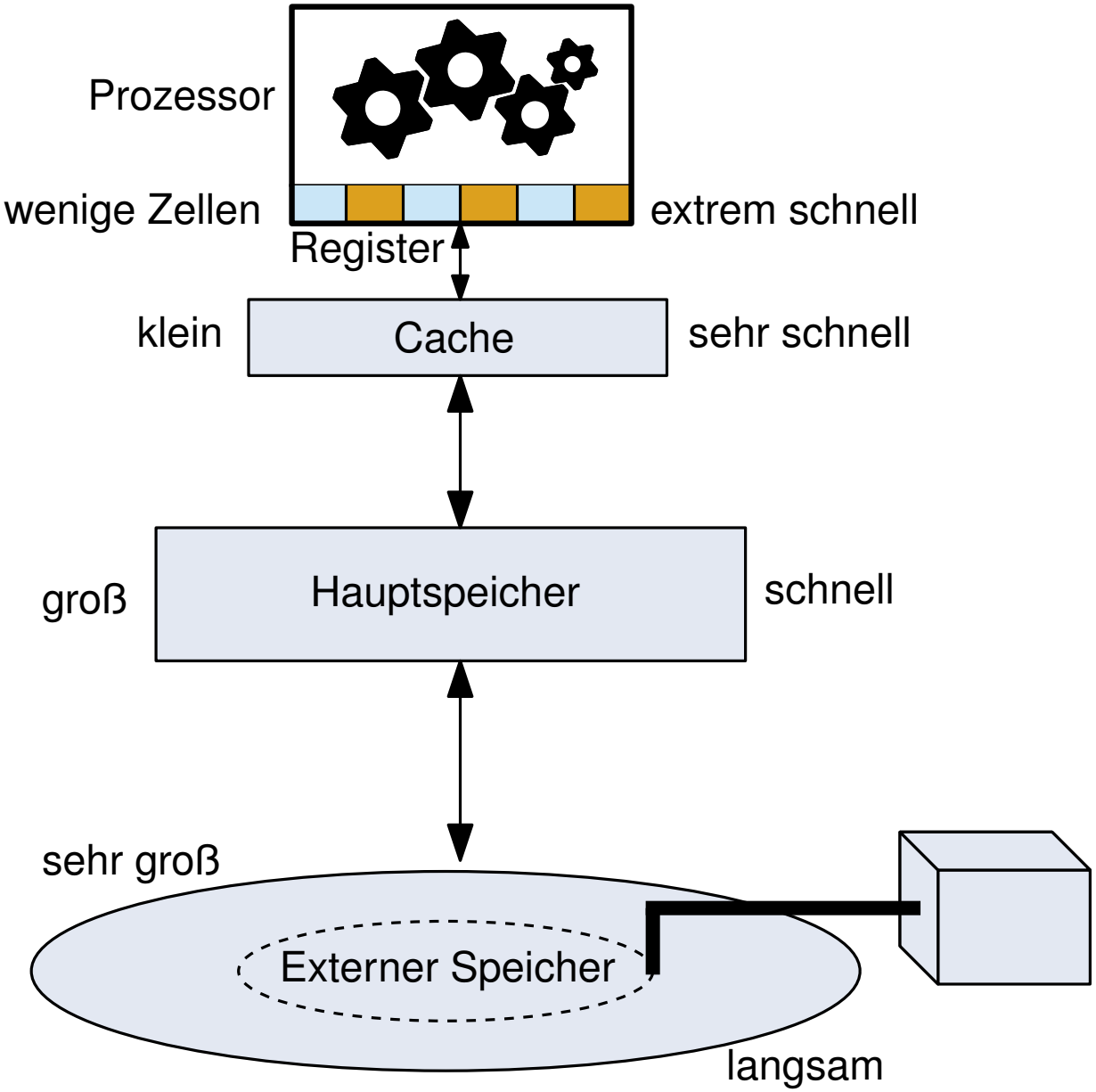
Karlsruher Institut für Technologie

Algorithmen II
Vorlesung am 31.01.2013

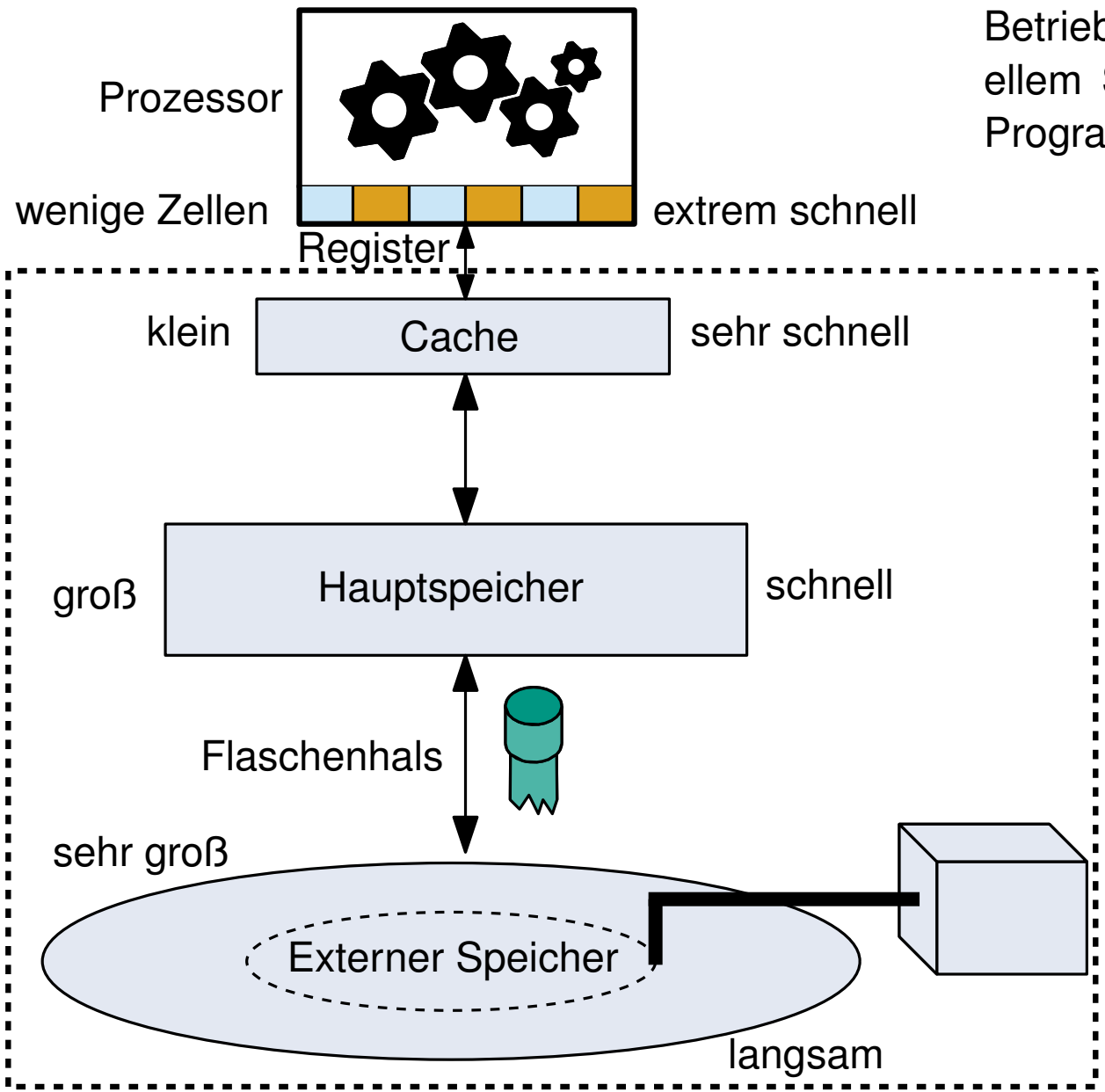
Algorithmen für externen Speicher

INSTITUT FÜR THEORETISCHE INFORMATIK · PROF. DR. DOROTHEA WAGNER

Einfaches Rechnermodell



Einfaches Rechnermodell



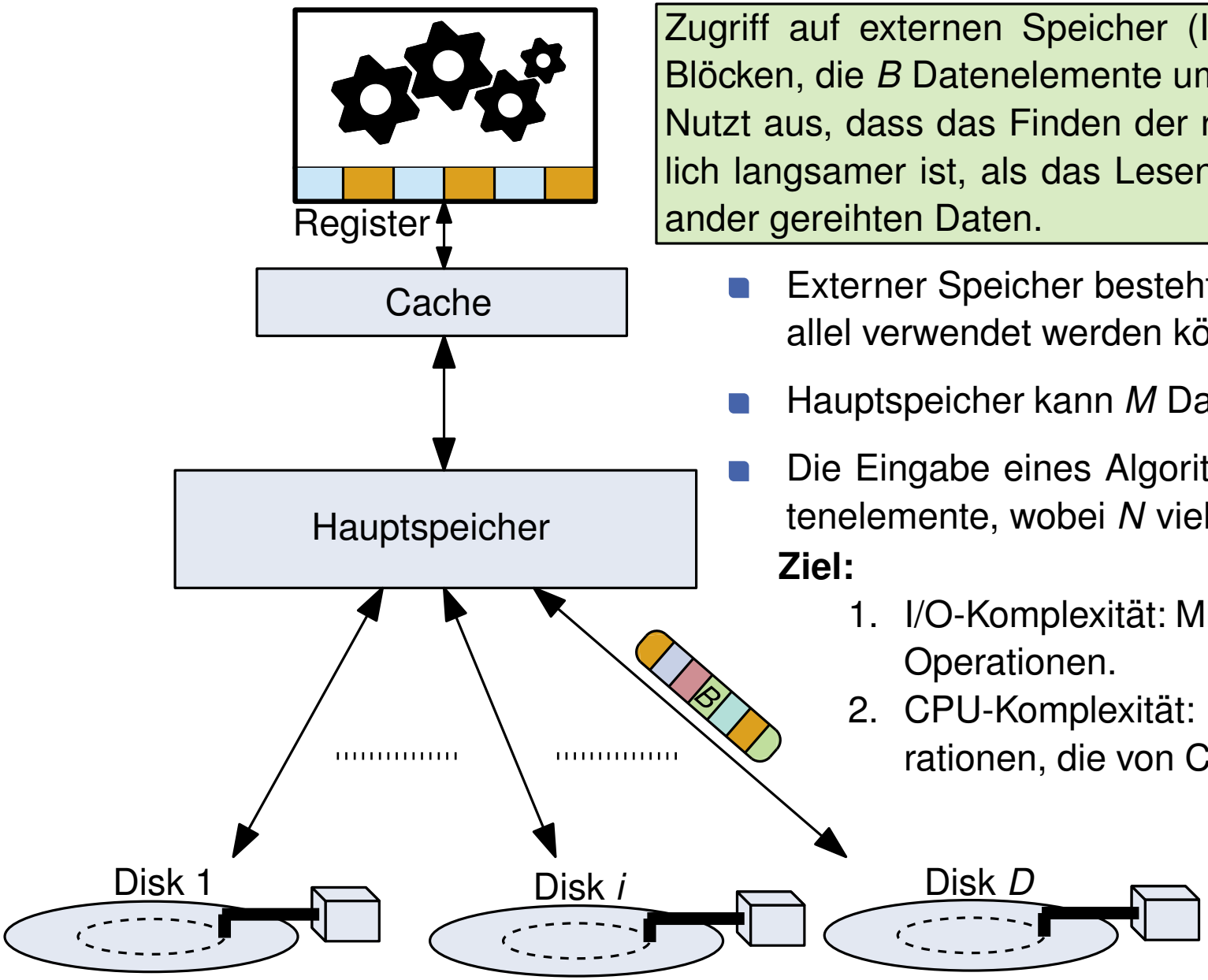
Betriebsystem bildet mithilfe von virtuellem Speicher eine Abstraktionsebene: Programm *sieht* nur einen Speicher.

Problem: Wenn auf großen Datenmengen gearbeitet wird, dann muss Betriebsystem viele Daten zwischen Hauptspeicher und externem Speicher transportieren, um virtuellen Speicher zu simulieren.

Deshalb: Manchmal besser nicht die Abstraktion zu verwenden.

Institut für Theoretische Informatik
Prof. Dr. Dorothea Wagner

Parallel Disk Modell



Zugriff auf externen Speicher (I/O) passiert immer in Blöcken, die *B* Datenelemente umfassen:
Nutzt aus, dass das Finden der richtigen Position deutlich langsamer ist, als das Lesen/Schreiben von aneinander gereihten Daten.

- Externer Speicher besteht aus *D* Platten, die parallel verwendet werden können.
- Hauptspeicher kann *M* Datenelemente speichern.
- Die Eingabe eines Algorithmus hat Größe *N* Datenelemente, wobei *N* viel größer ist als *M*.

Ziel:

1. I/O-Komplexität: Minimiere Anzahl der I/O-Operationen.
2. CPU-Komplexität: Minimiere Anzahl Operationen, die von CPU ausgeführt werden.

Institut für Theoretische Informatik
Prof. Dr. Dorothea Wagner

Grundprinzipien:

Grundprinzipien, die beim Design von Algorithmen auf externem Speicher eingehalten werden sollten:

- **Interne Effizienz:** Die interne Arbeit, die vom Algorithmus durchgeführt wird, sollte mit der vergleichbar sein, die von den besten Algorithmen mit internem Speicher geleistet wird.
- **Räumliche Lokalität:** Wenn auf ein Block B im externen Speicher zugegriffen wird, dann sollte dieser möglichst viel nützliche Information enthalten.
- **Zeitliche Lokalität:** Sobald Daten im internen Speicher (Hauptspeicher) sind, dann sollte möglichst viel nützliche Arbeit auf diesen Daten ausgeführt werden, bevor sie in den externen Speicher zurückgeschrieben werden.

Zugriff auf externen Speicher ist teuer!

Grundoperationen

Viele Algorithmen mit externem Speicher verwenden die Grundoperationen *Scannen*, und *Sortieren*:

Scannen: Anstatt jedes Datenelement einzeln aus dem externen Speicher zu laden, sollte dies in Blöcken geschehen. Das Laden von N Datenelementen aus dem externen Speicher sollte im optimalen Fall $\texttt{scan}(N) := \Theta(\frac{N}{D \cdot B})$ I/Os kosten.

Sortieren: Externes Sortieren von N Elementen ist mit $\texttt{sort}(N) := \Theta(\frac{N}{D \cdot B} \cdot \log_{\frac{M}{B}} \frac{N}{B})$ I/Os möglich. Sinnvoll, wenn Daten nicht so vorliegen, dass externer Scann möglich ist (siehe spätere Folien).

Notation:

- M: Größe des Hauptspeichers.
- N: Größe der Instanz:
- D: Anzahl der Platten des externen Speichers.
- B: Anzahl Elemente in einem Block.

Interner Stack

Interner Stack (arbeitet nur auf Hauptspeicher), der maximal m Elemente enthält, kann mithilfe eines Arrays S der Größe m und einem Zähler top implementiert werden. Hierzu initialisiere top mit -1.

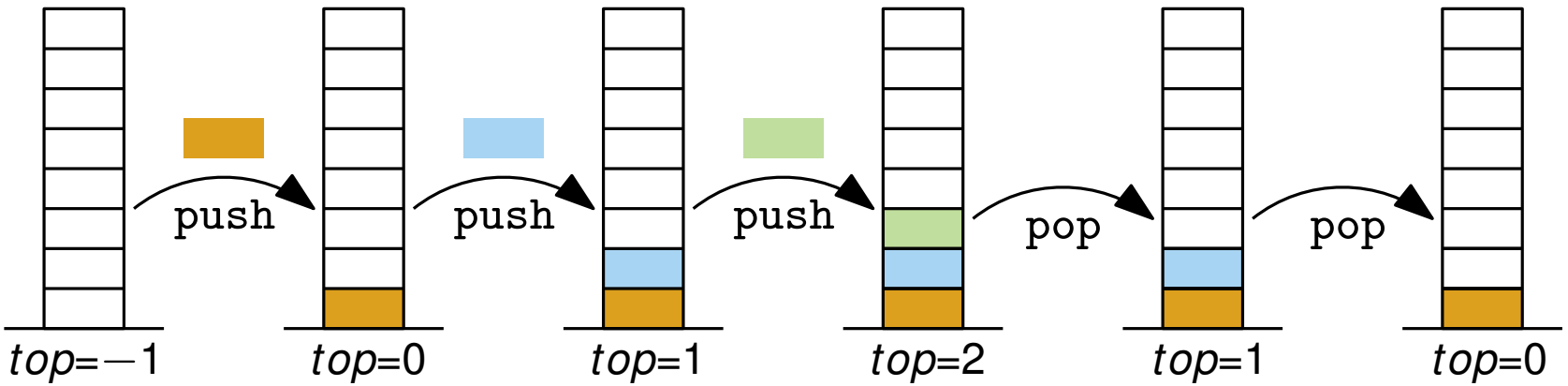
Operation `push`(el):
Effekt: Legt Element el auf den Stack.

wenn $top < m - 1$ **dann**
$top \leftarrow top + 1$
$S[top] \leftarrow el$

Operation `clear`():
Effekt: Löscht alle Elemente vom Stack.
$top \leftarrow -1$

Operation `pop`():
Effekt: Entfernt das oberste Element vom Stack.

wenn *top = -1* **dann return** *nil*
$temp \leftarrow S[top]$
$top \leftarrow top - 1$
return temp



Institut für Theoretische Informatik
Prof. Dr. Dorothea Wagner

Externer Stack

Kann mithilfe eines internen Stacks S der Größe $2 \cdot B$ umgesetzt werden:

Operation `externalPush`(*el*):
Effekt: Legt *el* auf externen Stack.

wenn *S voll* **dann**
Kopiere S in externen Speicher.
S.`clear`().
S.`push`(*el*)

Operation `externalPop`():
Effekt: Entfernt das oberste Element vom externen Stack.

wenn *S leer* **dann**
Kopiere die B zuletzt geschriebenen Elemente aus dem externen Speicher in S.
S.`pop`()

Analyse:

1. I/Os treten nur auf, wenn Puffer im Hauptspeicher leer oder voll ist.
2. Amortisiert ergibt sich damit $O(\frac{1}{B})$ I/Os pro Operation.
3. Nicht besser möglich: Pro I/O können maximal B Elemente gelesen oder geschrieben werden.

