1/4

order1 = [0, 1, 2, 3, 4]
order2 = [1, 4, 2, 0, 3]

- Einfach(st)es Beispiel
  - Wir addieren die n Elemente eines Feldes auf
    … in der natürlichen Reihenfolge: 1 + 2 + 3 + 4 + 5
    … in einer zufälligen Reihenfolge: 2 + 5 + 3 + 1 + 4
  - Das Ergebnis ist in beiden Fällen **identisch**
  - Die Anzahl der Operationen ist ebenfalls **identisch**
  - Beobachtung: die Laufzeit ist bei zufälliger Reihenfolge sehr deutlich länger

Bei n = 100M , 20 mal länger

Lokalität Speicherzugriffe 2/4



- CPU Cache, Prinzip
  - Zugriff auf ein Byte im Hauptspeicher kostet ca. 50 - 100ns
  - Zugriff auf ein Byte im Level-1 (L1) Cache kostet ca. 1ns
  - Bei Zugriff auf ein oder mehrere Bytes im Hauptspeicher holt man gleich einen ganzen Block (cache line) in den Cache

    Typische Größe einer cache line: **64 Bytes**
  - Solange dieser Block im Cache ist, braucht man für Bytes aus diesem Block nicht mehr auf den Hauptspeicher zuzugreifen
  - Der Cache hat Platz für viele solcher Blöcke

    Typische Größe eines L1-Cache: **64 KB** (=1000 cache lines)

Lokalität Speicherzugriffe 3/4

UNI FREIBURG

- Disk Cache, Prinzip
  - Zugriff auf ein Byte auf Platte kostet ~ 5ms

    Lesekopf muss an die Stelle bewegt werden (seek time)
  - Transferrate für folgende Bytes ist 50 MB / s und mehr

    Kopf bleibt stehen, Platte dreht sich schnell weiter
  - Deshalb geht das Betriebssystem wie folgt vor

    Wird ein Byte von der Platte gelesen, wird gleich ein ganzer Block eingelesen … z.B. 128 KB auf einmal

    Solange dieser Block im Speicher ist, braucht man für Bytes aus diesem Block nicht mehr auf die Platte zugreifen

    Also dasselbe Prinzip wie beim CPU Cache

Lokalität Speicherzugriffe 4/4

UNI FREIBURG

- Wenn der Cache voll ist
  - … muss einer der Blöcke entfernt werden, dafür gibt es zahlreiche Strategien, zum Beispiel:

    LRU (Least Recently Used) = der Block, für den es am längsten her ist, das darauf zugegriffen wurde

    LFU (Least Frequently Used) = der Block, auf den am wenigsten zugegriffen wurde, seit er im Cache ist
  - Das ist aber nicht das Thema der Vorlesung heute

    Für unsere einfachen Analysen heute und auf dem Übungsblatt spielt es keine Rolle, welche Strategie verwendet wird

Blockoperationen 1/11

UNI FREIBURG

- Abstraktion der bisherigen Beobachtungen
  - Wir haben einen langsamen und einen schnellen Speicher
  - Der langsame Speicher ist in Blöcke der Größe B unterteilt
  - Der schnelle Speicher ist M groß (Platz für M/B Blöcke)
  - Stehen die Daten nicht im schnellen Speicher, wird der entsprechende Block in den schnellen Speicher geladen
  - Das Programm kann sich aussuchen, welche Blöcke im schnellen Speicher gehalten werden
  - Wir zählen nur die **Anzahl der Blockoperationen**

Blockoperationen 2/11

UNI FREIBURG

- Was wir alles vernachlässigen
  - Sämtliche Berechnung auf einem Block im schnellen Speicher

    Grund: die Zeit, die man braucht, um einen Block in den schnellen Speicher zu holen, dominiert oft alles andere
  - Kosten für das Verwalten der Blöcke im schnellen Speicher

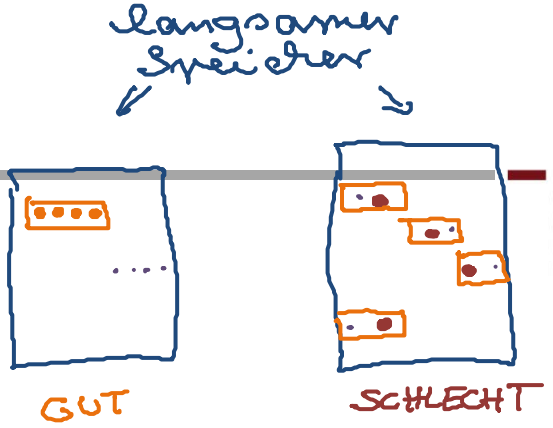
    Insbesondere die Implementierung der verwendeten Strategie, wie z.B. LRU oder LFU
  - Wie genau der langsame Speicher in Blöcke unterteilt ist

