Externspeicheralgorithmen I

Speichermodell

Einfache Datenstrukturen

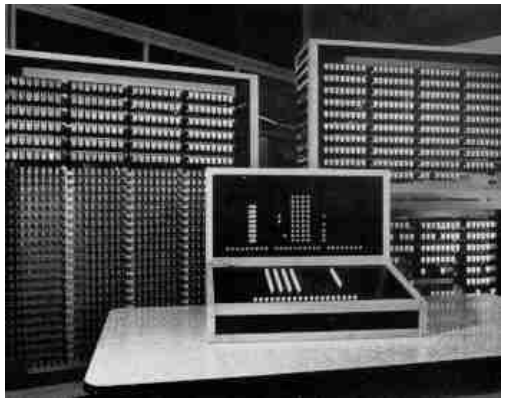
Sortieren

Maschinenmodell

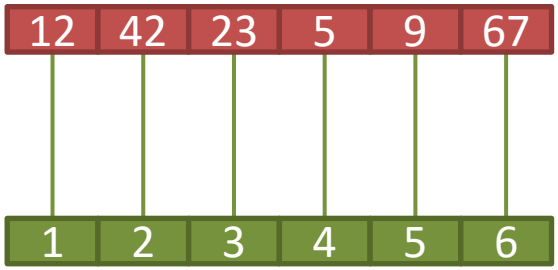
von Neumann Modell

RAM-Modell

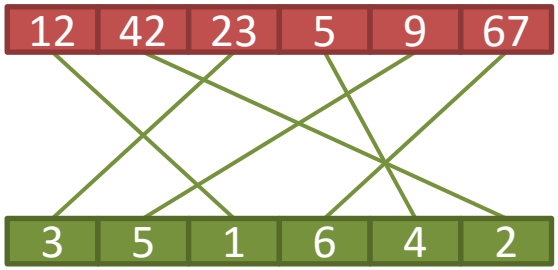
- In jedem Rechenschritt kann jederzeit direkt auf eine beliebige Speicheradresse zugegriffen werden (lesend&schreibend)
- **Früher** tatsächlich ohne „extra“ Wartezeit



Oops: Sequentiell VS Random Access



12 + 42 + 23 + 5 + 9 + 67 = 158



23 + 9 + 12 + 67 + 5 + 42 = 158

```
int data[N]
int idx[N]

for i = 1..N:
  idx[i] = i
```

sequenziell

```
sum = 0
for i = 1..N:
  sum += data[idx[i]]
```

```
permute(idx)
```

