Externspeicheralgorithmen I

Speichermodell

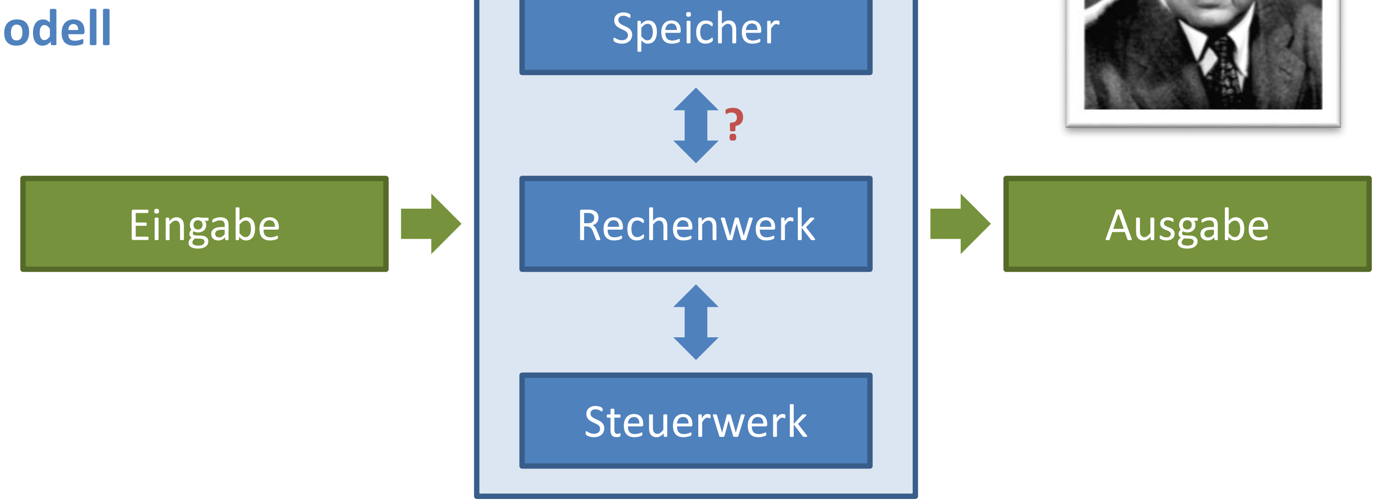
Einfache Datenstrukturen

Sortieren

Markus Chimani – Uni Osnabrück

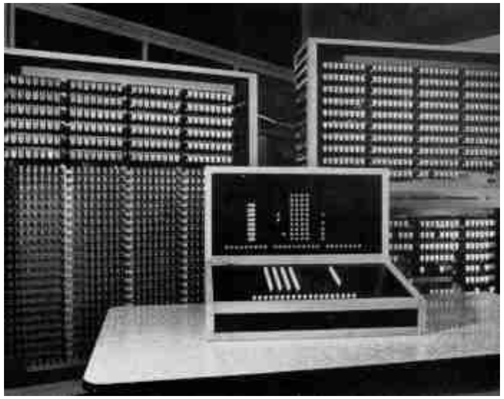
Maschinenmodell

von Neumann Modell

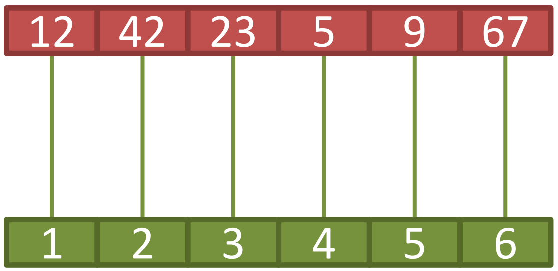


RAM-Modell

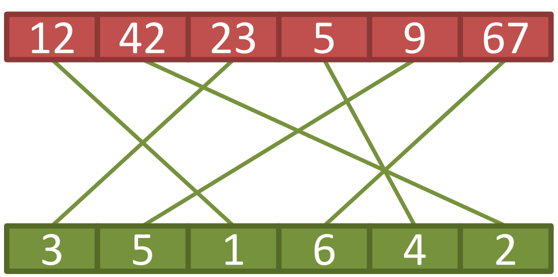
- In jedem Rechenschritt kann jederzeit direkt auf eine beliebige Speicheradresse zugegriffen werden (lesend&schreibend)
- **Früher** tatsächlich ohne „extra“ Wartezeit



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]]
```

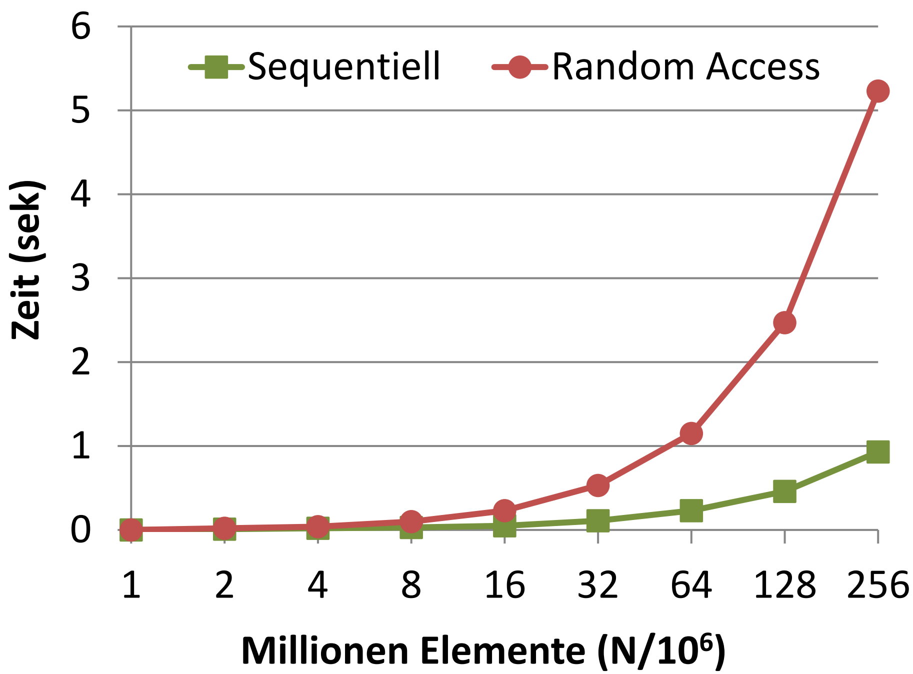
sequenziell vs. random access

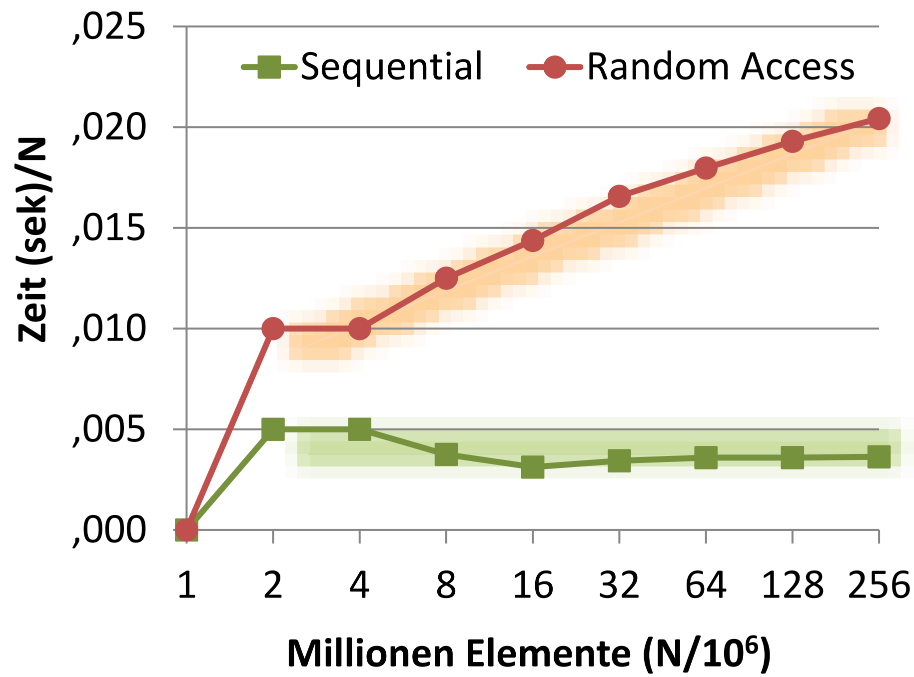