    Das macht höchstens einen Faktor von 2 Unterschied
  - Ob eine Operation 1, 2, 4 oder 8 Bytes liest

    Abhängig von der Maschine, aber nicht vom Algorithmus

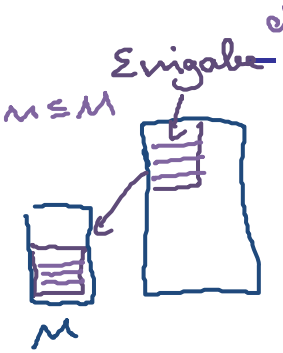
Blockoperationen 3/11



- Gute vs. Schlechte Lokalität
  - Für B Operationen hat man also:

    Im "best case" nur 1 Blockoperation **gute Lokalität**

    Im "worst case" B Blockoperationen **schlechte Lokalität**



Für einen Algorithmus, der n Eingabeelemente lesen muss, mit $n \leq M$, hat man dann trivial $\lceil n/B \rceil$ Blockoperationen

Dann Analyse der Anzahl Blockoperationen erst interessant, wenn die Eingabe nicht ganz in den schnellen Speicher passt

Für Operationen, die nur einen Teil der Eingabe bearbeiten müssen (z.B. lookup in einer HashMap), Analyse aber evtl. auch schon für kleine n interessant

Blockoperationen 4/11

UNI FREIBURG

- Typische Werte (für einen Server)
  - CPU Cache: B = 64 Bytes, M = 64 KB
  - Disk Cache: B = 64 Kilobytes, M = 64 GB

    Die meisten Betriebssysteme benutzen alles, was vom Hauptspeicher gerade nicht genutzt wird, als Disk Cache
  - Sinnvollerweise wählt man dabei B so, dass die transfer time für einen Block ungefähr gleich der seek time ist

    Wenn man schon die viele Zeit für ein "seek" aufwendet, kann man auch gerade noch mal so viel Zeit aufwenden, um möglichst viele Elemente auf einmal zu lesen

Blockoperationen 5/11

UNI FREIBURG

- Terminologie
  - Blockoperationen nennt man

    … beim CPU Cache in der Regel **cache misses**

    Weil sie dann nötig werden, wenn ein Stück vom (langsamen) Hauptspeicher nicht im (schnellen) Cache ist

    … beim Disk Cache oft **IOs**

    IO oder I/O = Input/Output … eher historische Bezeichnung für Datentransfer von der oder auf die Platte
  - Man nennt spricht deshalb im Zusammenhang mit der Analyse der Anzahl Blockoperationen auch von der **Cache-Effizienz** oder **IO-Effizienz** eines Algorithmus

Blockoperationen 6/11

Blockgröße B

- IO-Effizienz von ArraySumMain 1/2
  - Wenn wir über die n Elemente in der Reihenfolge 1, 2, 3, ... iterieren, Anzahl Blockoperationen immer = $\lceil n/B \rceil$
  - Wenn wir über die n Elemente in einer zufälligen Reihenfolge iterieren, Anzahl Blockoperationen im worst case = n

Blockoperationen 7/11

UNI FREIBURG

- IO-Effizienz von ArraySumMain 2/2
  - In der Praxis ist der Unterschied zwischen den beiden Varianten $< B = 64$

    Auch bei der zufälligen Reihenfolge wird pro Element auf 4 benachbarte Bytes (ein int) auf einmal zugegriffen

    Außerdem wird, wenn nicht $n \gg M$, das nächste Element manchmal zufällig schon im schnellen Speicher stehen