random access

```
sum = 0
for i = 1..N:
  sum += data[idx[i]]
```

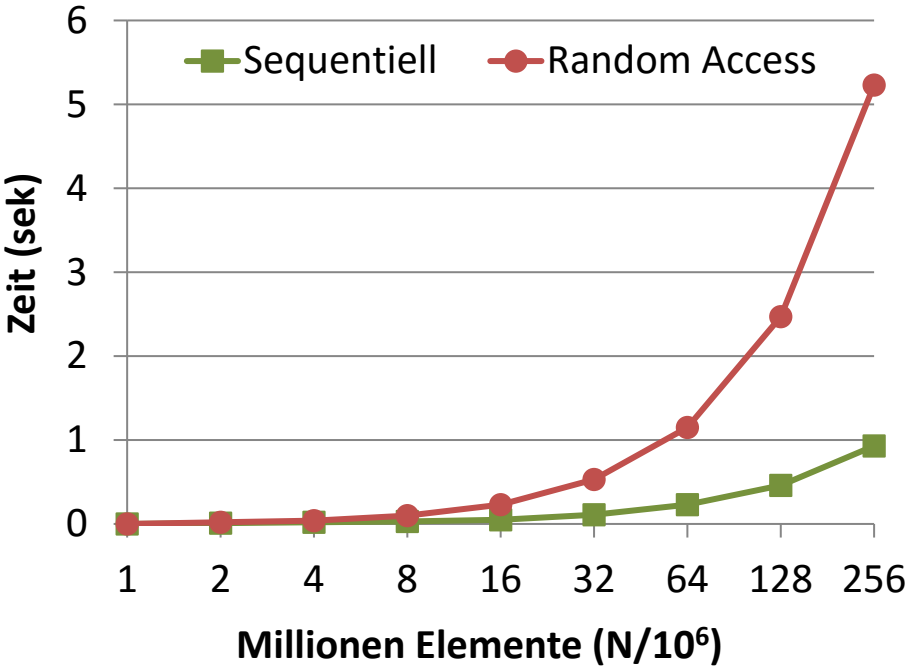
sequenziell vs. random access

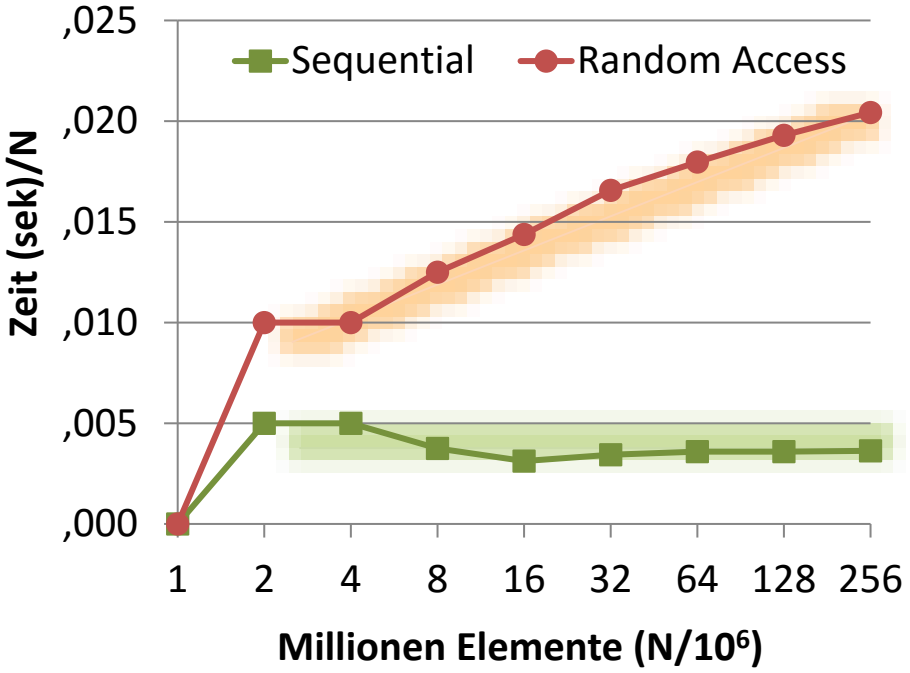
Resultat: ident

O-Notation: **ident** O(N)

Oops: Sequentiell VS Random Access

O(*N*) sollte eine waagrechte Linie sein





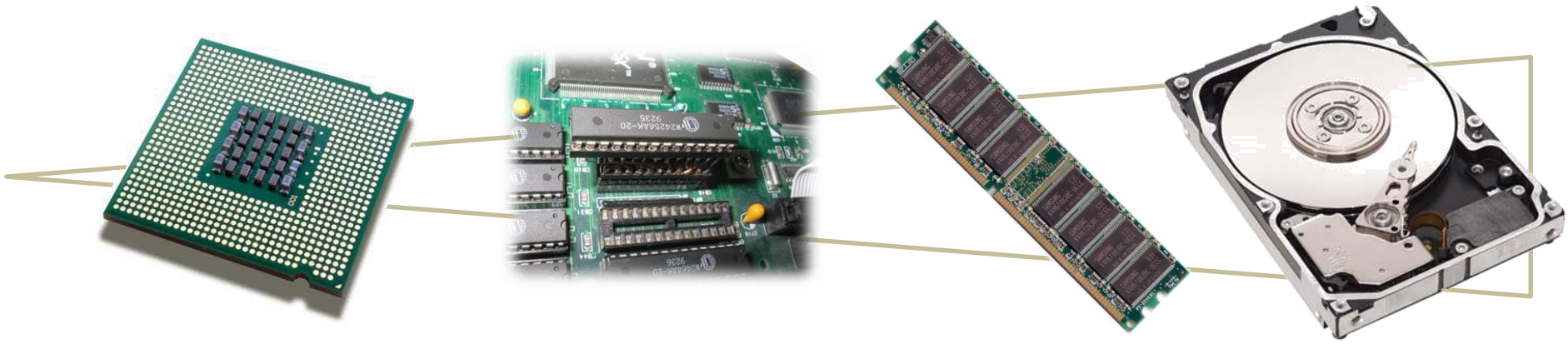
- Intel Core i7 860, 2.80GHz, QuadCore, 8GB RAM
- 1 Core, 32bit, g++ 4.4 –O0, Ubuntu 10.10

Bei schreibendem Zugriff wäre es noch ausgeprägter!

```
sum = 0
for i = 1..N:
  sum += data[idx[i]]
```

```
for i = 1..N:
  data[idx[i]] += 1
```