Institut für Theoretische Informatik
Prof. Dr. Dorothea Wagner

Externer Stack

Kann mithilfe eines internen Stacks S der Größe $2 \cdot B$ umgesetzt werden:

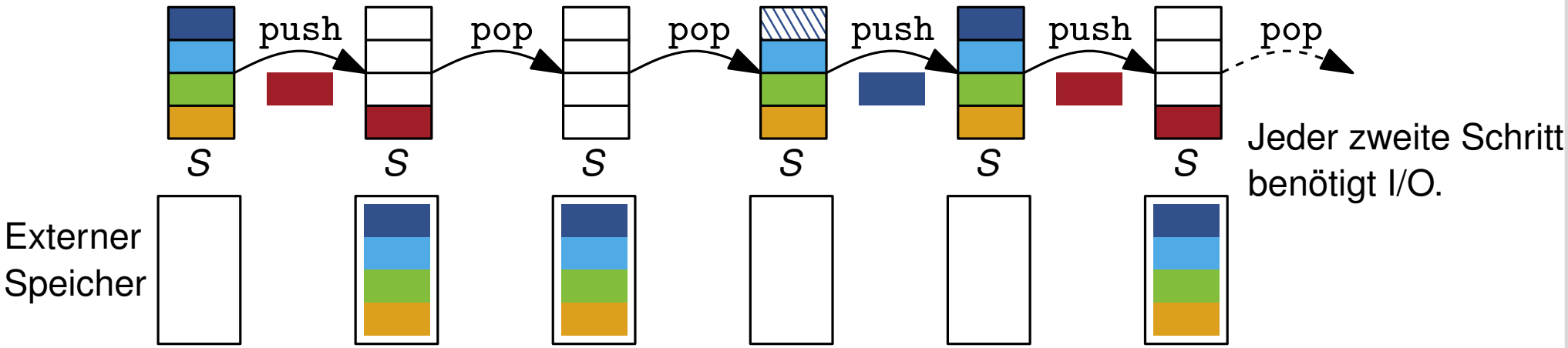
Operation `externalPush`(*el*):
Effekt: Legt *el* auf externen Stack.

wenn *S voll* **dann**
Kopiere S in externen Speicher.
S.`clear`().
S.`push`(*el*)

Operation `externalPop`():
Effekt: Entfernt das oberste Element vom externen Stack.

wenn *S leer* **dann**
Kopiere die B zuletzt geschriebenen Elemente aus dem externen Speicher in S.
S.`pop`()

Warum hat Stack die Größe $2 \cdot B$ und nicht B? Annahme S hat Größe B.



Externe Warteschlange

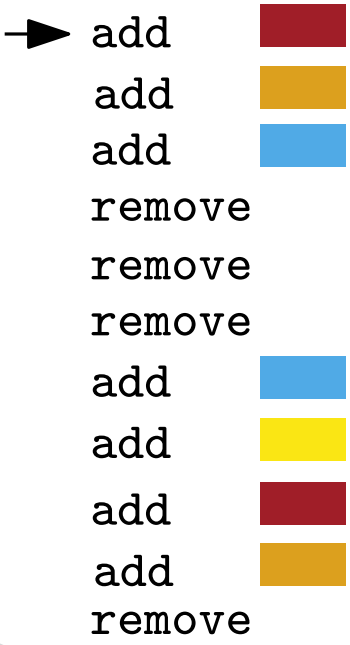
Kann mithilfe von zwei internen Stacks S_1 und S_2 der Größe B umgesetzt werden.
Idee: S_1 ist Schreib-Puffer und S_2 ist Lese-Puffer.

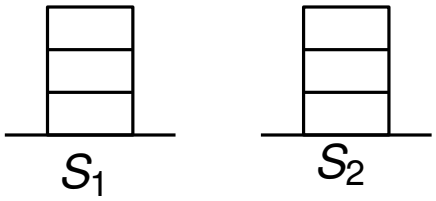
Operation `add`(*el*):
Effekt: Hängt *el* an externe Warteschlange.

wenn S_1 *voll* **dann**
Kopiere S_1 in externen Speicher in verkehrter Reihenfolge.
S_1.`clear`()
S_1.`push`(*el*)

Operation `remove`():
Effekt: Entfernt erstes Element aus externer Warteschlange.

wenn S_2 *leer* **dann**
Kopiere die B zuletzt geschriebenen Elemente aus dem externen Speicher in S_2.
wenn S_2 *immer noch leer* **dann**
Kopiere S_1 in S_2 in verkehrter Reihenfolge.
S_1.*clear*()
S_2.`pop`()





Externer Speicher:

Institut für Theoretische Informatik
Prof. Dr. Dorothea Wagner

Externe Warteschlange

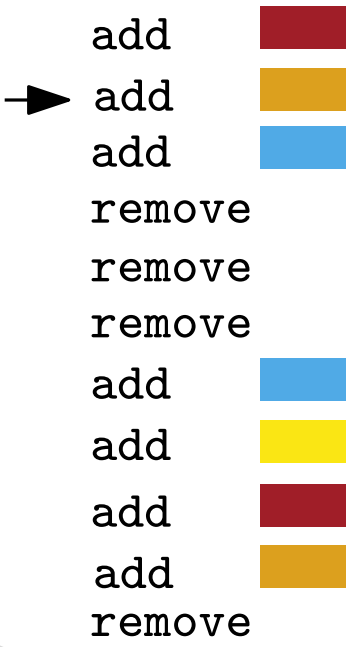
Kann mithilfe von zwei internen Stacks S_1 und S_2 der Größe B umgesetzt werden.
Idee: S_1 ist Schreib-Puffer und S_2 ist Lese-Puffer.

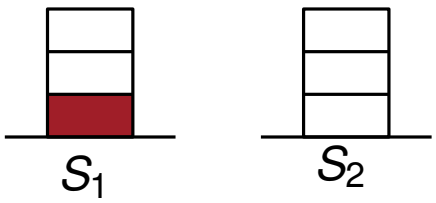
Operation `add`(*el*):
Effekt: Hängt *el* an externe Warteschlange.

wenn S_1 *voll* **dann**
Kopiere S_1 in externen Speicher in verkehrter Reihenfolge.
S_1.`clear`()
S_1.`push`(*el*)

Operation `remove`():
Effekt: Entfernt erstes Element aus externer Warteschlange.

wenn S_2 *leer* **dann**
Kopiere die B zuletzt geschriebenen Elemente aus dem externen Speicher in S_2.
wenn S_2 *immer noch leer* **dann**
Kopiere S_1 in S_2 in verkehrter Reihenfolge.
S_1.*clear*()
S_2.`pop`()





Externer Speicher:

Externe Warteschlange

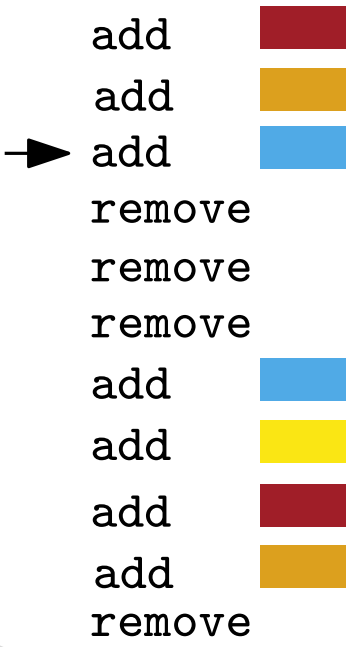
Kann mithilfe von zwei internen Stacks S_1 und S_2 der Größe B umgesetzt werden.
Idee: S_1 ist Schreib-Puffer und S_2 ist Lese-Puffer.

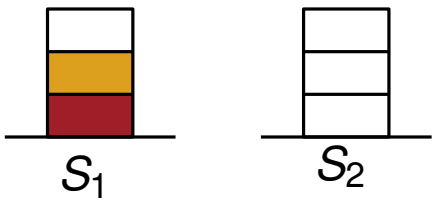
Operation `add`(*el*):
Effekt: Hängt *el* an externe Warteschlange.

wenn S_1 *voll* **dann**
Kopiere S_1 in externen Speicher in verkehrter Reihenfolge.
S_1.`clear`()
S_1.`push`(*el*)

Operation `remove`():
Effekt: Entfernt erstes Element aus externer Warteschlange.

wenn S_2 *leer* **dann**
Kopiere die B zuletzt geschriebenen Elemente aus dem externen Speicher in S_2.
wenn S_2 *immer noch leer* **dann**
Kopiere S_1 in S_2 in verkehrter Reihenfolge.
S_1.*clear*()
S_2.`pop`()





Externer Speicher:

Institut für Theoretische Informatik
Prof. Dr. Dorothea Wagner

Externe Warteschlange

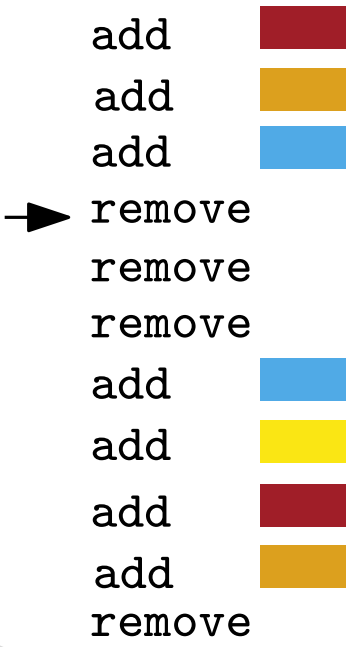
Kann mithilfe von zwei internen Stacks S_1 und S_2 der Größe B umgesetzt werden.
Idee: S_1 ist Schreib-Puffer und S_2 ist Lese-Puffer.

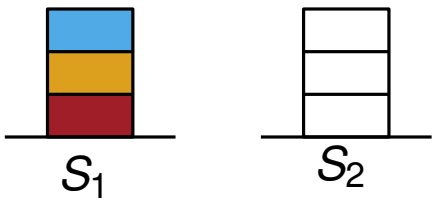
Operation `add`(*el*):
Effekt: Hängt *el* an externe Warteschlange.

wenn S_1 *voll* **dann**
Kopiere S_1 in externen Speicher in verkehrter Reihenfolge.
S_1.`clear`()
S_1.`push`(*el*)

Operation `remove`():
Effekt: Entfernt erstes Element aus externer Warteschlange.

wenn S_2 *leer* **dann**
Kopiere die B zuletzt geschriebenen Elemente aus dem externen Speicher in S_2.
wenn S_2 *immer noch leer* **dann**
Kopiere S_1 in S_2 in verkehrter Reihenfolge.
S_1.*clear*()
S_2.`pop`()





Externer Speicher:

Institut für Theoretische Informatik
Prof. Dr. Dorothea Wagner

Externe Warteschlange

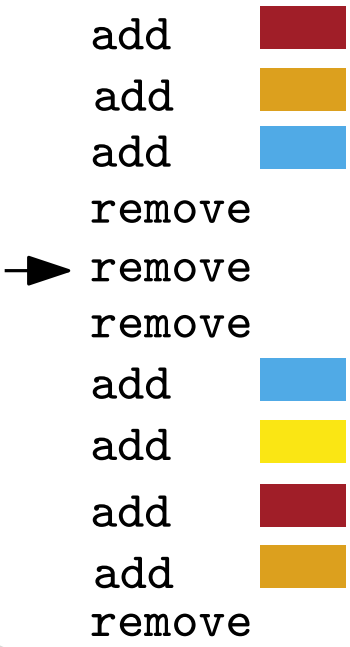
Kann mithilfe von zwei internen Stacks S_1 und S_2 der Größe B umgesetzt werden.
Idee: S_1 ist Schreib-Puffer und S_2 ist Lese-Puffer.

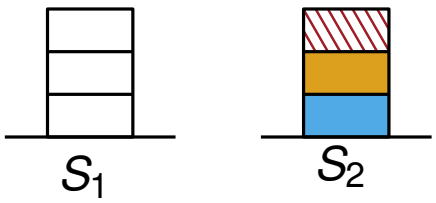
Operation `add`(*el*):
Effekt: Hängt *el* an externe Warteschlange.

wenn S_1 *voll* **dann**
Kopiere S_1 in externen Speicher in verkehrter Reihenfolge.
S_1.`clear`()
S_1.`push`(*el*)

Operation `remove`():
Effekt: Entfernt erstes Element aus externer Warteschlange.

wenn S_2 *leer* **dann**
Kopiere die B zuletzt geschriebenen Elemente aus dem externen Speicher in S_2.
wenn S_2 *immer noch leer* **dann**
Kopiere S_1 in S_2 in verkehrter Reihenfolge.
S_1.*clear*()
S_2.`pop`()





Externer Speicher:

Externe Warteschlange

Kann mithilfe von zwei internen Stacks S_1 und S_2 der Größe B umgesetzt werden.
Idee: S_1 ist Schreib-Puffer und S_2 ist Lese-Puffer.

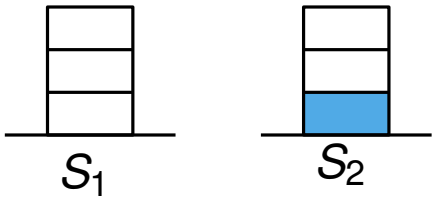
Operation `add`(*el*):
Effekt: Hängt *el* an externe Warteschlange.

wenn S_1 *voll* **dann**
Kopiere S_1 in externen Speicher in verkehrter Reihenfolge.
S_1.`clear`()
S_1.`push`(*el*)

Operation `remove`():
Effekt: Entfernt erstes Element aus externer Warteschlange.

wenn S_2 *leer* **dann**
Kopiere die B zuletzt geschriebenen Elemente aus dem externen Speicher in S_2.
wenn S_2 *immer noch leer* **dann**
Kopiere S_1 in S_2 in verkehrter Reihenfolge.
S_1.*clear*()
S_2.`pop`()

`add`
`add`
`add`
`remove`
`remove`
→ `remove`
`add`
`add`
`add`
`add`
`remove`



Externer Speicher:

Institut für Theoretische Informatik
Prof. Dr. Dorothea Wagner

Externe Warteschlange

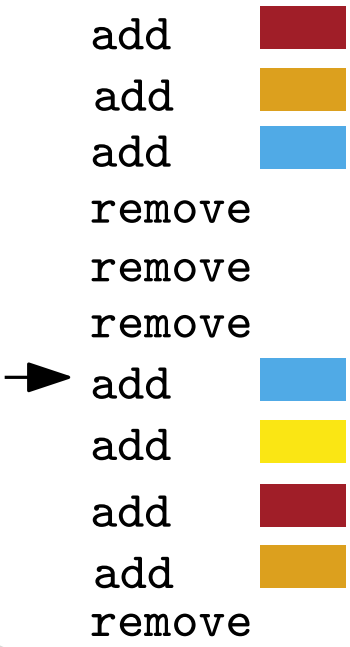
Kann mithilfe von zwei internen Stacks S_1 und S_2 der Größe B umgesetzt werden.
Idee: S_1 ist Schreib-Puffer und S_2 ist Lese-Puffer.

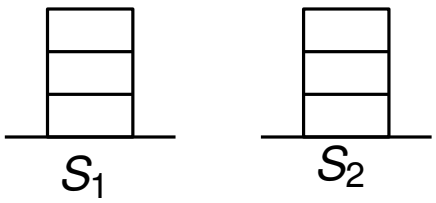
Operation `add`(*el*):
Effekt: Hängt *el* an externe Warteschlange.

wenn S_1 *voll* **dann**
Kopiere S_1 in externen Speicher in verkehrter Reihenfolge.
S_1.`clear`()
S_1.`push`(*el*)

Operation `remove`():
Effekt: Entfernt erstes Element aus externer Warteschlange.

wenn S_2 *leer* **dann**
Kopiere die B zuletzt geschriebenen Elemente aus dem externen Speicher in S_2.
wenn S_2 *immer noch leer* **dann**
Kopiere S_1 in S_2 in verkehrter Reihenfolge.
S_1.*clear*()
S_2.`pop`()





Externer Speicher:

Externe Warteschlange

Kann mithilfe von zwei internen Stacks S_1 und S_2 der Größe B umgesetzt werden.
Idee: S_1 ist Schreib-Puffer und S_2 ist Lese-Puffer.

