Kapitel 8: Externspeicheralgorithmen

VO Algorithm Engineering

Professor Dr. Petra Mutzel

Lehrstuhl für Algorithm Engineering, LS11

22. VO

26. Juni 2007

Literatur für diese VO

- U. Meyer, P. Sanders und J. Sibeyn (Eds.), Algorithms for Memory Hierarchies, Advances Lectures, Lecture Notes in Computer Science 2625, Springer 2003:
  - Kapitel 2: R. Pagh: Basic External Memory Data Structures, S. 14-35
  - Kapitel 3: A. Maheshwari und N. Zeh: A Survey of Techniques for Designing I/O-Efficient Algorithms

Überblick

8.1 Einführung
- Motivation
- Das Externspeichermodell
- Grundlegende externe Datenstrukturen

8.2 Sortierverfahren

8.3 Externe Array Heaps (Prioritätswarteschlange)

8.1 Einführung

Durchwandern eines Arrays:

```
for (i=0; i<N; i++) D[i]=i
C=Permute(D)
```

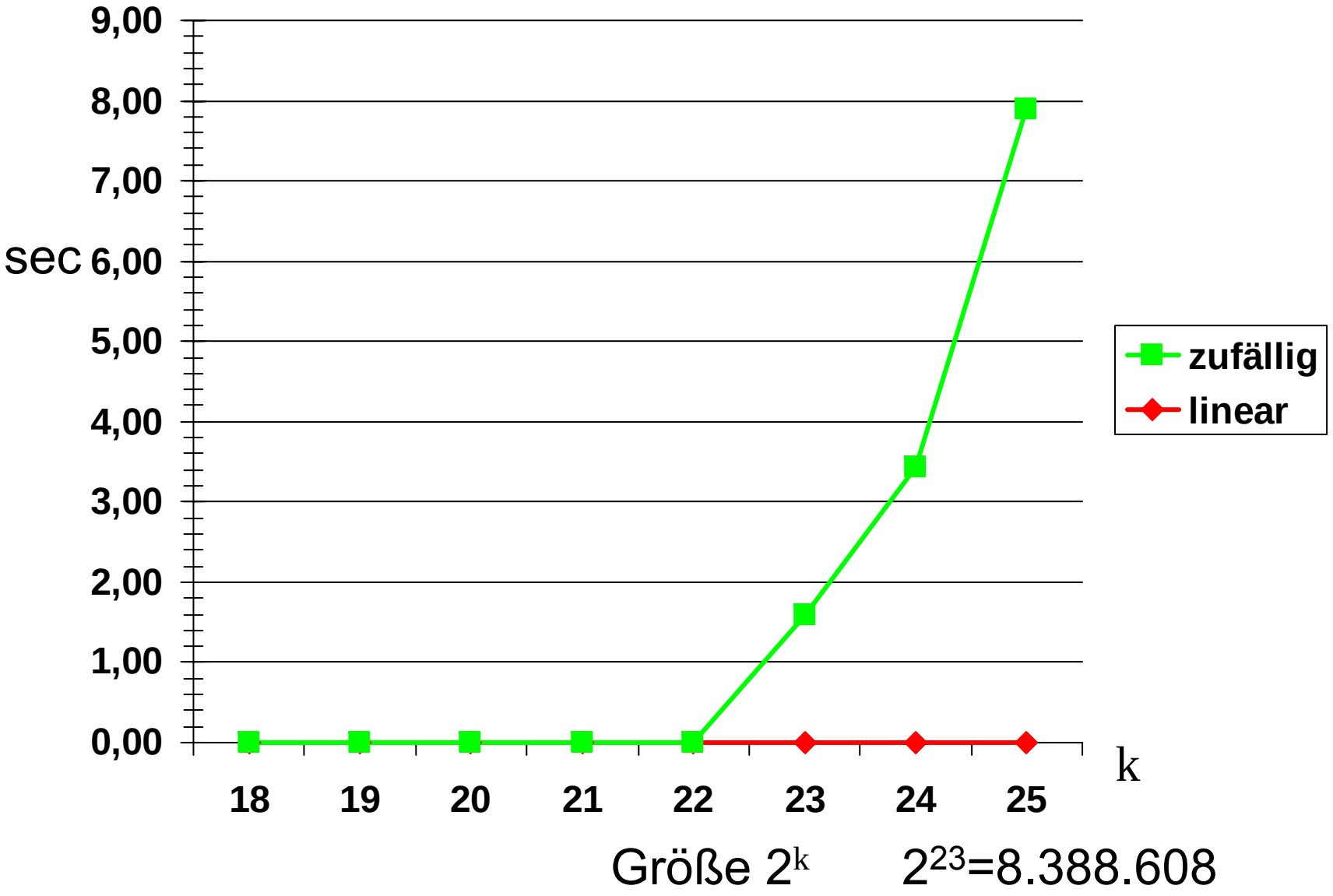
Lineares Durchlaufen:

```
for (i=0; i<N; i++) A[D[i]]=A[D[i]]+1
```

