

Algorithmen II

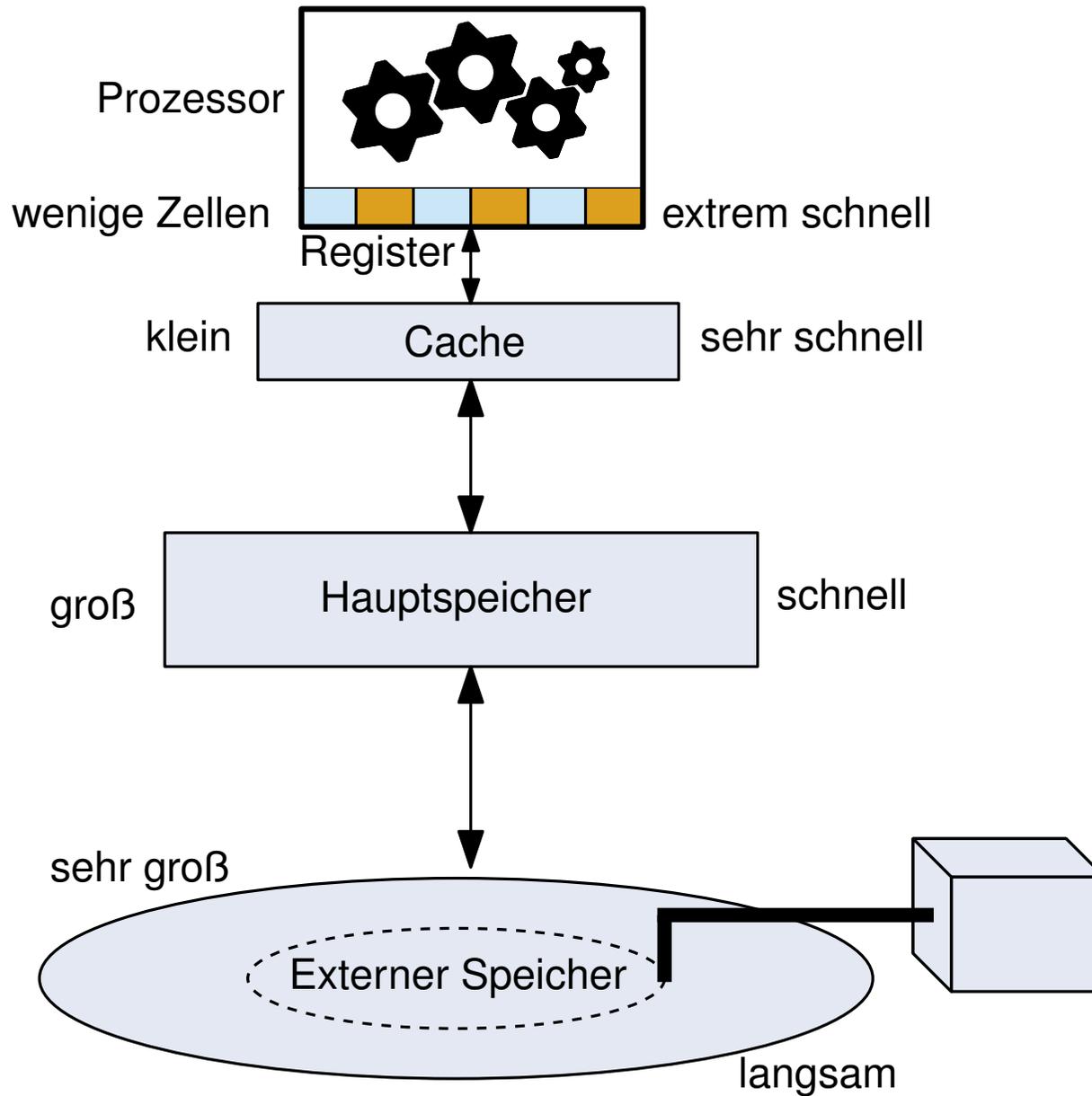
Vorlesung am 11.02.2014

Algorithmen für externen Speicher

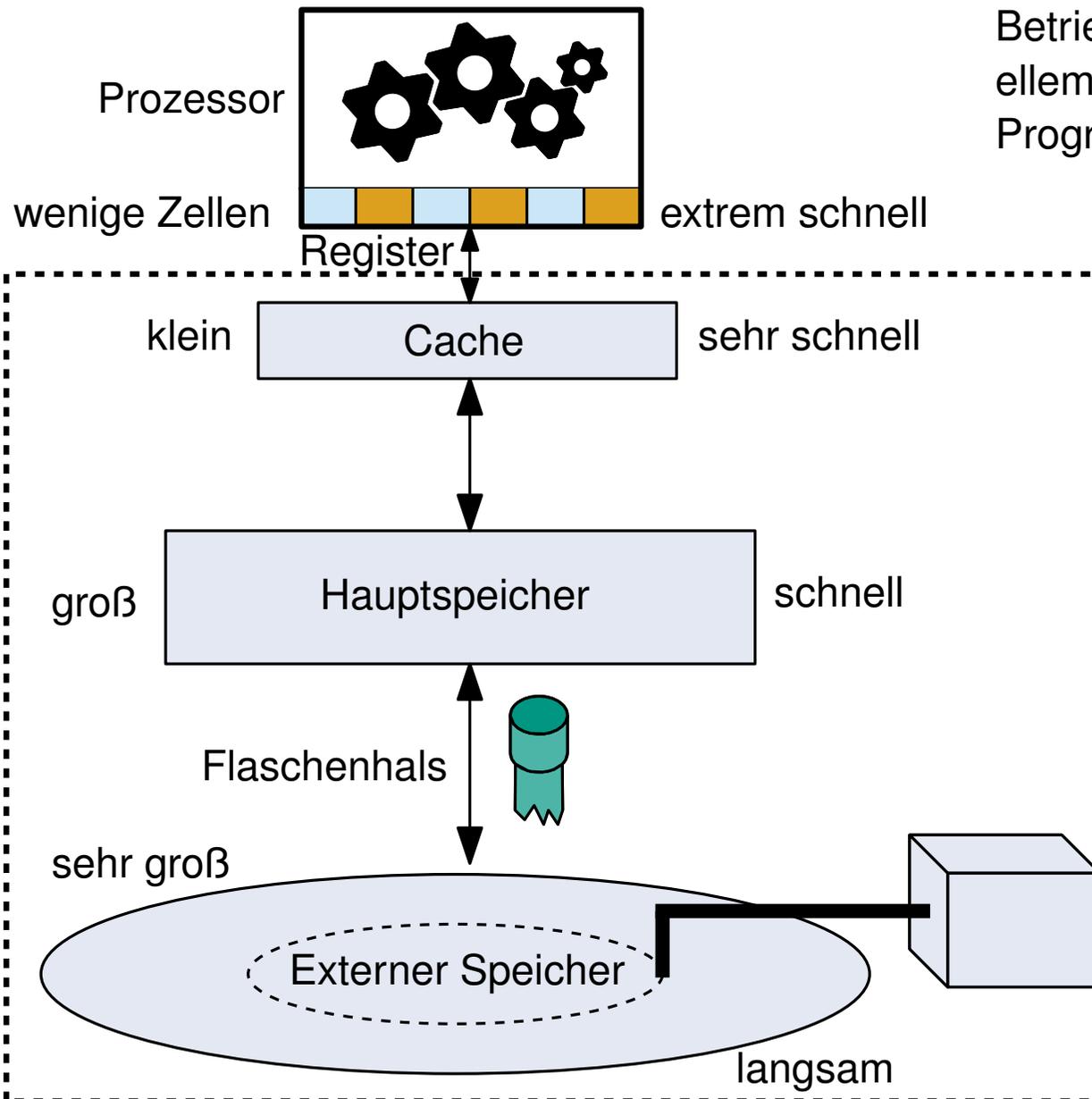
INSTITUT FÜR THEORETISCHE INFORMATIK · PROF. DR. DOROTHEA WAGNER



Einfaches Rechnermodell



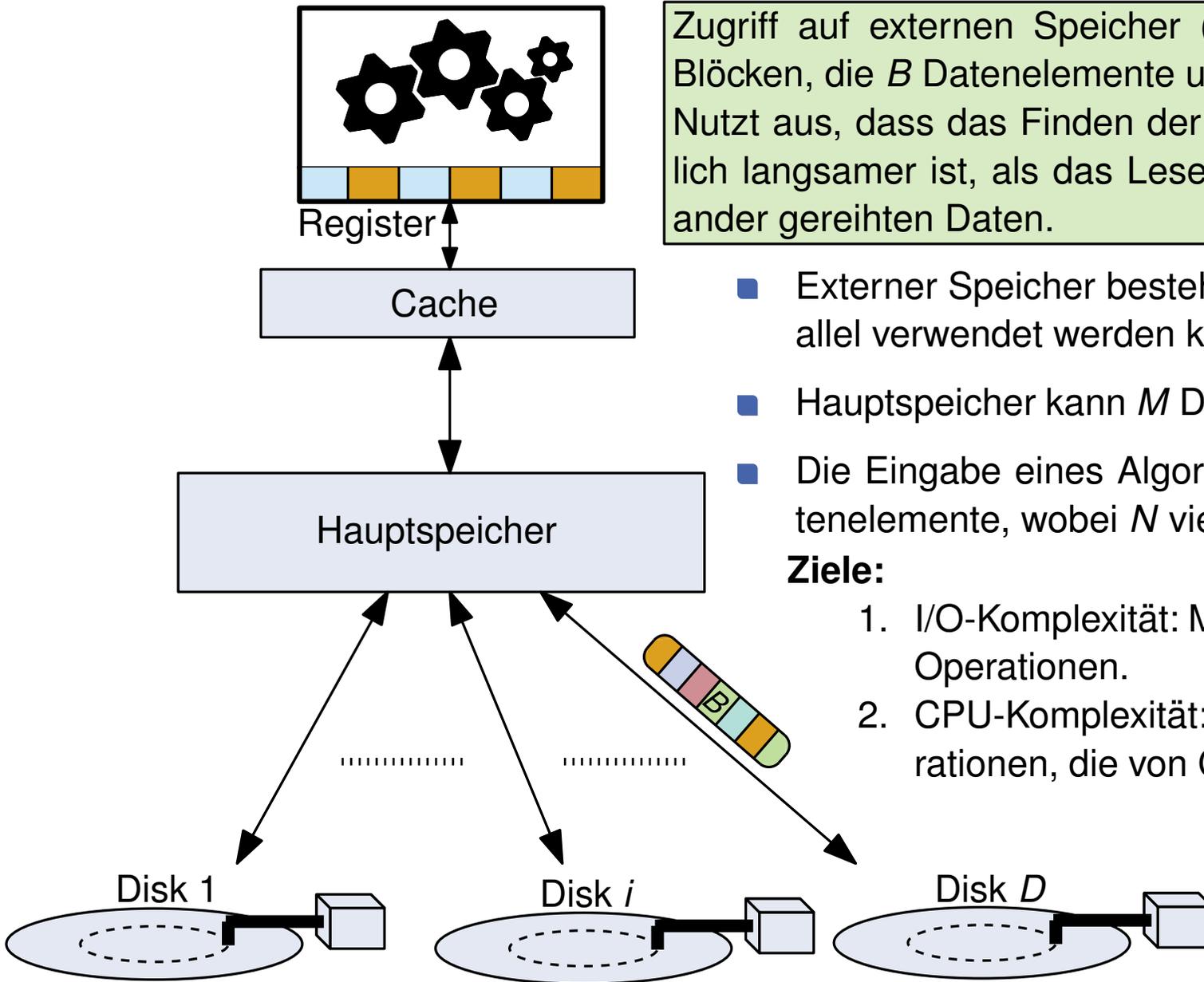
Einfaches Rechnermodell



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

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

Deshalb: Manchmal besser nicht die Abstraktion zu verwenden.



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 .

Ziele:

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

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!

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 $\text{scan}(N) := \Theta\left(\frac{N}{D \cdot B}\right)$ I/Os kosten.

Sortieren: Externes Sortieren von N Elementen ist mit $\text{sort}(N) := \Theta\left(\frac{N}{D \cdot B} \cdot \log_{\frac{M}{B}} \frac{N}{B}\right)$ 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 eines Zählers top implementiert werden. Hierzu initialisiere top mit -1 .

Operation $push(e)$:
Effekt: Legt Element e auf den Stack.

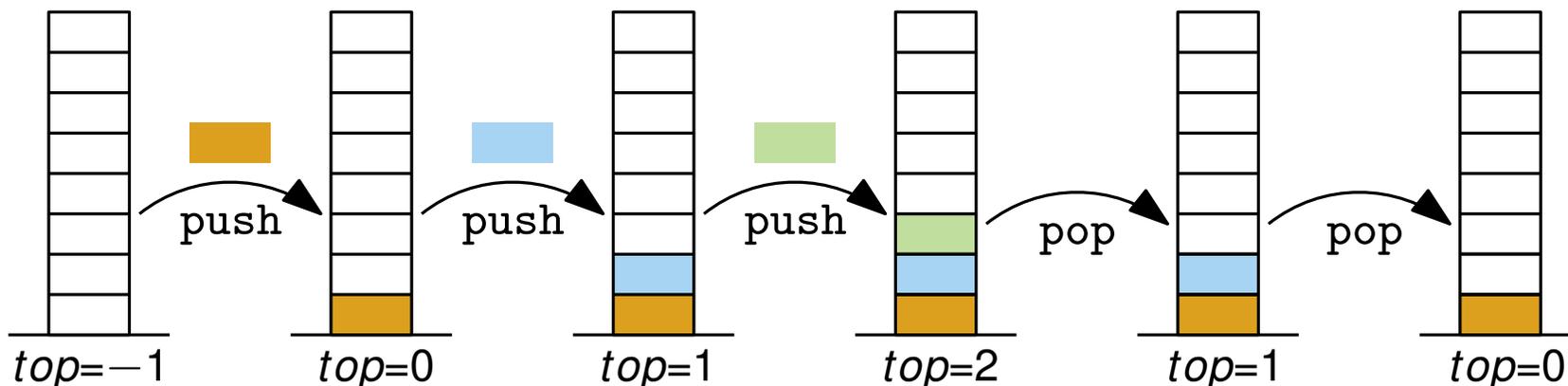
```
wenn  $top < m - 1$  dann  
|    $top \leftarrow top + 1$   
|    $S[top] \leftarrow e$ 
```

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

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

Operation $clear()$:
Effekt: Löscht alle Elemente vom Stack.

```
 $top \leftarrow -1$ 
```



Externer Stack

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

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

wenn S voll dann

| Kopiere S in externen Speicher.

| `S.clear()`.

`S.push(e)`

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.

Externer Stack

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

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

wenn S voll dann

 Kopiere S in externen Speicher.

`S.clear()`.

`S.push(e)`

Operation `externalPop()`:

Effekt: Entfernt das oberste Element vom externen Stack.

wenn S leer dann

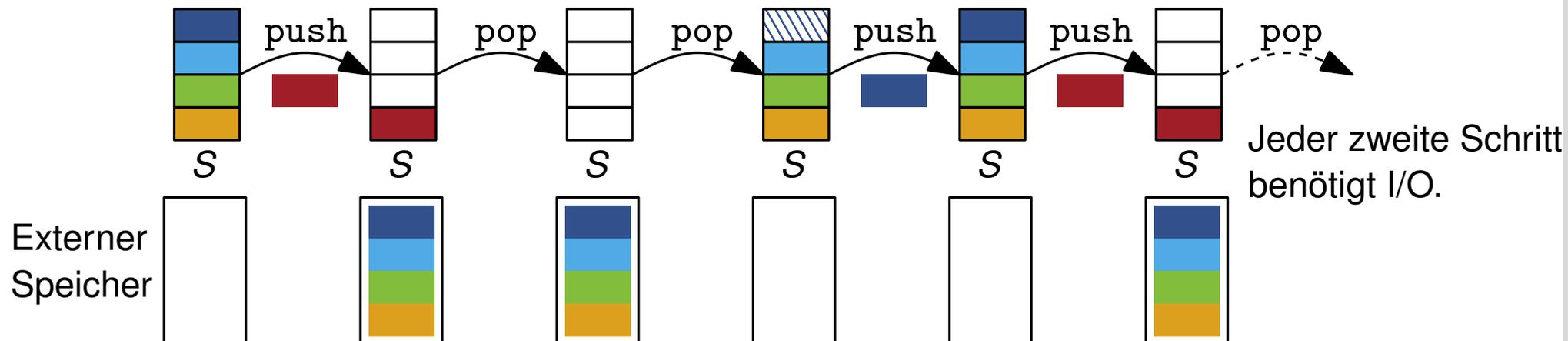
 Kopiere die B zuletzt geschriebenen

 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(e)`:

Effekt: Hängt e an externe Warteschlange.

wenn S_1 voll dann

Kopiere S_1 in externen Speicher in
verkehrter Reihenfolge.