Resultat: ident

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

Oops: Sequentiell VS Random Access

O(*N*) sollte eine waagrechte Linie sein





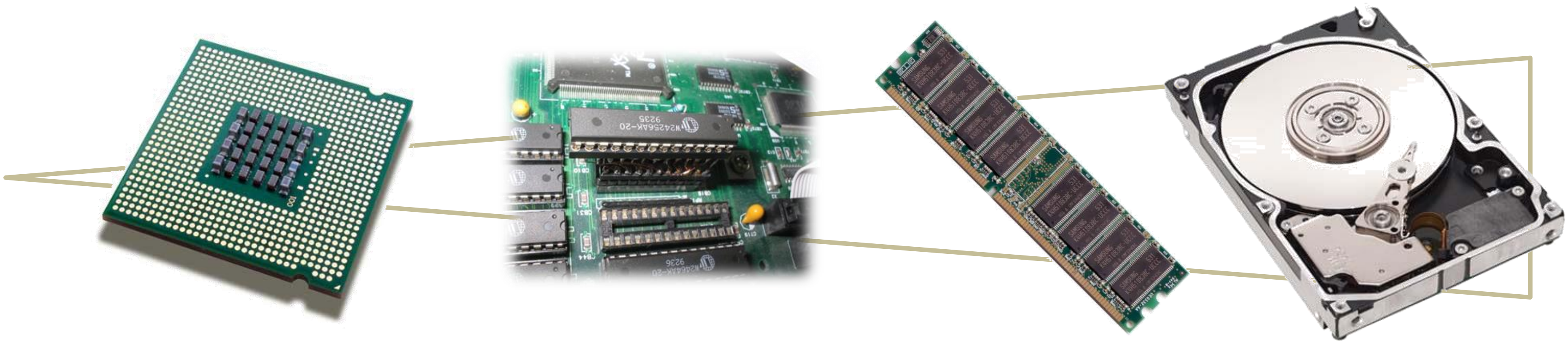
- Intel Core i7 860, 2.80GHz, QuadCore, 8GB RAM
- 1 Core, 32bit, g++ 4.4 –O0, Ubuntu 10.10

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



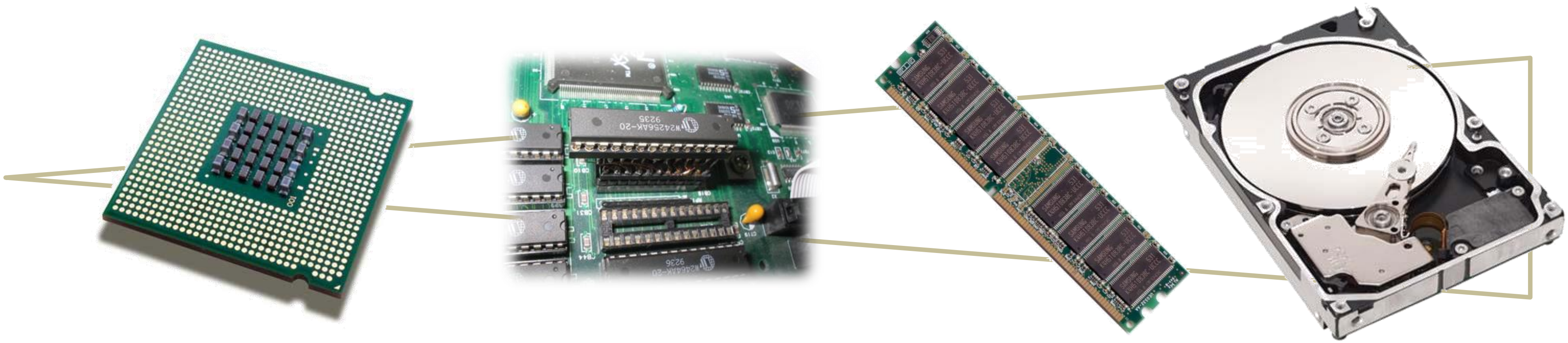
Schnell aber klein

Langsam aber groß

	CPU Register	Cache			RAM	HDD
		Level 1	Level 2	Level 3		
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**

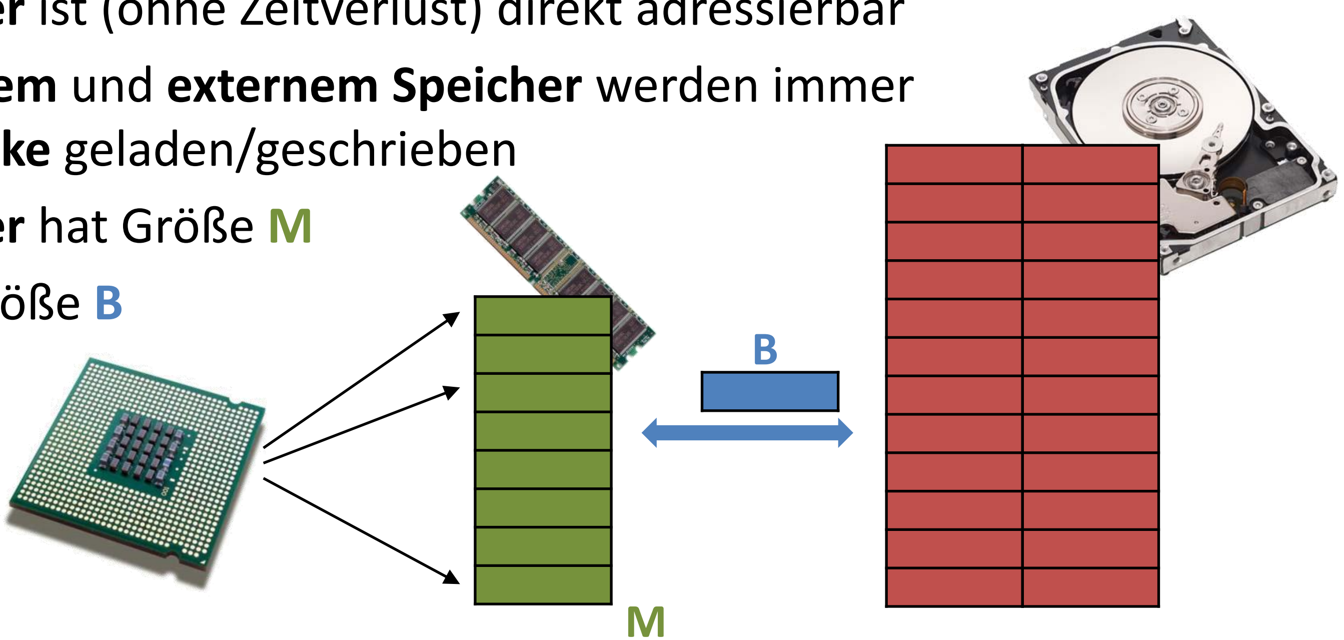
- Jede Operation benötigt gleich viel Zeit (1 Zeiteinheit)