Zufälliges Durchlaufen:

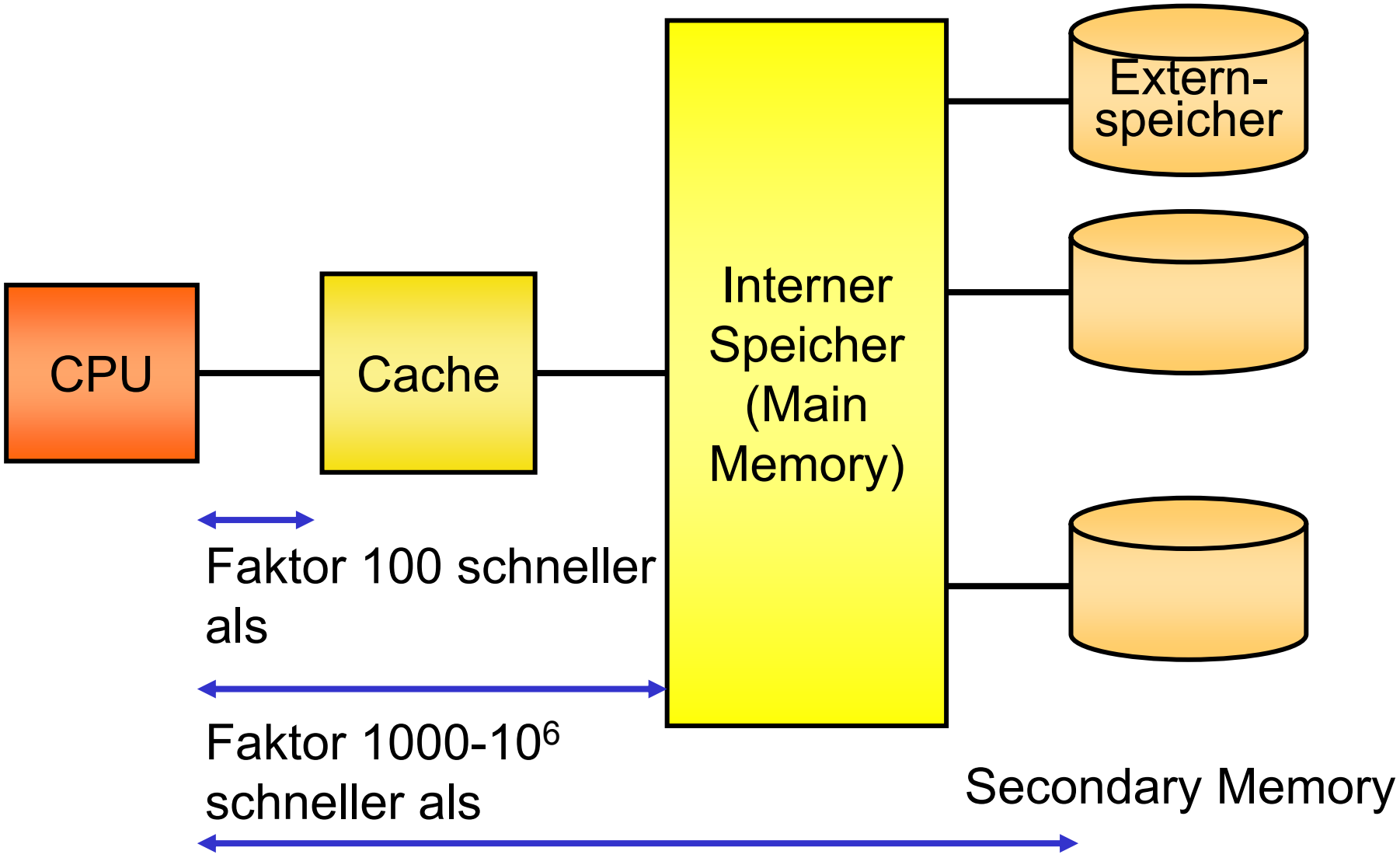
```
for (i=0; i<N; i++) A[C[i]]=A[C[i]]+1
```