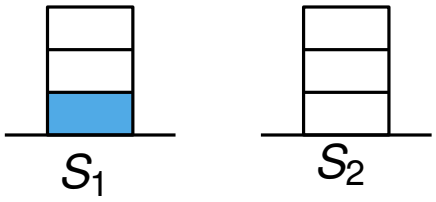
Operation `add`(*el*):
Effekt: Hängt *el* an externe Warteschlange.

wenn S_1 *voll* **dann**
Kopiere S_1 in externen Speicher in verkehrter Reihenfolge.
S_1.`clear`()
S_1.`push`(*el*)

Operation `remove`():
Effekt: Entfernt erstes Element aus externer Warteschlange.

wenn S_2 *leer* **dann**
Kopiere die B zuletzt geschriebenen Elemente aus dem externen Speicher in S_2.
wenn S_2 *immer noch leer* **dann**
Kopiere S_1 in S_2 in verkehrter Reihenfolge.
S_1.*clear*()
S_2.`pop`()

`add`
`add`
`add`
`remove`
`remove`
`remove`
`add`
`add`
`add`
`add`
`remove`



Externer Speicher:

Externe Warteschlange

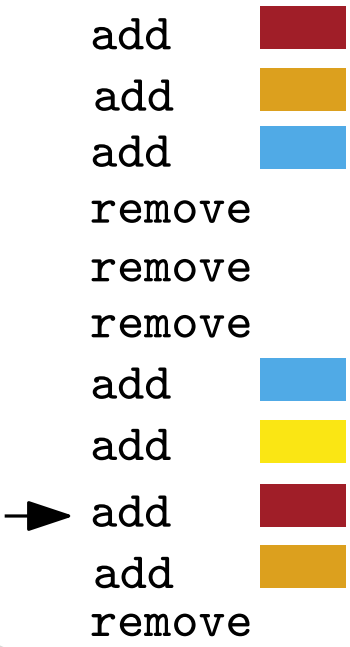
Kann mithilfe von zwei internen Stacks S_1 und S_2 der Größe B umgesetzt werden.
Idee: S_1 ist Schreib-Puffer und S_2 ist Lese-Puffer.

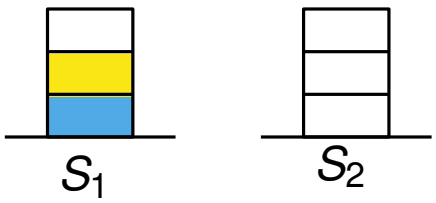
Operation `add`(*el*):
Effekt: Hängt *el* an externe Warteschlange.

wenn S_1 *voll* **dann**
Kopiere S_1 in externen Speicher in verkehrter Reihenfolge.
S_1.`clear`()
S_1.`push`(*el*)

Operation `remove`():
Effekt: Entfernt erstes Element aus externer Warteschlange.

wenn S_2 *leer* **dann**
Kopiere die B zuletzt geschriebenen Elemente aus dem externen Speicher in S_2.
wenn S_2 *immer noch leer* **dann**
Kopiere S_1 in S_2 in verkehrter Reihenfolge.
S_1.*clear*()
S_2.`pop`()





Externer Speicher:

Externe Warteschlange

Kann mithilfe von zwei internen Stacks S_1 und S_2 der Größe B umgesetzt werden.
Idee: S_1 ist Schreib-Puffer und S_2 ist Lese-Puffer.

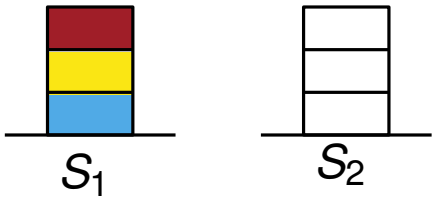
Operation `add`(*el*):
Effekt: Hängt *el* an externe Warteschlange.

wenn S_1 *voll* **dann**
Kopiere S_1 in externen Speicher in verkehrter Reihenfolge.
S_1.`clear`()
S_1.`push`(*el*)

Operation `remove`():
Effekt: Entfernt erstes Element aus externer Warteschlange.

wenn S_2 *leer* **dann**
Kopiere die B zuletzt geschriebenen Elemente aus dem externen Speicher in S_2.
wenn S_2 *immer noch leer* **dann**
Kopiere S_1 in S_2 in verkehrter Reihenfolge.
S_1.*clear*()
S_2.`pop`()

`add`
`add`
`add`
`remove`
`remove`
`remove`
`add`
`add`
`add`
`add`
`remove`



Externer Speicher:

Institut für Theoretische Informatik
Prof. Dr. Dorothea Wagner

Externe Warteschlange

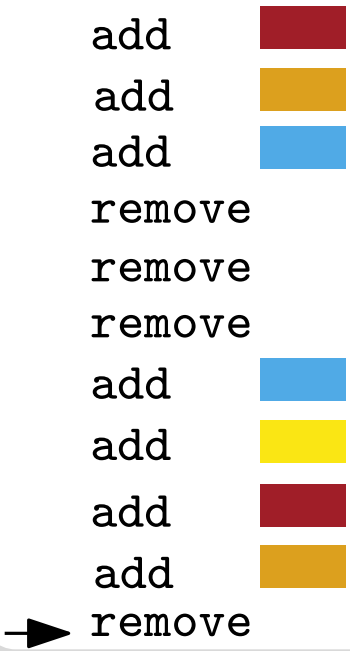
Kann mithilfe von zwei internen Stacks S_1 und S_2 der Größe B umgesetzt werden.
Idee: S_1 ist Schreib-Puffer und S_2 ist Lese-Puffer.

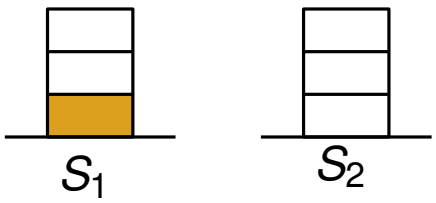
Operation `add`(*el*):
Effekt: Hängt *el* an externe Warteschlange.

wenn S_1 *voll* **dann**
Kopiere S_1 in externen Speicher in verkehrter Reihenfolge.
S_1.`clear`()
S_1.`push`(*el*)

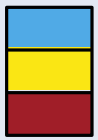
Operation `remove`():
Effekt: Entfernt erstes Element aus externer Warteschlange.

wenn S_2 *leer* **dann**
Kopiere die B zuletzt geschriebenen Elemente aus dem externen Speicher in S_2.
wenn S_2 *immer noch leer* **dann**
Kopiere S_1 in S_2 in verkehrter Reihenfolge.
S_1.*clear*()
S_2.`pop`()





Externer Speicher:



Institut für Theoretische Informatik
Prof. Dr. Dorothea Wagner

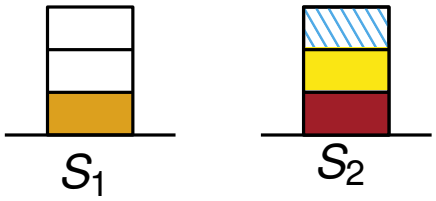
Externe Warteschlange

Kann mithilfe von zwei internen Stacks S_1 und S_2 der Größe B umgesetzt werden.
Idee: S_1 ist Schreib-Puffer und S_2 ist Lese-Puffer.

Operation `add`(*el*):
Effekt: Hängt *el* an externe Warteschlange.

wenn S_1 *voll* **dann**
Kopiere S_1 in externen Speicher in verkehrter Reihenfolge.
S_1.`clear`()
S_1.`push`(*el*)

`add`
`add`
`add`
`remove`
`remove`
`remove`
`add`
`add`
`add`
`add`
`remove`



Operation `remove`():
Effekt: Entfernt erstes Element aus externer Warteschlange.

wenn S_2 *leer* **dann**
Kopiere die B zuletzt geschriebenen Elemente aus dem externen Speicher in S_2.
wenn S_2 *immer noch leer* **dann**
Kopiere S_1 in S_2 in verkehrter Reihenfolge.
S_1.*clear*()
S_2.`pop`()

Externer Speicher:

Institut für Theoretische Informatik
Prof. Dr. Dorothea Wagner

Multiway Merge Sort

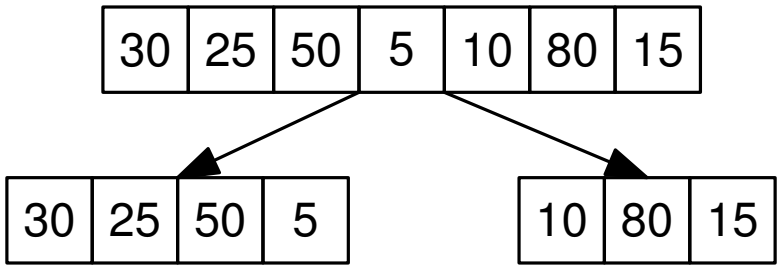
Erinnerung: Prinzip von Merge Sort

Laufzeit: $O(n \log n)$

Multiway Merge Sort

Erinnerung: Prinzip von Merge Sort

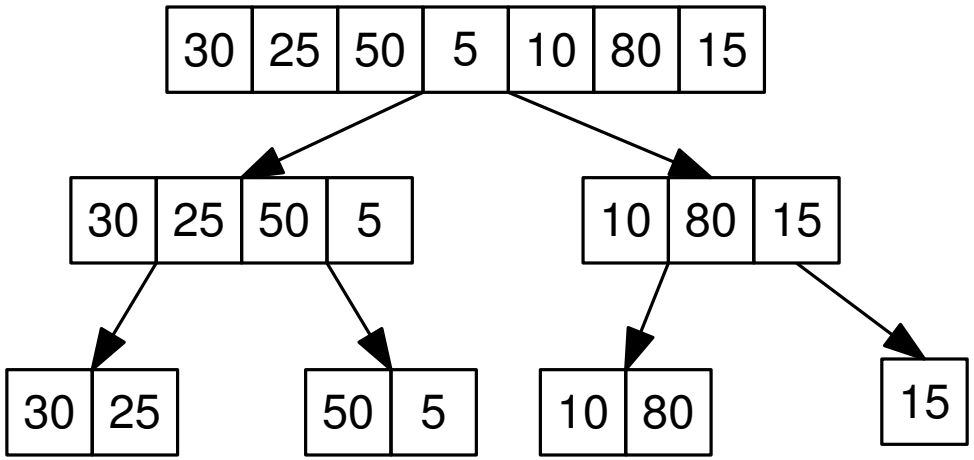
Laufzeit: $O(n \log n)$



Multiway Merge Sort

Erinnerung: Prinzip von Merge Sort

Laufzeit: $O(n \log n)$

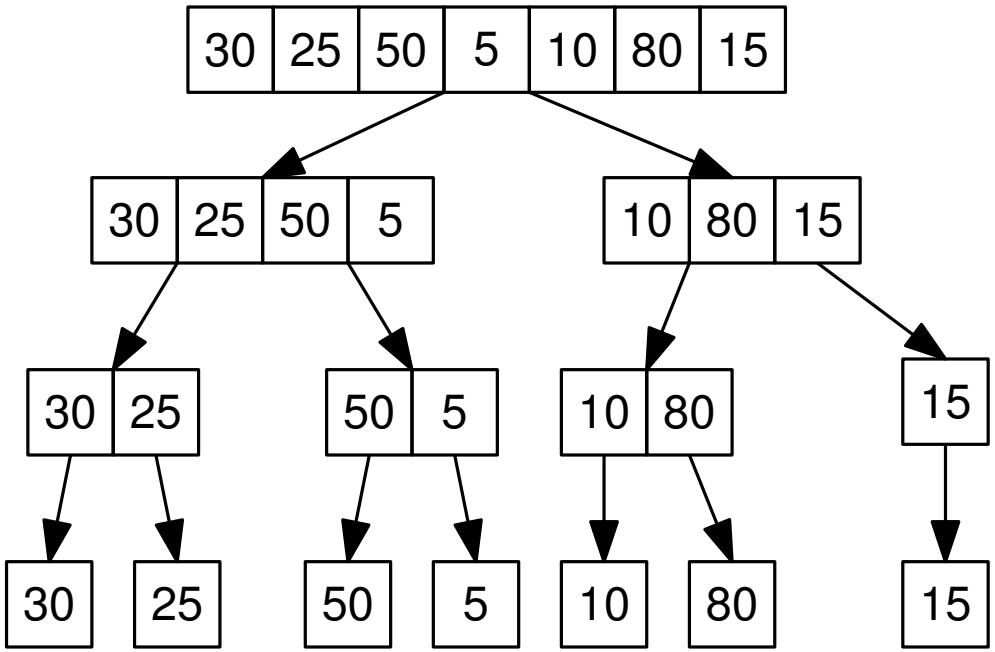


Institut für Theoretische Informatik
Prof. Dr. Dorothea Wagner

Multiway Merge Sort

Erinnerung: Prinzip von Merge Sort

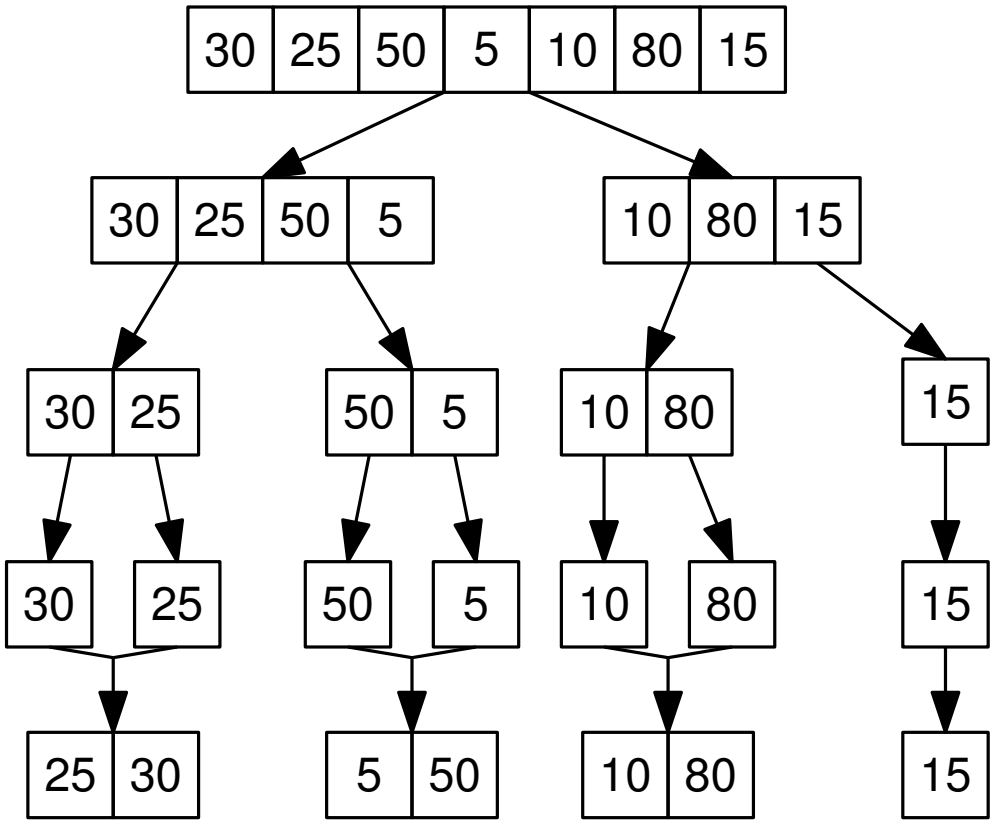
Laufzeit: $O(n \log n)$



Multiway Merge Sort

Erinnerung: Prinzip von Merge Sort

Laufzeit: $O(n \log n)$

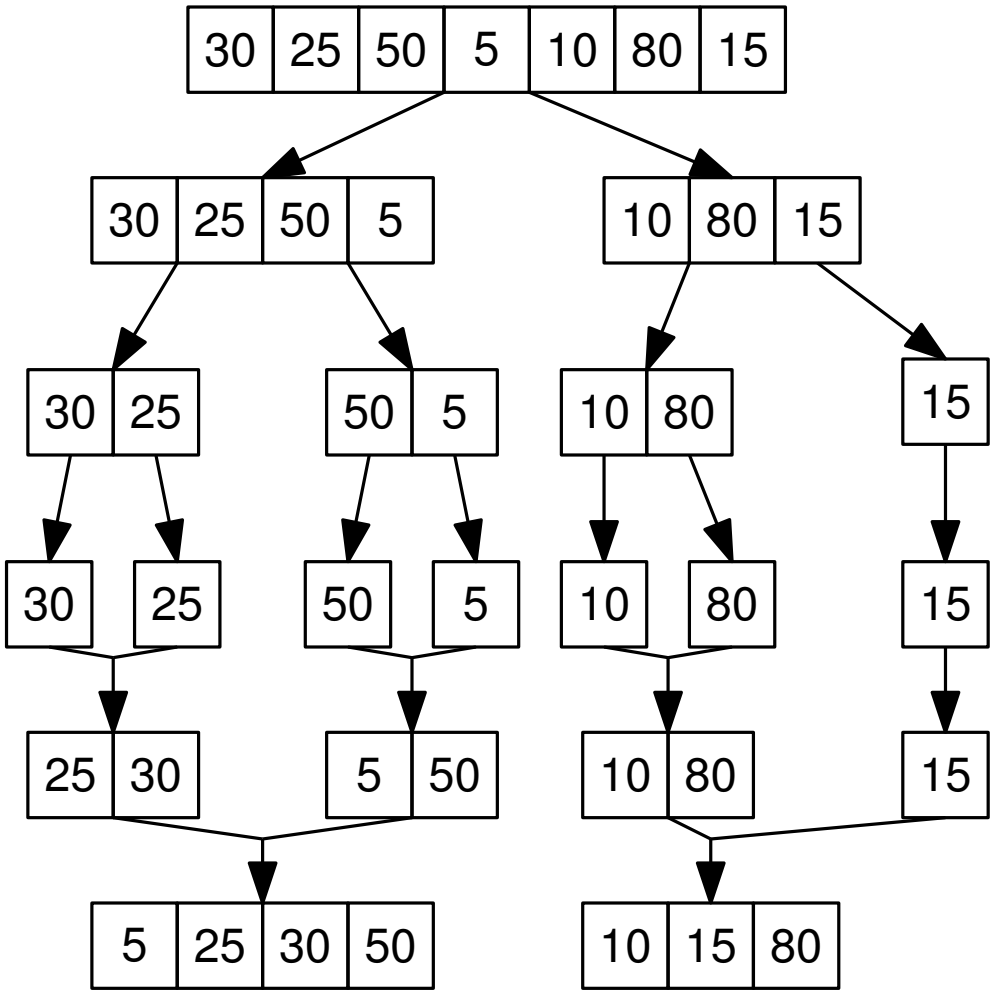


Institut für Theoretische Informatik
Prof. Dr. Dorothea Wagner

Multiway Merge Sort

Erinnerung: Prinzip von Merge Sort

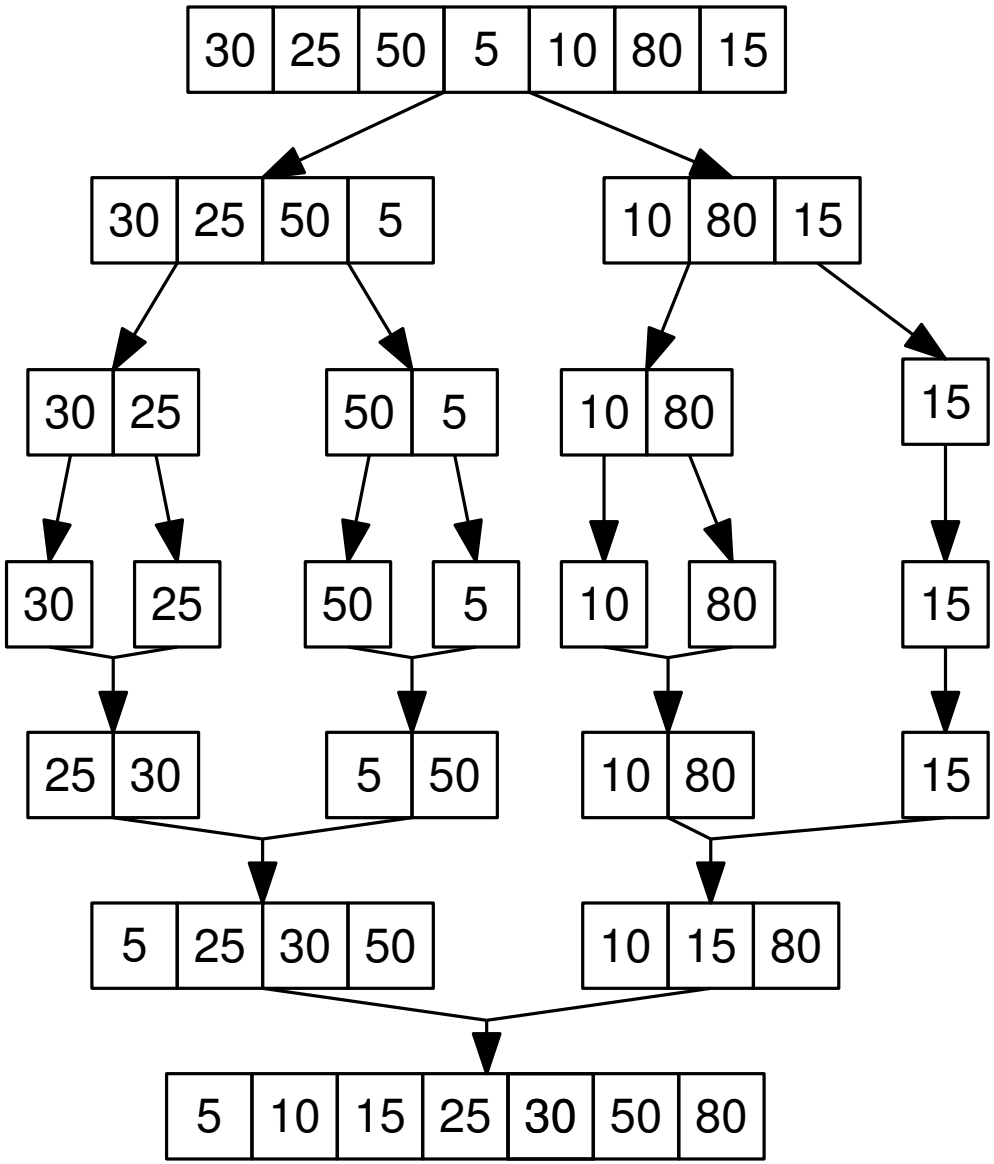
Laufzeit: $O(n \log n)$



Multiway Merge Sort

Erinnerung: Prinzip von Merge Sort

Laufzeit: $O(n \log n)$



Institut für Theoretische Informatik
Prof. Dr. Dorothea Wagner

Multiway Merge Sort

Eingabe: $a_1, \ldots, a_n$ Elemente, die im externen Speicher liegen.
Ausgabe: Sortierung von $a_1, \ldots, a_n$.

1. Phase: Run Formation

1. Teile $a_1, \ldots, a_n$ in m Gruppen $G_1, \ldots, G_m$ der Größe $\Theta(M)$ auf.
2. Lade jede Gruppe G_i in den Hauptspeicher, sortiere sie und schreibe sie zurück in den externen Speicher: Man erhält $\mathcal{R} = \{R_1, \ldots, R_m\}$ sortierte *Runs*.

2. Phase: Merging

3. Vereinige die einzelnen Runs ordnungserhaltend zu größeren, bis schließlich einer übrig bleibt. Vermenge hierzu möglichst viele Runs pro Durchgang.

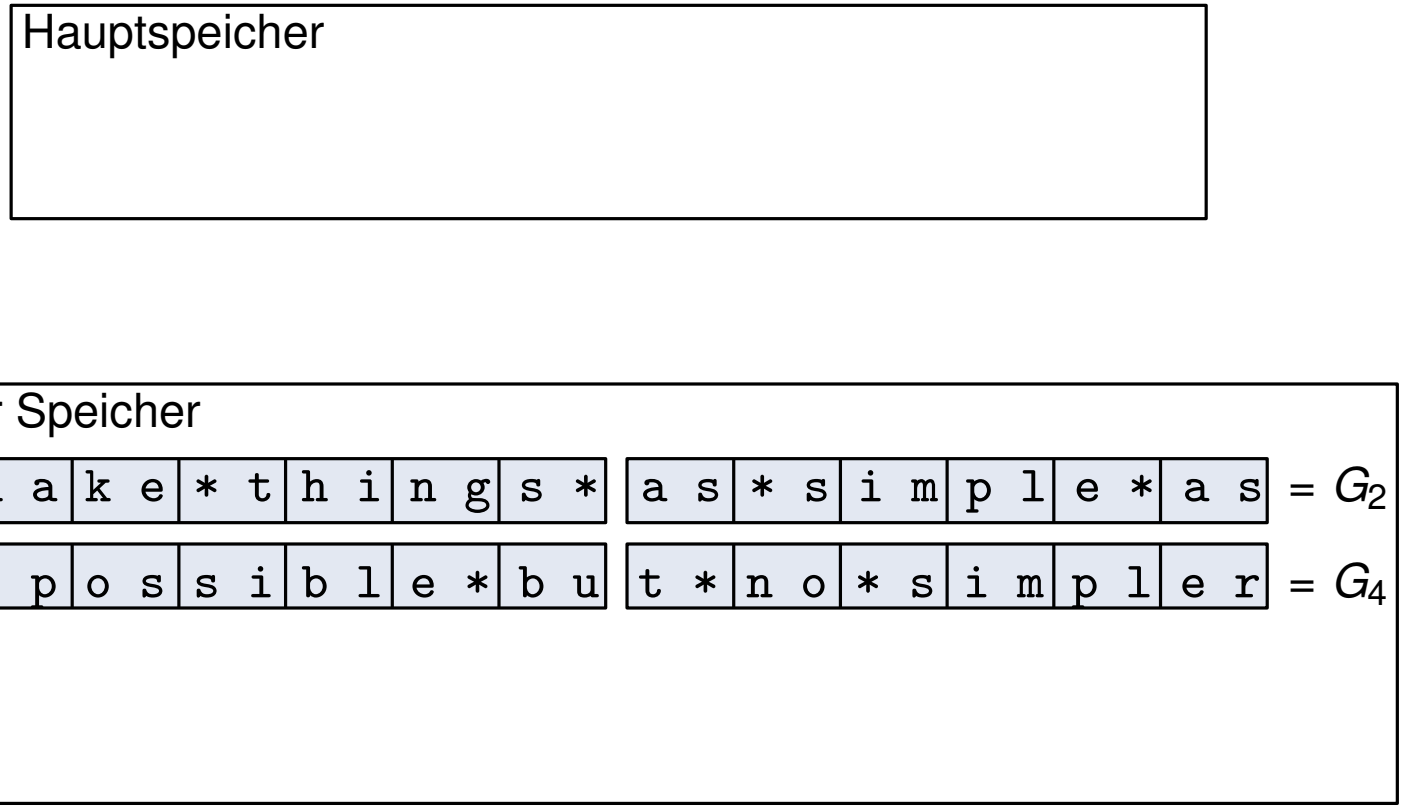
Die folgenden Folien basieren auf dem Paper:
Asynchronous parallel disk sorting, R. Dementiev and P. Sanders. In 15th ACM Symposium on Parallelism in Algorithms and Architectures, pages 138–148, San Diego, 2003.
Entsprechend sind die Nummerierungen gewählt.

1. Phase: Run Formation

1. Phase: Run Formation

1. Teile $a_1, \ldots, a_n$ in m Gruppen $G_1, \ldots, G_m$ der Größe $\Theta(M)$ auf.
2. Lade jede Gruppe G_i in den Hauptspeicher, sortiere sie und schreibe sie zurück in den externen Speicher: Man erhält $\mathcal{R} = \{R_1, \ldots, R_m\}$ sortierte *Runs*.

Beispiel $m = 4$ und $B = 2$:



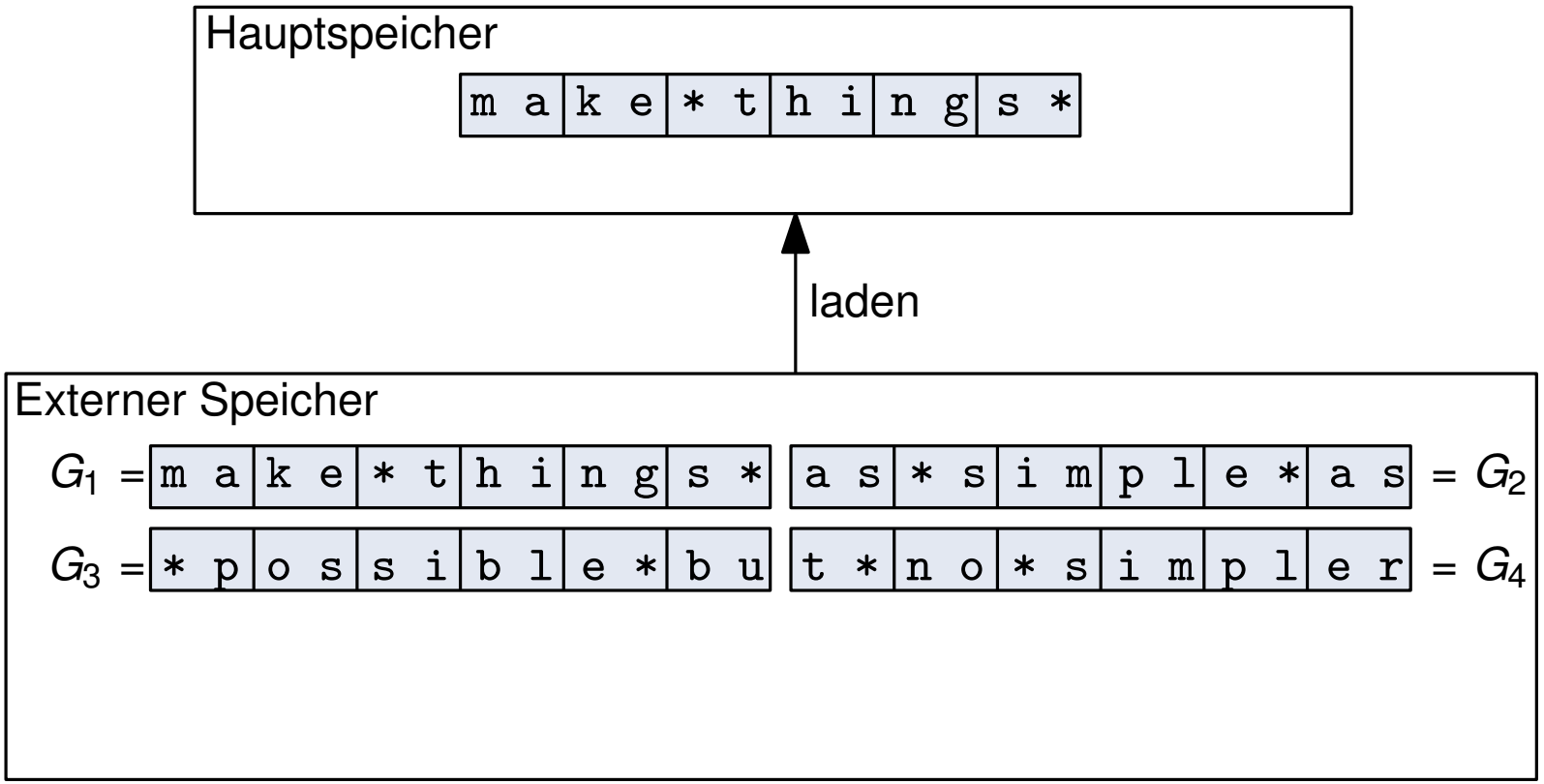
Institut für Theoretische Informatik
Prof. Dr. Dorothea Wagner

1. Phase: Run Formation

1. Phase: Run Formation

1. Teile $a_1, \ldots, a_n$ in m Gruppen $G_1, \ldots, G_m$ der Größe $\Theta(M)$ auf.
2. Lade jede Gruppe G_i in den Hauptspeicher, sortiere sie und schreibe sie zurück in den externen Speicher: Man erhält $\mathcal{R} = \{R_1, \ldots, R_m\}$ sortierte *Runs*.

Beispiel $m = 4$ und $B = 2$:



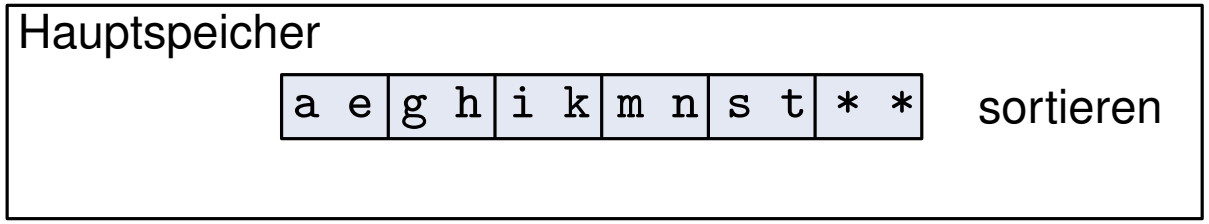
Institut für Theoretische Informatik
Prof. Dr. Dorothea Wagner

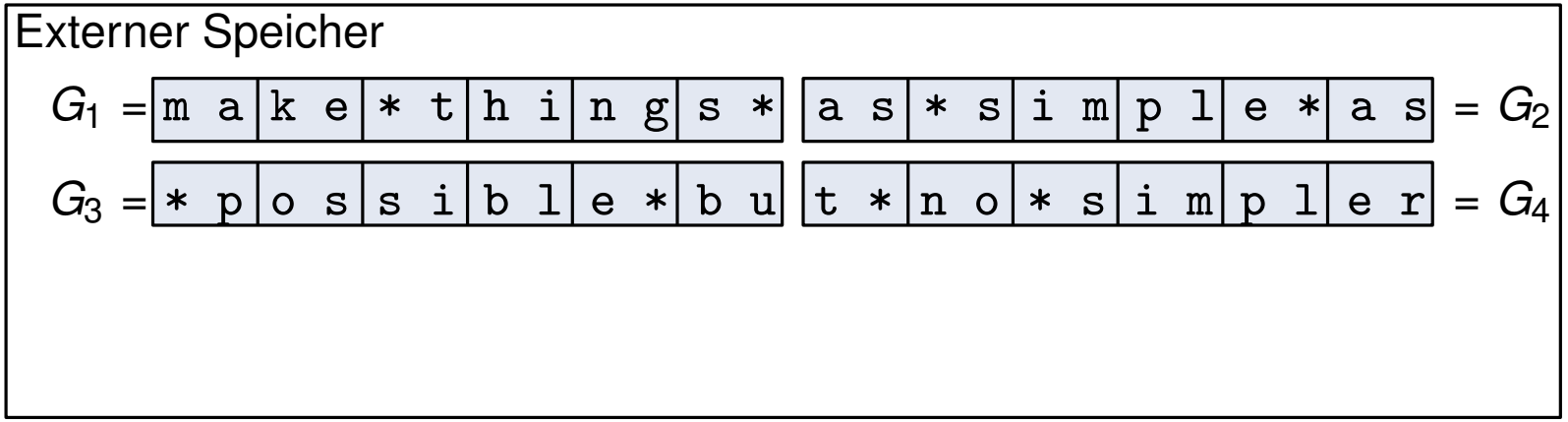
1. Phase: Run Formation

1. Phase: Run Formation

1. Teile $a_1, \ldots, a_n$ in m Gruppen $G_1, \ldots, G_m$ der Größe $\Theta(M)$ auf.
2. Lade jede Gruppe G_i in den Hauptspeicher, sortiere sie und schreibe sie zurück in den externen Speicher: Man erhält $\mathcal{R} = \{R_1, \ldots, R_m\}$ sortierte *Runs*.