`S_1 .clear()`

`S_1 .push(e)`

Operation `remove()`:

Effekt: Entfernt erstes Element aus externer Warteschlange.

wenn S_2 leer und externer Speicher nicht leer dann

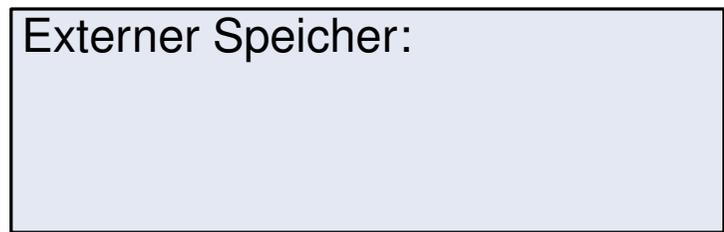
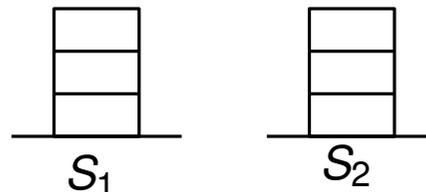
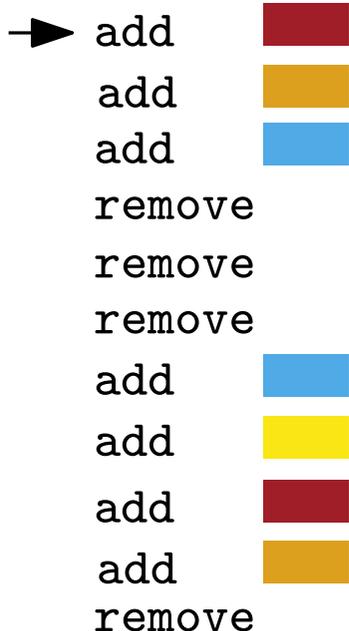
Kopiere die B zuerst geschriebenen
Elemente aus dem externen Speicher
in S_2 .

sonst wenn S_2 leer dann

Kopiere S_1 in S_2 in verkehrter
Reihenfolge.

`S_1 .clear()`

`S_2 .pop()`



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(e)`:

Effekt: Hängt e an externe Warteschlange.

wenn S_1 voll dann

Kopiere S_1 in externen Speicher in
verkehrter Reihenfolge.

`S_1 .clear()`

`S_1 .push(e)`

Operation `remove()`:

Effekt: Entfernt erstes Element aus externer Warteschlange.

wenn S_2 leer und externer Speicher nicht leer dann

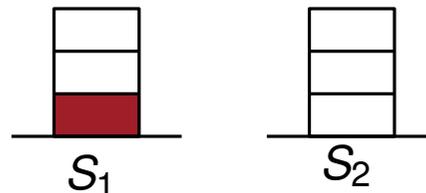
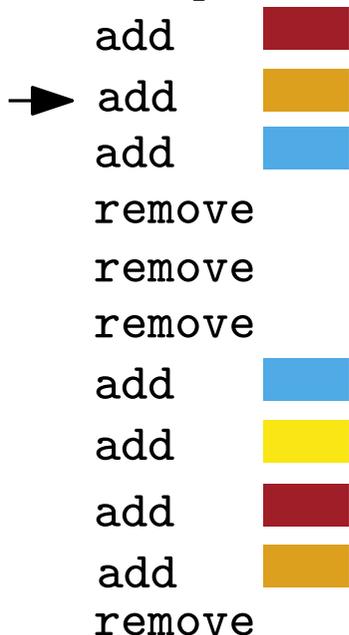
Kopiere die B zuerst geschriebenen
Elemente aus dem externen Speicher
in S_2 .

sonst wenn S_2 leer dann

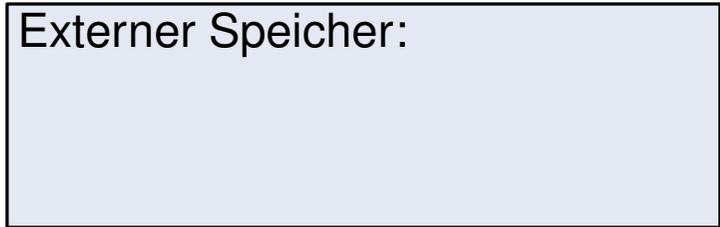
Kopiere S_1 in S_2 in verkehrter
Reihenfolge.

`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(e)`:

Effekt: Hängt e an externe Warteschlange.

wenn S_1 voll dann

Kopiere S_1 in externen Speicher in
verkehrter Reihenfolge.

`S_1 .clear()`

`S_1 .push(e)`

Operation `remove()`:

Effekt: Entfernt erstes Element aus externer Warteschlange.

wenn S_2 leer und externer Speicher nicht leer dann

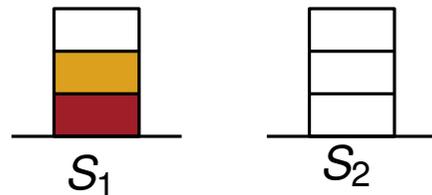
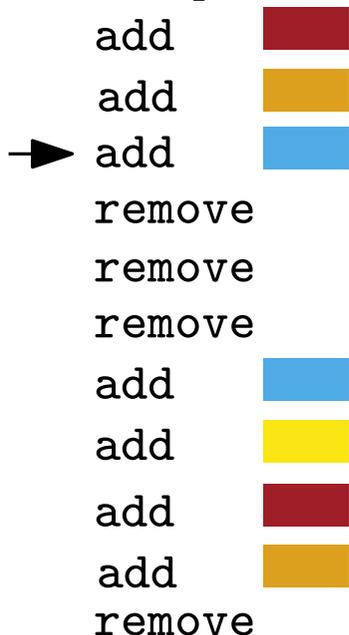
Kopiere die B zuerst geschriebenen
Elemente aus dem externen Speicher
in S_2 .

sonst wenn S_2 leer dann

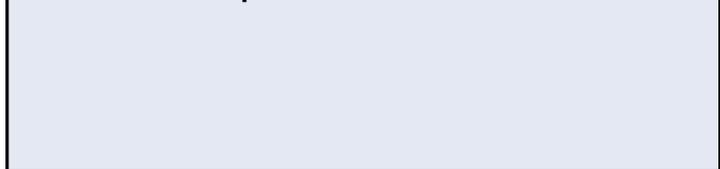
Kopiere S_1 in S_2 in verkehrter
Reihenfolge.

`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(e)`:

Effekt: Hängt e an externe Warteschlange.

wenn S_1 voll dann

Kopiere S_1 in externen Speicher in
verkehrter Reihenfolge.

`S_1 .clear()`

`S_1 .push(e)`

Operation `remove()`:

Effekt: Entfernt erstes Element aus externer Warteschlange.

wenn S_2 leer und externer Speicher nicht leer dann

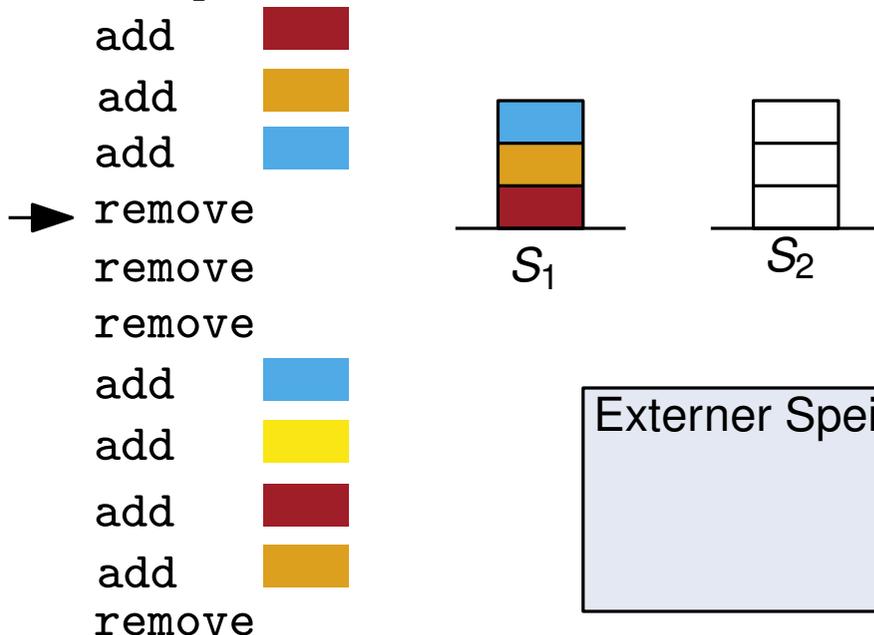
Kopiere die B zuerst geschriebenen
Elemente aus dem externen Speicher
in S_2 .

sonst wenn S_2 leer dann

Kopiere S_1 in S_2 in verkehrter
Reihenfolge.

`S_1 .clear()`

`S_2 .pop()`



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(e)`:

Effekt: Hängt e an externe Warteschlange.

wenn S_1 voll dann

Kopiere S_1 in externen Speicher in
verkehrter Reihenfolge.

`S_1 .clear()`

`S_1 .push(e)`

Operation `remove()`:

Effekt: Entfernt erstes Element aus externer Warteschlange.

wenn S_2 leer und externer Speicher nicht leer dann

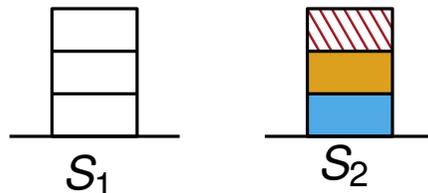
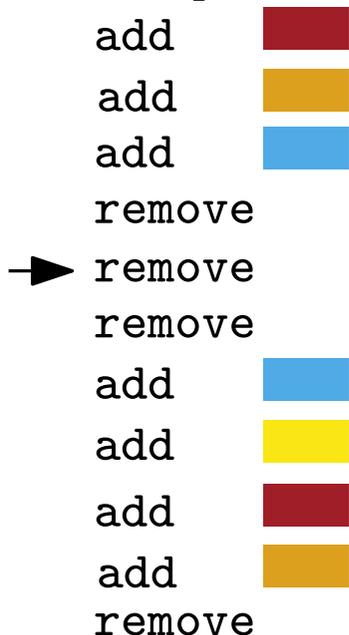
Kopiere die B zuerst geschriebenen
Elemente aus dem externen Speicher
in S_2 .

sonst wenn S_2 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(e)`:

Effekt: Hängt e an externe Warteschlange.

wenn S_1 voll dann

Kopiere S_1 in externen Speicher in
verkehrter Reihenfolge.

`S_1 .clear()`

`S_1 .push(e)`

Operation `remove()`:

Effekt: Entfernt erstes Element aus externer Warteschlange.

wenn S_2 leer und externer Speicher nicht leer dann

Kopiere die B zuerst geschriebenen
Elemente aus dem externen Speicher
in S_2 .

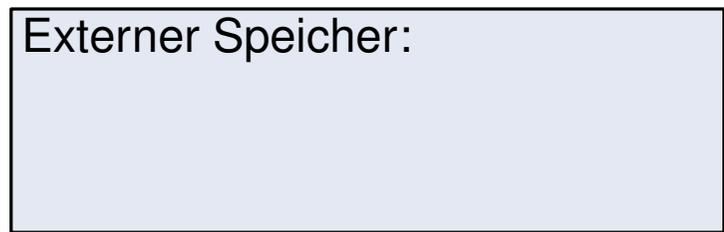
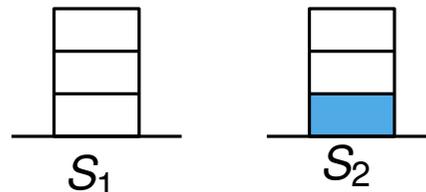
sonst wenn S_2 leer dann

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



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 $\text{add}(e)$:

Effekt: Hängt e an externe Warteschlange.

wenn S_1 voll dann

Kopiere S_1 in externen Speicher in
verkehrter Reihenfolge.

$S_1.\text{clear}()$

$S_1.\text{push}(e)$

Operation $\text{remove}()$:

Effekt: Entfernt erstes Element aus externer Warteschlange.

wenn S_2 leer und externer Speicher nicht leer dann

Kopiere die B zuerst geschriebenen
Elemente aus dem externen Speicher
in S_2 .

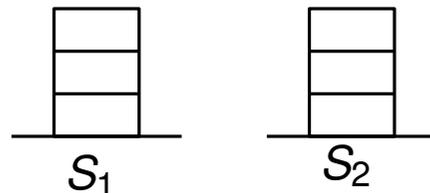
sonst wenn S_2 leer dann

Kopiere S_1 in S_2 in verkehrter
Reihenfolge.

$S_1.\text{clear}()$

$S_2.\text{pop}()$

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(e)`:

Effekt: Hängt e an externe Warteschlange.

wenn S_1 voll dann

Kopiere S_1 in externen Speicher in
verkehrter Reihenfolge.

`S_1 .clear()`

`S_1 .push(e)`

Operation `remove()`:

Effekt: Entfernt erstes Element aus externer Warteschlange.

wenn S_2 leer und externer Speicher nicht leer dann

Kopiere die B zuerst geschriebenen
Elemente aus dem externen Speicher
in S_2 .

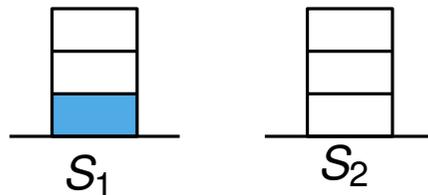
sonst wenn S_2 leer dann

Kopiere S_1 in S_2 in verkehrter
Reihenfolge.

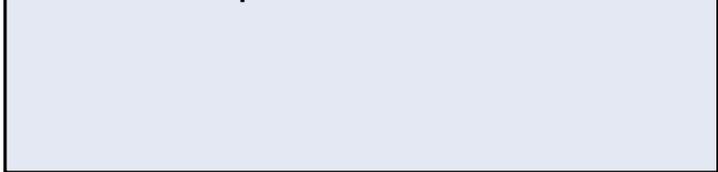
`S_1 .clear()`

`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(e)`:

Effekt: Hängt e an externe Warteschlange.

wenn S_1 voll dann

Kopiere S_1 in externen Speicher in
verkehrter Reihenfolge.

`S_1 .clear()`

`S_1 .push(e)`

Operation `remove()`:

Effekt: Entfernt erstes Element aus externer Warteschlange.

wenn S_2 leer und externer Speicher nicht leer dann

Kopiere die B zuerst geschriebenen
Elemente aus dem externen Speicher
in S_2 .

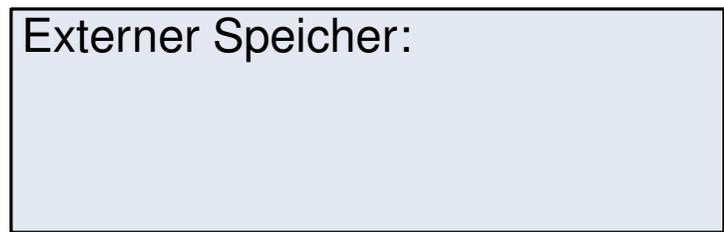
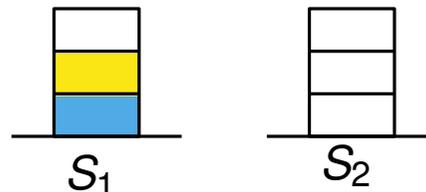
sonst wenn S_2 leer dann

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



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(e)`:

Effekt: Hängt e an externe Warteschlange.

wenn S_1 voll dann

Kopiere S_1 in externen Speicher in
verkehrter Reihenfolge.

`S_1 .clear()`

`S_1 .push(e)`

Operation `remove()`:

Effekt: Entfernt erstes Element aus externer Warteschlange.

wenn S_2 leer und externer Speicher nicht leer dann

Kopiere die B zuerst geschriebenen
Elemente aus dem externen Speicher
in S_2 .

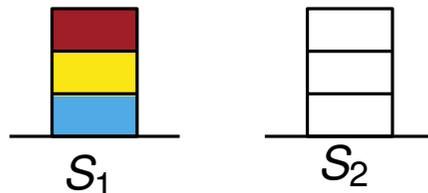
sonst wenn S_2 leer dann

Kopiere S_1 in S_2 in verkehrter
Reihenfolge.

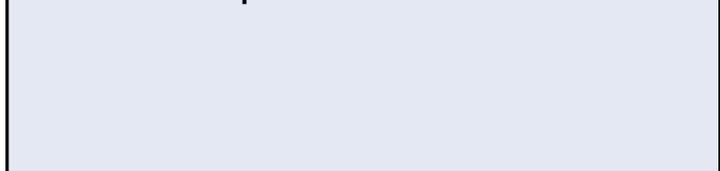
`S_1 .clear()`

`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(e)`:

Effekt: Hängt e an externe Warteschlange.

wenn S_1 voll dann

Kopiere S_1 in externen Speicher in
verkehrter Reihenfolge.

`S_1 .clear()`

`S_1 .push(e)`

Operation `remove()`:

Effekt: Entfernt erstes Element aus externer Warteschlange.

wenn S_2 leer und externer Speicher nicht leer dann

Kopiere die B zuerst geschriebenen
Elemente aus dem externen Speicher
in S_2 .