Speicherhierarchien



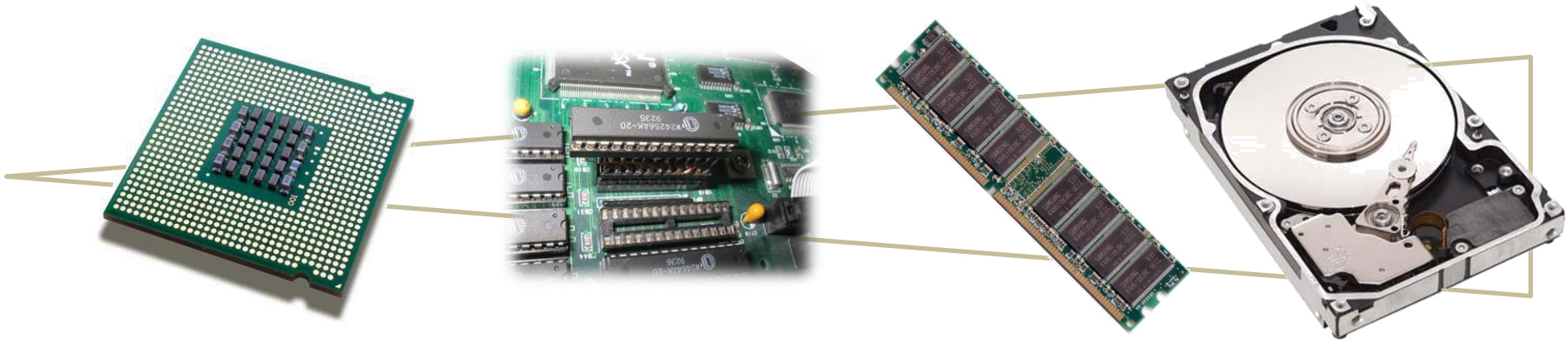
Schnell aber klein

Langsam aber groß

	CPU Register	Cache Level 1	Cache Level 2	Cache Level 3	RAM	HDD
Größe	16 (64bit)	32+32 KB	256 KB	8MB	8GB	1TB
Latenz	0,5 ns	0,5 ns	3 ns	20ns	40–100 ns	10 ms
…in CPU Zyklen	1	1-2	3-7	30-40	80-200	10^7

Größenordnungen bei einem Intel Core i7

Speicherhierarchien



Schnell aber klein

Langsam aber groß

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

[IEEE InfoVis 2003 Call-For-Papers]

Entwicklung der Geschwindigkeiten:

CPU	ca. + 30% / Jahr
Speicher	+ 7–10% / Jahr

Betrachtung von Externspeicheralgorithmen wird immer wichtiger!

I/O-Modell

Klassische O-Notation benutzt **RAM-Modell**

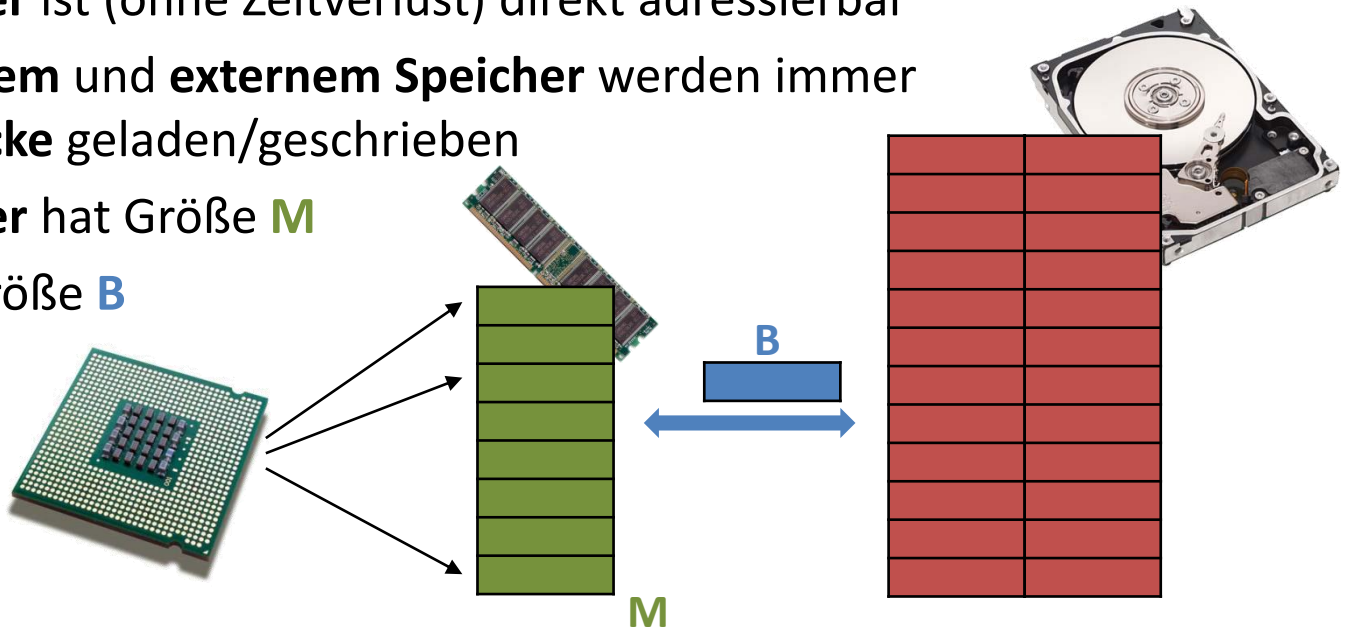
- Jede Operation benötigt gleich viel Zeit (1 Zeiteinheit)
- Jede gewünschte Speicheradresse steht direkt zum Lesen/Schreiben bereit