Beispiel $m = 4$ und $B = 2$:





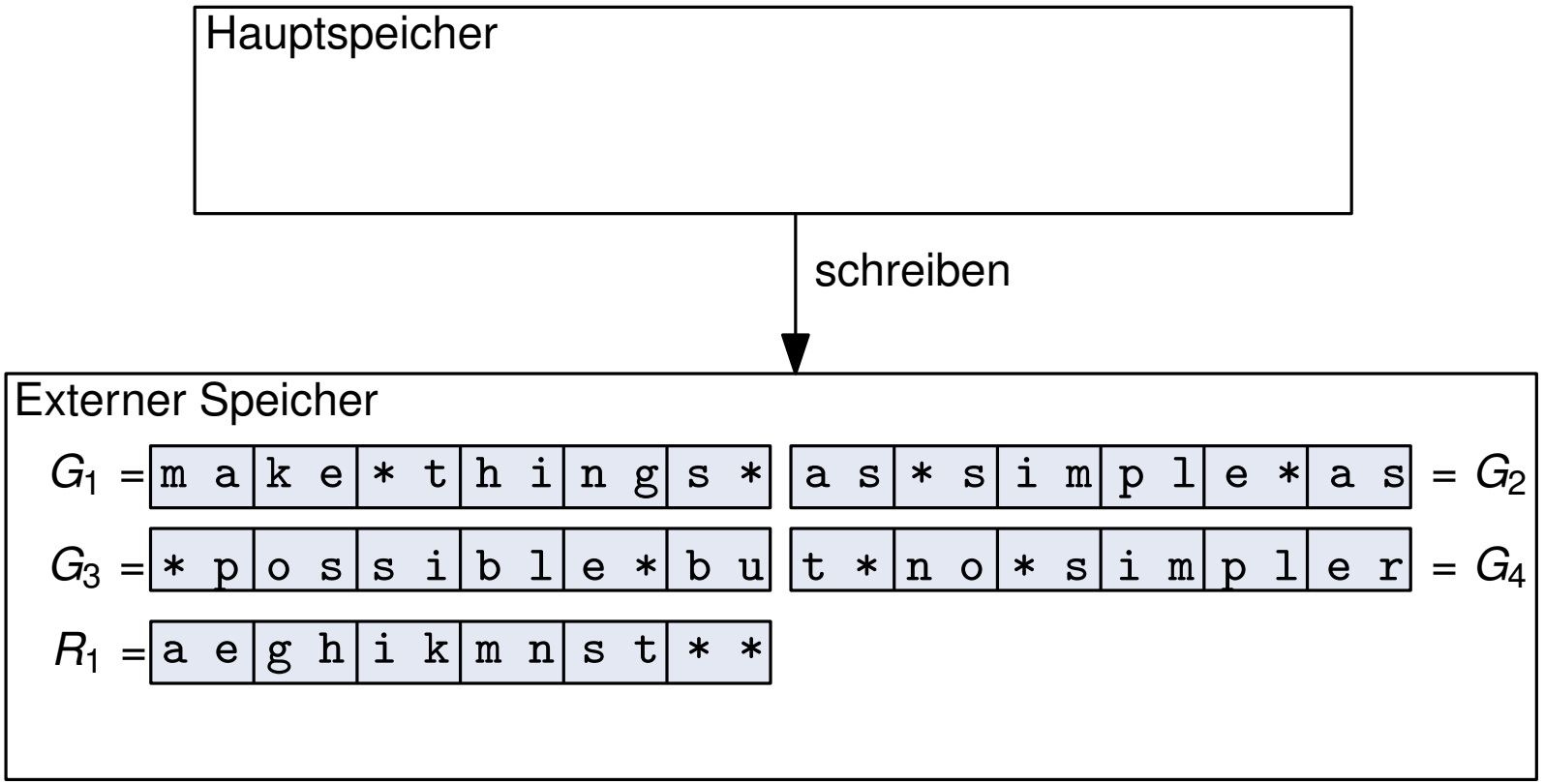
Institut für Theoretische Informatik
Prof. Dr. Dorothea Wagner

1. Phase: Run Formation

1. Phase: Run Formation

1. Teile $a_1, \ldots, a_n$ in m Gruppen $G_1, \ldots, G_m$ der Größe $\Theta(M)$ auf.
2. Lade jede Gruppe G_i in den Hauptspeicher, sortiere sie und schreibe sie zurück in den externen Speicher: Man erhält $\mathcal{R} = \{R_1, \ldots, R_m\}$ sortierte *Runs*.

Beispiel $m = 4$ und $B = 2$:



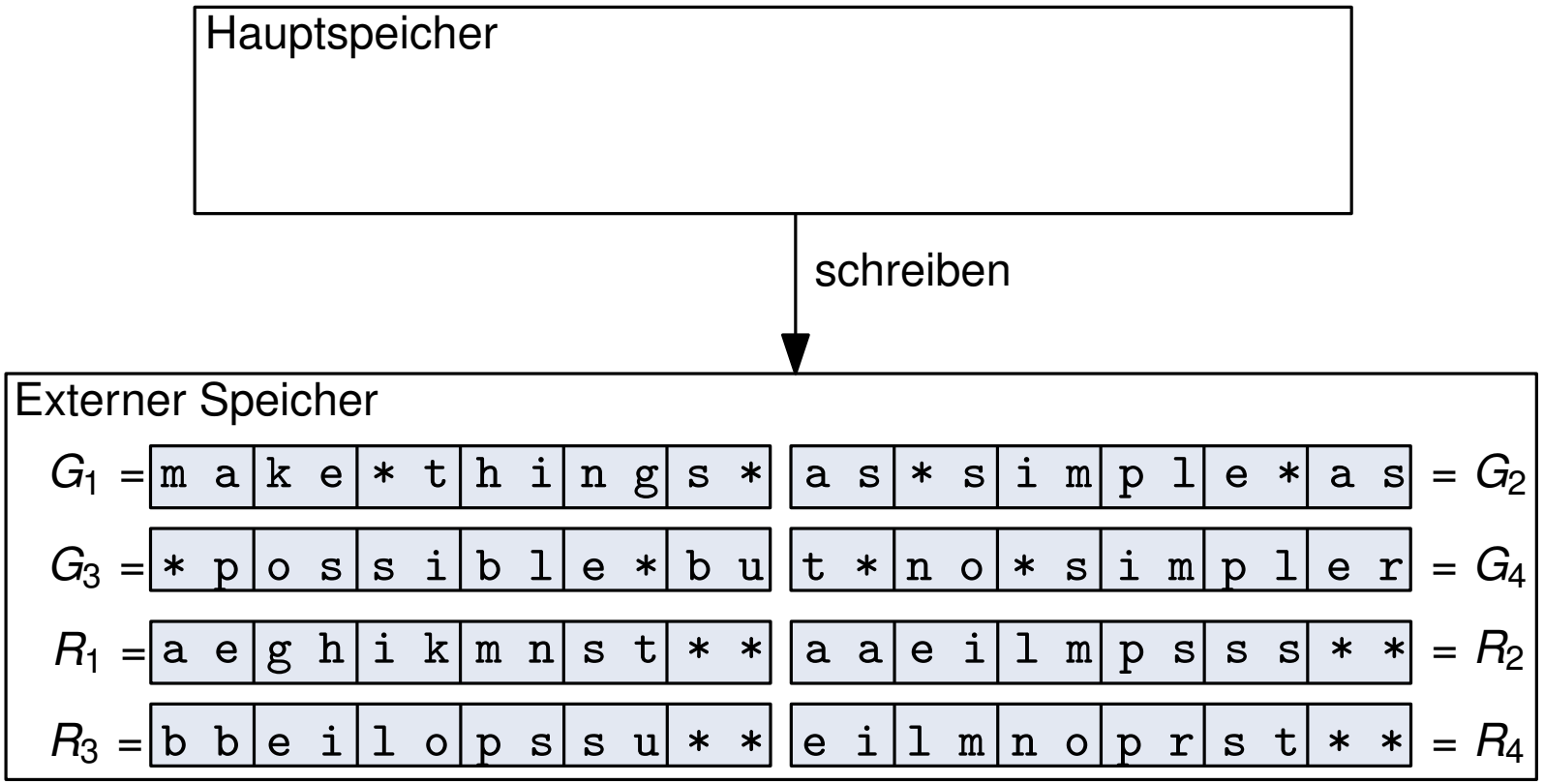
Institut für Theoretische Informatik
Prof. Dr. Dorothea Wagner

1. Phase: Run Formation

1. Phase: Run Formation

1. Teile $a_1, \ldots, a_n$ in m Gruppen $G_1, \ldots, G_m$ der Größe $\Theta(M)$ auf.
2. Lade jede Gruppe G_i in den Hauptspeicher, sortiere sie und schreibe sie zurück in den externen Speicher: Man erhält $\mathcal{R} = \{R_1, \ldots, R_m\}$ sortierte *Runs*.

Beispiel $m = 4$ und $B = 2$:



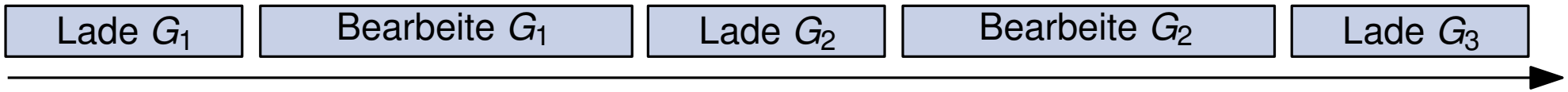
Institut für Theoretische Informatik
Prof. Dr. Dorothea Wagner

1. Phase: Run Formation

1. Phase: Run Formation

1. Teile $a_1, \ldots, a_n$ in m Gruppen $G_1, \ldots, G_m$ der Größe $\Theta(M)$ auf.
2. Lade jede Gruppe G_i in den Hauptspeicher, sortiere sie und schreibe sie zurück in den externen Speicher: Man erhält $\mathcal{R} = \{R_1, \ldots, R_m\}$ sortierte *Runs*.

- *I/O-beschränkt*: I/O-Operationen benötigen mehr Zeit, als das Sortieren.
- *berechnungsbeschränkt*: Sortieren benötigt mehr Zeit als I/O-Operationen.



Beobachtung: G_2 könnte bereits in den Speicher geladen werden, solange G_1 bearbeitet wird.

Verwende *Overlapping*-Technik: Teile Arbeit auf zwei Threads auf:

- Thread *A*: Verantwortlich für das Laden und Schreiben der Daten.
- Thread *B*: Verantwortlich für die eigentliche Arbeit auf den geladenen Daten.

1. Phase: Run Formation

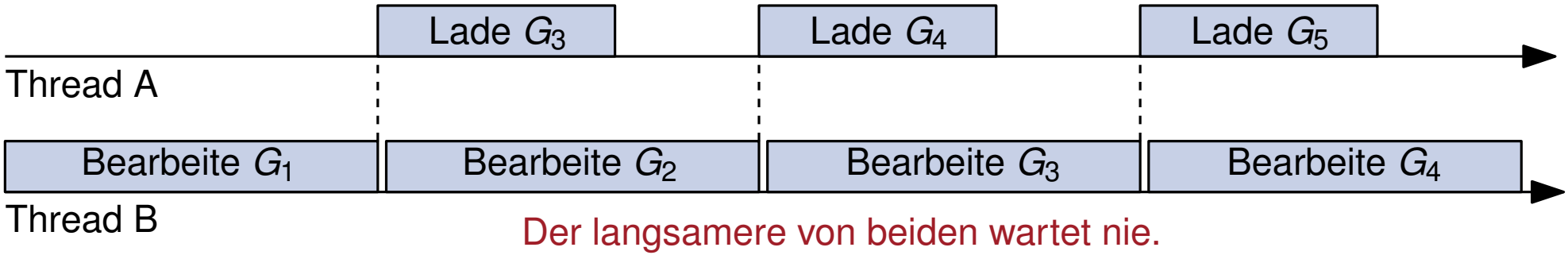
1. Phase: Run Formation

1. Teile $a_1, \ldots, a_n$ in m Gruppen $G_1, \ldots, G_m$ der Größe $\Theta(M)$ auf.
2. Lade jede Gruppe G_i in den Hauptspeicher, sortiere sie und schreibe sie zurück in den externen Speicher: Man erhält $\mathcal{R} = \{R_1, \ldots, R_m\}$ sortierte *Runs*.

Lade G_1 und G_2 in den Hauptspeicher.

Thread A:	Thread B:
für $i = 1, \ldots, m-2$ **tue** **Warte** bis G_i in den externen Speicher geschrieben wurde. Lade G_{i+2} in den Hauptspeicher.	**für** $i = 1, \ldots, m$ **tue** **Warte** bis G_i in den Haupspeicher geladen wurde. Sortiere Gruppe G_i. Schreibe G_i in externen Speicher.

Annahme: berechnungsbeschränkt



Der langsamere von beiden wartet nie.

1. Phase: Run Formation

Korollar 2: Eine Eingabe der Größe N kann in Zeit

$$\max\{2T_{sort}(\frac{M}{2})\frac{N}{M}, \frac{2LN}{DB}\} + O(\frac{LM}{DB})$$

in sortierte Runs der Größe $\frac{M}{2} - O(DB)$ transformiert werden. Dabei bezeichnet $T_{sort}(n)$ die Zeit, die benötigt wird um n Elemente intern zu sortieren und L die Zeit, die für einen I/O-Schritt benötigt wird.

Notation:

- M: Größe des Hauptspeichers.
- N: Größe der Instanz:
- D: Anzahl der Platten des externen Speichers.
- B: Anzahl Elemente in einem Block.
- L: Zeit, die für einen I/O-Schritt benötigt wird (Latenz).

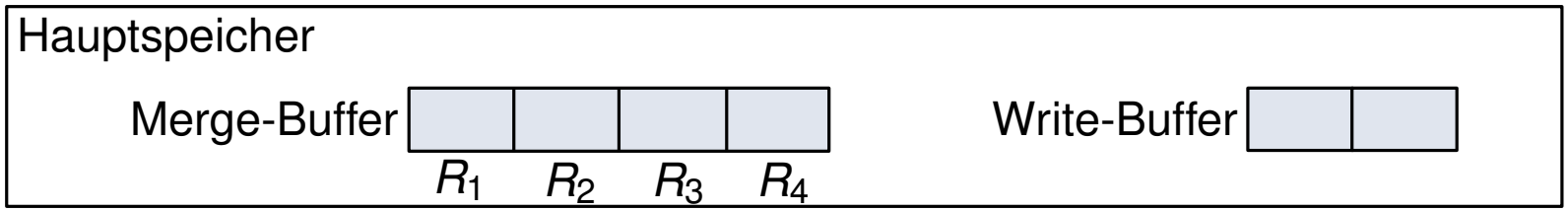
2. Phase: Merging

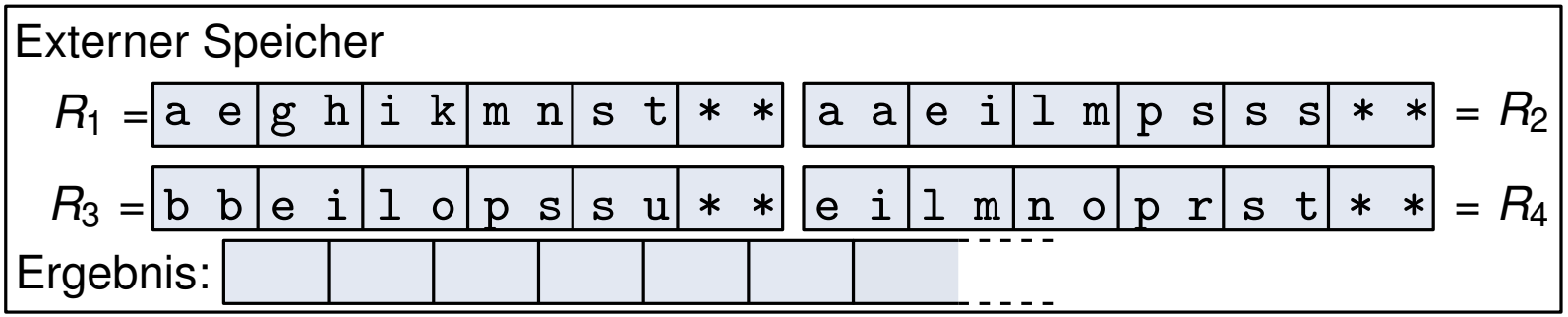
2. Phase: Merging

3. Vereinige die einzelnen Runs ordnungserhaltend zu größeren, bis schließlich einer übrig bleibt. Es können $k = O(\frac{M}{B})$ Runs in einem Durchgang vereinigt werden.