Durchwandern eines Arrays



Rechner: CPU 2.4 GHz mit Cache 512 KB: für $N=2^{25}$: 0,39 Sek. vs. 7,89 Sek.

Hierarchisches Speichermodell moderner Computer



Probleme klassischer Algorithmen

- Ein Zugriff im Hauptspeicher spricht jeweils eine Speicherzelle an und liefert jeweils eine Einheit zurück

- Ein Zugriff im Externspeicher (ein I/O) liefert jeweils einen ganzen Block von Daten zurück

- Meist keine Lokalität bei Speicherzugriffen, und deswegen mehr Speicherzugriffe als nötig

Problem ist aktueller denn je, denn

- Geschwindigkeit der Prozessoren verbessert sich zwischen 30%-50% im Jahr
- Geschwindigkeit des Speichers nur um 7%-10% pro Jahr

- „One of the few resources increasing faster than the speed of computer hardware is the amount of data to be processed.“

Donald E. Knuth: The Art of Computer Programming 1967 (Neuauflage 1998):

- When this book was first written, magnetic tapes were abundant and disk drives were expensive. But disks became enormously better during the 1980s,... . Therefore the once-crucial topic of patterns for tape merging has become of limited relevance to current needs. Yet many of the patterns are quite beautiful, and the associated algorithms reflect some of the best research done in computer science during its early years;
- The techniques are just too nice to be discarded abruptly onto the rubbish heap of history. ...
- Therefore merging patterns are discussed carefully and completely below, in what may be their last grand appearance before they accept a final curtain call.