$\Rightarrow$ Zähle # Operationen

I/O-Modell (nach Aggarwal und Vitter), auch: „**cache-aware**“

Noch immer vereinfacht, aber guter Tradeoff zw. Realität und Analysierbarkeit

- **Interner Speicher (zB. RAM) vs. Externer Speicher (zB. HDD)**
- **Interner Speicher** ist (ohne Zeitverlust) direkt adressierbar
- Zwischen **internem** und **externem Speicher** werden immer ganze Daten**blöcke** geladen/geschrieben
- **Interner Speicher** hat Größe **M**
- **Blöcke** haben Größe **B**



I/O-Modell

Klassische O-Notation benutzt **RAM-Modell**

⇒ Zähle # Operationen

I/O-Modell (nach Aggarwal und Vitter), auch: „**cache-aware**“

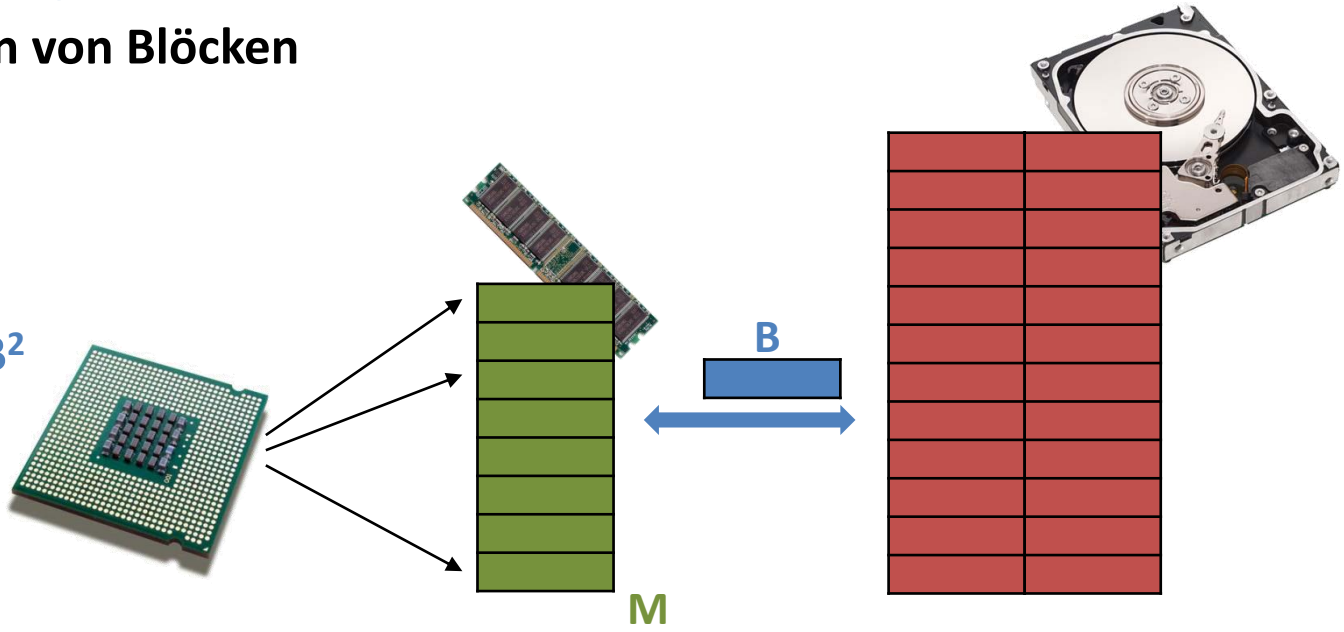
Noch immer vereinfacht, aber guter Tradeoff zw. Realität und Analysierbarkeit