- Jede gewünschte Speicheradresse steht direkt zum Lesen/Schreiben bereit

⇒ 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**

$\Rightarrow$ Zähle # Operationen

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

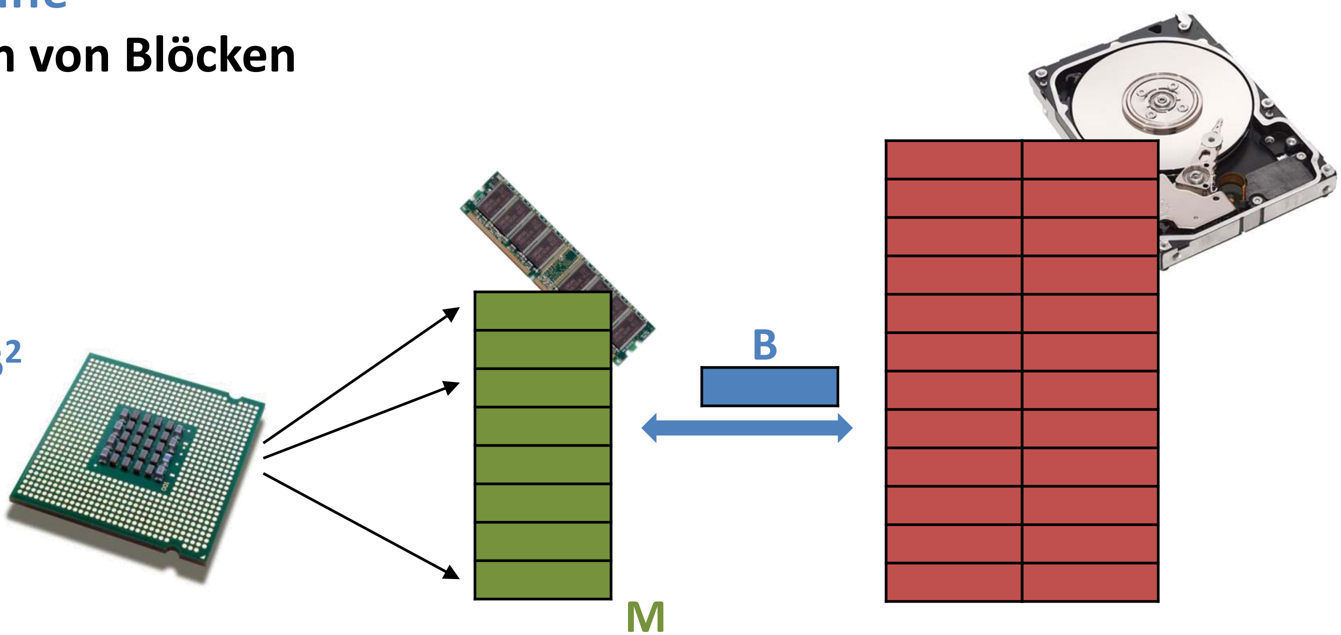
Noch immer vereinfacht, aber guter Tradeoff zw. Realität und Analysierbarkeit

$\Rightarrow$ **Zähle # interne Operationen**
Ziel: Möglichst gleich mit RAM-Modell

$\Rightarrow$ **Zähle # I/O Zugriffe**
Laden/Schreiben von Blöcken

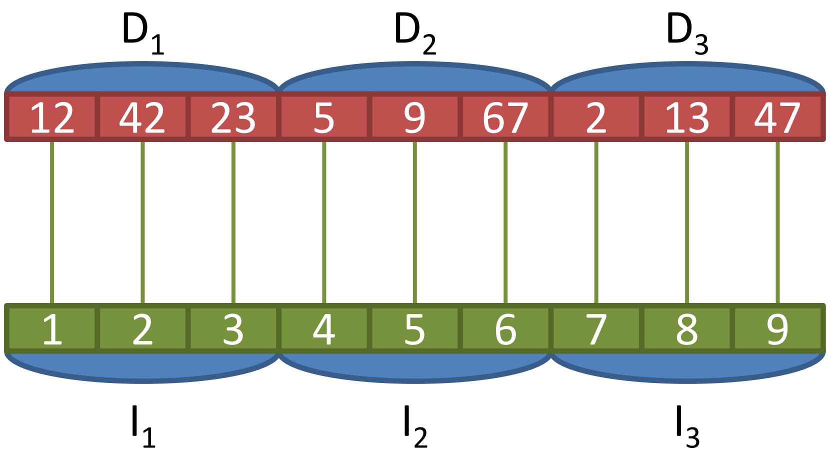
Annahmen

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

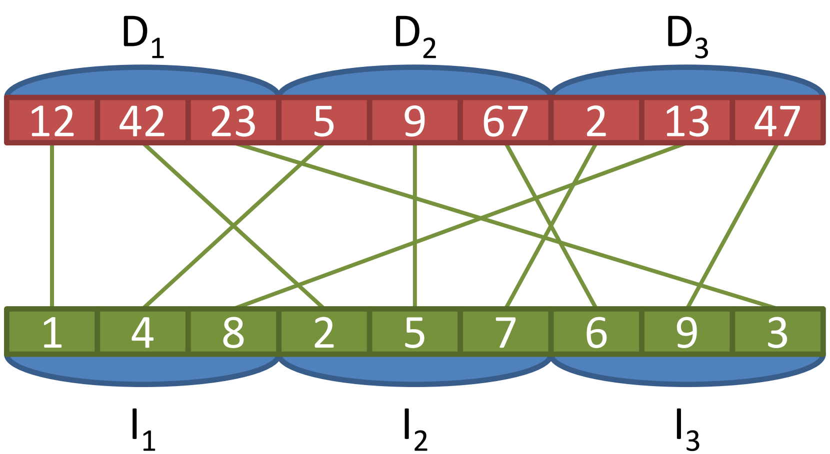


Lokalität

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!