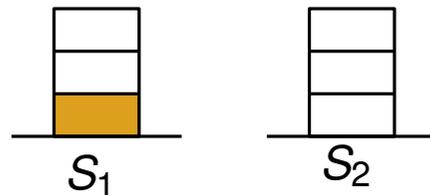
sonst wenn S_2 leer dann

Kopiere S_1 in S_2 in verkehrter
Reihenfolge.

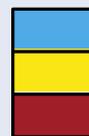
`S_1 .clear()`

`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(e)`:

Effekt: Hängt e an externe Warteschlange.

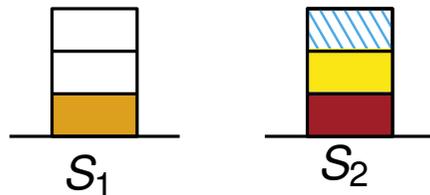
wenn S_1 voll dann

Kopiere S_1 in externen Speicher in
verkehrter Reihenfolge.

`S_1 .clear()`

`S_1 .push(e)`

add 
 add 
 add 
 remove
 remove
 remove
 add 
 add 
 add 
 add 
 remove



Operation `remove()`:

Effekt: Entfernt erstes Element aus externer Warteschlange.

wenn S_2 leer und externer Speicher nicht leer dann

Kopiere die B zuerst geschriebenen
Elemente aus dem externen Speicher
in S_2 .

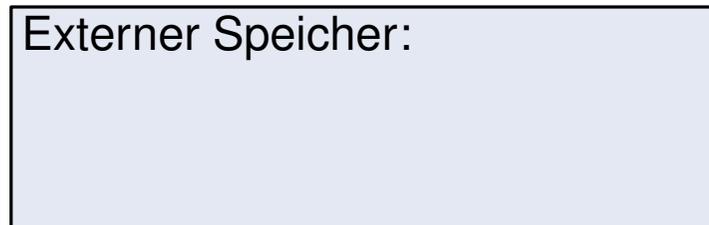
sonst wenn S_2 leer dann

Kopiere S_1 in S_2 in verkehrter
Reihenfolge.

`S_1 .clear()`

`S_2 .pop()`

Externer Speicher:



Multiway Merge Sort

Multiway Merge Sort

Erinnerung: Prinzip von Merge Sort

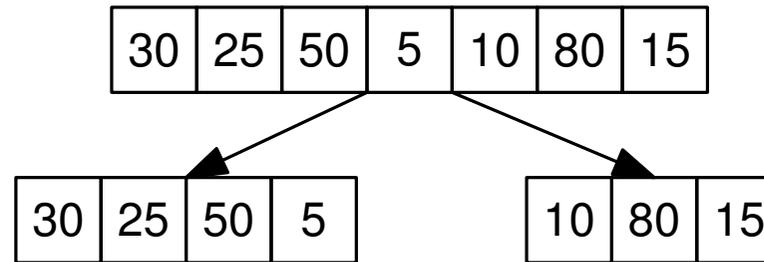
Laufzeit: $O(n \log n)$

30	25	50	5	10	80	15
----	----	----	---	----	----	----

Multiway Merge Sort

Erinnerung: Prinzip von Merge Sort

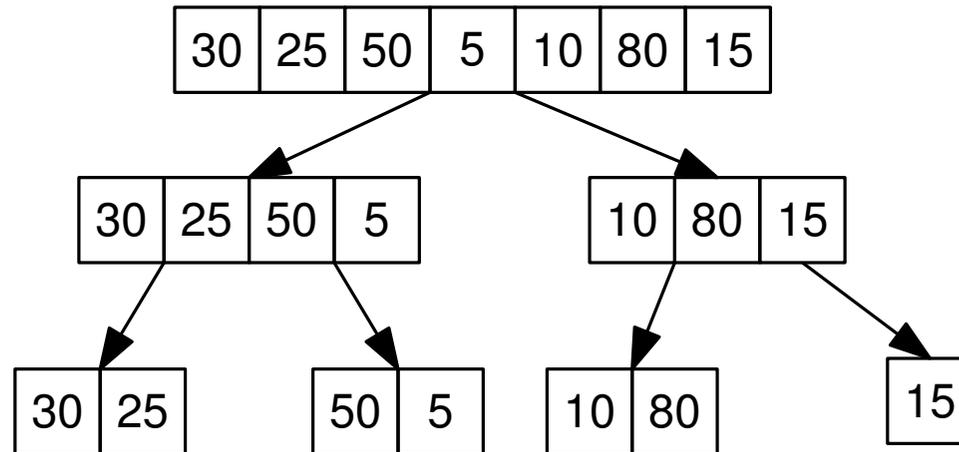
Laufzeit: $O(n \log n)$



Multiway Merge Sort

Erinnerung: Prinzip von Merge Sort

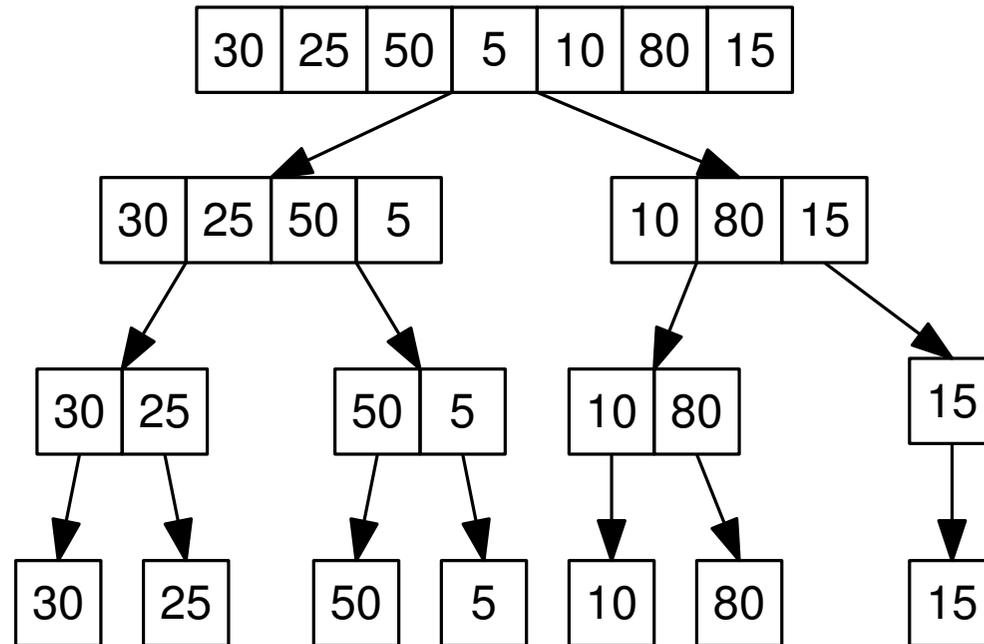
Laufzeit: $O(n \log n)$



Multiway Merge Sort

Erinnerung: Prinzip von Merge Sort

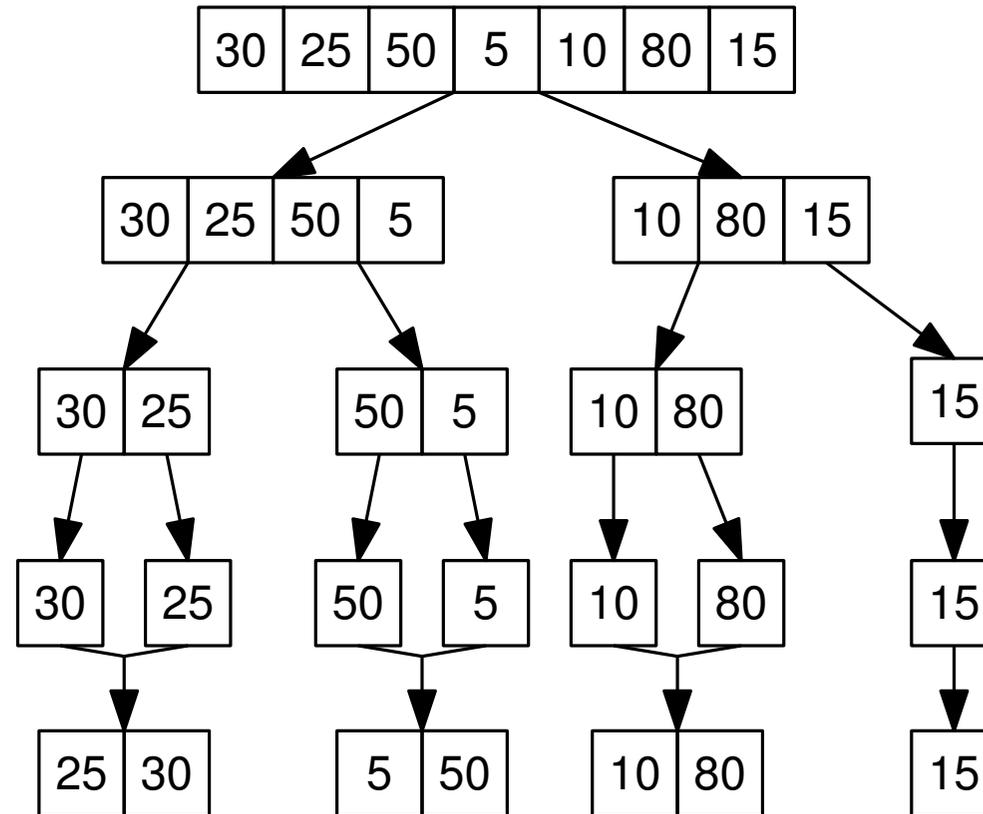
Laufzeit: $O(n \log n)$



Multiway Merge Sort

Erinnerung: Prinzip von Merge Sort

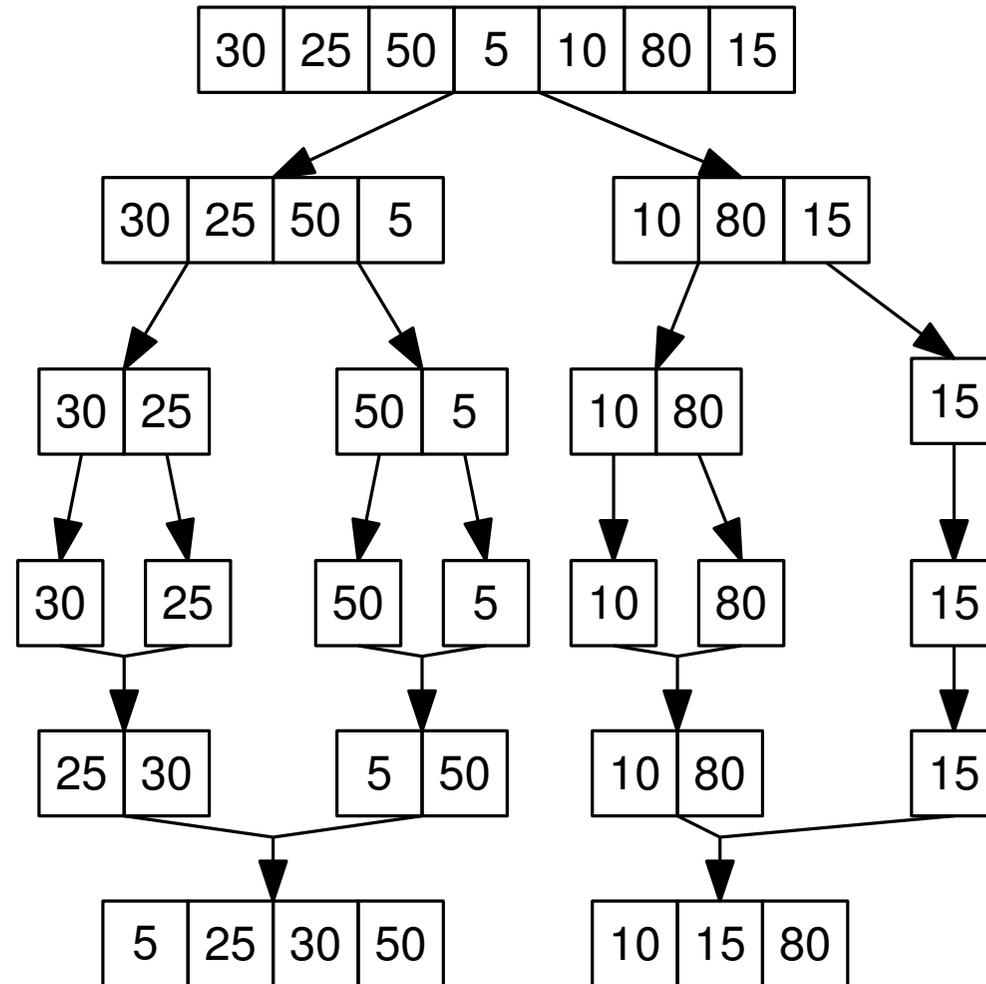
Laufzeit: $O(n \log n)$



Multiway Merge Sort

Erinnerung: Prinzip von Merge Sort

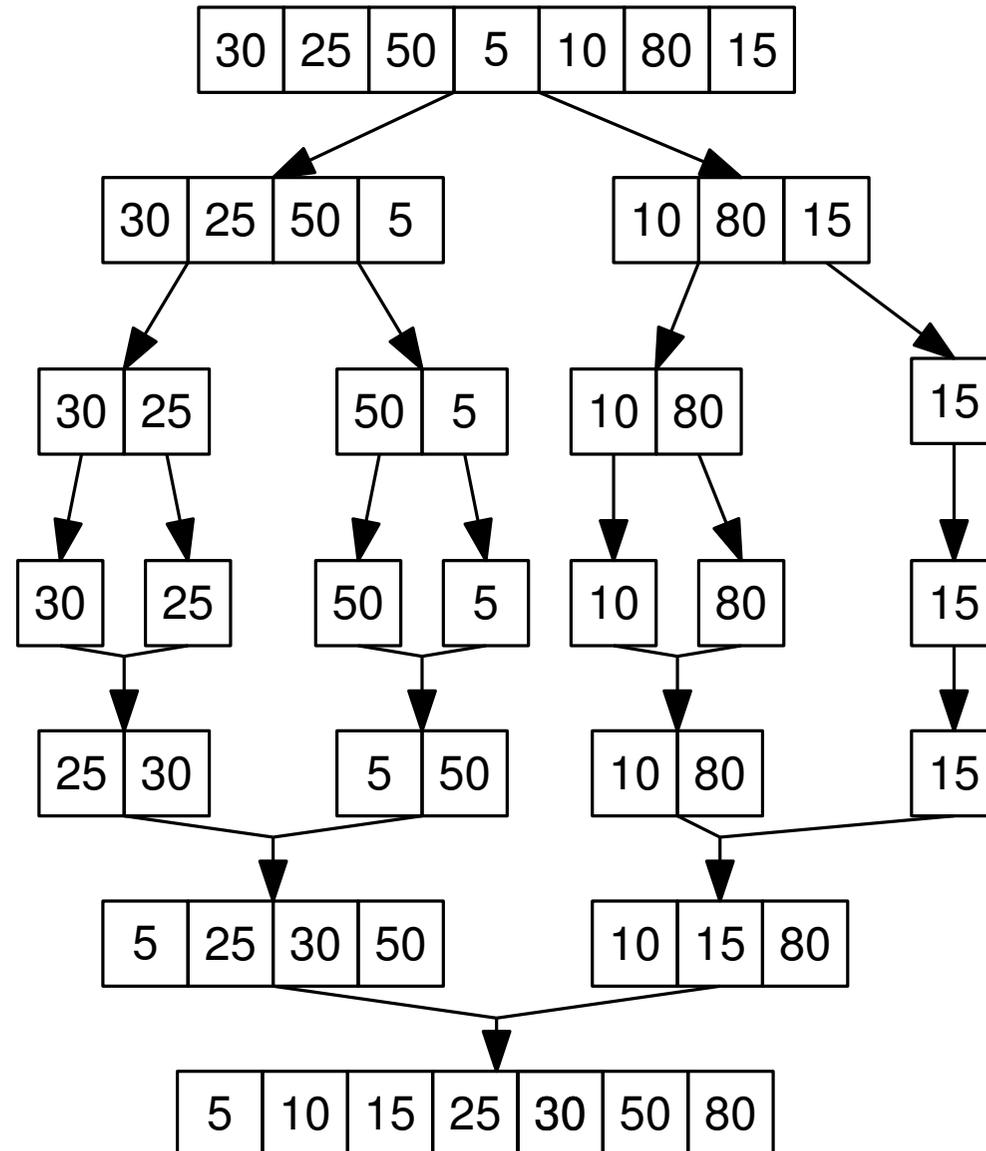
Laufzeit: $O(n \log n)$



Multiway Merge Sort

Erinnerung: Prinzip von Merge Sort

Laufzeit: $O(n \log n)$



Multiway Merge Sort

Eingabe: a_1, \dots, a_n Elemente, die im externen Speicher liegen.

Ausgabe: Sortierung von a_1, \dots, a_n .

1. Phase: Run Formation

1. Teile a_1, \dots, a_n in m Gruppen G_1, \dots, 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 $\mathcal{R} = \{R_1, \dots, R_m\}$ 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, \dots, a_n in m Gruppen G_1, \dots, 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 $\mathcal{R} = \{R_1, \dots, R_m\}$ sortierte *Runs*.