⇒ **Zähle # interne Operationen**
Ziel: Möglichst gleich mit RAM-Modell

⇒ **Zähle # I/O Zugriffe**
Laden/Schreiben von Blöcken

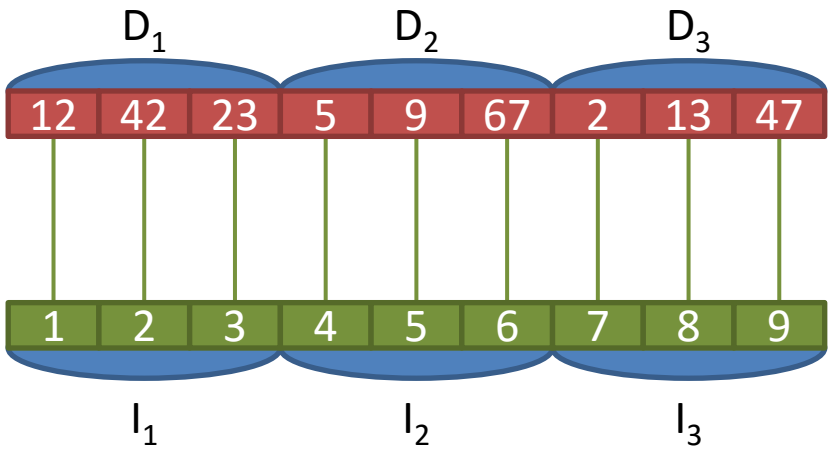
Annahmen

- immer: $M \geq 2B$
- tall-cache: $M \geq B^2$
- $M \geq B^{1+\varepsilon}$

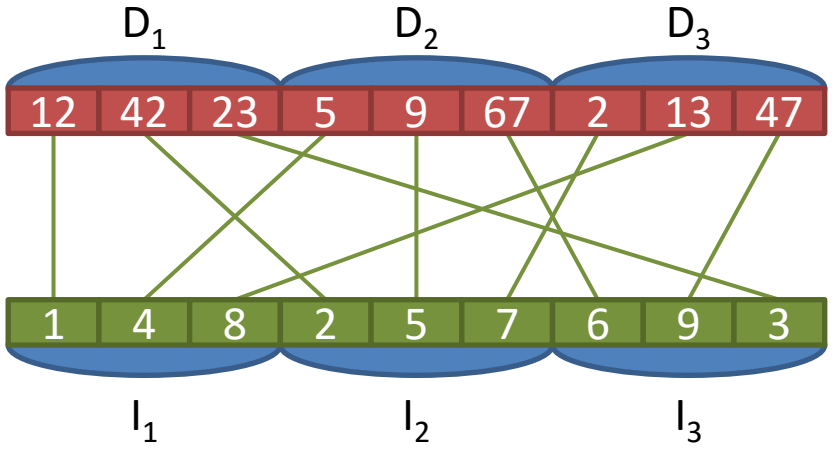


Lokalität

Warum war **Sequentiell** schneller als **Random Access** ?

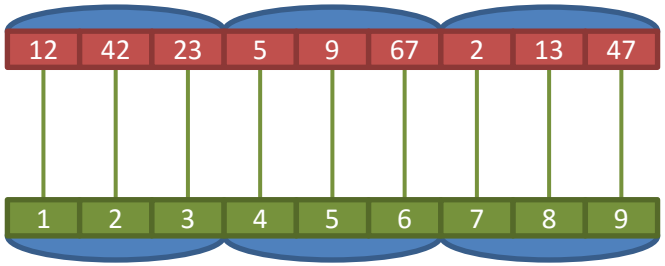


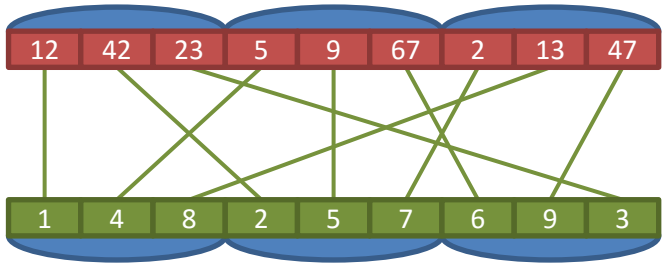
data[idx[1]]: load(I_1), load(D_1)
data[idx[2]]: (schon geladen)
data[idx[3]]: (schon geladen)
data[idx[4]]: load(I_2), load(D_2)
data[idx[5]]: (schon geladen)
…
 # loads: N/B + N/B
O(N/B)



data[idx[1]]: load(I_1), load(D_1)
data[idx[2]]: load(D_2)
data[idx[3]]: load(D_3)
data[idx[4]]: load(I_2), load$^?$(D_1)
data[idx[5]]: load$^?$(D_2)
…
 # loads: ≤ N/B + N
O(N)

Lokalität





Ziele bei der Entwicklung von Externspeicheralgorithmen

Örtliche Lokalität

Ein gelesener Block sollte möglichst viel nutzbare Information enthalten.

Zeitliche Lokalität

Möglichst viele Daten im internen Speicher bearbeiten, bevor sie wieder rausgeschrieben werden.

Interne Effizienz

Optimiere obige Lokalitäten, ohne (große) Einbußen bzgl. der internen Operationen gegenüber dem optimalen Algorithmus im RAM-Modell.

Externe Datenstrukturen

Stack

push(type *v*) Legt *v* oben auf den Stack
type **pop**() Liefert oberstes Element des Stacks und entfernt es

Implementierungen (optimal im RAM-Modell)

- Array + Zeiger auf oberstes Element
- Zeigerverkettete Liste

Anzahl der I/Os? (für beliebige Abfolge von Operationen)

O(1) pro Operation!