$\Longrightarrow$ **O(1/B)** I/Os pro Operation (amortisiert)

$\Longrightarrow$ Dies ist bestmöglich, da nur B Elemente pro I/O

Externe Datenstrukturen

Weitere einfache Datenstrukturen

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

Komplizierterere Datenstrukturen

- **Priority Queue?** → nächste Woche

Zunächst

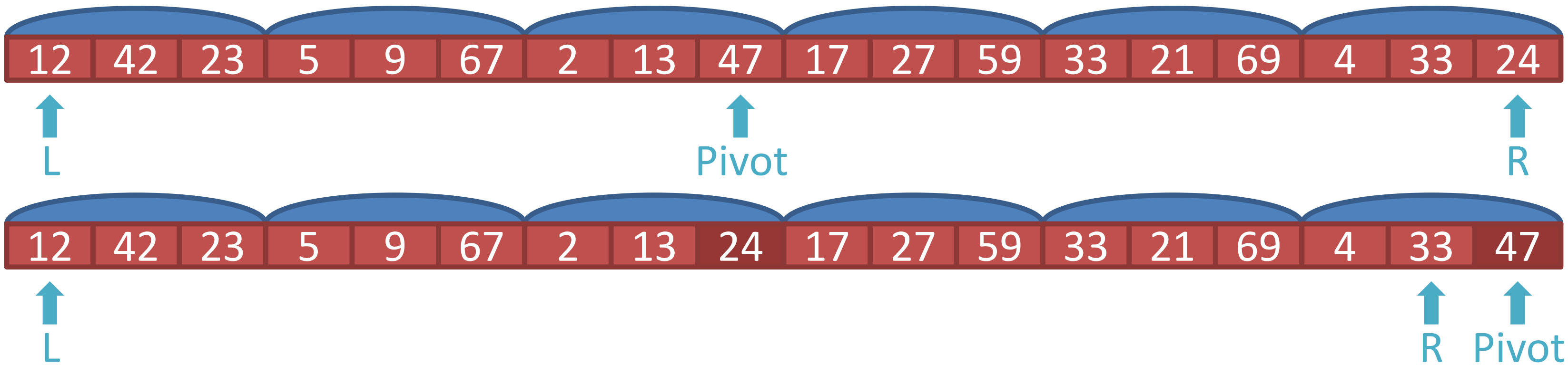
Einfache Algorithmen

- **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 *)* $\rightarrow$ ersparbar wenn man gleich letztes Element wählt

load_block_of(L *), load_block_of(* R *)*

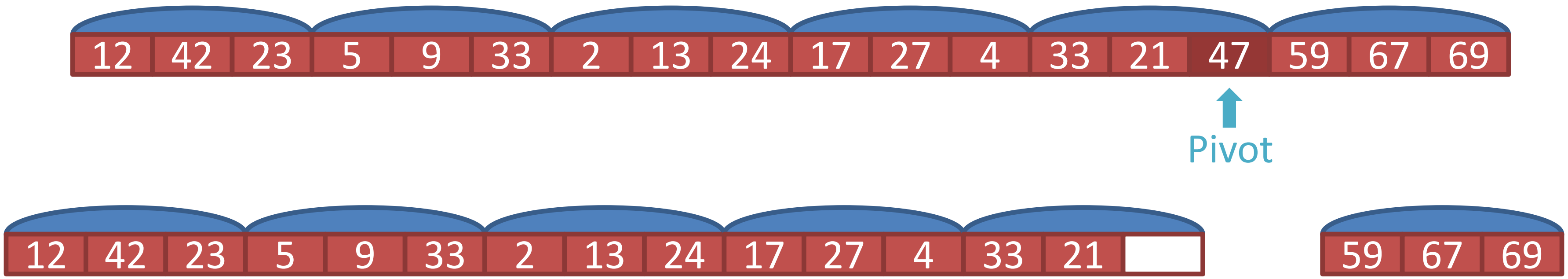
Laufe mit L nach rechts, mit R nach links:

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

Partitionieren von N_0 Elementen: $O(N_0/B)$ I/Os

$\rightarrow$ 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: $O(N)$

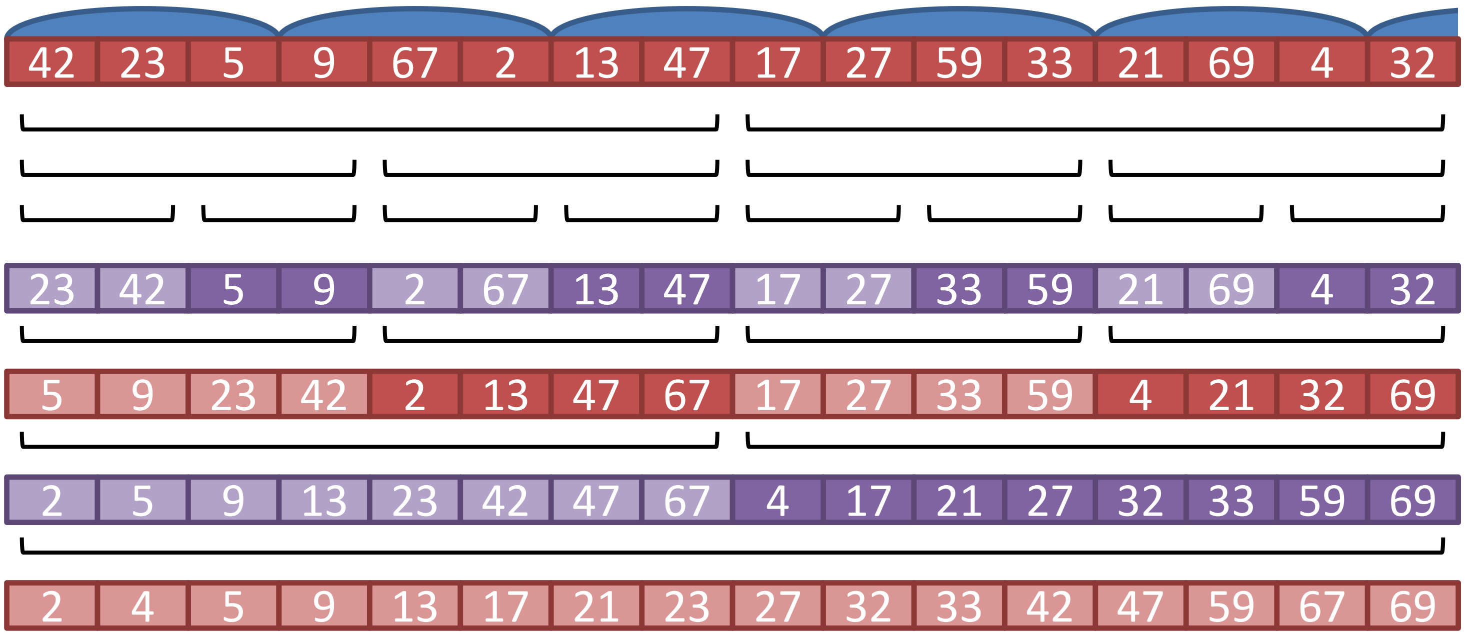
Rekursionstiefe solange I/Os benötigt werden (Analyse wie traditionell):

- 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: $O(N/B + (N/M+N/B) \log_2 (N/M))$ $\quad (N/M < N/B)$
  $= O((N/B) \log_2 (N/M))$

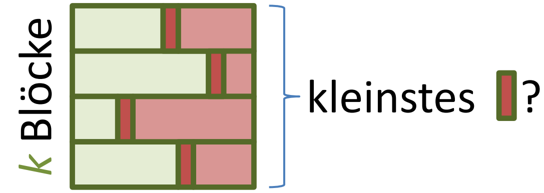
Externes Merge-Sort

Beschleunige Merge-Sort (2)

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

- $k = \frac{1}{2}\,(M/B)$ → 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



$k = ½ M/B$

Verschmelzen von *k* Runs: Finde Minimum

Naïv: lineare Suche, $O(k)$

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