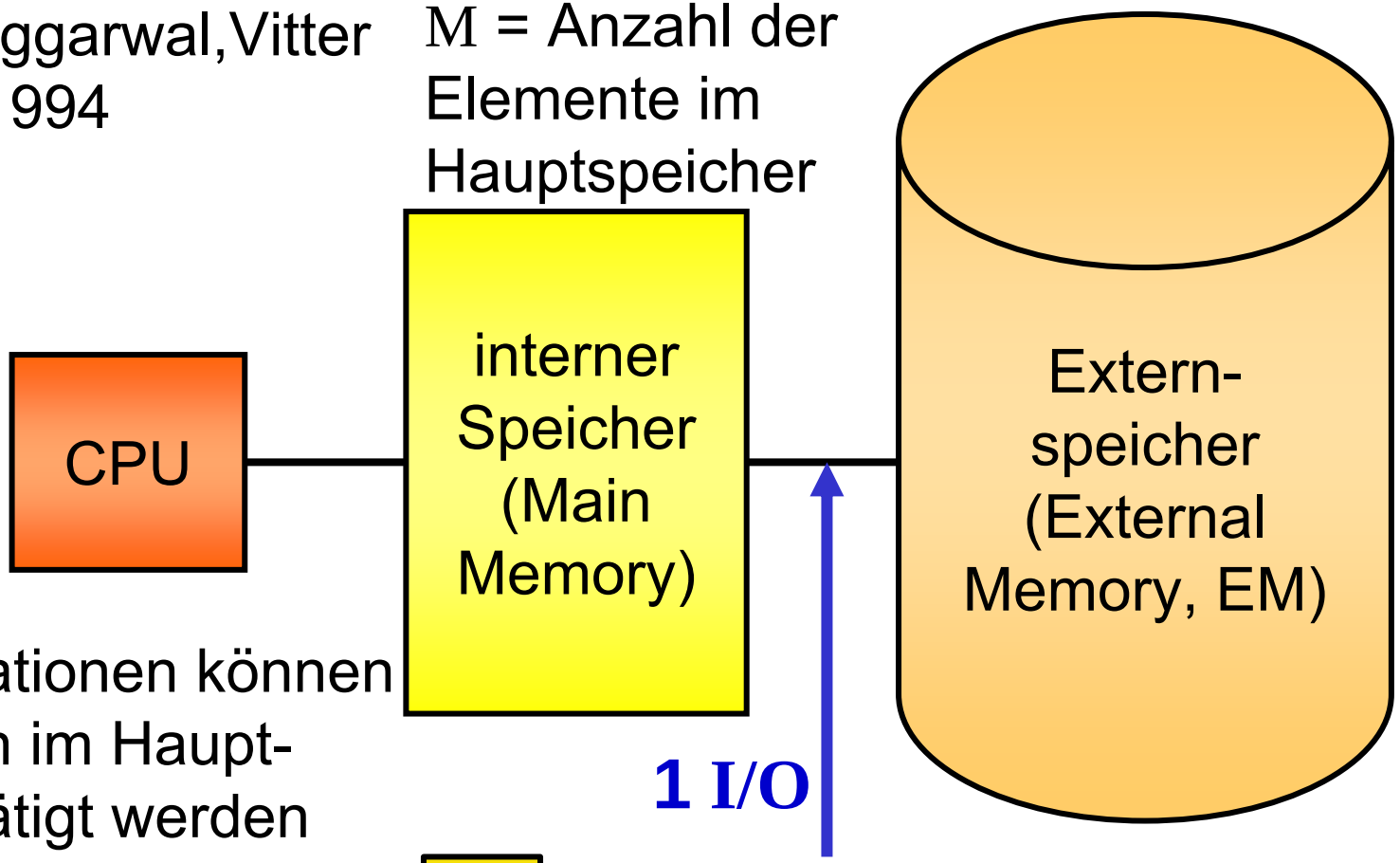
Pavel Curtis in Knuth:The Art of Computer Programming 1967 (Neuauflage 1998):

- For all we know now, these techniques may well become crucial once again.

Das Externspeichermodell

Modell von Aggarwal,Vitter und Shriver 1994

M = Anzahl der Elemente im Hauptspeicher



Rechenoperationen können nur mit Daten im Haupt-speicher getätigt werden

B = Anzahl der Elemente, die in einen Block passen

Annahme: $B<M/2$

Analyse von Externen Algorithmen

- Anzahl der ausgeführten I/O-Operationen

- Anzahl der ausgeführten CPU-Operationen im RAM-Modell

- Anzahl der belegten Blöcke auf dem Sekundärspeicher

Ziele beim Entwurf Externer Algorithmen

- **Interne Effizienz:**
  - Anzahl der RAM-Operationen vergleichbar zu den besten internen Algorithmen

- **Örtliche Lokalität:**
  - Ein r/w Block sollte möglichst viele nützliche Daten enthalten

- **Zeitliche Lokalität:**
  - Daten, die im internen Speicher sind, sollten möglichst verarbeitet werden, bevor sie wieder herausgeschrieben werden.