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

Hauptspeicher

Externer Speicher

$G_1 =$

m	a	k	e	*	t	h	i	n	g	s	*
---	---	---	---	---	---	---	---	---	---	---	---

a	s	*	s	i	m	p	l	e	*	a	s
---	---	---	---	---	---	---	---	---	---	---	---

 $= G_2$

$G_3 =$

*	p	o	s	s	i	b	l	e	*	b	u
---	---	---	---	---	---	---	---	---	---	---	---

t	*	n	o	*	s	i	m	p	l	e	r
---	---	---	---	---	---	---	---	---	---	---	---

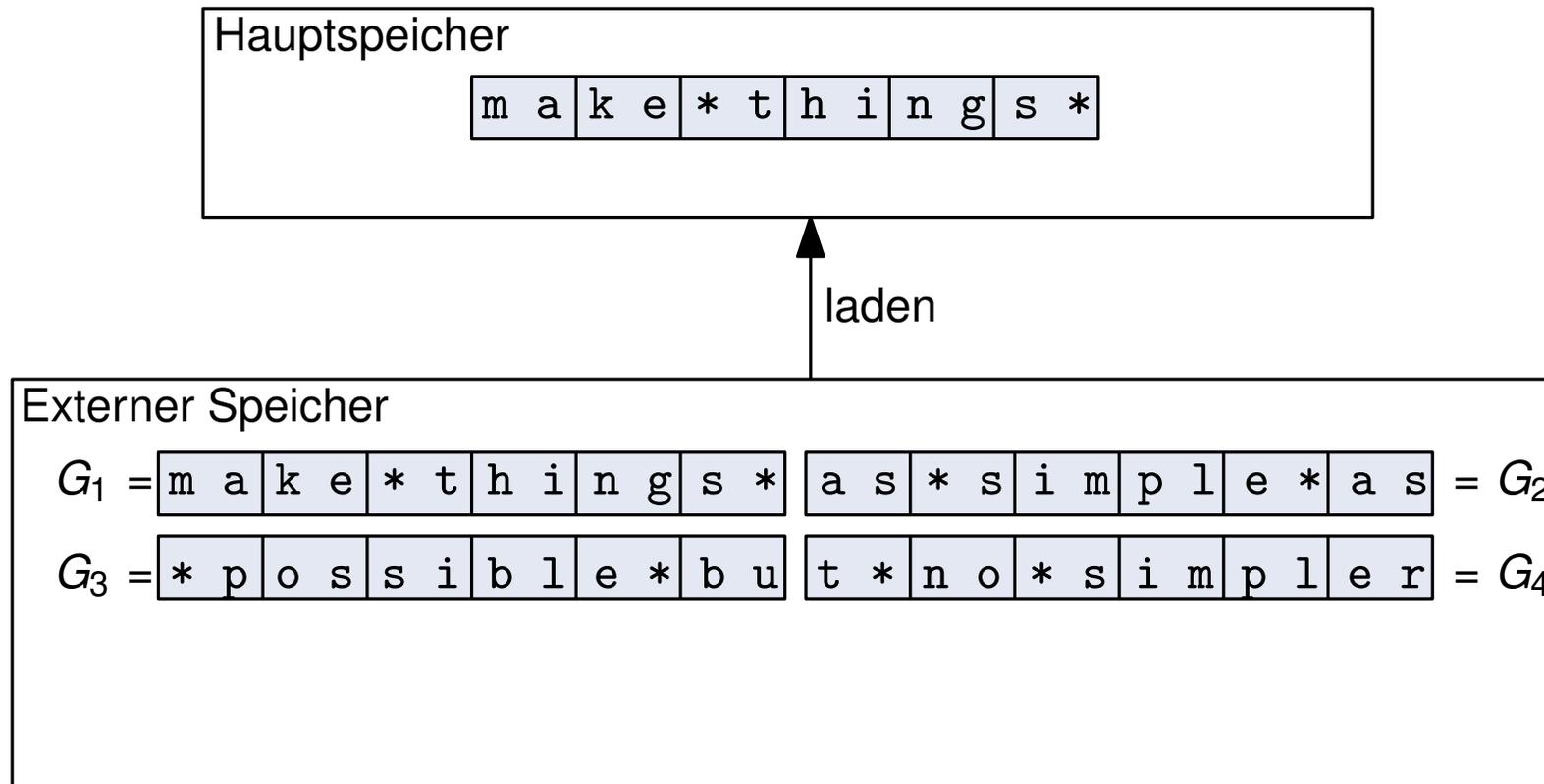
 $= G_4$

1. Phase: Run Formation

1. Phase: Run Formation

1. Teile a_1, \dots, a_n in m Gruppen G_1, \dots, 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 $\mathcal{R} = \{R_1, \dots, R_m\}$ sortierte *Runs*.

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

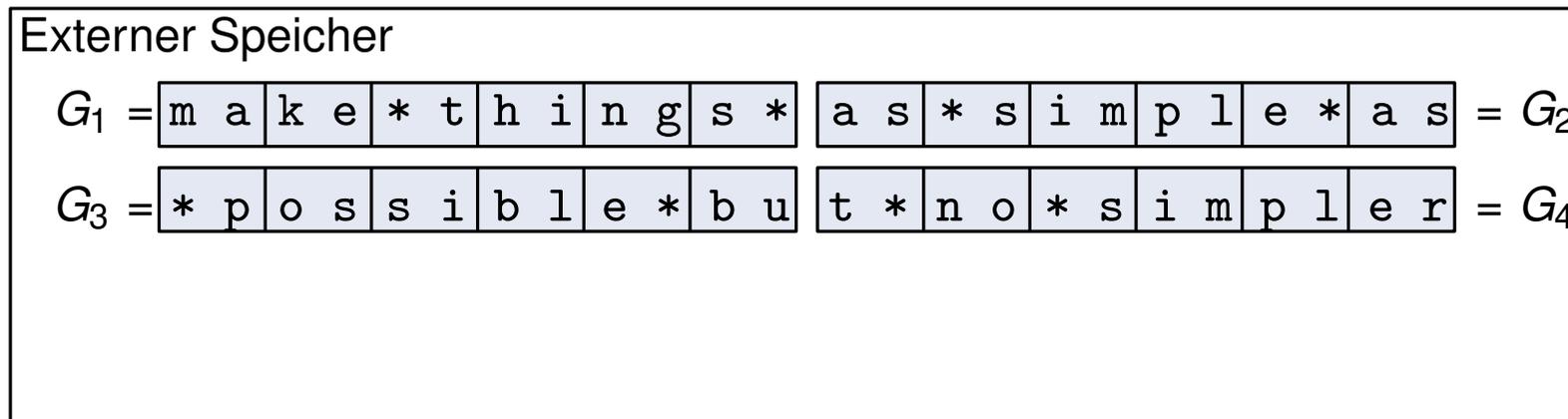
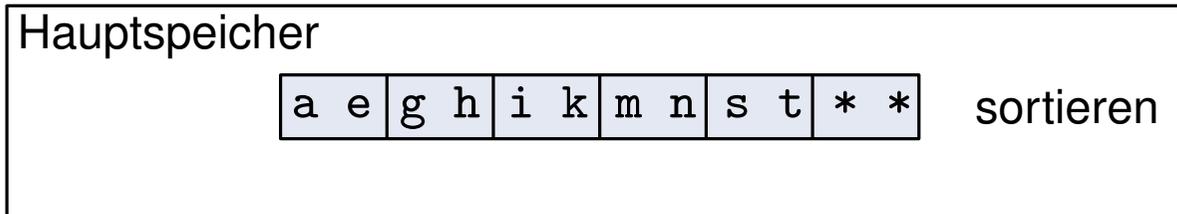


1. Phase: Run Formation

1. Phase: Run Formation

1. Teile a_1, \dots, a_n in m Gruppen G_1, \dots, 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 $\mathcal{R} = \{R_1, \dots, R_m\}$ sortierte *Runs*.

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

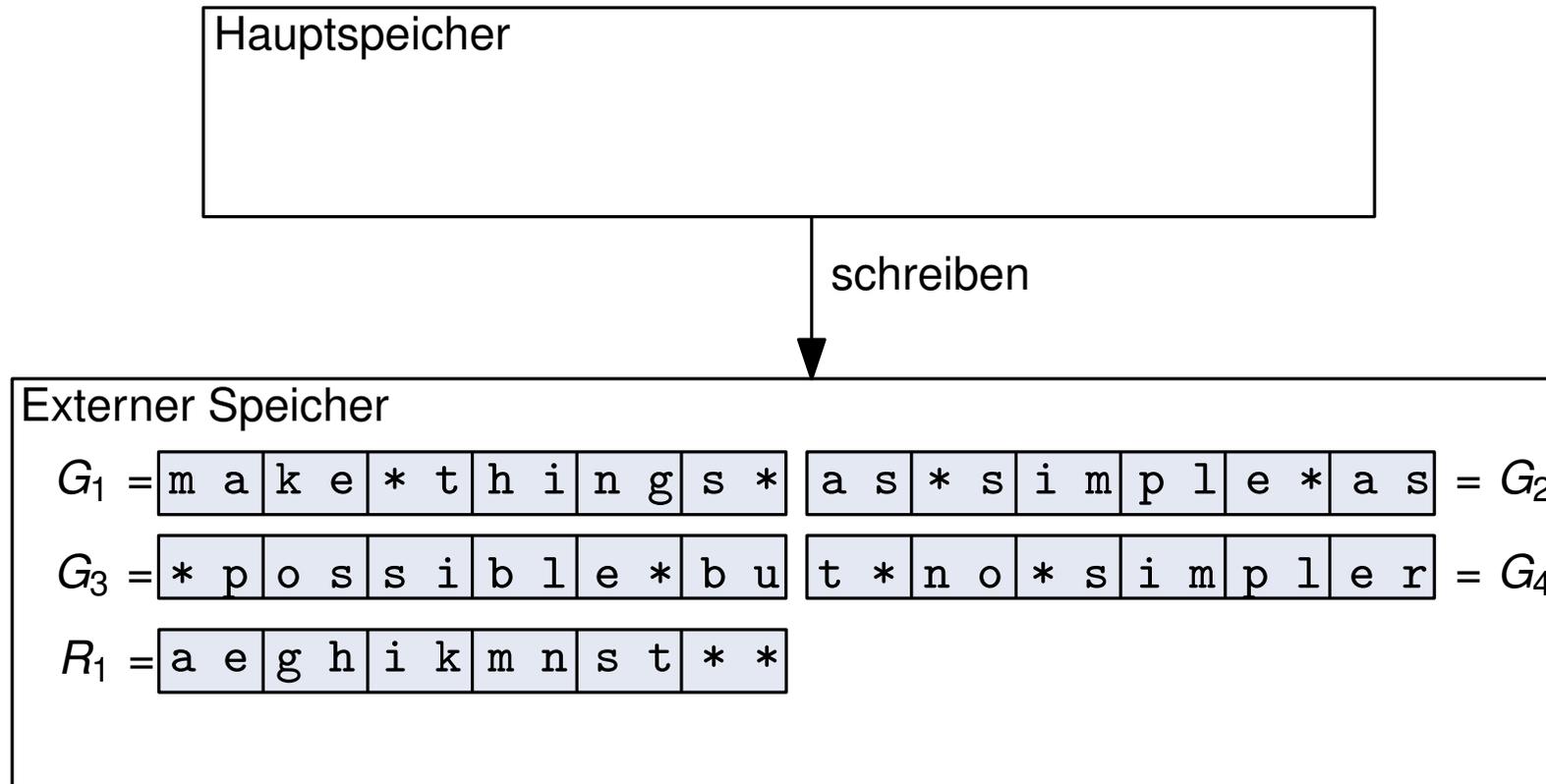


1. Phase: Run Formation

1. Phase: Run Formation

1. Teile a_1, \dots, a_n in m Gruppen G_1, \dots, 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 $\mathcal{R} = \{R_1, \dots, R_m\}$ sortierte *Runs*.

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

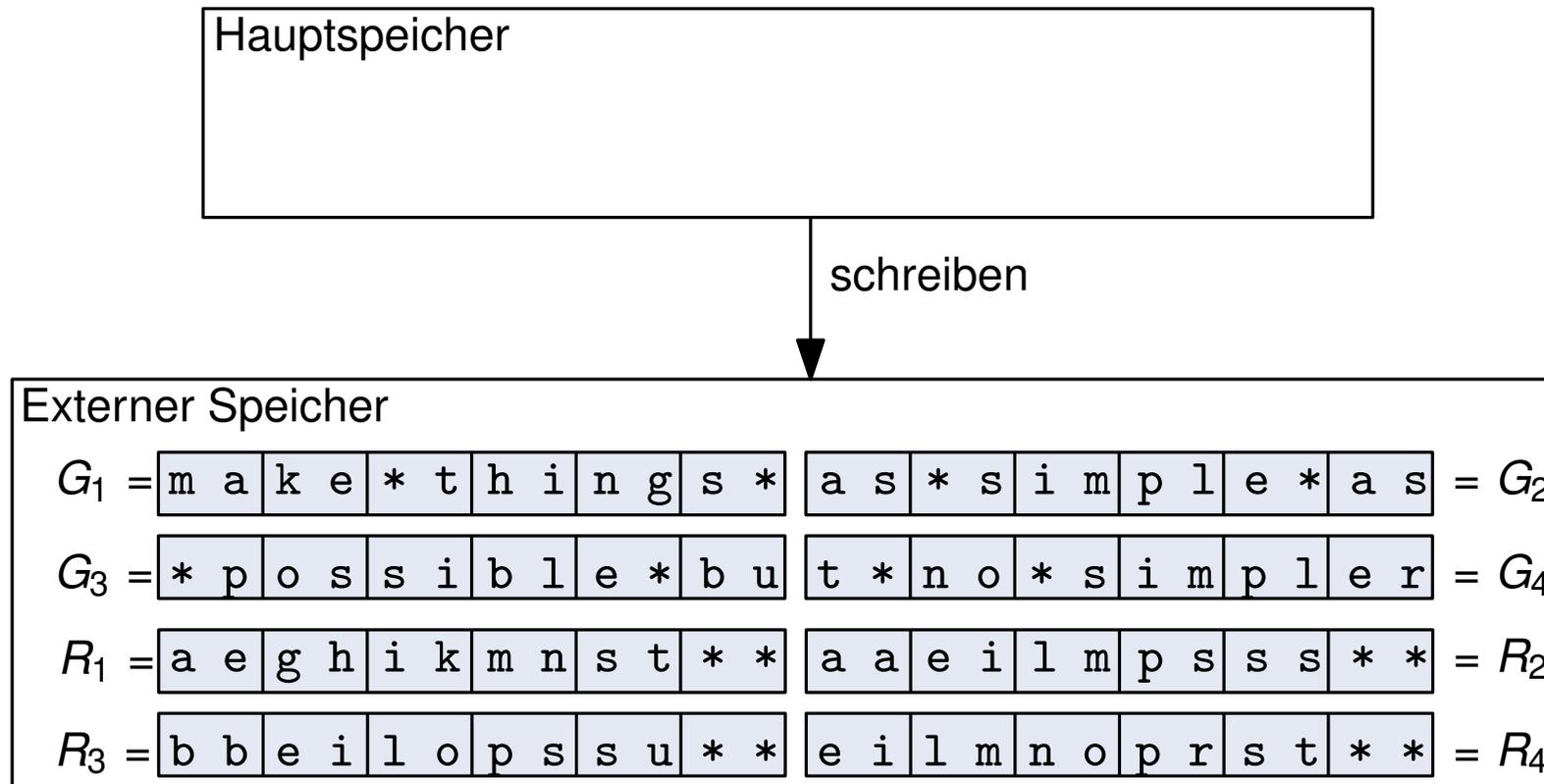


1. Phase: Run Formation

1. Phase: Run Formation

1. Teile a_1, \dots, a_n in m Gruppen G_1, \dots, 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 $\mathcal{R} = \{R_1, \dots, R_m\}$ sortierte *Runs*.