Besser: **Extern-Stack**

Extern-Stack

Extern-Stack

- Interner Speicher („Puffer“): Array J der Größe $2B$; restlichen Daten extern
- J enthält zu jedem Zeitpunkt die $k \leq 2B$ obersten Elemente

push(type ***v*****)**

- Falls $k \leq 2B$ („meistens“): Füge v in J ein. → **Kein I/O**
- Falls $k = 2B$ („Puffer voll“): Lagere die untersten B Elemente von J auf den externen Speicher aus; füge v in J ein. → **1 I/O**

type **pop()**

- Falls $k > 0$ („meistens“): Entferne oberstes Element aus J. → **Kein I/O**
- Falls $k = 0$ („Puffer leer“): Lade die obersten B Elemente aus dem externen Speicher nach J; entferne oberstes Element aus J. → **1 I/O**

Beobachtung

Nach jedem I/O-Zugriff mindestens B viele Operationen ohne I/O!

⇒ **O(1/B)** I/Os pro Operation (amortisiert)

⇒ Dies ist bestmöglich, da nur B Elemente pro I/O

Externe Datenstrukturen

Weitere einfache Datenstrukturen

- Analog für Queue → Übung
- Wie für Listen? → Übung

Kompliziertere Datenstrukturen

- **Priority Queue?** → nächste Woche

Zunächst

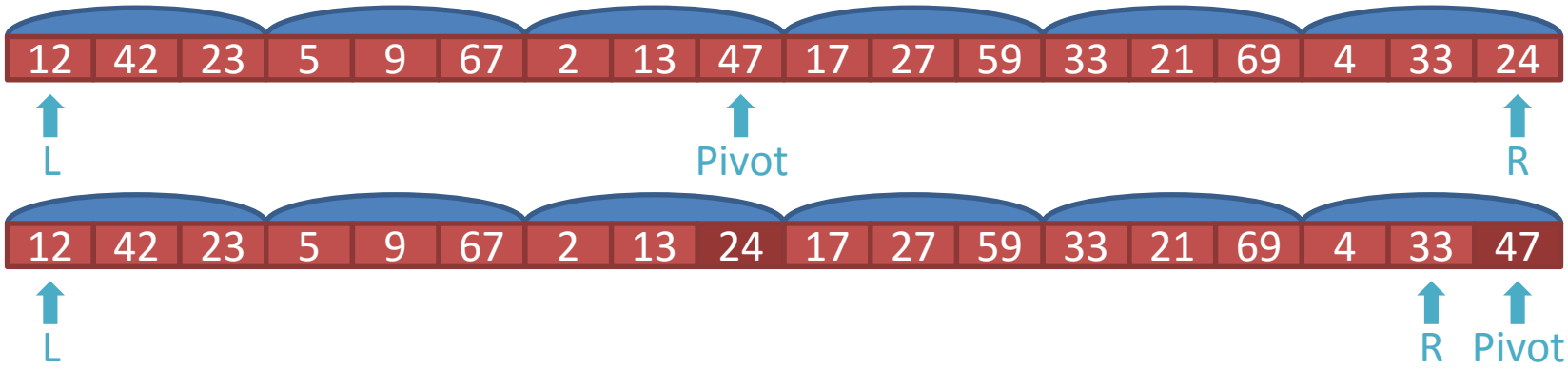
Einfache Algorithmen

- **Sortieren** (vergleichsbasiert)

im RAM-Modell am effizientesten…

- **Quick-Sort** $O(N^2)$, randomisiert/erwartet: $O(N \log N)$
- **Merge-Sort** $O(N \log N)$
- **Heap-Sort** $O(N \log N)$

Quick-Sort



Partitionierungsschritt (N_0 viele Elemente)

load_block_of(Pivot) → ersparbar wenn man gleich letztes Element wählt

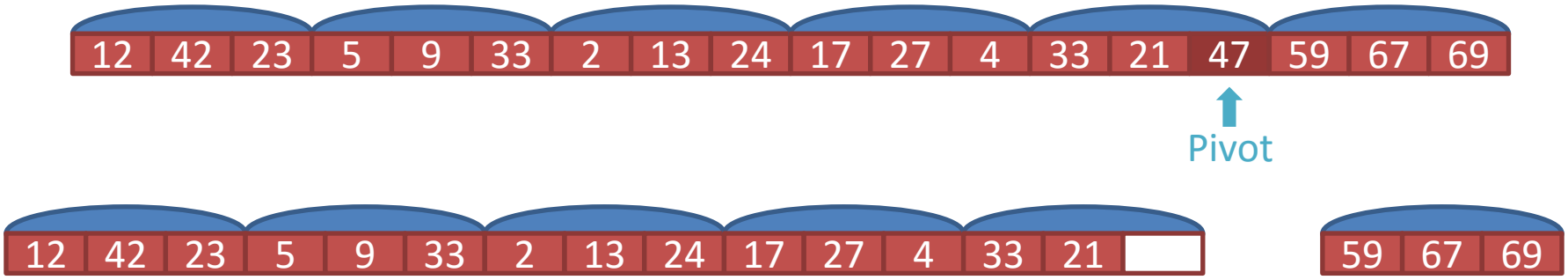
load_block_of(L), load_block_of(R)