Externe Datenstrukturen: Stacks

- Ein Stack S repräsentiert eine dynamische Menge von Elementen (maximal N)

- **Operationen:**
  - Insert(x): Einfügen eines neuen Elements in S
  - Delete: Ausgabe und Entfernung des letzten eingefügten Elements aus S

Interne Algorithmen für Stack der Größe N:
Array der Länge N und zwei Zeiger

Im Worst Case: 1 I/O per Insert und Delete Operation

Externe Stacks

- **Buffer für externe Stacks:**
  - Array im internen Speicher der Länge $2B$
  - enthält zu jedem Zeitpunkt die letzten k eingefügten Elemente, wobei $k \leq 2B$

- **Insert(x):**
  - Meistens 0 I/Os benötigt, außer: wenn Buffer voll ist
  - Dann: die B ältesten Elemente werden in EM ausgelagert.

- **Delete:**
  - Meistens 0 I/Os benötigt, außer: wenn Buffer leer
  - Dann: 1 I/O holt die nächsten B Elemente aus dem EM (diejenigen, die als letztes ausgelagert wurden)

Externe Stacks

- **Analyse der Operationen:**
  - Insert(x): $1/B$ I/Os amortisiert
  - Delete: $1/B$ I/Os amortisiert

- **Dies ist bestmöglich!**
- Denn: mit einer I/O können nicht mehr als B Elemente gleichzeitig gespeichert oder gelesen werden

Externe Datenstrukturen: Queues

- **Operationen:**
  - Insert(x): Einfügen eines neuen Elements in S
  - Delete: Ausgabe und Entfernung des ältesten Elem. aus S

- **Zwei Buffer: R und W Buffer:**
  - Zwei Arrays im internen Speicher der Länge jeweils B

- **Insert(x):**
  - zu W Buffer, außer: wenn voll
  - Dann: schreibe alle B Elemente in EM

Analyse: $1/B$ I/Os

- **Delete:**
  - aus R Buffer, außer: wenn Buffer leer
  - Dann: 1 I/O holt die nächsten B Elemente aus dem EM (wenn keine dort, dann aus W Buffer (Buchführung!))

Analyse: $1/B$ I/Os

Externe Datenstrukturen: Lineare Listen

- Eine lineare Liste L liefert eine effiziente Implementierung für dynamische geordnete Listen von Elementen

- **Operationen:**
  - Search(x): Sucht ein Element in L mit Schlüssel x
  - Insert(x): Einfügen in L eines neuen Elements mit Schlüssel x an die richtige Stelle
  - Delete(x): Entfernung des Elements mit Schlüssel x aus L

Interne Algorithmen für Lineare Listen:

Im Worst Case: 1 I/O für jeden einzelnen Zeiger, der verfolgt wird

Lineare Listen

- **Idee: Erhalte räumliche Lokalität:**
  - Füge jeweils ungefähr k konsekutive Listenelemente in einem Block zusammen
  - Verlinke jeweils benachbarte Blöcke

- **Konkrete Umsetzung:**
  - **Invariante:** In jedem Paar konsekutiver Blöcke existieren mehr als $2/3B$ konsekutive Listen-Elemente

- **Search(x):**
  - Traversiere die Liste bis zur richtigen Stelle: $O(N/B)$ I/Os
  - Die Einführung der Invariante führt zu einer Erhöhung der benötigten I/Os auf höchstens $3N/B$.