Vereinige k Runs in einem Durchlauf (MULTIWAY MERGING):

1. Halte nur den Block pro Run im Hauptspeicher (*Merge-Buffer*), der das aktuell kleinste Element enthält.
2. Vereinige Blöcke im Merge-Buffer schrittweise und schreibe Ergebnis in den Write-Buffer.
3. Lade bei Bedarf Blöcke nach und schreibe Write-Buffer in externen Speicher.





$B = 2$
$D = 1$

Institut für Theoretische Informatik
Prof. Dr. Dorothea Wagner

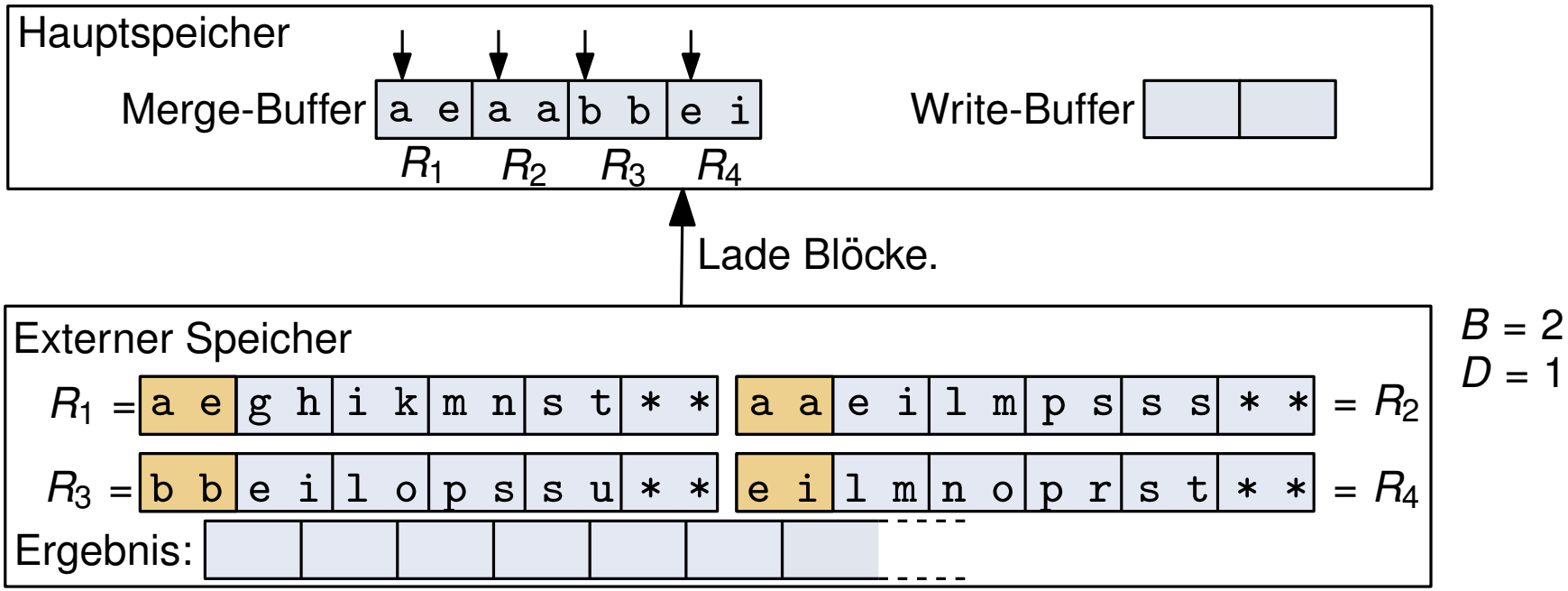
2. Phase: Merging

2. Phase: Merging

3. Vereinige die einzelnen Runs ordnungserhaltend zu größeren, bis schließlich einer übrig bleibt. Es können $k = O(\frac{M}{B})$ Runs in einem Durchgang vereinigt werden.

Vereinige k Runs in einem Durchlauf (MULTIWAY MERGING):

1. Halte nur den Block pro Run im Hauptspeicher (*Merge-Buffer*), der das aktuell kleinste Element enthält.
2. Vereinige Blöcke im Merge-Buffer schrittweise und schreibe Ergebnis in den Write-Buffer.
3. Lade bei Bedarf Blöcke nach und schreibe Write-Buffer in externen Speicher.



Institut für Theoretische Informatik
Prof. Dr. Dorothea Wagner

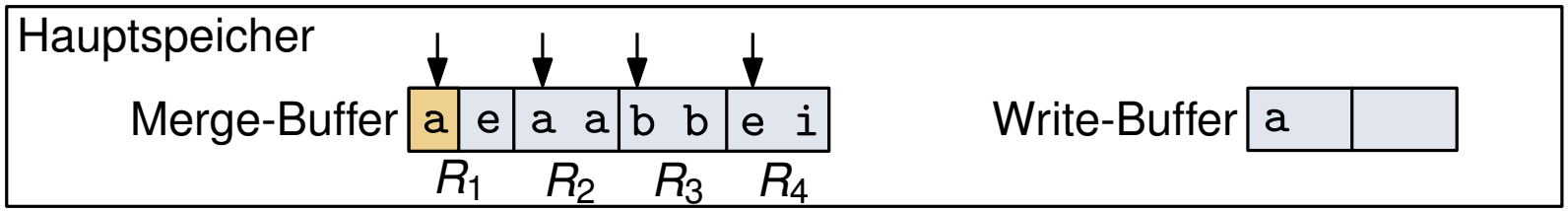
2. Phase: Merging

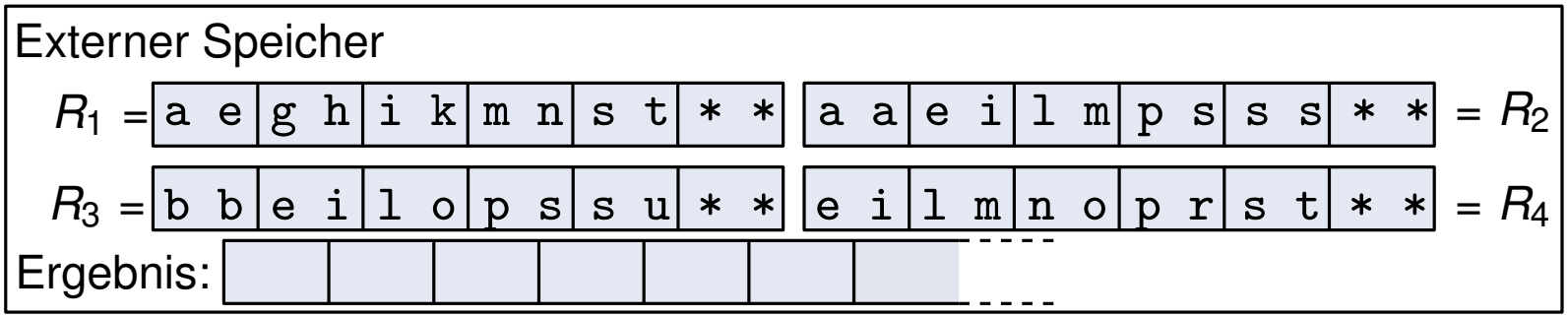
2. Phase: Merging

3. Vereinige die einzelnen Runs ordnungserhaltend zu größeren, bis schließlich einer übrig bleibt. Es können $k = O(\frac{M}{B})$ Runs in einem Durchgang vereinigt werden.

Vereinige k Runs in einem Durchlauf (MULTIWAY MERGING):

1. Halte nur den Block pro Run im Hauptspeicher (*Merge-Buffer*), der das aktuell kleinste Element enthält.
2. Vereinige Blöcke im Merge-Buffer schrittweise und schreibe Ergebnis in den Write-Buffer.
3. Lade bei Bedarf Blöcke nach und schreibe Write-Buffer in externen Speicher.





$B = 2$
$D = 1$

Institut für Theoretische Informatik
Prof. Dr. Dorothea Wagner

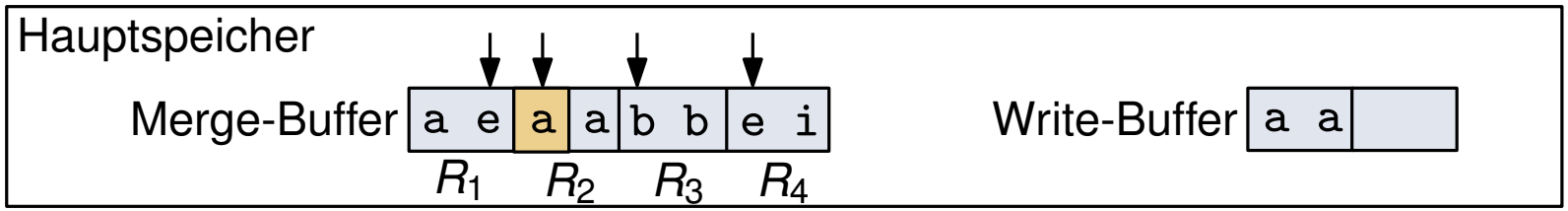
2. Phase: Merging

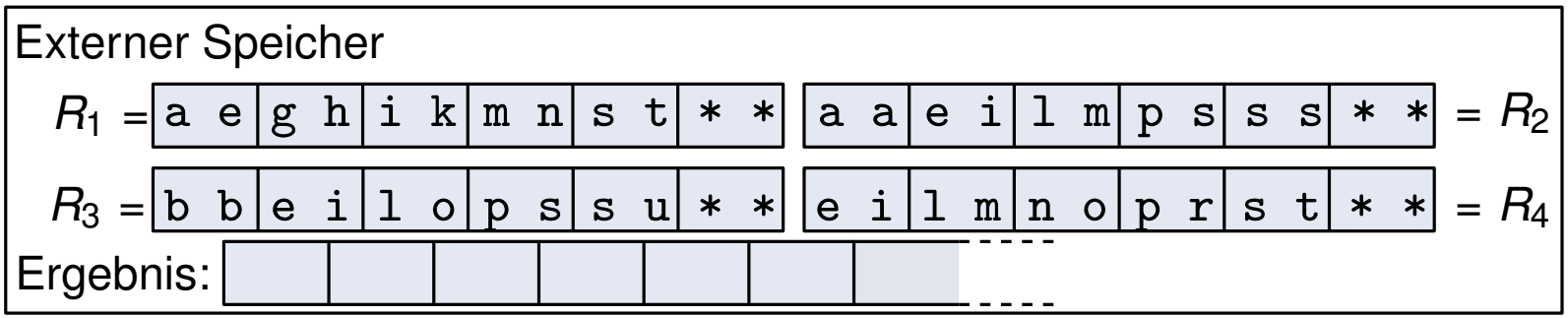
2. Phase: Merging

3. Vereinige die einzelnen Runs ordnungserhaltend zu größeren, bis schließlich einer übrig bleibt. Es können $k = O(\frac{M}{B})$ Runs in einem Durchgang vereinigt werden.

Vereinige k Runs in einem Durchlauf (MULTIWAY MERGING):

1. Halte nur den Block pro Run im Hauptspeicher (*Merge-Buffer*), der das aktuell kleinste Element enthält.
2. Vereinige Blöcke im Merge-Buffer schrittweise und schreibe Ergebnis in den Write-Buffer.
3. Lade bei Bedarf Blöcke nach und schreibe Write-Buffer in externen Speicher.





$B = 2$
$D = 1$

Institut für Theoretische Informatik
Prof. Dr. Dorothea Wagner

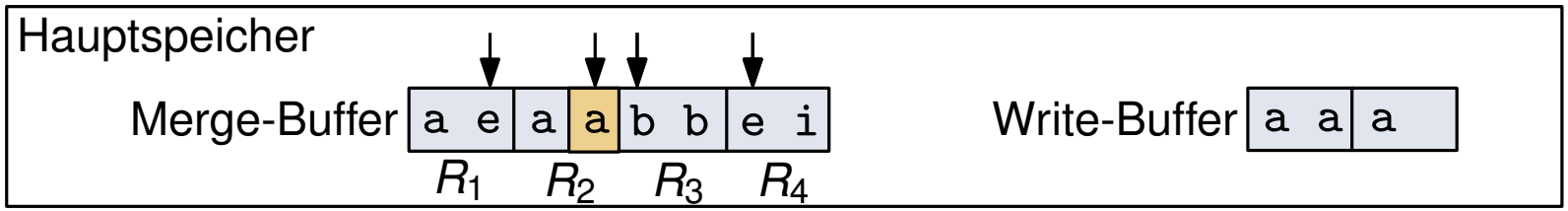
2. Phase: Merging

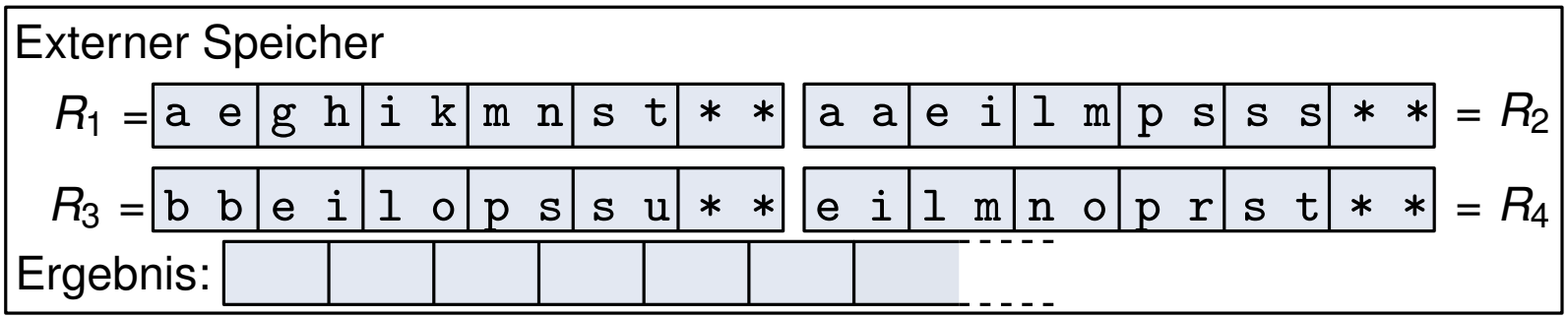
2. Phase: Merging

3. Vereinige die einzelnen Runs ordnungserhaltend zu größeren, bis schließlich einer übrig bleibt. Es können $k = O(\frac{M}{B})$ Runs in einem Durchgang vereinigt werden.

Vereinige k Runs in einem Durchlauf (MULTIWAY MERGING):

1. Halte nur den Block pro Run im Hauptspeicher (*Merge-Buffer*), der das aktuell kleinste Element enthält.
2. Vereinige Blöcke im Merge-Buffer schrittweise und schreibe Ergebnis in den Write-Buffer.
3. Lade bei Bedarf Blöcke nach und schreibe Write-Buffer in externen Speicher.





$B = 2$
$D = 1$

Institut für Theoretische Informatik
Prof. Dr. Dorothea Wagner

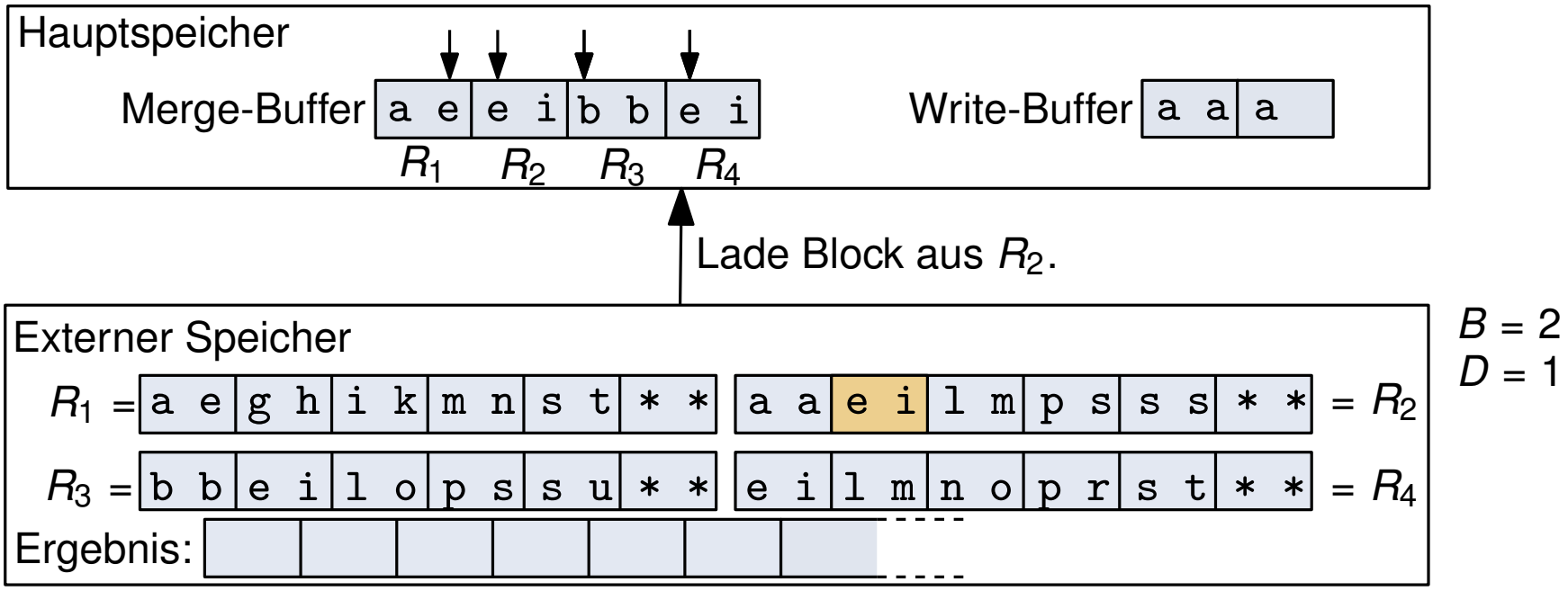
2. Phase: Merging

2. Phase: Merging

3. Vereinige die einzelnen Runs ordnungserhaltend zu größeren, bis schließlich einer übrig bleibt. Es können $k = O(\frac{M}{B})$ Runs in einem Durchgang vereinigt werden.