Priority Queue

- Kleinstes Element pro Block in eine Priority-Queue (zB. Min-Heap) (Größe: $k = ½ M/B$, 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: $O(N \log_2 M + N \log_2 (M/B) \log_{M/B} (N/M)) = O(N \log_2 N)$

→ 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	$O(N \log_2 N)$	$\mathbf{O((N/B) \log_{M/B} (N/M))}$

I/O Aufwand fundamentaler Operationen

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

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

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

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

Externspeicheralgorithmen II

Priority-Queue – Externer Array-Heap

Details in:

[Andreas Crauser. *LEDA-SM: External Memory Algorithms and Data Structures in Theory and Practice.* Dissertation, Saarbrücken, 2001]

Wiederholung: I/O-Modell

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

⇒ Zähle # Operationen

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

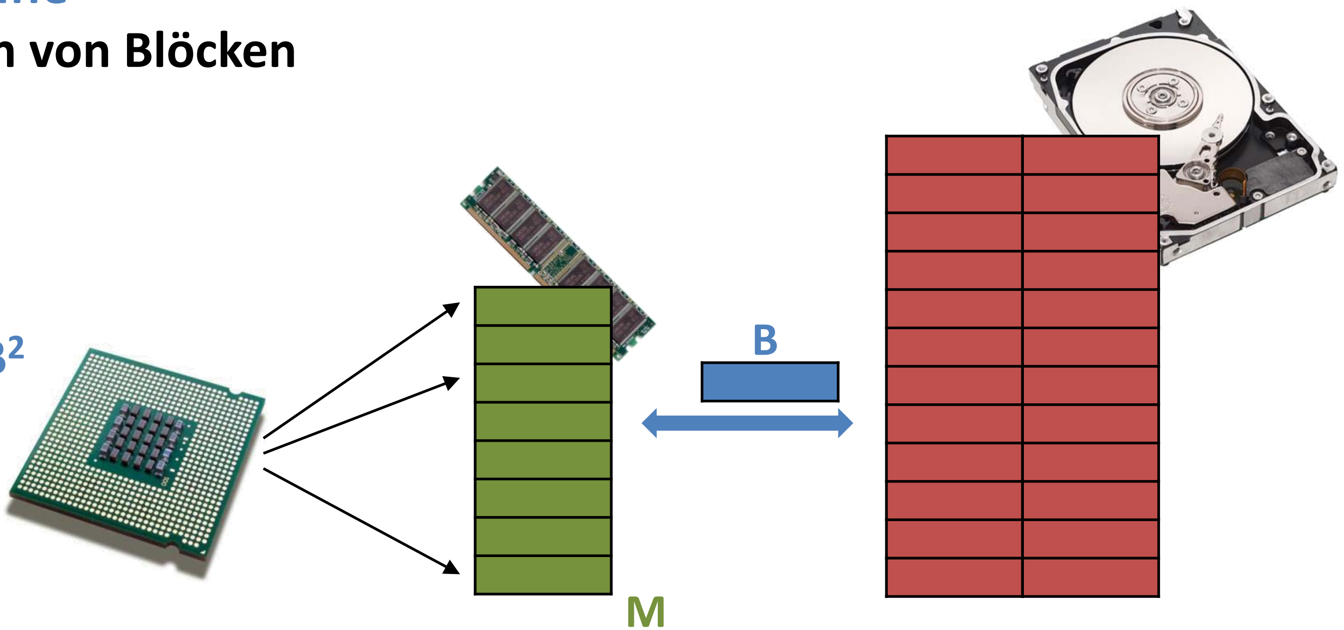
Noch immer vereinfacht, aber guter Tradeoff zw. Realität und Analysierbarkeit

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

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

Annahmen

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



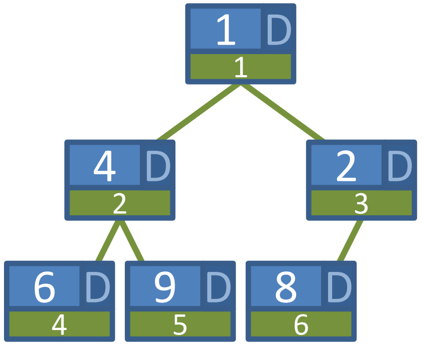
Priority Queue

Priority Queue (PQ)

Operationen

- **insert(key,data)**
  Füge Key/Data-Paar in die PQ ein.