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



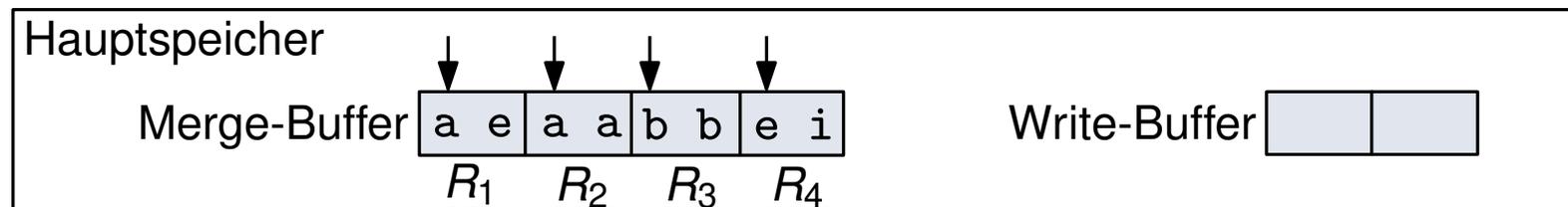
2. Phase: Merging

2. Phase: Merging

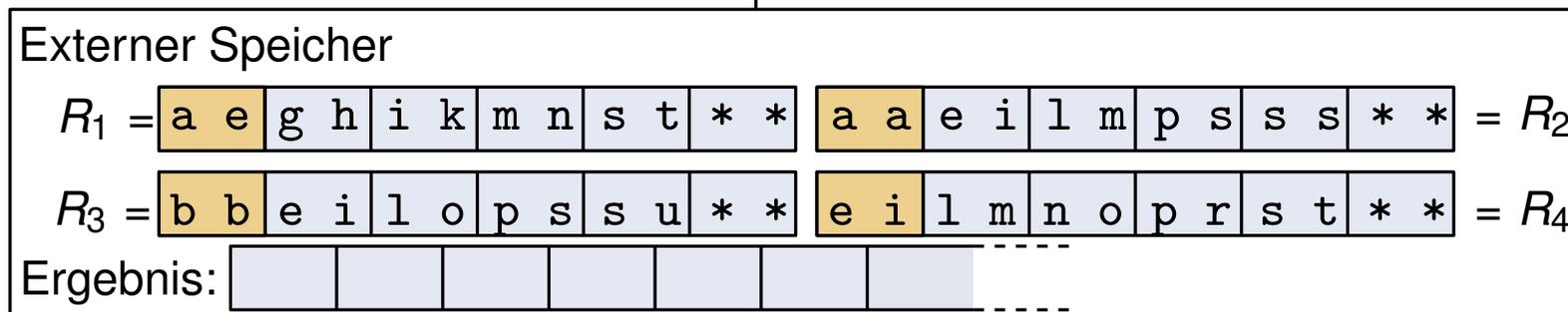
3. Vereinige die einzelnen Runs ordnungserhaltend zu größeren, bis schließlich einer übrig bleibt. Es können $k = O(\frac{M}{B})$ Runs in einem Durchgang vereinigt werden.

Vereinige k Runs in einem Durchlauf (MULTIWAY MERGING):

1. Pro Run nur der Block im *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.



Lade Blöcke.



$B = 2$
 $D = 1$

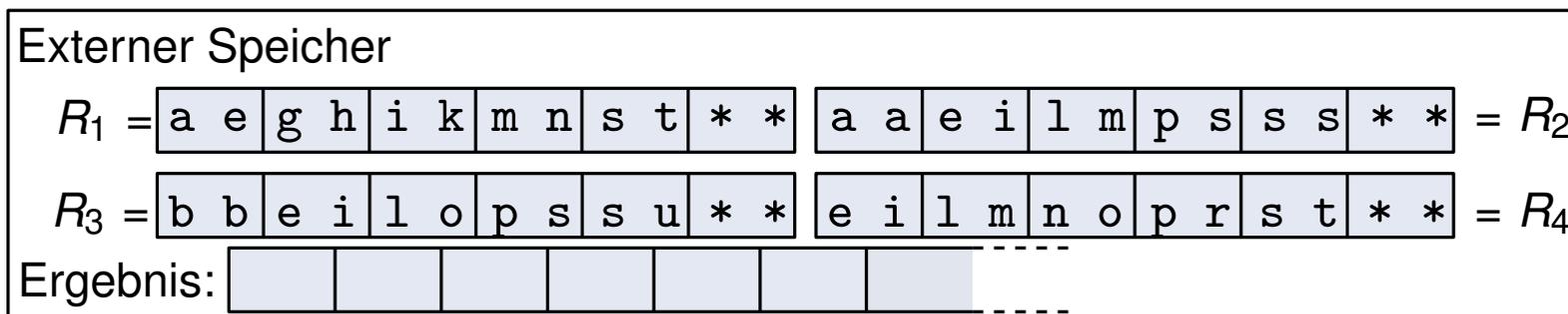
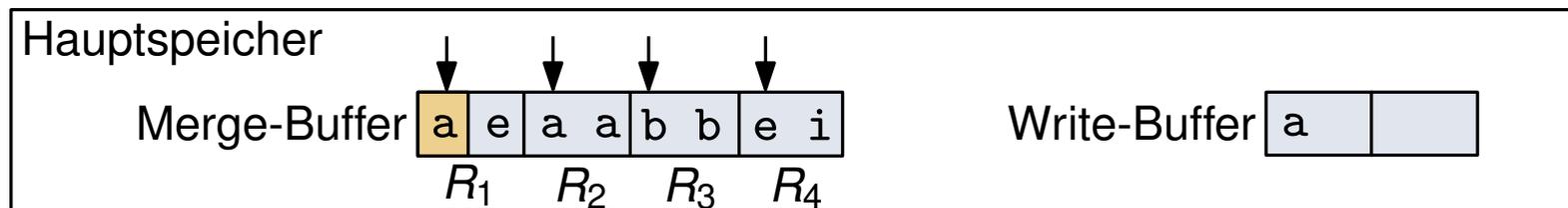
2. Phase: Merging

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. Pro Run nur der Block im *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.



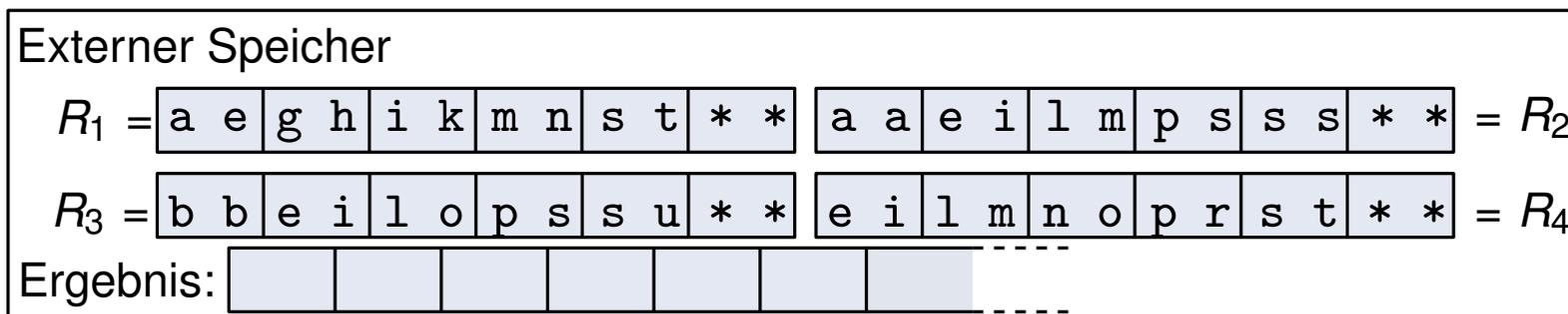
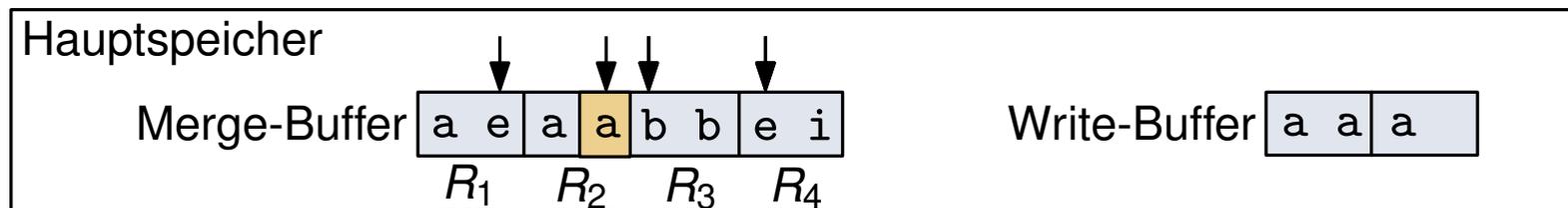
2. Phase: Merging

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. Pro Run nur der Block im *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.



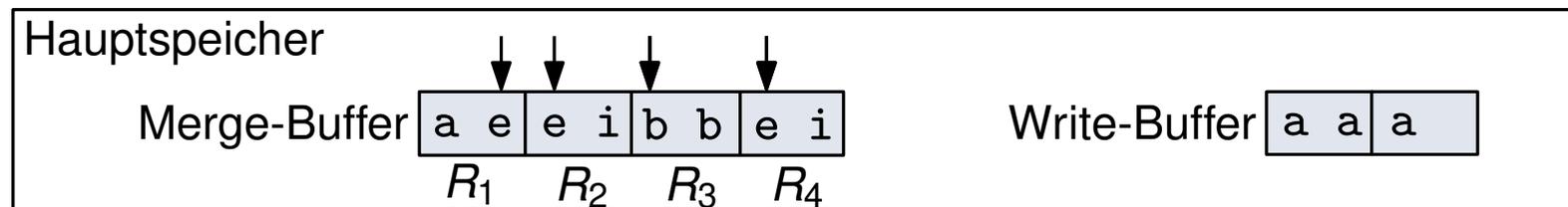
2. Phase: Merging

2. Phase: Merging

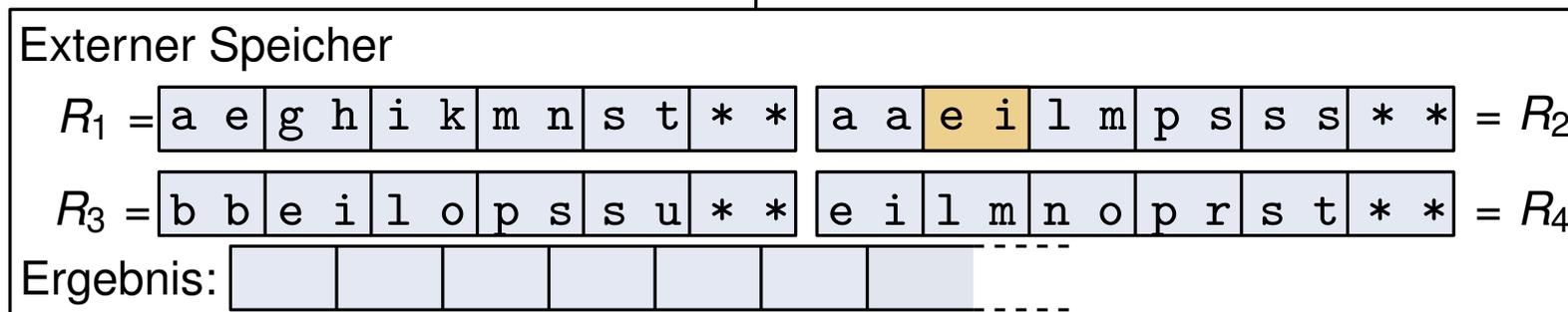
3. Vereinige die einzelnen Runs ordnungserhaltend zu größeren, bis schließlich einer übrig bleibt. Es können $k = O(\frac{M}{B})$ Runs in einem Durchgang vereinigt werden.

Vereinige k Runs in einem Durchlauf (MULTIWAY MERGING):

1. Pro Run nur der Block im *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.



Lade Block aus R_2 .



$B = 2$
 $D = 1$

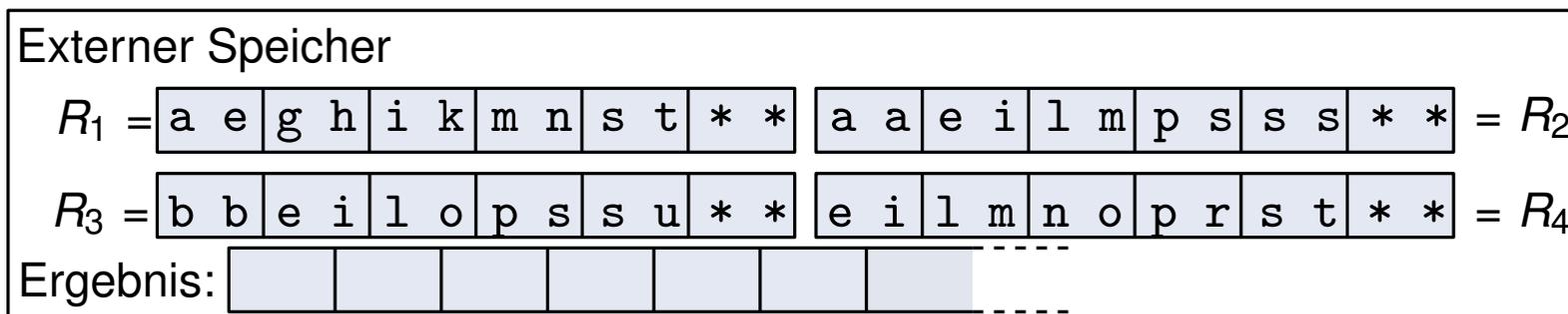
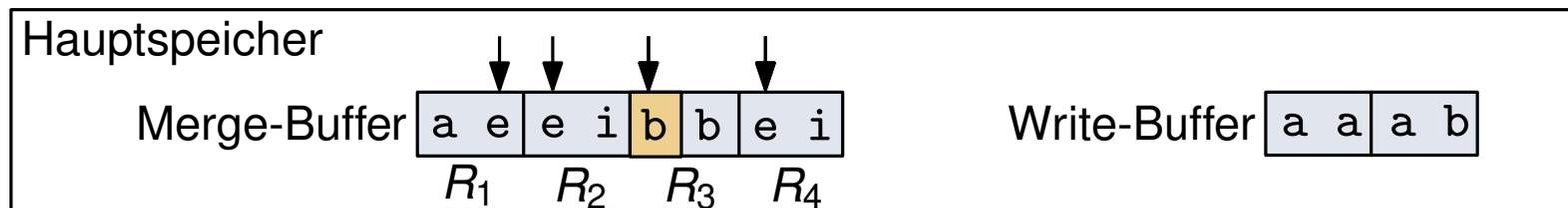
2. Phase: Merging

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. Pro Run nur der Block im *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.



$B = 2$
 $D = 1$

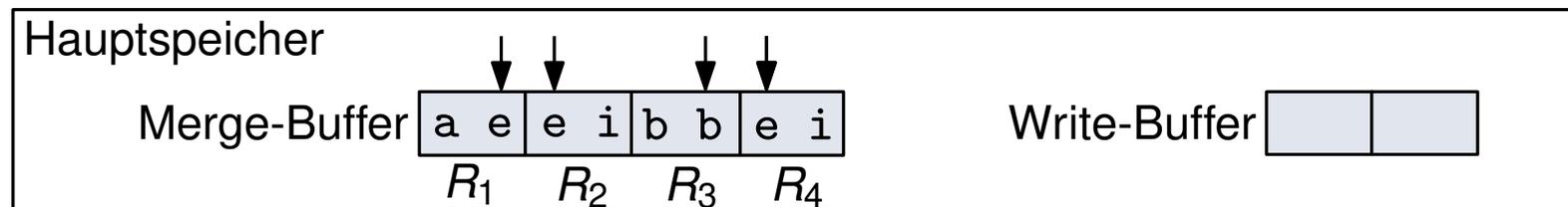
2. Phase: Merging

2. Phase: Merging

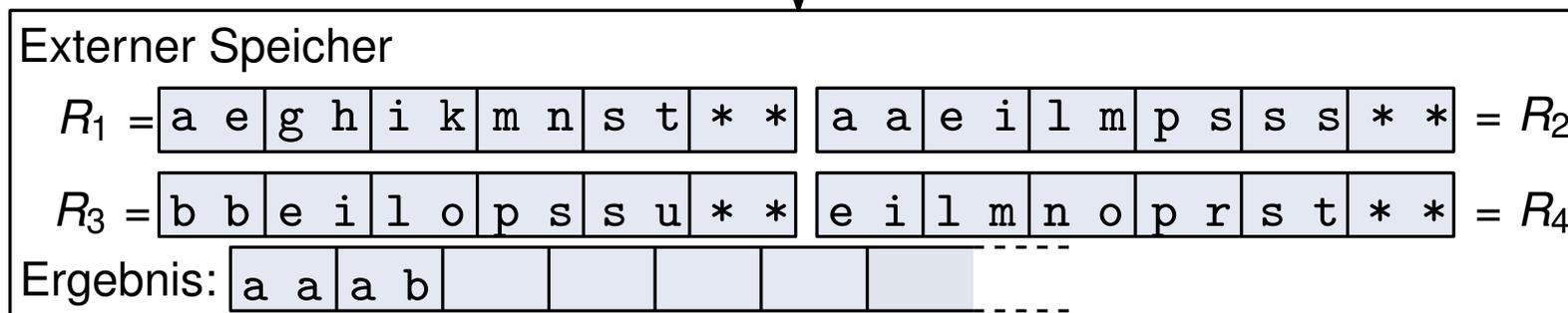
3. Vereinige die einzelnen Runs ordnungserhaltend zu größeren, bis schließlich einer übrig bleibt. Es können $k = O(\frac{M}{B})$ Runs in einem Durchgang vereinigt werden.

Vereinige k Runs in einem Durchlauf (MULTIWAY MERGING):

1. Pro Run nur der Block im *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.



Kopiere Write-Buffer in externen Speicher.