Vereinige k Runs in einem Durchlauf (MULTIWAY MERGING):

1. Halte nur den Block pro Run im Hauptspeicher (*Merge-Buffer*), der das aktuell kleinste Element enthält.
2. Vereinige Blöcke im Merge-Buffer schrittweise und schreibe Ergebnis in den Write-Buffer.
3. Lade bei Bedarf Blöcke nach und schreibe Write-Buffer in externen Speicher.



Institut für Theoretische Informatik
Prof. Dr. Dorothea Wagner

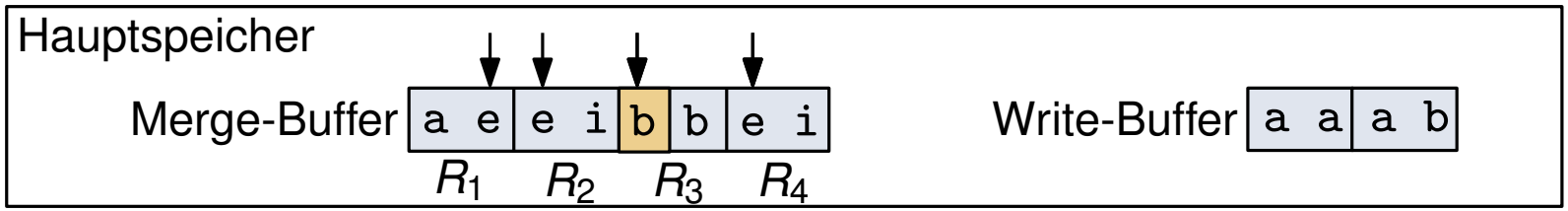
2. Phase: Merging

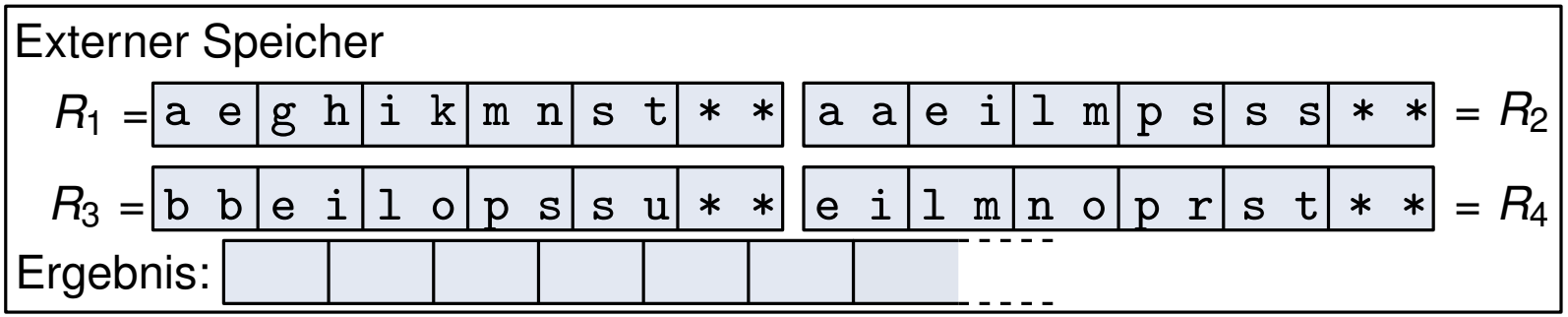
2. Phase: Merging

3. Vereinige die einzelnen Runs ordnungserhaltend zu größeren, bis schließlich einer übrig bleibt. Es können $k = O(\frac{M}{B})$ Runs in einem Durchgang vereinigt werden.

Vereinige k Runs in einem Durchlauf (MULTIWAY MERGING):

1. Halte nur den Block pro Run im Hauptspeicher (*Merge-Buffer*), der das aktuell kleinste Element enthält.
2. Vereinige Blöcke im Merge-Buffer schrittweise und schreibe Ergebnis in den Write-Buffer.
3. Lade bei Bedarf Blöcke nach und schreibe Write-Buffer in externen Speicher.





$B = 2$
$D = 1$

Institut für Theoretische Informatik
Prof. Dr. Dorothea Wagner

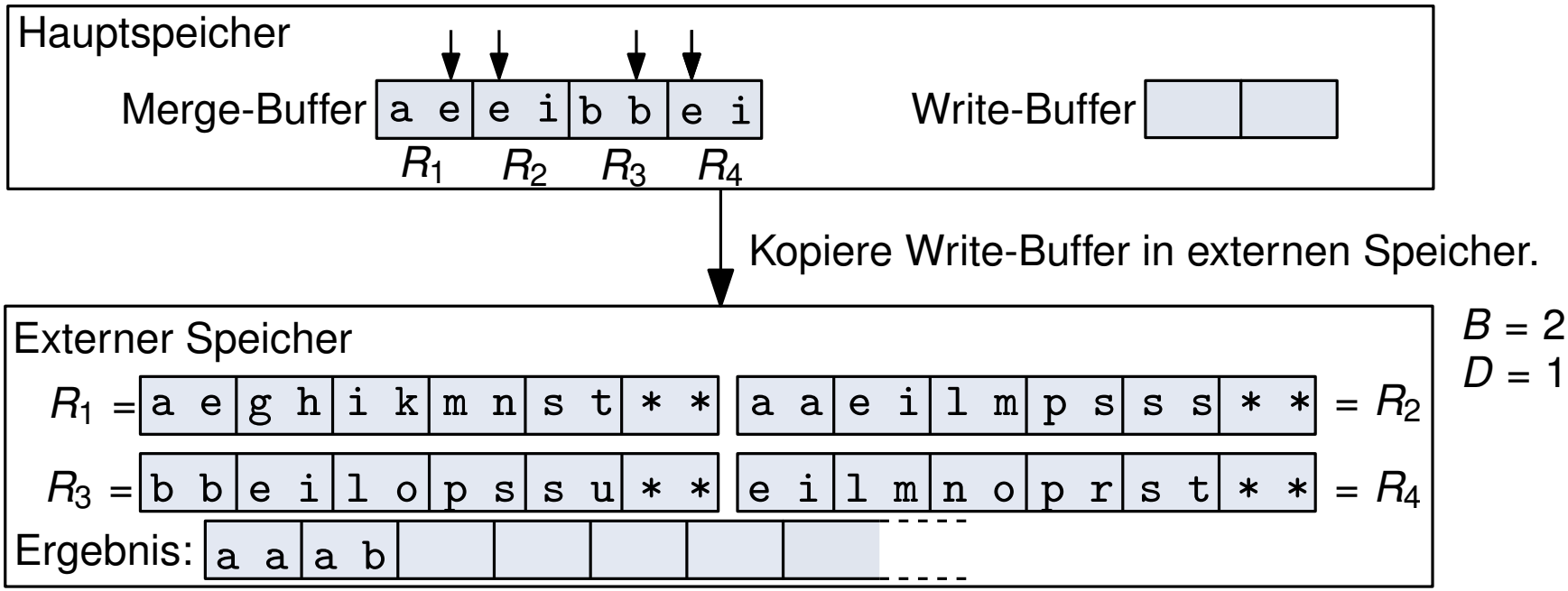
2. Phase: Merging

2. Phase: Merging

3. Vereinige die einzelnen Runs ordnungserhaltend zu größeren, bis schließlich einer übrig bleibt. Es können $k = O(\frac{M}{B})$ Runs in einem Durchgang vereinigt werden.

Vereinige k Runs in einem Durchlauf (MULTIWAY MERGING):

1. Halte nur den Block pro Run im Hauptspeicher (*Merge-Buffer*), der das aktuell kleinste Element enthält.
2. Vereinige Blöcke im Merge-Buffer schrittweise und schreibe Ergebnis in den Write-Buffer.
3. Lade bei Bedarf Blöcke nach und schreibe Write-Buffer in externen Speicher.



Institut für Theoretische Informatik
Prof. Dr. Dorothea Wagner

2. Phase: Merging

Lemma 3: Zu jedem Zeitpunkt von MULTIWAY MERGING ist das kleinste Element im Merge-Buffer unter allen Elementen, die noch nicht von der Merge-Prozedur betrachtet worden sind, minimal.

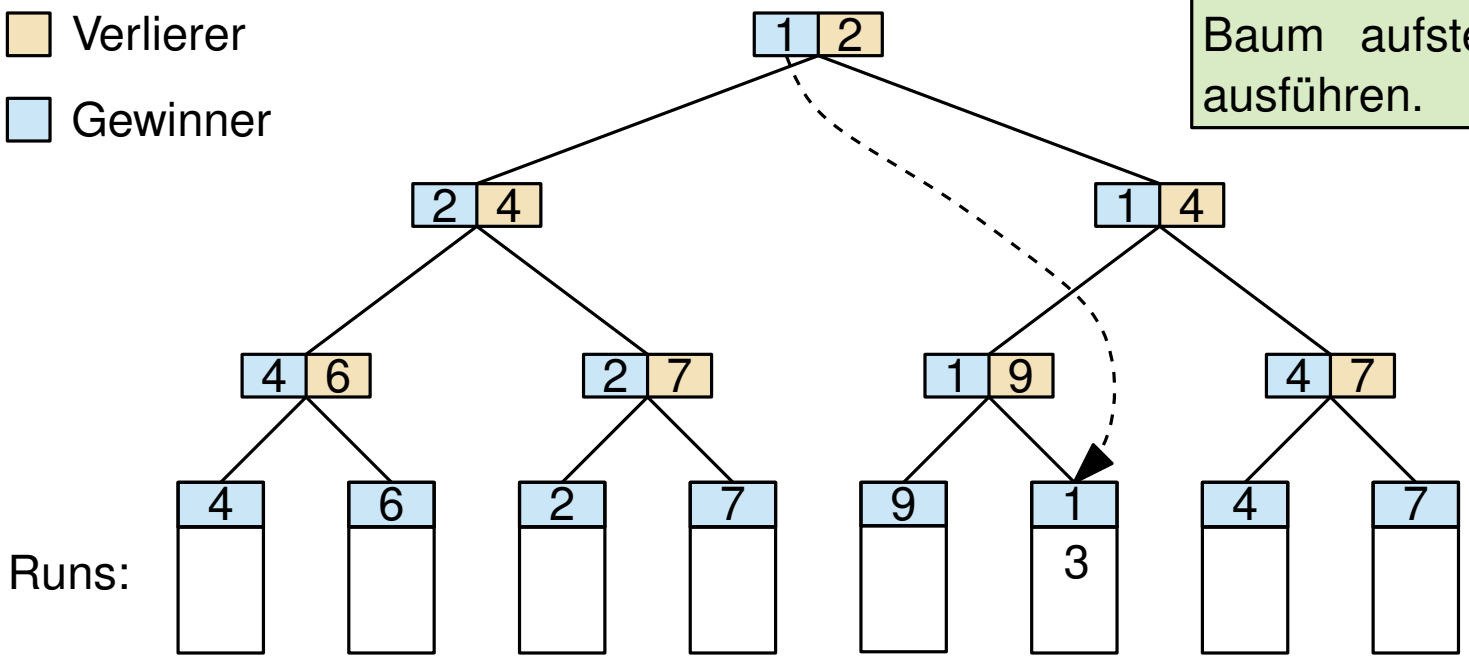
Tournament-Bäume

Wie kann das kleinste Element im Merge-Buffer schnell gefunden werden?

Ein *Tournament-Baum* ist ein binärer Baum mit k Blättern, sodass

- das i-te Blatt das kleinste Element vom i-ten Run enthält, und
- jeder innere Knoten den Gewinner und den Verlierer des Wettkampfes zwischen den Gewinnern seiner zwei Kindern enthält. Es gilt: Gewinner $<$ Verlierer und Blätter enthalten nur Gewinner.

Minimum entfernen und Baum aktualisieren ist in $O(\log n)$ Zeit möglich.
Grundidee: Lasse neuen Wert im Baum aufsteigen und Wettkämpfe ausführen.



Institut für Theoretische Informatik
Prof. Dr. Dorothea Wagner

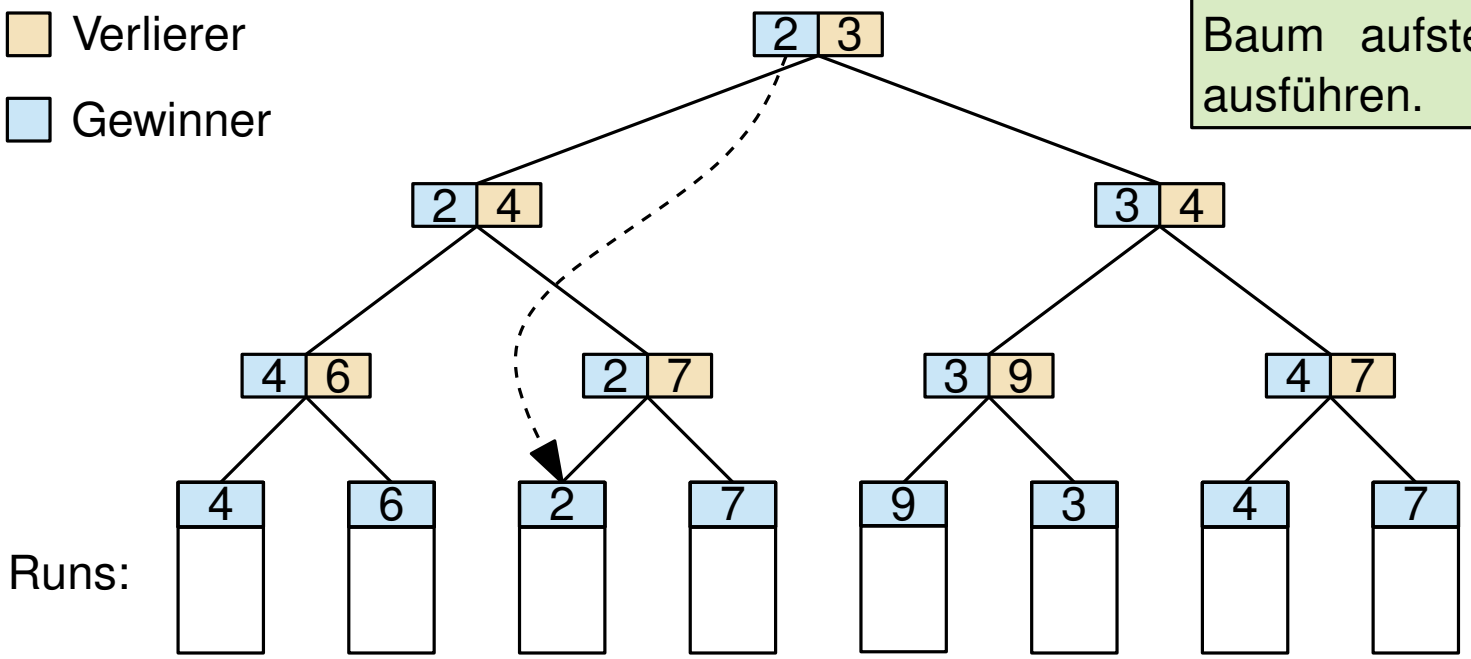
Tournament-Bäume

Wie kann das kleinste Element im Merge-Buffer schnell gefunden werden?

Ein *Tournament-Baum* ist ein binärer Baum mit k Blättern, sodass

- das i-te Blatt das kleinste Element vom i-ten Run enthält, und
- jeder innere Knoten den Gewinner und den Verlierer des Wettkampfes zwischen den Gewinnern seiner zwei Kindern enthält. Es gilt: Gewinner $<$ Verlierer und Blätter enthalten nur Gewinner.

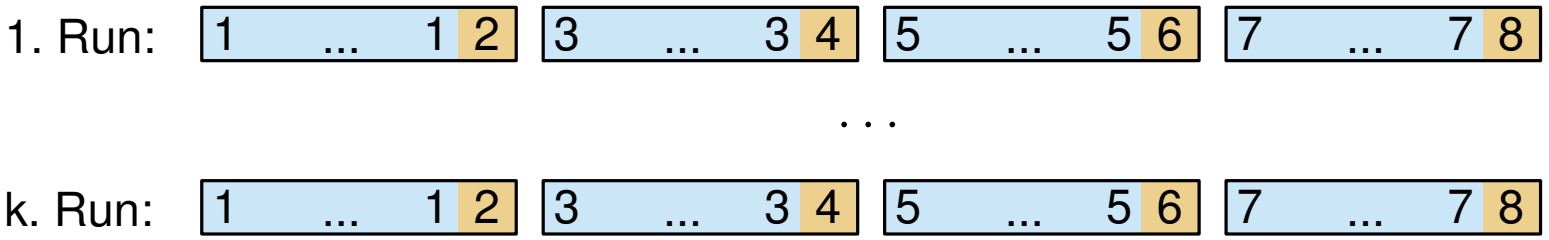
Minimum entfernen und Baum aktualisieren ist in $O(\log n)$ Zeit möglich.
Grundidee: Lasse neuen Wert im Baum aufsteigen und Wettkämpfe ausführen.