- **decreaseKey(entry, newKey)**
  Vermindere den Schlüssel eines gegebenen Key/Data-Paares in der PQ
- (key,data) **deleteMinimum()**
  Liefere und entferne das Key/Data-Paar mit kleinstem Schlüssel

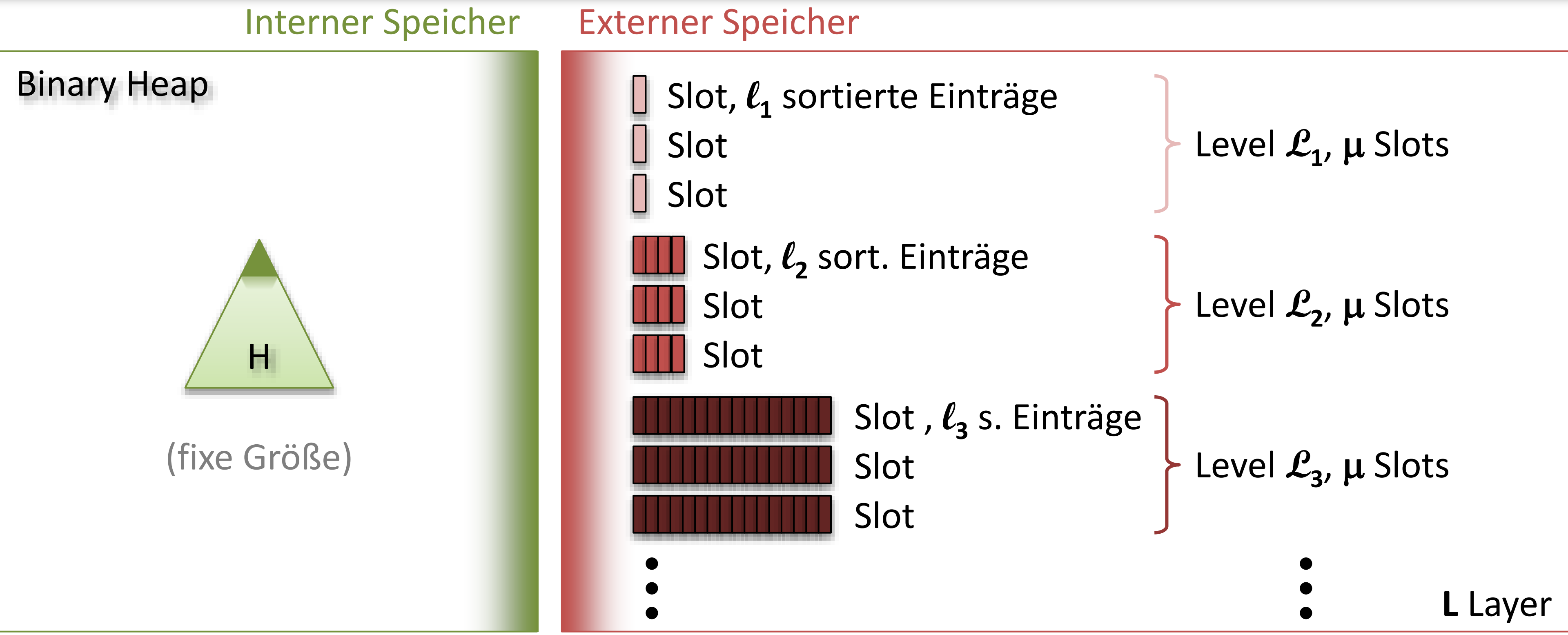
Klassischste Implementierung (intern):
Binary Heap



Problem
Jeder Sprung in eine benachbarte Schicht benötigt (womöglich) einen I/O Zugriff!

→ Externer Array-Heap
als Verbesserung des klassischen Binary Heaps für Externspeicher

Priority Queue: Externer Array-Heap



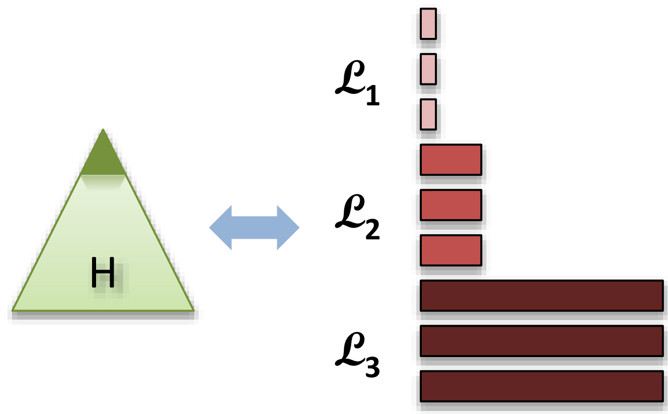
Wähle Konstante **c** < **1**. (Typischer Wert: $^1/_7$)

M/B = max. Anzahl der Blöcke im internen Speicher $\rightarrow \alpha := \mathbf{c} \cdot M/B \in \mathbb{N}$

$\ell_i := B \cdot \alpha^i$

$\mu := \alpha - 1$ $\quad \Rightarrow \quad \ell_1 = \mathbf{c} \cdot M \quad \Rightarrow \quad \ell_{i+1} = \ell_i \cdot (\mu+1)$

insert(key, data)



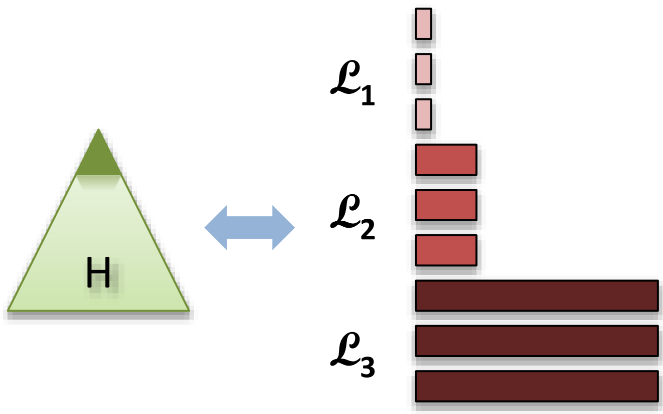
$\mu = \alpha - 1$ Slots pro Level

Slotgröße:
$\ell_1 = \mathbf{c} \cdot M$
$\ell_{i+1} = \ell_i \cdot (\mu+1)$

Operation **insert**:

- Füge neues Element in internen Heap **H** ein
- Falls kein Platz in **H**:
  - Verschiebe ℓ_1 Einträge **S** in den externen Speicher
  - Falls ein Slot in $\mathcal{L}_1$ frei: Lege **S** dort ab
  - Sonst: **S** = Overflow-Folge
    $\rightarrow$ Fasse **S** mit alle Listen in den Slots von $\mathcal{L}_1$ zusammen
    $\rightarrow$ Gesamtliste hat $\leq \ell_1 \cdot (\mu+1)$ Einträge = Größe eines Slots in $\mathcal{L}_2$
    $\rightarrow$ Falls ein Slot in $\mathcal{L}_2$ frei ist: Lege Gesamtliste dort ab