$B = 2$
 $D = 1$

Analyse und Details

1. Phase: Run Formation

1. Phase: Run Formation

1. Teile a_1, \dots, a_n in m Gruppen G_1, \dots, 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 $\mathcal{R} = \{R_1, \dots, R_m\}$ sortierte *Runs*.

Definition:

- *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, \dots, a_n in m Gruppen G_1, \dots, 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 $\mathcal{R} = \{R_1, \dots, R_m\}$ sortierte *Runs*.

Lade G_1 und G_2 in den Hauptspeicher.

Thread A:

für $i = 1, \dots, m - 2$ **tue**

Warte bis G_i in den externen Speicher geschrieben wurde.

 Lade G_{i+2} in den Hauptspeicher.

Thread B:

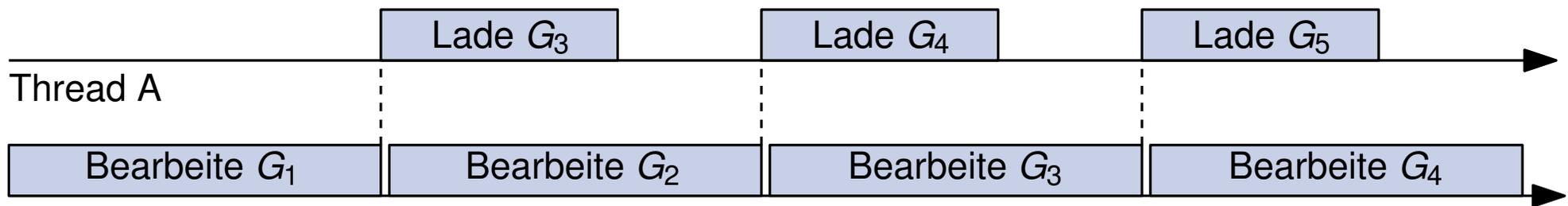
für $i = 1, \dots, m$ **tue**

Warte bis G_i in den Hauptspeicher geladen wurde.

 Sortiere Gruppe G_i .

 Schreibe G_i in externen Speicher.

Annahme: berechnungsbeschränkt



Der langsamere Thread von beiden wartet nie.

1. Phase: Run Formation

Korollar 2: Eine Eingabe der Größe N kann in Zeit

$$\max\left\{T_{\text{sort}}\left(\frac{M}{2}\right) \frac{2N}{M}, L \cdot 2 \cdot \frac{N}{DB}\right\} + O\left(\frac{LM}{DB}\right)$$

in sortierte Runs der Größe $\frac{M}{2}$ transformiert werden.

$T_{\text{sort}}\left(\frac{M}{2}\right)$ = Laufzeit für Sortieren eines Runs. $O\left(\frac{LM}{DB}\right)$ = Initialisierung

$\frac{2N}{M}$ = Anzahl Runs.

$2 \cdot \frac{N}{DB}$ = Anzahl I/O-Operationen: Jeder Run muss geladen und gespeichert werden.

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).
- $T_{\text{sort}}(n)$: Zeit um n Elemente intern zu sortieren.

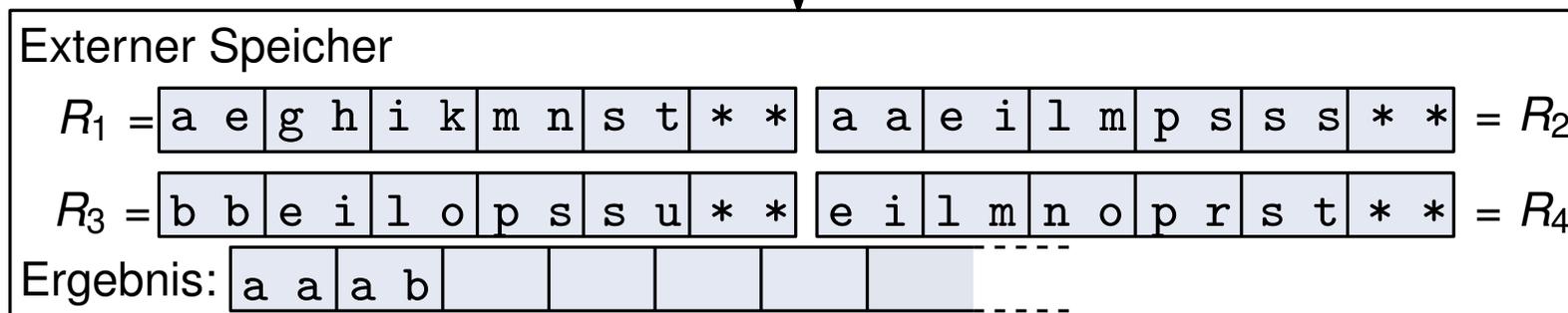
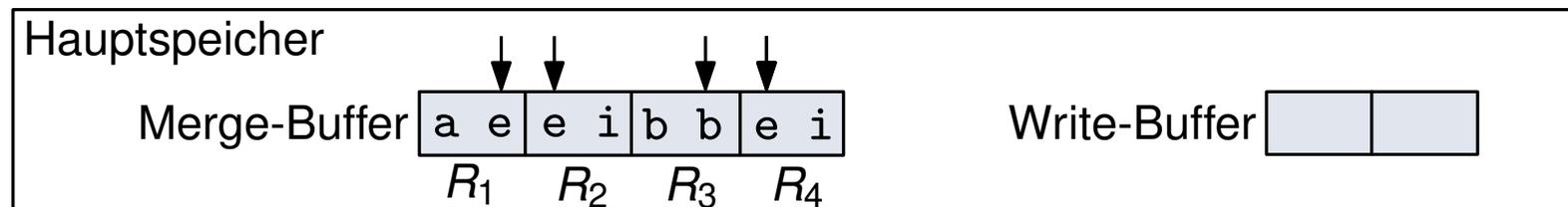
2. Phase: Merging

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. Pro Run nur der Block im *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.



$B = 2$
 $D = 1$

Tournament-Bäume

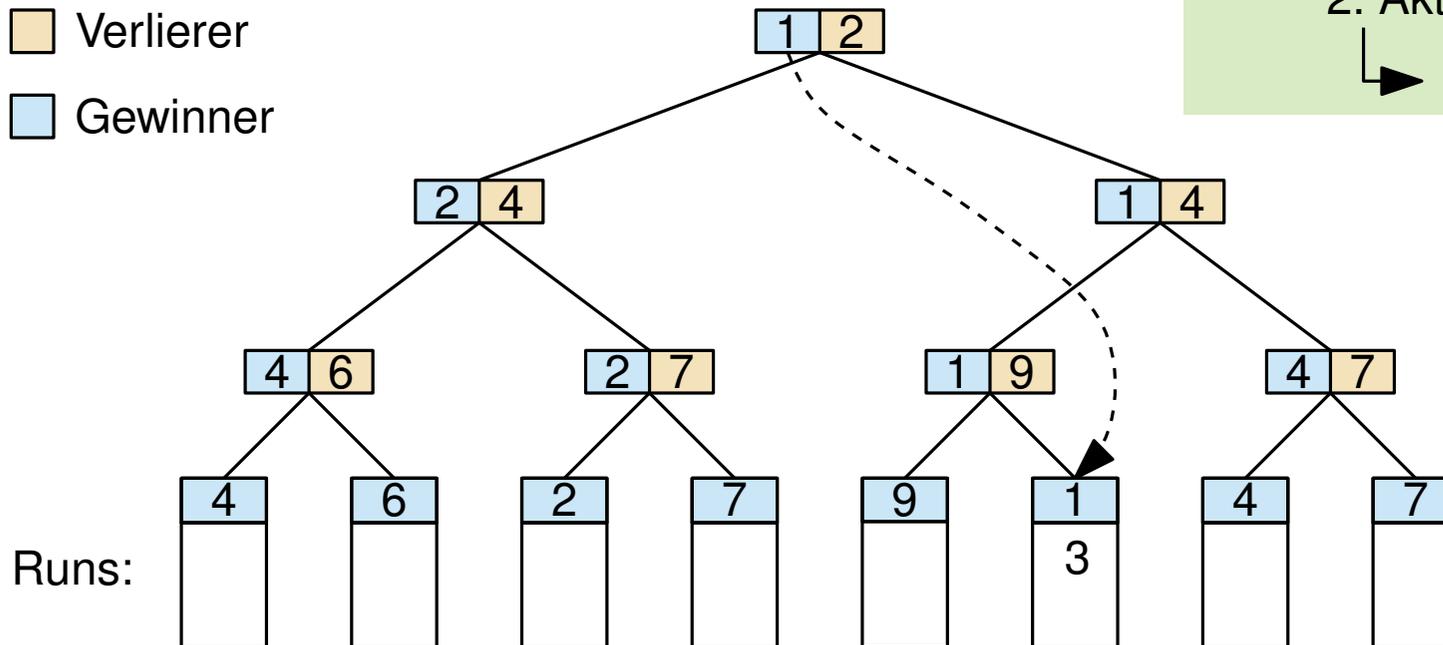
Wie kann das kleinste Element im Merge-Buffer schnell gefunden werden?

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.

Operation: Minimum entfernen.
1. Entferne Minimum
2. Aktualisiere Baum
 ▶ Führe Wettkämpfe aus.

■ Verlierer
■ Gewinner



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.

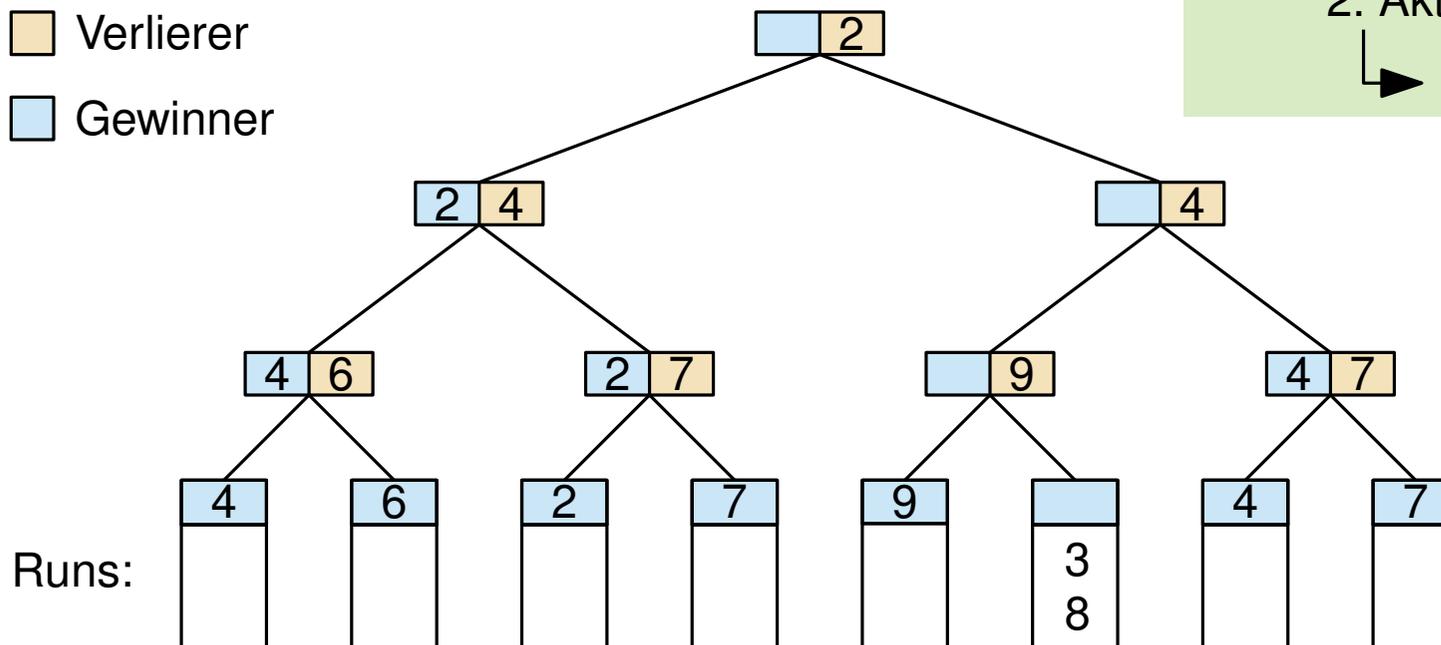
Operation: Minimum entfernen.

1. Entferne Minimum

2. Aktualisiere Baum

↳ Führe Wettkämpfe aus.

■ Verlierer
■ Gewinner



Tournament-Bäume

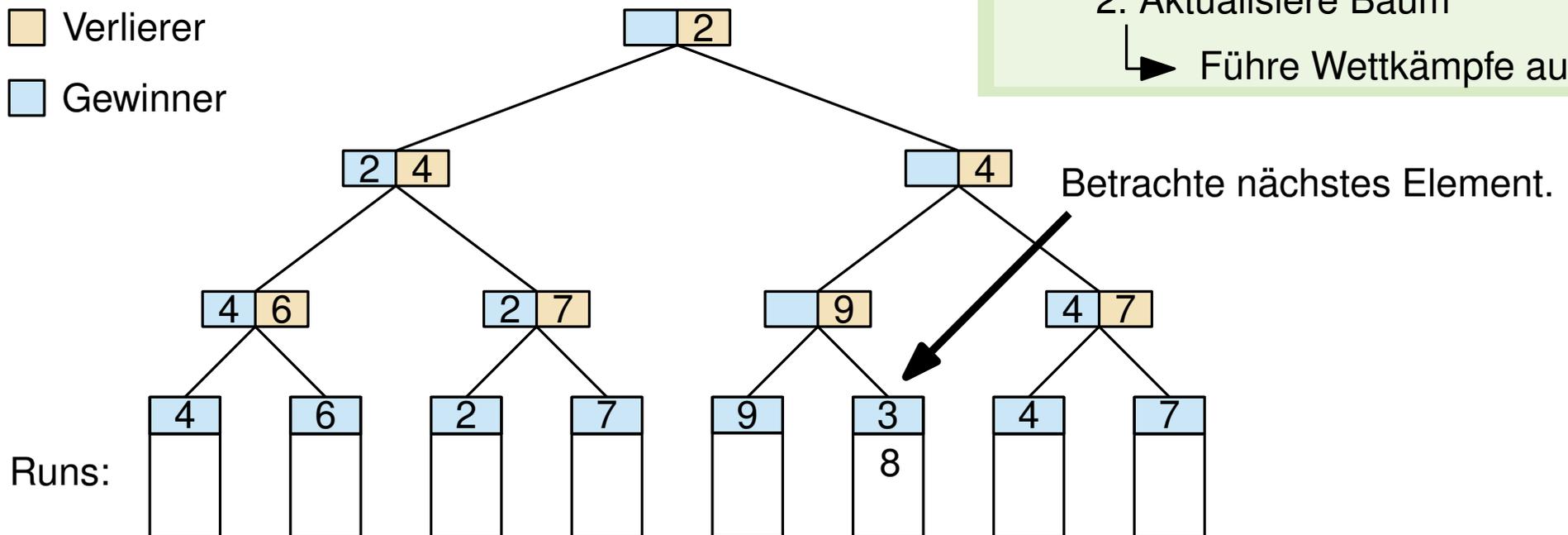
Wie kann das kleinste Element im Merge-Buffer schnell gefunden werden?

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.

Operation: Minimum entfernen.
1. Entferne Minimum
2. Aktualisiere Baum
↳ Führe Wettkämpfe aus.

■ Verlierer
■ Gewinner



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.

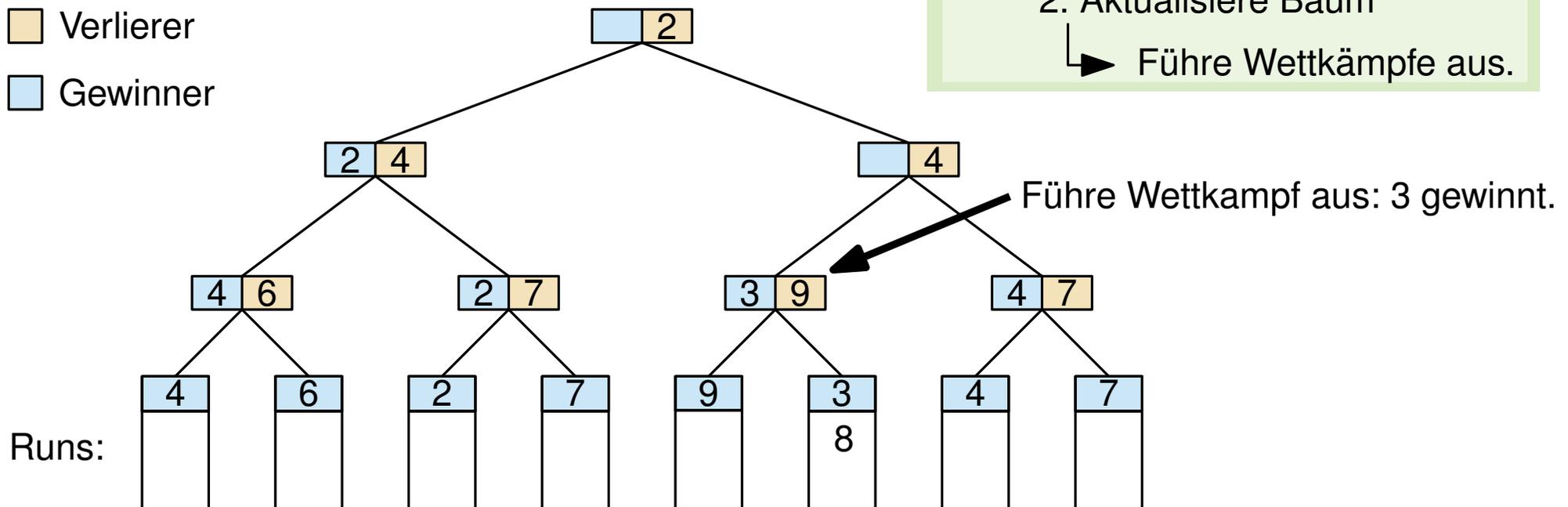
Operation: Minimum entfernen.