Lineare Listen

- **Insert(x):**

Analyse: O(1+N/B) I/Os

  - Traversiere die Liste bis zur richtigen Stelle: $O(N/B)$ I/Os
  - Meistens 1 I/O benötigt, außer: wenn Block voll ist
  - Dann: Falls ein benachbarter Block noch Platz hat: tausche 1 Element aus: $O(1)$ I/Os
  - Sonst: Teile den Block in 2 Teile der Größe $B/2$: $O(1)$ I/Os

Danach sind mind. $B/3$ Delete-Operationen notwendig, damit L an dieser Stelle wieder verschmolzen werden muß

Verbesserung aus VO: statt Austausch eines Elements zu Nachbar:
Falls Nachbar weniger als $B/6$ Elemente besitzt, dann tausche gleich $B/2$ aus.
Dies verhindert Problem bei wiederholtem DEL aus Nachbar und INS(x)

Lineare Listen

- **Delete(x):**

  Analyse: $O(1+N/B)$ I/Os

  – Traversiere die Liste bis zur richtigen Stelle: $O(N/B)$ I/Os
  – Meistens 1 I/Os benötigt, außer: wenn B danach mit benachbarten Blöcken ≤ $2/3B$ Elemente besitzt:
  – Dann: Verschmelze die beiden Blöcke: $O(1)$ I/Os

8.2 Externes Sortieren

- Einführung Externes Sortieren

- Externes Merge-Sort

- Untere Schranke für Externes Sortieren

Wichtigste Externe Sortierparadigmen

Distribution Paradigma

1. **Partitionierungsphase:** Partitioniere die Input-Folge S in Teilfolgen $S_0,\ldots,S_k$, so dass für alle $0 \leq i < j \leq k$ und zwei beliebige Elemente $x \in S_i$ und $y \in S_j$: $x \leq y$. Dazu wähle Splitters $x_1 \leq \ldots \leq x_k$ von S.

2. **Sortierphase:** Sortiere jede Teilmenge S_i rekursiv.

3. **End:** Hänge die sortierten Teilfolgen aneinander.

- Beispiel Quicksort:
  - Falls alle Teilmengen ungefähr $|S|/(k+1)$ Elemente besitzen, dann Laufzeit: $O(N \log N)$

Wichtigste Externe Sortierparadigmen

Merging Paradigma

1. **Run-Formationsphase:** Partitioniere die Input-Folge in sortierte Teilsequenzen: „Runs“

2. **Merging-Phase:** Verschmelze diese „Runs“ solange, bis nur noch ein „Run“ existiert

- Beispiel Merge Sort:
  - Die Runs sind zu Beginn 1-elementig
  - Verschmelzen mit 2-Way Merging (in Paaren aufgeteilt).
  - Laufzeit: $O(N \log N)$

Externer Merge-Sort

I/O-Komplexität des internen Merge-Sort:

1. **Run-Formationsphase:** 0 I/Os

2. **Merging-Phase:**
   - Verschmelzen der Teilfolgen S_1 und S_2: $O(1+(|S_1|+|S_2|)/B)$ I/Os
   - Anzahl der I/Os auf einer Schicht: $O(N+N/B)$ I/Os
   - Über alle Schichten: $O((N+N/B) \log N)$ I/Os

Externer Merge-Sort

1. Verbesserung: Verhindere in der Run-Formationsphase 1-elementige Mengen!

- Beende die Aufteilung bei Teilmengen der Länge M
- Lade die N/M Stücke nacheinander in das Main Memory
- Sortiere diese im Main Memory
- Schreibe die sortierte Teilfolge zurück nach EM

- Diese Aufteilung kostet zusätzlich: $O((N/M)(M/B))=O(N/B)$ I/Os für das Hin- und Herkopieren (das Sortieren an sich kostet kein I/O)
- Anzahl der I/Os auf einer Schicht: $O(N/M+N/B)$ I/Os
- Über alle Schichten: $O((N/M+N/B) \log (N/M))$ I/Os
- Dies ist gleich $O(N/B(1+\log(N/B)))$

Externer Merge-Sort

2. Verbesserung: Verschmelze jeweils $k=M/(2B)$ Runs

- Kopiere die jeweils kleinsten Elemente $x_1,\ldots,x_k$ der Runs $S_1,\ldots,S_k$ in das MM
- Kopiere das Minimum x_i der Elemente in den Output Run
- Lese das nächste Elemente von S_i usw.