    $\rightarrow$ Sonst: wiederhole Vorgehen für $\mathcal{L}_3$,…

deleteMinimum()



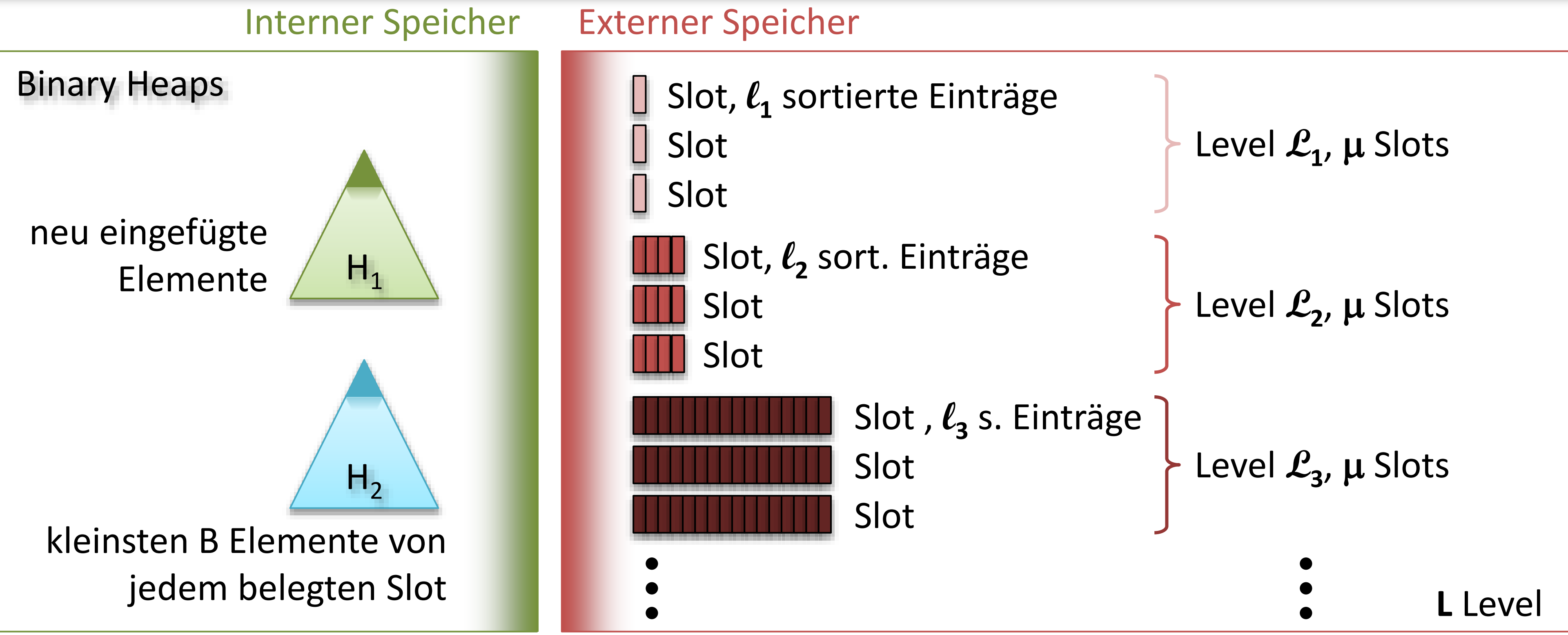
$\mu = \alpha - 1$ Slots pro Level

Slotgröße:
$\ell_1 = \mathbf{c} \cdot M$
$\ell_{i+1} = \ell_i \cdot (\mu+1)$

Operation **deleteMinimum**:

- **oops… schwer…**
- **Invariante um das Minimum schnell zu finden:**
  Das kleinste Element liegt immer im internen Speicher (Heap **H**)
- **Also:** nach deleteMinimum ggf. **H** wieder auffüllen
- **Dazu:** Benutze 2 Heaps $\mathbf{H_1}$, $\mathbf{H_2}$ statt einem Heap **H**.

Priority Queue: Externer Array-Heap



- $|H_1| = 2\mathbf{c}M$
- $|H_2| = B \cdot \mathbf{L}\mu = B \cdot \mathbf{L} \cdot (\mathbf{c} \cdot M/B - 1) \leq \mathbf{L} \cdot \mathbf{c}M$
- Zusätzlich: Mergen von μ Slots + Overflow-Folge $\rightarrow B\alpha = \mathbf{c}M$

$\Rightarrow$ Interner Speicher: $(3+\mathbf{L})\mathbf{c}M$

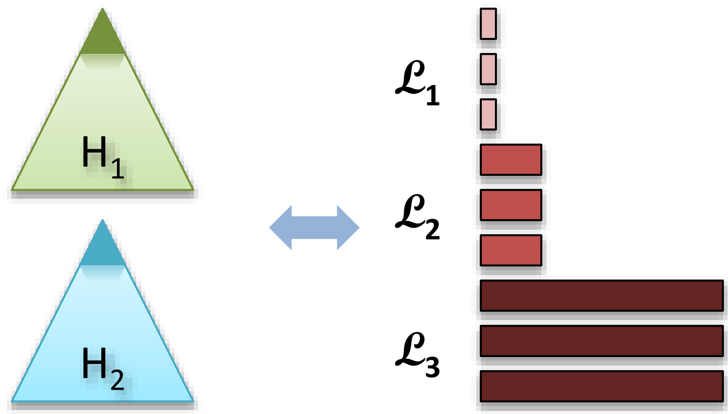
$\Rightarrow (3+\mathbf{L})\mathbf{c} \leq 1$

Praxis: $\mathbf{c} = {}^1/_7$, $\mathbf{L} = 4$ ✓

$\alpha := \mathbf{c} \cdot M/B$
$\ell_i := B \cdot \alpha^i$
$\mu := \alpha - 1$
$\ell_{i+1} = \ell_i \cdot (\mu+1)$

$\rightarrow$ Wieviele Daten können maximal verwaltet werden? $\rightarrow$ später…

Hilfsoperationen (1)



$\mu = \alpha - 1$ Slots pro Level

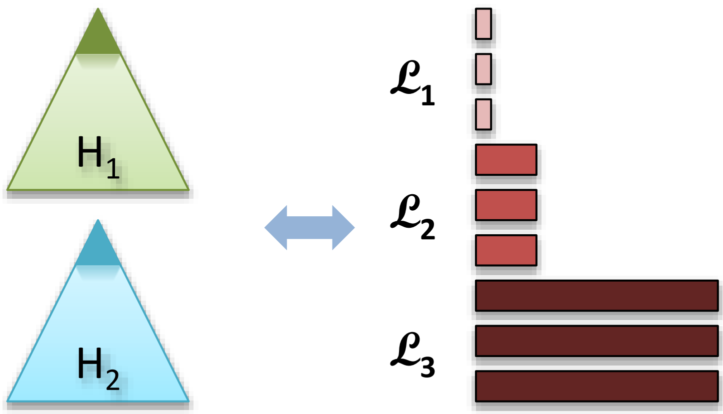
Slotgröße:
$\ell_1 = c \cdot M$
$\ell_{i+1} = \ell_i \cdot (\mu+1)$

Merge(i,S,S‘) $O(\ell_{i+1}/B)$ I/Os
Verschmelze alle Slots aus $\mathcal{L}_i$ (inkl. deren Blöcke in H_2) und die Folge **S** ($|S| \le \ell_i$) zu einer neuen Folge **S‘**.