Laufe mit L nach rechts, mit R nach links:

- stoppe jeweils wenn Element kleiner (größer) als Pivot. Vertausche.
  → sequenziell! $O(N_0/B)$ I/Os
- fertig wenn R links von L. Tausche Pivot in die „Mitte“.
  → Mitte ist schon geladen, *load_block(ganz-rechts)* → geladen falls $M \geq 3B$

Partitionieren von N_0 Elementen: $O(N_0/B)$ I/Os
→ jeder Block wird nur „1 mal“ angeschaut

Quick-Sort



Rekursion

- Pro Rekursionstiefe: **O(N/B)** I/Os
- Sobald N<M: Lade alle M/B Blöcke und sortiere rekursiv ohne weitere I/Os.
- Rekursionstiefe?
  average: $O(\log_2 N)$, worst: $O(N)$

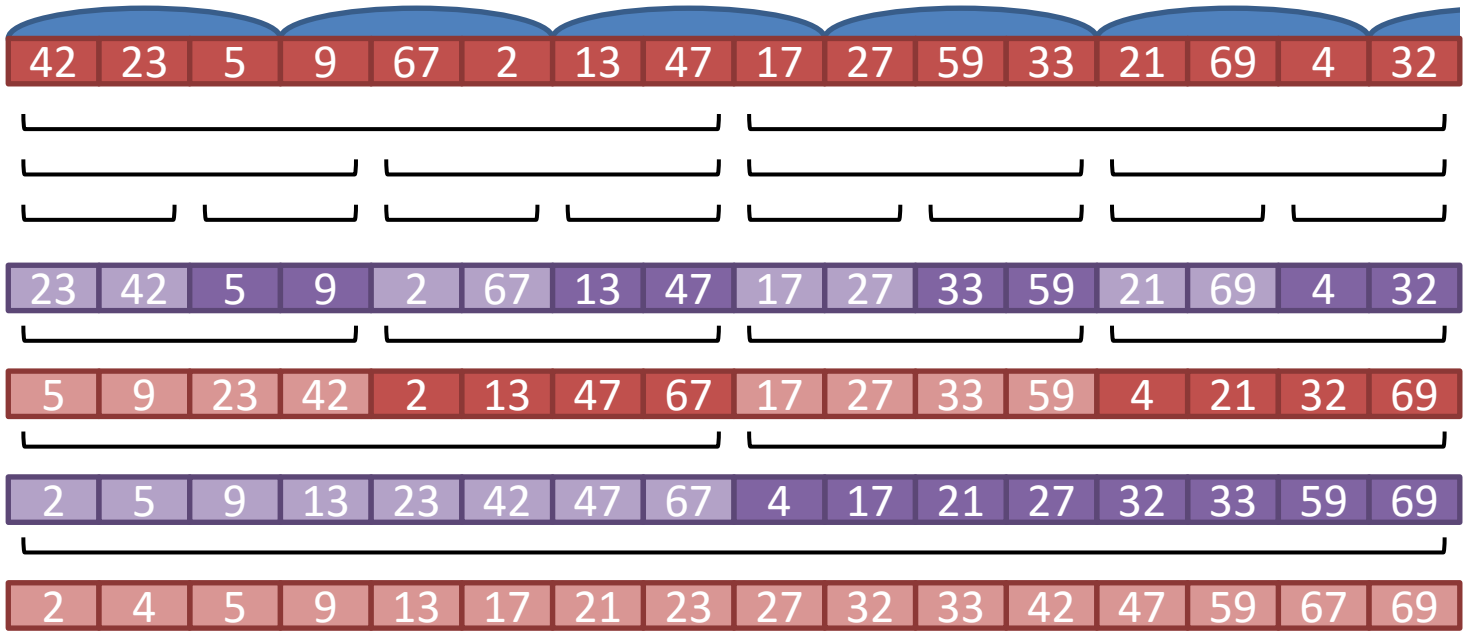
Rekursionstiefe solange I/Os benötigt werden (Analyse wie traditionell):

- average: $O(\log_2 (N/B))$, worst: $O(N/B)$

Gesamt # I/Os:

- average: $O((N/B) \log_2 (N/B))$, worst: $O(N^2/B^2)$

Merge-Sort



Rekursiv unterteilen: nur Index-Berechnungen, keine I/Os

Bottom-up: Teilsequenzen („Runs“) mergen, Hilfsarray.