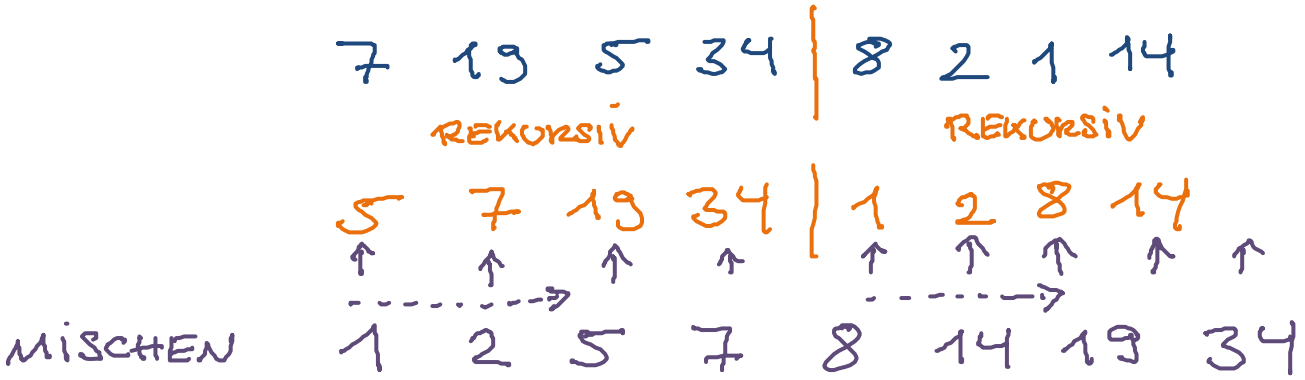
Blockoperationen 8/11

- IO-Effizienz von MergeSort 1/4
  - Kurze Wiederholung der Funktionsweise:

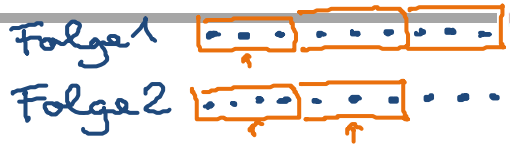
    Teile das Feld in zwei gleich große Teile

    Sortiere die beiden Teile rekursiv

    Mische die beiden sortierten Folgen zu einer sortierten Folge



Blockoperationen 9/11



- IO-Effizienz von MergeSort 2/4
  - Mischen von zwei Folgen der Gesamtlänge n geht mit $IO(n) \leq \lceil n/B \rceil$ Blockoperationen

    Man liest in jeder der beiden Folgen einen Block nach dem anderen, und zu jedem Zeitpunkt braucht man nur zwei Blöcke

    
  - Außerdem ist $IO(n) = 1$ für $1 \leq n \leq B$

    Wenn die Eingabe in einen Block passt, braucht man auch nur eine Blockoperation

Blockoperationen 10/11

UNI FREIBURG

- IO-Effizienz von MergeSort 3/4
  - Die benötigte Anzahl Blockoperationen für MergeSort ist damit $IO(n) = \Theta(n/B \cdot \log_2(n/B))$

$\lceil n/B \rceil \leq 2 \cdot n/B$

$$IO(n) \leq 2 \cdot IO(n/2) + 2 \cdot n/B$$
$$\leq 2 \cdot [2 \cdot IO(n/4) + 2 \cdot \tfrac{n}{2}/B] + 2 \cdot n/B$$
$$= 4 \cdot IO(n/4) + 2 \cdot 2n/B$$
$$\ldots \leq 2^k \cdot IO(n/2^k) + 2 \cdot k \cdot n/B$$

$k = \log_2 \frac{n}{B}$ $\longrightarrow$ $\leq n/B \cdot \underbrace{IO(B)}_{=1} + 2 \cdot \log_2 \frac{n}{B} \cdot n/B$

$2^k = n/B$

$n/2^k = B$

$$= O\left(\frac{n}{B} \cdot \log_2 \frac{n}{B}\right)$$ ∎

Blockoperationen 11/11

UNI FREIBURG

- IO-Effizienz von MergeSort 4/4
  - Man kann sogar zeigen: $IO(n) = \Theta(n/B \cdot \log_{M/B}(n/B))$
  - Dazu teilt man auf jeder Rekursionsstufe in $k = M/B$ Teile (statt nur in 2 Teile), sortiert die rekursiv und mischt die k sortierten Teilfolgen dann zu einer Folge

    Beim Mischen von k Folgen braucht man zu jedem Zeitpunkt für jede Folge genau einen Block, also $k \cdot B = M$ Blöcke insgesamt

    Gute Zusatzaufgabe zum Verständnis / Lernen

Literatur / Links

UNI FREIBURG

- Cache-Effizienz / IO-Effizienz
  - In Mehlhorn/Sanders:

    2 Introduction 2.2.1 External Memory
  - In Wikipedia

    http://en.wikipedia.org/wiki/Cache

    http://de.wikipedia.org/wiki/Cache

    Da wird das Prinzip eines Caches beschrieben, es gibt aber keinen separaten Artikel zur Cache-Effizienz bei Algorithmen