Store(i,S) $O(\ell_i/B)$ I/Os
Voraussetzung: $\mathcal{L}_i$ hat einen freien Slot. Speichere die Folge **S** in einen freien Slot von $\mathcal{L}_i$, und verschiebe die kleinsten B Elemente nach H_2.

Load(i,j) $O(1)$ I/Os
Lade die (nächsten) B kleinsten Elemente aus Slot **j** von $\mathcal{L}_i$ nach H_2.

Hilfsoperationen (2)



$\mu = \alpha - 1$ Slots pro Level

Slotgröße:
$\ell_1 = \mathbf{c} \cdot M$
$\ell_{i+1} = \ell_i \cdot (\mu + 1)$

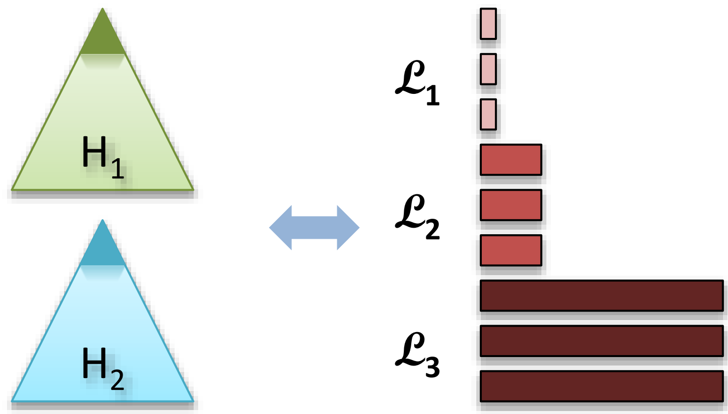
Compact(i) **$O(\ell_i / B)$ I/Os**

Immer ausführen falls: es existieren mindestens zwei Slots von $\mathcal{L}_i$ die **in Summe** (inkl. ihrer Blöcke in H_2) maximal ℓ_i Elemente enthalten.

Verschmelze diese Slots (inkl. ihrer Blöcke in H_2) und verschiebe den Minimum-Block der neuen Liste nach H_2.

Dadurch wird mindestens ein Slot frei.

insert(key, data)



$\mu = \alpha - 1$ Slots pro Level

Slotgröße:

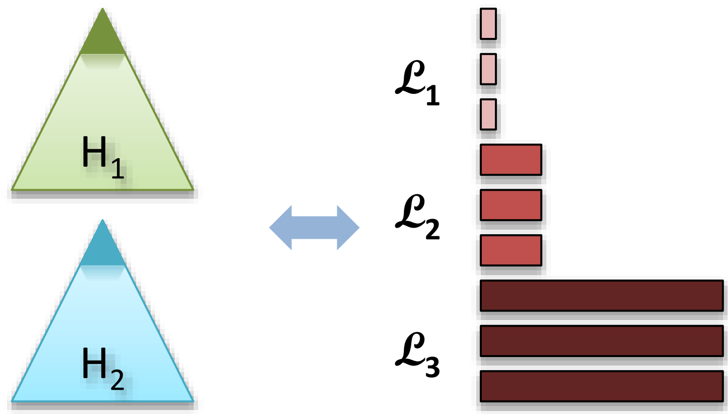
$\ell_1 = \mathbf{c} \cdot M$

$\ell_{i+1} = \ell_i \cdot (\mu + 1)$

Operation **insert**:

- Füge neues Element in internen Heap $\mathbf{H_1}$ ein
- Falls kein Platz in $\mathbf{H_1}$:
  - Verschiebe $\boldsymbol{\ell_1}$ Einträge **S** in den externen Speicher
  - Falls ein Slot in $\mathcal{L}_1$ frei: **store(1,S)**
  - Sonst: (alle bis auf max. 1 Slot aus $\mathcal{L}_1$ haben mind. $\ell_1/2$ Elemente)
    $\rightarrow$ **merge(1,S,S‘)**
    $\rightarrow$ Gesamtliste hat $\leq \ell_1 \cdot (\mu+1)$ Einträge = Größe eines Slots in $\mathcal{L}_2$
    $\rightarrow$ Falls ein Slot in $\mathcal{L}_2$ frei ist: **store(2,S‘)**
    $\rightarrow$ Sonst: merge $\mathcal{L}_2$,… etc. …

deleteMinimum()



$\mu = \alpha - 1$ Slots pro Level

Slotgröße:

$\ell_1 = \mathbf{c} \cdot M$

$\ell_{i+1} = \ell_i \cdot (\mu + 1)$

Operation **deleteMinimum**:

- Entferne kleinstes Element **x** aus $\mathbf{H_1}$ bzw. $\mathbf{H_2}$.
- Falls **x** aus $\mathbf{H_2}$ kam:
  - Sei $\mathcal{L}_i$ das Level, und **j** der Slot in diesem Level, aus dem **x** kam.
  - Falls **x** das letzte Element aus dem Minimum-Block von Slot j war:
    - Lade Daten nach, **load(i,j)**,
    - Rufe **compact(i)** nach Bedarf auf.

Analyse

Theorie

- Anzahl I/Os?
- Speicherplatz-Bedarf?
  Speicherplatz-Beschränkung?

Praxis

- Bringt‘s was? Wieviel?