1. Entferne Minimum

2. Aktualisiere Baum

↳ Führe Wettkämpfe aus.

■ Verlierer
■ Gewinner



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.

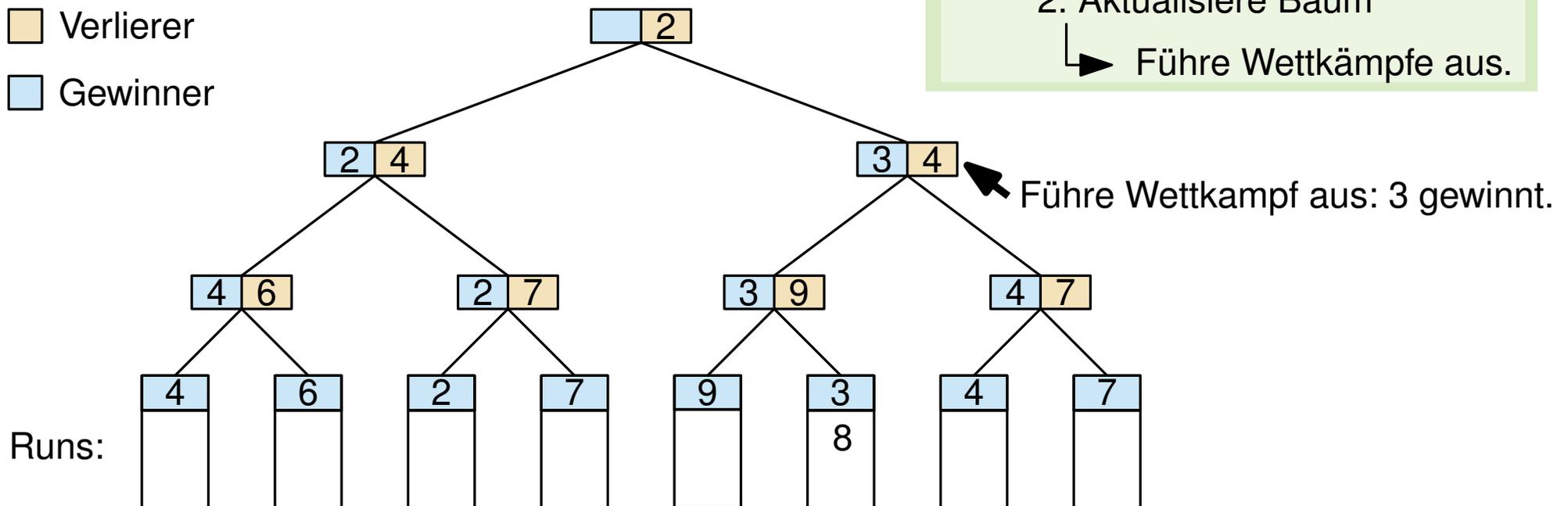
Operation: Minimum entfernen.

1. Entferne Minimum

2. Aktualisiere Baum

↳ Führe Wettkämpfe aus.

■ Verlierer
■ Gewinner



Tournament-Bäume

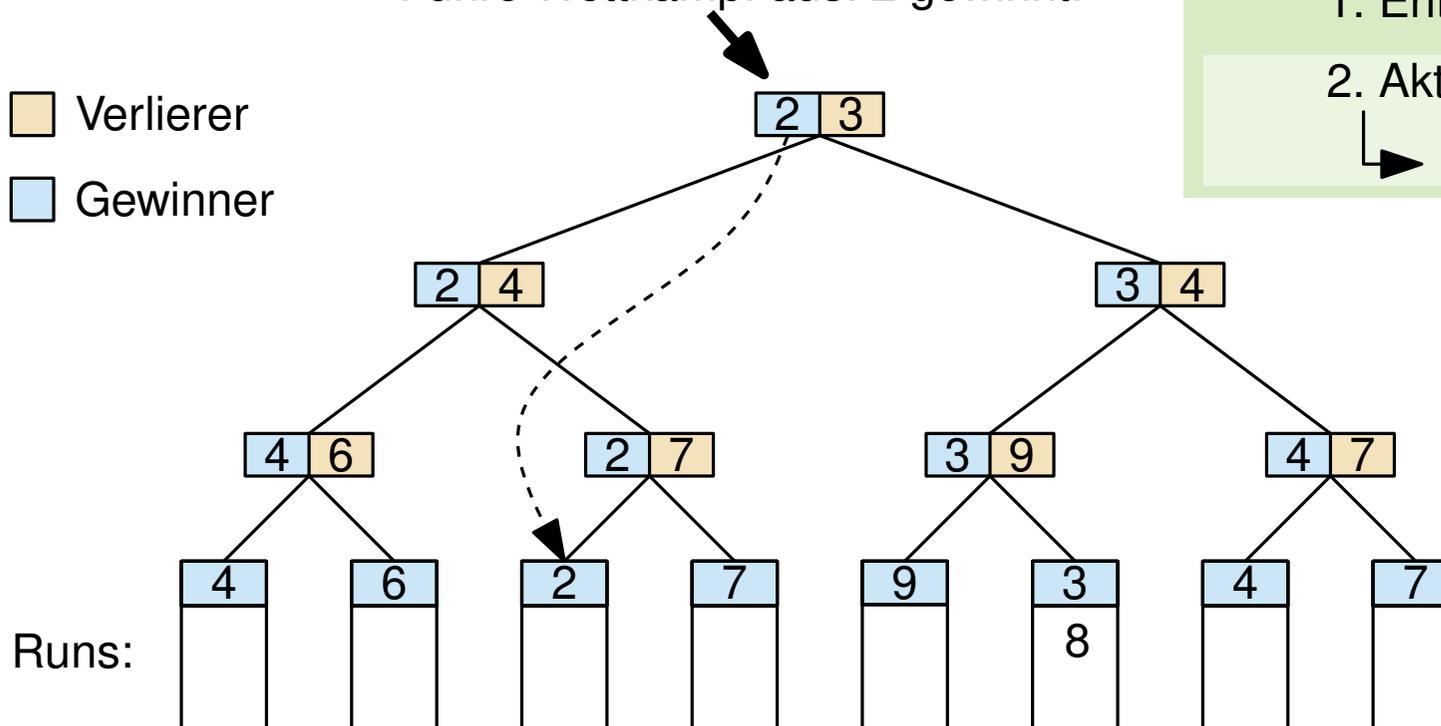
Wie kann das kleinste Element im Merge-Buffer schnell gefunden werden?

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.

Führe Wettkampf aus: 2 gewinnt.

■ Verlierer
■ Gewinner



Operation: Minimum entfernen.

1. Entferne Minimum

2. Aktualisiere Baum

↳ Führe Wettkämpfe aus.

Tournament-Bäume

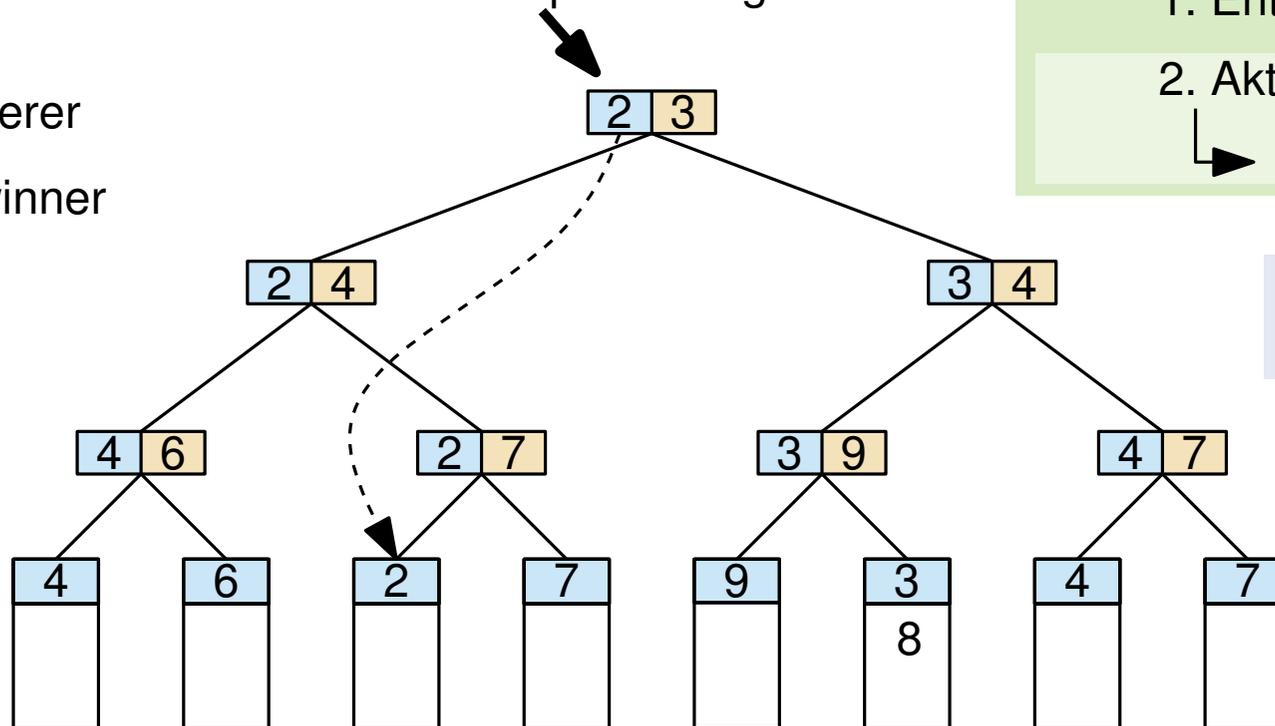
Wie kann das kleinste Element im Merge-Buffer schnell gefunden werden?

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.

Führe Wettkampf aus: 2 gewinnt.

Verlierer
 Gewinner



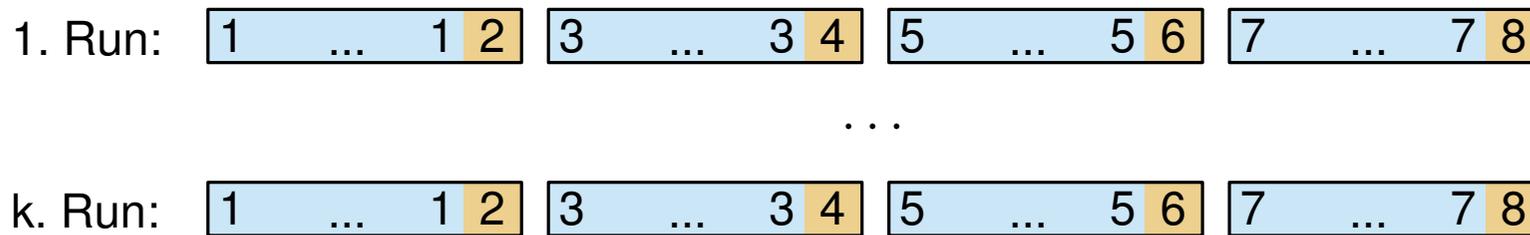
Runs:

Operation: Minimum entfernen.
 1. Entferne Minimum
 2. Aktualisiere Baum
 ↳ Führe Wettkämpfe aus.

Operation in $O(\log k)$
 Zeit ausführbar.

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.

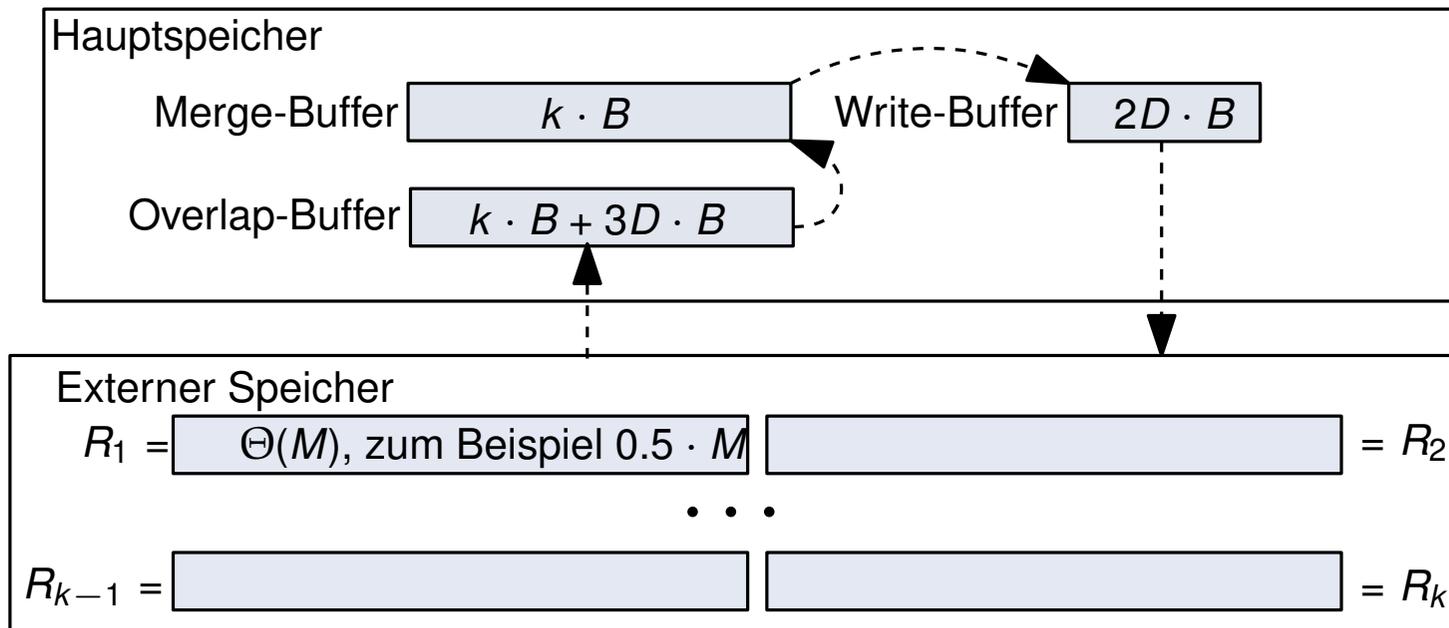
Verbesserung: Verwende wieder Overlapping-Technik, um I/O von Verarbeitung zu trennen:

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

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.



