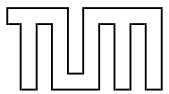
Grundlagen: Algorithmen und Datenstrukturen

Prof. Dr. Hanjo Täubig

Lehrstuhl für Effiziente Algorithmen
(Prof. Dr. Ernst W. Mayr)
Institut für Informatik
Technische Universität München

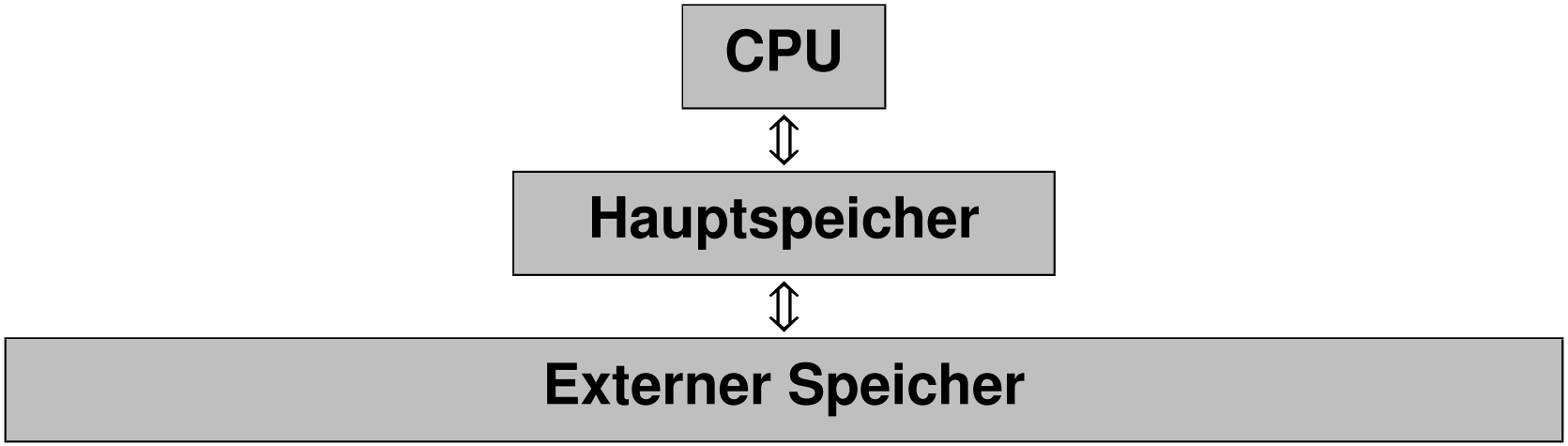
Sommersemester 2010



Übersicht

Externes Sortieren

Heutige Computer:



- Hauptspeicher hat Größe M
- Transfer zwischen Hauptspeicher und externem Speicher mit Blockgröße B

Externes Sortieren

Problem:
Minimiere Anzahl Blocktransfers zwischen internem und externem Speicher

Lösung: Verwende MergeSort

Vorteil: MergeSort verwendet oft konsekutive Elemente (Scanning)
(geht auf Festplatte schneller als Random Access-Zugriffe)

Externes Sortieren

- Eingabe: großes Feld auf der Festplatte
- Annahme: Anzahl der Elemente n ist durch B teilbar (sonst z.B. Auffüllen mit maximalem Schlüssel)

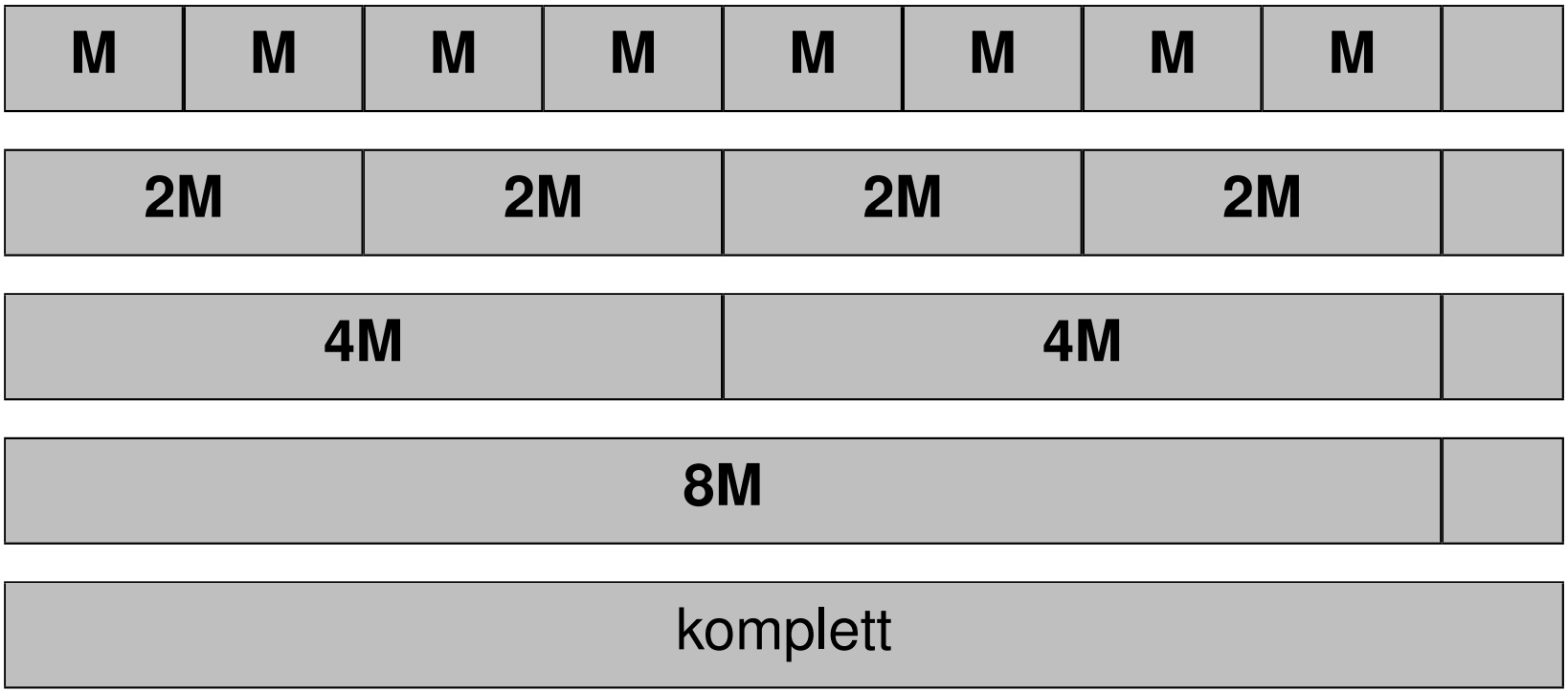
Run Formation Phase:

- Lade Teilfelder der Größe M in den Speicher,
- sortiere sie mit einem in-place-Sortierverfahren,
- schreibe das sortierte Teilfeld wieder zurück auf die Festplatte
- benötigt n/B Blocklese- und n/B Blockschreiboperationen
- Laufzeit: 2n/B Transfers
- ergibt sortierte Bereiche der Größe M

Externes Sortieren

Merge Phasen

- Merge von jeweils 2 Teilfolgen in $O(\lceil \log(n/M) \rceil)$ Phasen
- dabei jeweils Verdopplung der Größe der sortierten Teile



Externes Sortieren

Wie erfolgt dabei der Merge von zwei Runs?

- von jedem der beiden Runs und von der Ausgabesequenz bleibt ein Block im Hauptspeicher

$\Rightarrow$ 3 Puffer

- Anfang: beide Eingabepuffer mit B Elementen (1 Block) laden, Ausgabepuffer leer
- Dann: jeweils führende Elemente der beiden Eingabepuffer vergleichen und das kleinere in den Ausgabepuffer schreiben
- Wenn Eingabepuffer leer $\Rightarrow$ neuen Block laden
- Wenn Ausgabepuffer voll $\Rightarrow$ Block auf Festplatte schreiben und Ausgabepuffer leeren

- In jeder Merge-Phase wird das ganze Feld einmal gelesen und geschrieben

$\Rightarrow$ $(2n/B)(1 + \lceil \log(n/M) \rceil)$ Block-Transfers

Multiway-MergeSort

- Verfahren funktioniert, wenn 3 Blöcke in den Speicher passen
- Wenn mehr Blöcke in den Speicher passen, kann man gleich mehr als zwei Runs (k) mergen.
- Benutze Prioritätswarteschlange (Priority Queue) zur Minimumermittlung, wobei die Operationen $O(\log k)$ Zeit kosten
- $(k+1)$ Blocks und die PQ müssen in den Speicher passen

$\Rightarrow$ $(k+1)B + O(k) \leq M$, also $k \in O(M/B)$

- Anzahl Merge-Phasen reduziert auf $\lceil \log_k(n/M) \rceil$

$\Rightarrow$ $(2n/B)(1 + \lceil \log_{M/B}(n/M) \rceil)$ Block-Transfers

- In der Praxis: Anzahl Merge-Phasen gering
- Wenn $n \leq M^2/B$: nur eine Merge-Phase
  (erst M/B Runs der Größe M, dann einmal Merge)

Übersicht

Prioritätswarteschlangen

M: Menge von Elementen

Jedes Element e hat eine zugeordnete Priorität prio(e)

Operationen:

- M.build($\{e_1, \ldots, e_n\}$): $M = \{e_1, \ldots, e_n\}$
- M.insert(Element e): $M = M \cup e$
- M.min(): gib ein e mit minimaler Priorität prio(e) zurück
- M.deleteMin(): entferne ein e mit minimaler Priorität prio(e) und gib es zurück

Adressierbare Prioritätswarteschlangen

Zusätzliche Operationen:

- insert(Element e): wie zuvor, gibt aber ein Handle (Referenz/Zeiger) auf das eingefügte Element zurück
- remove(Handle h): lösche Element spezifiziert durch Handle h
- decreaseKey(Handle h, int k): erniedrige den zugeordneten Prioritätswert des Elements (auf das h zeigt) auf Wert k (je nach Implementation evt. auch um Differenz k)
- M.merge(Q): $M = M \cup Q; \quad Q = \emptyset;$

Prioritätswarteschlangen mit Listen

Priority Queue mittels unsortierter Liste:

- build($\{e_1, \ldots, e_n\}$): Zeit $\mathcal{O}(n)$
- insert(Element e): Zeit $\mathcal{O}(1)$
- min(), deleteMin(): Zeit $\mathcal{O}(n)$

Priority Queue mittels sortierter Liste:

- build($\{e_1, \ldots, e_n\}$): Zeit $\mathcal{O}(n \log n)$
- insert(Element e): Zeit $\mathcal{O}(n)$
- min(), deleteMin(): Zeit $\mathcal{O}(1)$

$\Rightarrow$ Bessere Struktur als eine Liste notwendig!

Binärer Heap

Idee: verwende binären Baum
Bewahre zwei Invarianten:

- Form-Invariante: fast vollständiger Binärbaum
- Heap-Invariante:

  $\text{prio}(p) \leq \min\{\text{prio}(c1), \text{prio}(c2)\})$