Institut für Theoretische Informatik
Prof. Dr. Dorothea Wagner

2. Phase: Merging

Schwierig den internen Aufwand abzuschätzen, der beim Vereinigen entsteht, da Lesen und Schreiben nicht vom Vereinigen getrennt sind. Betrachte hierzu folgende k identische Runs:



Ablauf:

- Nach Initialisierung der Merge-Buffers werden zuerst $k \cdot (B - 1)$ Werte '1' verarbeitet.
- Nach Verarbeitung des Wertes '2' für alle k Runs werden die nächsten k Blöcke geladen.
- Diese werden wieder zuerst verarbeitet, bevor weitere Blöcke geladen werden, usw.

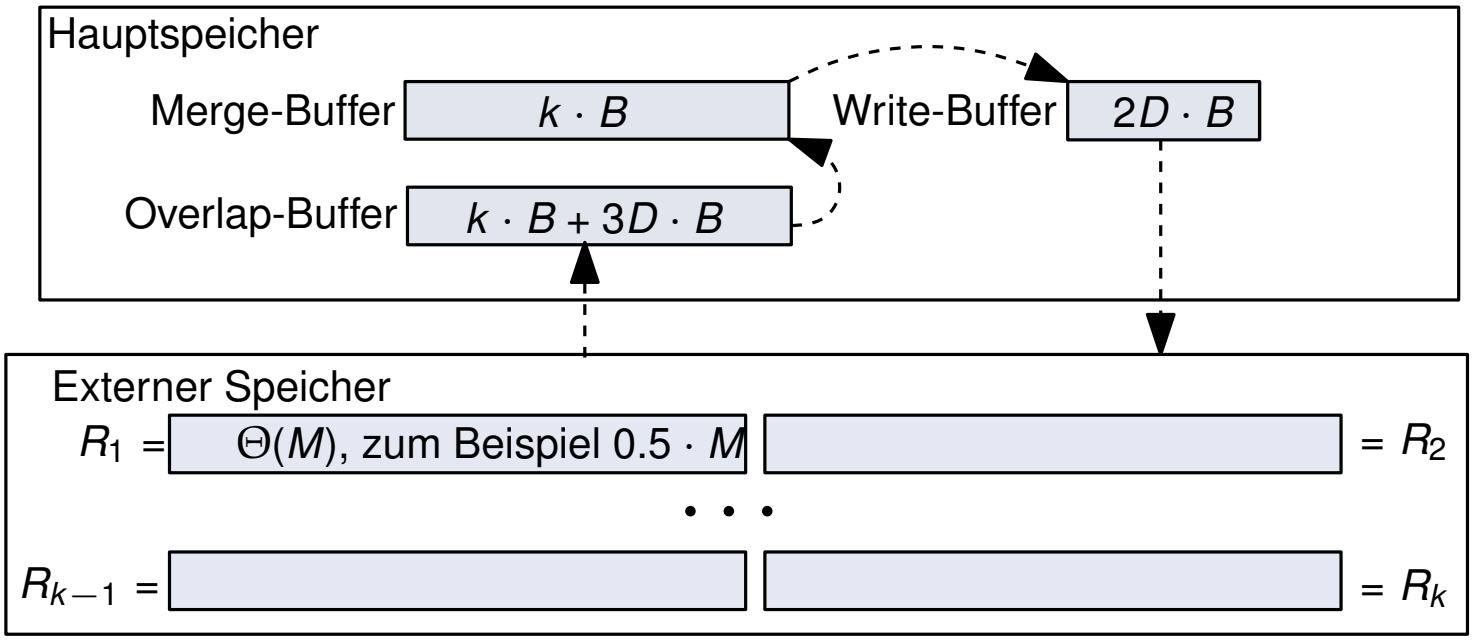
Verbesserung: Verwende wieder Overlapping-Technik, um I/O von Verarbeitung zu trennen:

- Thread *A*: Verantwortlich für I/O-Operationen.
- Thread *B*: Verantwortlich für das eigentliche Vermengen: Arbeitet ausschließlich auf dem Hauptspeicher.

2. Phase: Merging

Anpassungen:

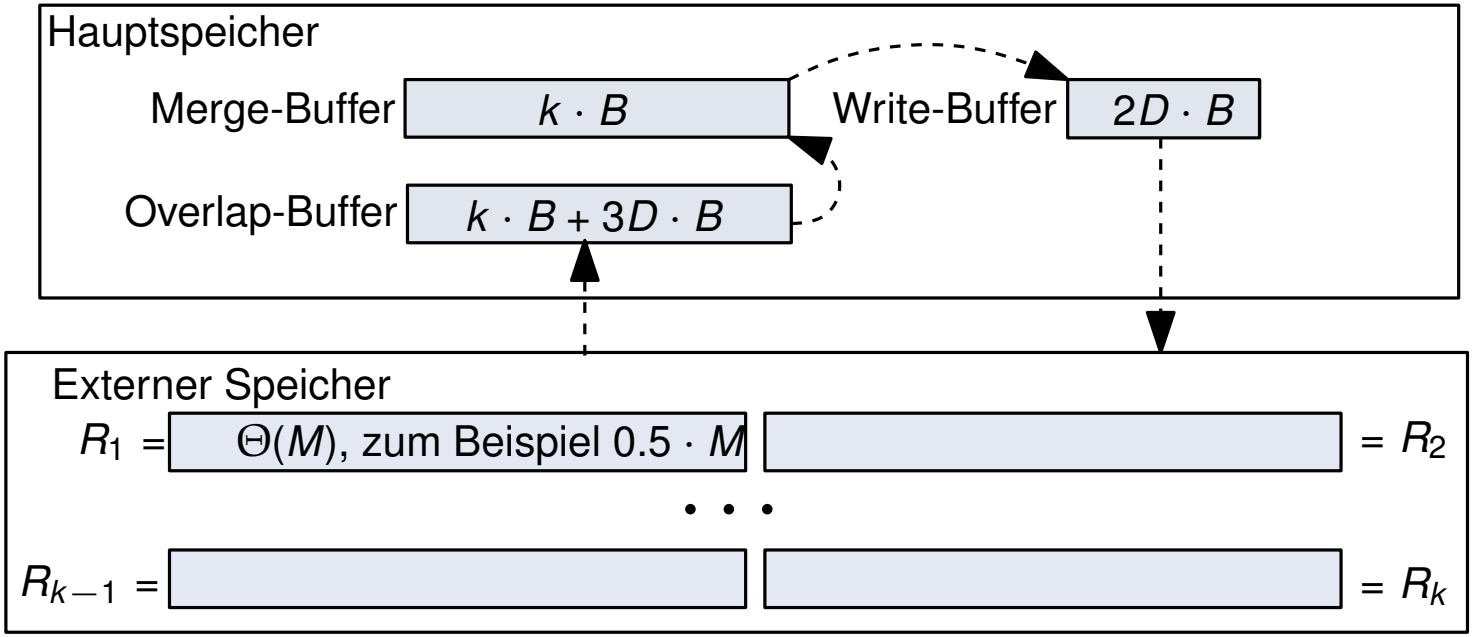
- Führe *Overlap-Buffer* ein, um Blöcke gepuffert lesen zu können.
- I/O-Thread (Thread A):
  1. Falls gerade kein I/O aktiv und mindestens $D \cdot B$ Elemente im Write-Buffer enthalten sind, dann schreibe Write-Buffer in den externen Speicher.
  2. Falls gerade kein I/O aktiv, weniger als D Blöcke im Write-Buffer sind und mindestens D Blöcke im Overlap-Buffer unbenutzt sind, dann lade die nächsten D Blöcke aus dem externen Speicher in den Overlap-Buffer.
- Merging-Thread (Thread B): Wie bisher, hole aber Daten aus dem Overlap-Buffer.



Institut für Theoretische Informatik
Prof. Dr. Dorothea Wagner

2. Phase: Merging

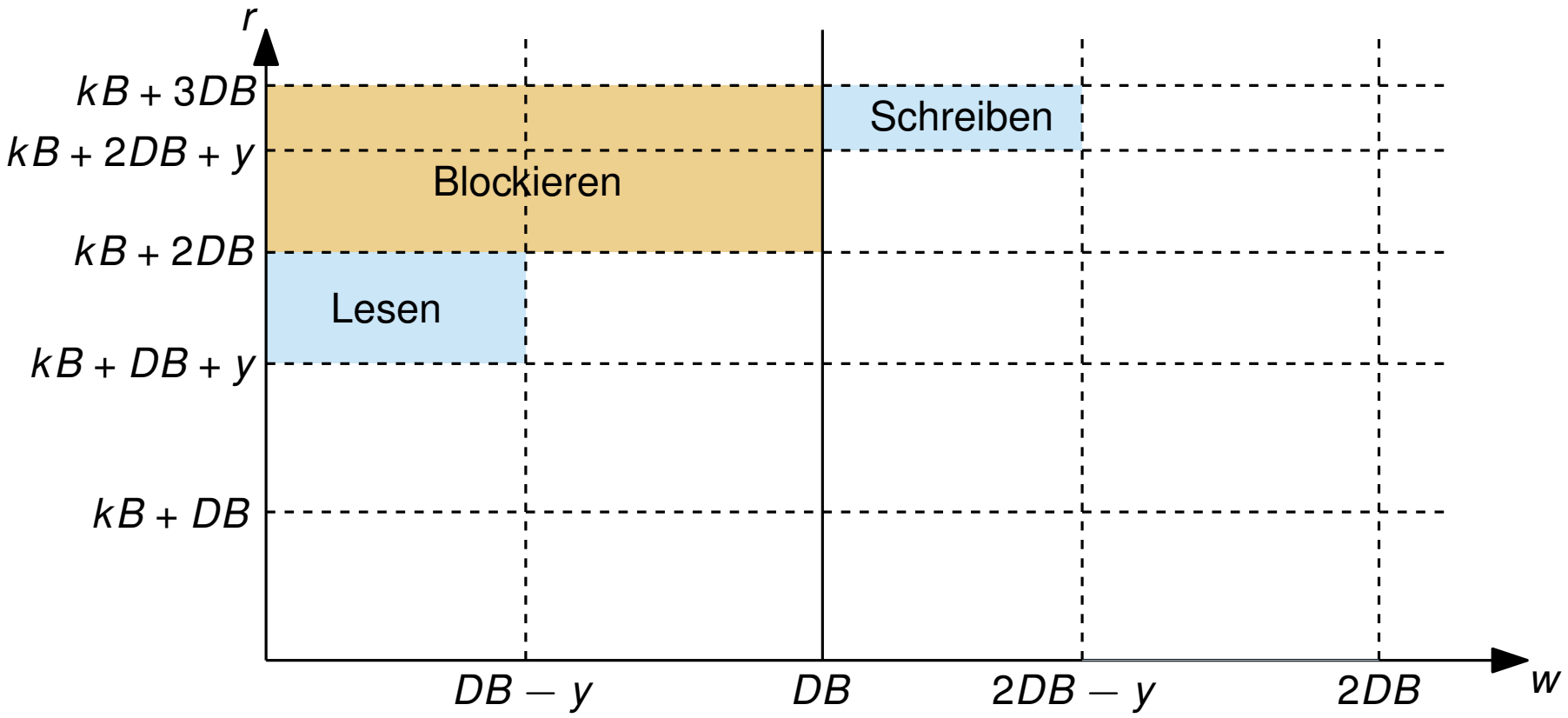
Lemma 5: Falls der Overlap-Buffer und der Merge-Buffer zusammen mindestens $k \cdot B$ Elemente enthalten, dann kann mindestens ein weiteres Element vom Merging-Thread verarbeitet werden, ohne dass ein neuer Block aus dem externen Speicher geladen werden muss.



Institut für Theoretische Informatik
Prof. Dr. Dorothea Wagner

2. Phase: Merging

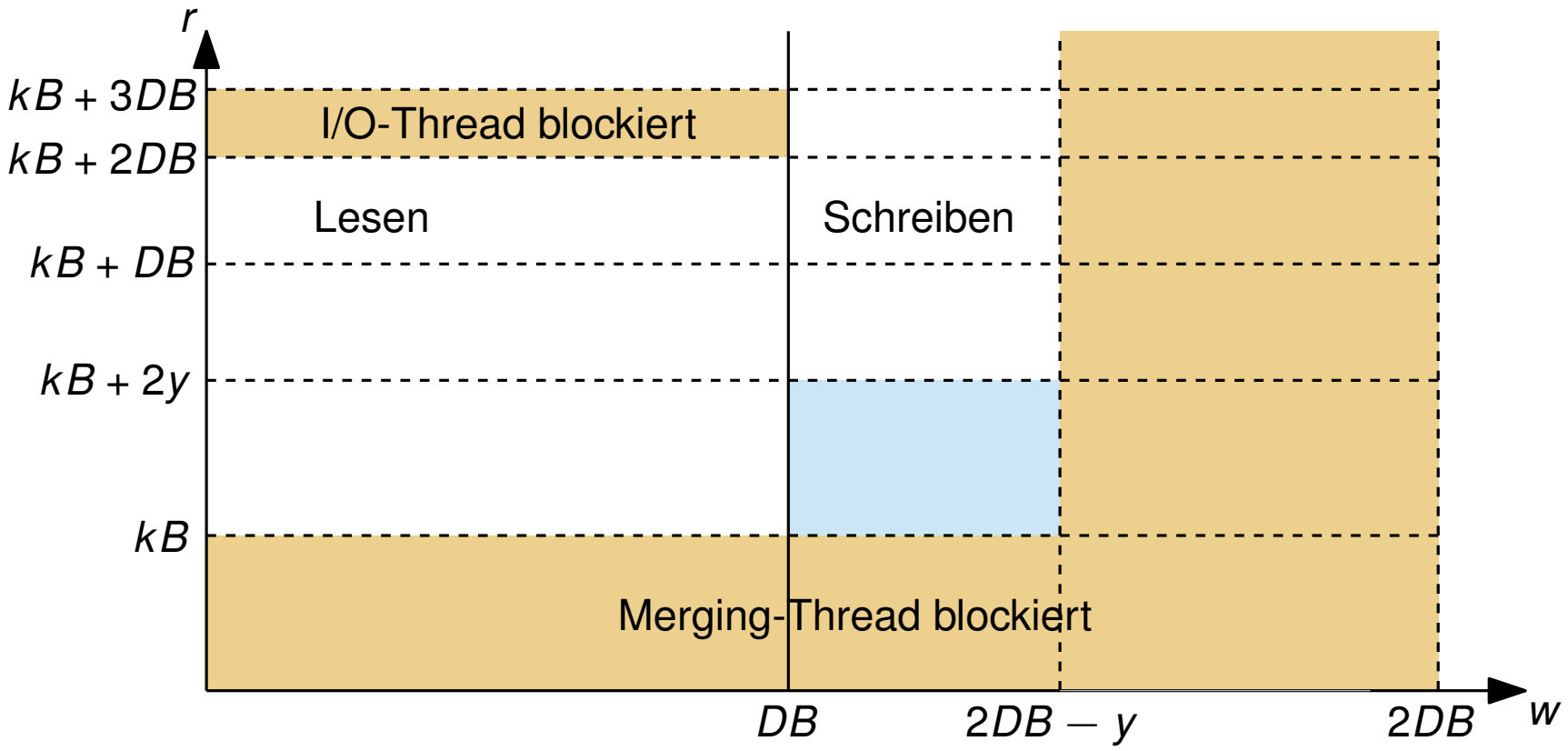
Lemma 6: Sei ℓ die Zeit, die der Merging-Thread benötigt um ein Element der Ausgabe zu erzeugen und sei L die Zeit, die gebraucht wird um D beliebige Blöcke zu laden/speichern.
Falls $2L \geq DB\ell$, dann blockiert (wartet) der I/O-Thread erst dann, wenn alle Eingabeblöcke gelesen worden sind.



Institut für Theoretische Informatik
Prof. Dr. Dorothea Wagner

2. Phase: Merging

Lemma 7: Sei ℓ die Zeit, die der Merging-Thread benötigt um ein Element der Ausgabe zu erzeugen und sei L die Zeit, die gebraucht wird um D beliebige Blöcke zu laden/speichern.
Falls $2L < DB\ell$, dann wird der Merging-Thread nach $\frac{k}{D}+1$ I/O-Schritten erst dann wieder blockieren, wenn alle Elemente vermengt worden sind.



Institut für Theoretische Informatik
Prof. Dr. Dorothea Wagner

2. Phase: Merging

Theorem 4: Sei ℓ die Zeit, die der Merging-Thread benötigt um ein Element der Ausgabe zu erzeugen und sei L die Zeit, die gebraucht wird um D beliebige Blöcke zu laden/speichern.
k sortierte Runs, die zusammen n Elemente enthalten, können in Zeit

$$\max\{\frac{2Ln}{DB}, \ell n\} + O(L\lceil \frac{k}{D} \rceil)$$

vereinigt werden.