Mergen zweier Runs der Längen N_1,N_2: $O(1+(N_1+N_2)/B)$ I/Os

Anzahl der Merge-Operationen per Rekursionsebene: $O(N)$

I/Os pro Rekursionsebene: $O(N + N/B)$

 # I/Os ingesamt: $O(N \log_2 N) \rightarrow$ ☹, Quick-Sort hatte $O((N/B) \log_2 (N/B))$

Externes Merge-Sort

Beschleunige Merge-Sort (1)

Verhindere I/Os für kleine Runs

- Sobald ein Run $\leq M/2$: Lade kompletten Run in Speicher, sortiere intern (ohne I/Os), schreibe die Lösung raus. $\rightarrow$ $O(M/B)$ I/Os
- Teile das Array in $2N/M$ **Chunks** der Größe $\leq M/2$, und sortiere intern: $O((N/M) \cdot (M/B)) = O(N/B)$ I/Os
- Merge diese **Chunks** nun gemäß Merge-Sort:
  - **Rekursionstiefe:** $O(\log_2 (N/M))$
  - **I/Os pro Rekursionsebene:** $O(N/M + N/B)$

- I/Os ingesamt: $O(N/B + (N/M+N/B) \log_2 (N/M))$ $(N/M < N/B)$
  $= O((N/B) \log_2 (N/M))$

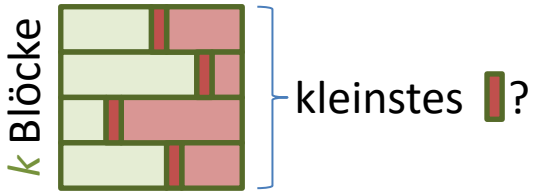
Externes Merge-Sort

Beschleunige Merge-Sort (2)

Merge nicht nur 2 Runs → *k*-way Merge

- $k = ½ (M/B) \rightarrow M/B$ = Anzahl der Blöcke die in internen Speicher passen
- Verschmelze immer k Runs:
  - Benutze jeweils einen Block für jeden Run.
  - Lade die ersten B Elemente jedes Runs in seinen Block. Lade immer einen Block nach, wenn geladener Block fertig abgearbeitet ist.
  - Iterativ: Verschiebe kleinstes der „obersten“ Elemente in Ausgabepuffer
    - **Interner Rechenaufwand?**
- Es ändert sich nur die **Rekursionstiefe:** $O(\log_{M/B} (N/M))$
- I/Os ingesamt: $O((N/B) \log_{M/B} (N/M))$

Externes Merge-Sort – Interner Aufwand



Verschmelzen von *k* Runs: Finde Minimum

Naïv: lineare Suche, O(k)

- Aufwand pro Rekursionsebene $O(k \cdot N)$ statt $O(N) \rightarrow$ ☹

Priority Queue

- Kleinstes Element pro Block in eine Priority-Queue (zB. Min-Heap) (Größe: $k = \frac{1}{2} M/B$, interner Speicher reicht dafür aus)
  - Wähle kleinstes Element in PQ ($O(1)$), und füge vom entsprechenden Block das nächstkleinste Element in PQ ein ($O(\log_2 k) = \log_2(M/B)$)

Gesamtaufwand (interne Rechenoperationen)

- Sortieren der Chunks: $O(N/M \cdot M \log_2 M) = O(N \log_2 M)$
- Rekursionstiefe: $O(\log_{M/B} (N/M))$
- Pro Rekursionsebene (inkl. Minimum-Finden): $O(N \log_2 (M/B))$

Gesamt: $O(N \log_2 M + N \log_2 (M/B) \log_{M/B} (N/M)) = O(N \log_2 N)$

$\rightarrow$ Effizient wie internes Merge-Sort!

Externes Sortieren

	Interne Operationen	I/Os
Internes Quick-Sort (Average, bzw. Randomisiert/Erwartungswert)	$\tilde{O}(N \log_2 N)$	$\tilde{O}((N/B) \log_2 (N/B))$
Internes Merge-Sort	$O(N \log_2 N)$	$O(N \log_2 N)$
Externes Merge-Sort	$O(N \log_2 N)$	**$O((N/B) \log_{M/B} (N/M))$**

I/O Aufwand fundamentaler Operationen

Durchsehen von N Elementen — $\mathbf{Scan}_{M,B}(N) = \Theta(N/B)$

Suchen in N Elementen — $\mathbf{Search}_{M,B}(N) = \Theta(\log_B N)$ → Übung

Sortieren von N Elementen — $\mathbf{Sort}_{M,B}(N) = \Theta((N/B) \log_{M/B} (N/M))$

→ Diese Komplexitäten werden oft als Black-Box innerhalb von anderen Algorithmen benutzt