- Verschmelzen der Teilfolgen $S_1,\ldots,S_k$: $O(k+(|S_1|+\ldots+|S_k|)/B)$ I/Os
- Anzahl der I/Os auf einer Schicht: $O(N/M+N/B)$ I/Os
- Über alle Schichten: $O((N/M+N/B)\ \log_{M/B}\ (N/B))$ I/Os
- Dies ist gleich $O(N/B(1+\log_{M/B}(N/B)))$

Externer Merge-Sort

Effiziente Verschmelzung von k Runs? (Intern)

Minimumsuche der Keys $x_1,\ldots,x_k$?

- Naiv: $O(k)$: zu teuer, denn dann würde die Laufzeit der Merge-Phase $O(k\ N\ \log_k(N/B))$ sein.
- Lösung: Halte die jeweils kleinsten Elemente $x_1,\ldots,x_k$ in einer Prioritätsschlange in MM
- Speicherplatz: $O(k)=O(M/B)$

Laufzeit von k-Multiway Merging

- Phase 1: Sortiere N/M Stücke der Länge M: $O((N/M)\ M \log M) = N \log M$
- pro Schicht: Verschmelze N Elemente inkl. Minimumsuche: $O(N \log k)$
- Tiefe des Baumes: $O(\log_k(N/B))$
- Insgesamt: $O(N \log M)+(N \log k)\ \log_k(N/B) = O(N \log N)$

- Verwende Logarithmus-Gesetz: $\log_b a=(\log_c a)/(\log_c b)$

Externes Sortieren

Theorem: Eine Menge von N Elementen kann mit Hilfe von k-Multiway Merging in interner Zeit $O(N \log N)$ mit $O(N/B(1+\log_{M/B}(N/B)))$ I/Os sortiert werden.

ENDE

Untere Schranken für Sortieren

- Wieviele I/O-Operationen sind mindestens notwendig um eine gewisse Permutation der Eingabefolge zu erhalten?

- Soviele sind auch notwendig um diese Folge zu sortieren.

Untere Schranken für Sortieren

Restriktionen unseres Modells:

1. MM kann M Elemente aufnehmen.
2. EM ist ein Array von Elementen.
3. EM kann nur an Blockbegrenzungen beschrieben werden
4. Zu Beginn enthalten die ersten N/B Blöcke des EM den Input
5. Alles andere ist leer (MM und restl. EM)
6. Nach Ablauf enthalten die ersten N/B Blöcke im EM den Output
7. Einzige erlaubte Operation: *move* (nicht split, duplicated)
8. *read*: bewegt B Elemente eines Blocks von EM nach MM
9. *write*: bewegt B Elemente von MM in einen Block von EM

Untere Schranken für Sortieren

Theorem: Um eine gegebene Permutation von N Elementen herzustellen wird mindestens die folgende Anzahl von I/Os benötigt:

$$t \geq 2\frac{N}{B}\frac{\log(N/eB)}{\log(eM/B) + 2\log(N/B)/B}$$

- Für $N=O((eM/B)^{B/2})$ dominiert der Term $\log(eM/B)$ den Nenner und wir erhalten:

$$2\frac{N}{B}\frac{\log(N/eB)}{O(\log(eM/B))} = \Omega(\frac{N}{B}\log_{M/B}\frac{N}{B})$$

(O.Bw.)

Untere Schranken im EM-Modell

- Einlesen einer Menge von N Elementen benötigt mindestens $\Theta(N/B)$ I/O's

- Sortieren einer Menge von N Elementen benötigt mindestens

  $\Theta(N/B \log_{1+M/B} (1+N/B))$

- Suche in dynamischen Daten von N Elementen benötigt mindestens Zeit

  $\Theta(\log N / \log B)$ I/O-Operationen