2. Phase: Merging

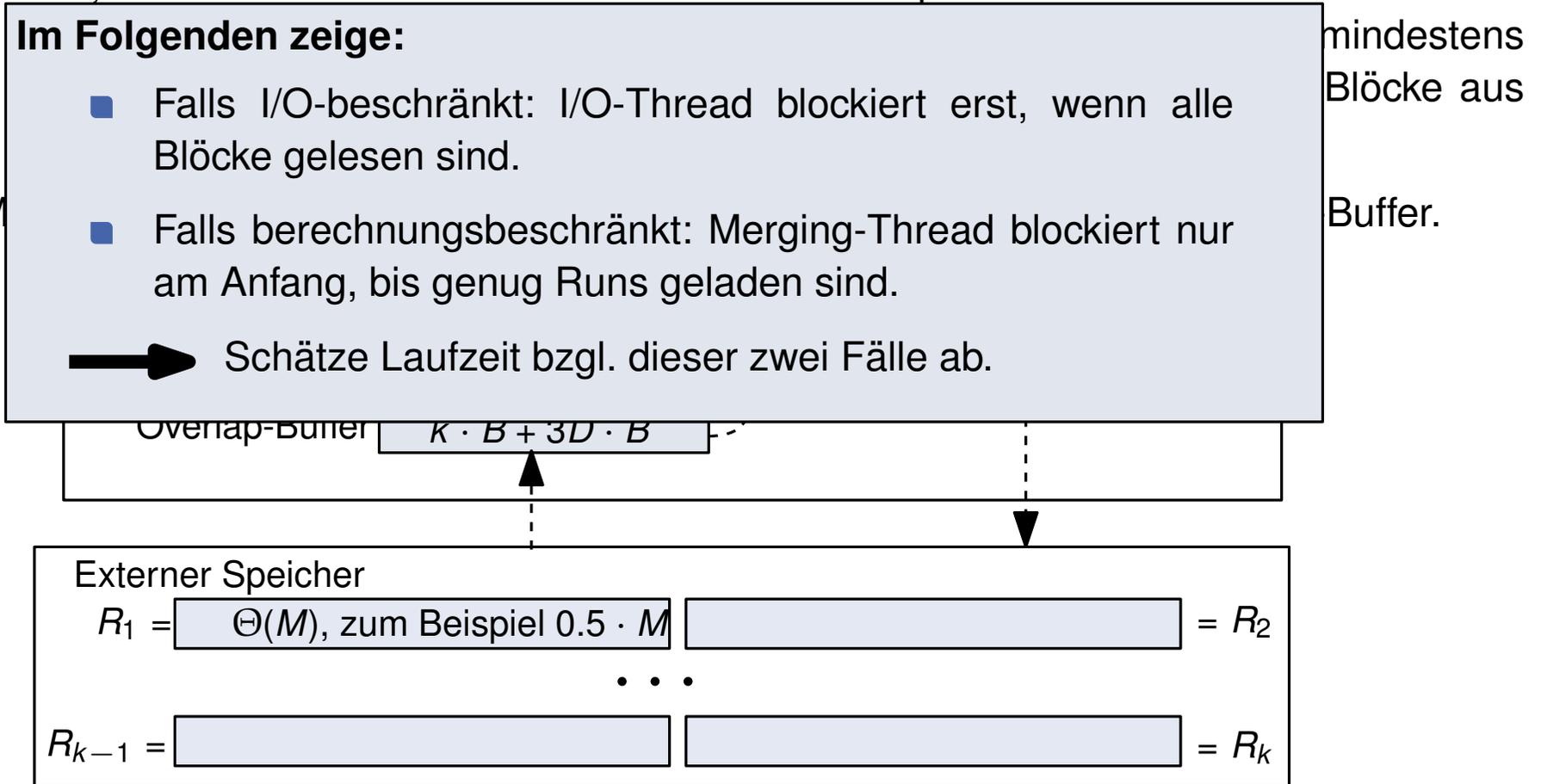
Anpassungen:

- 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. **Im Folgenden zeige:**

- Falls I/O-beschränkt: I/O-Thread blockiert erst, wenn alle Blöcke gelesen sind.
- Falls berechnungsbeschränkt: Merging-Thread blockiert nur am Anfang, bis genug Runs geladen sind.

➔ Schätze Laufzeit bzgl. dieser zwei Fälle ab.



2. Phase: Merging

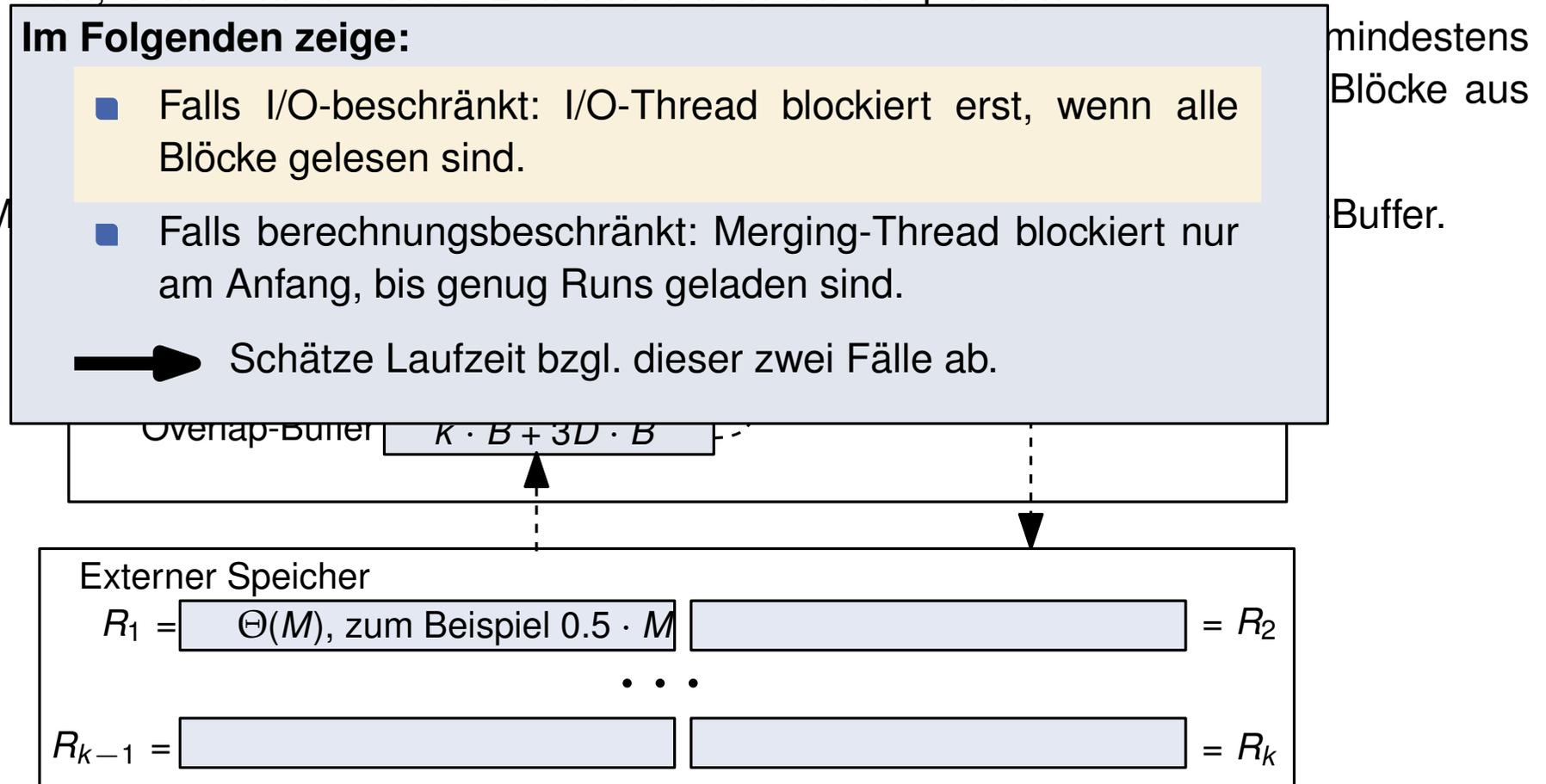
Anpassungen:

- 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. Im Folgenden zeige:

- Falls I/O-beschränkt: I/O-Thread blockiert erst, wenn alle Blöcke gelesen sind.
- Falls berechnungsbeschränkt: Merging-Thread blockiert nur am Anfang, bis genug Runs geladen sind.

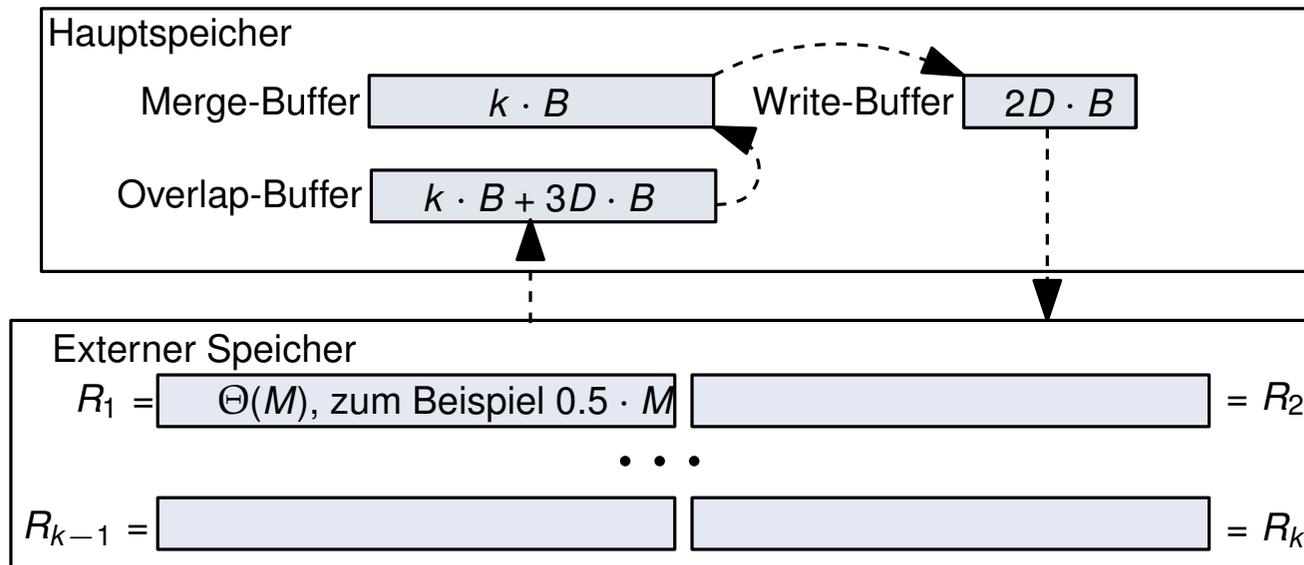
➔ Schätze Laufzeit bzgl. dieser zwei Fälle ab.



1. Fall: I/O-beschränkt

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.

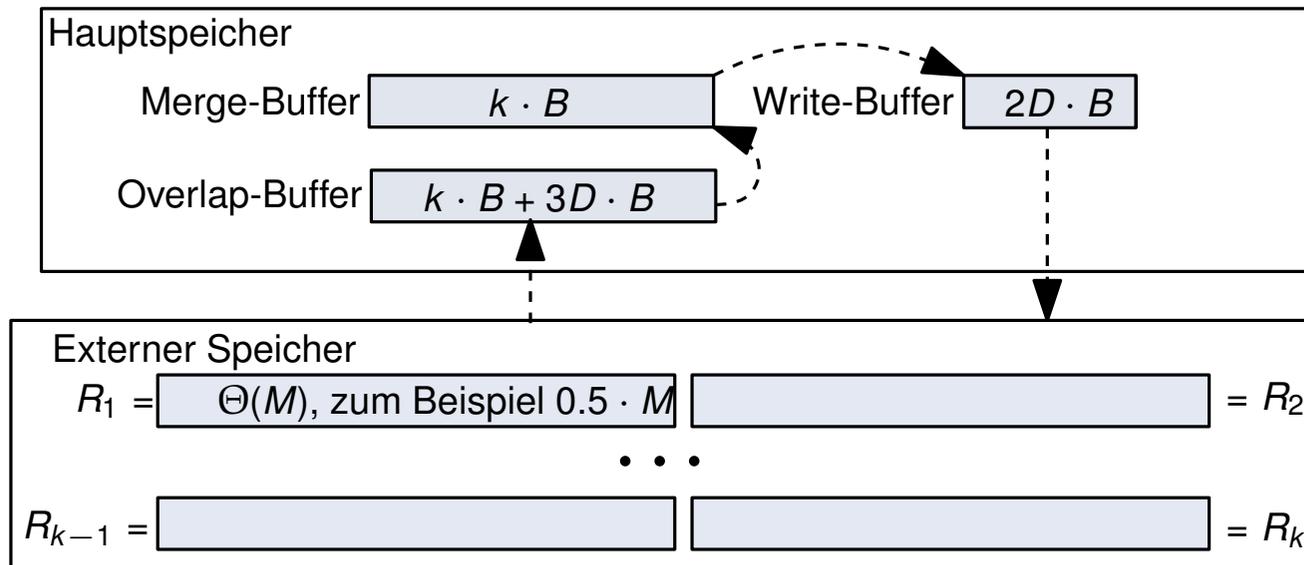
Lemma wird gleich benötigt, um zu zeigen, dass I/O-Thread nicht blockiert.



1. Fall: I/O-beschränkt

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.

Beweis: Annahme Merge- und Overlap-Buffer enthalten zusammen kB Elemente, aber ein neuer Block muss geladen werden.

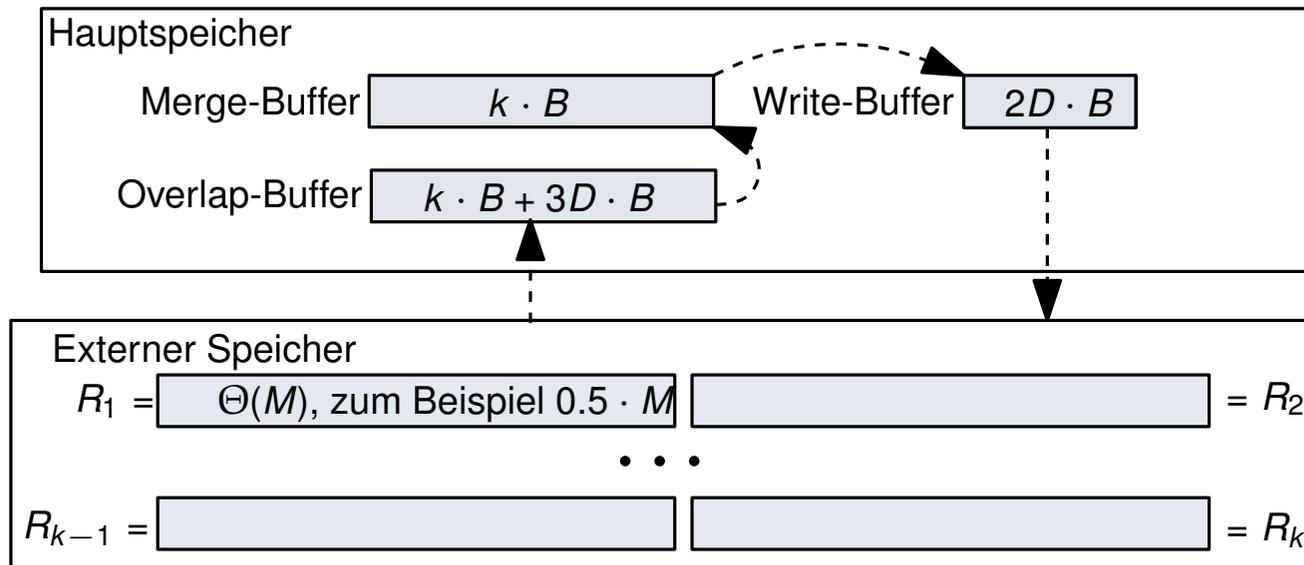


1. Fall: I/O-beschränkt

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.

Beweis: Annahme Merge- und Overlap-Buffer enthalten zusammen kB Elemente, aber ein neuer Block muss geladen werden.

- Overlap-Buffer ist leer, da ansonsten ein Laden aus dem externen Speicher nicht nötig ist.
- Damit enthält Merge-Buffer kB Elemente.



1. Fall: I/O-beschränkt

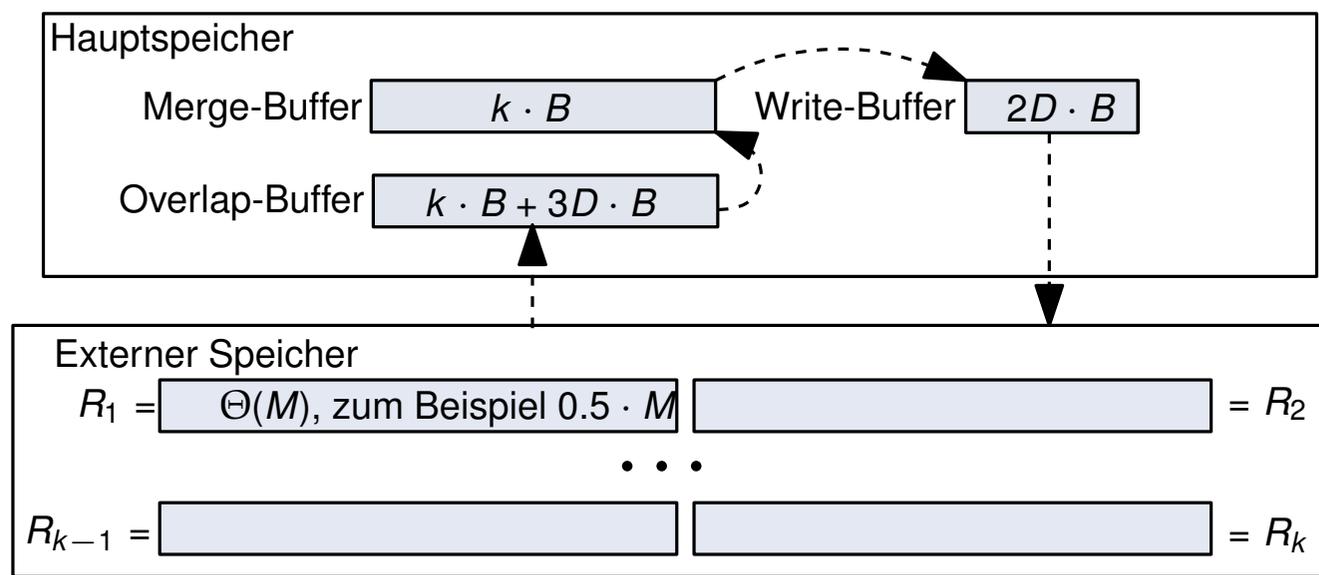
Lemma 5: Falls der Overlap-Buffer und der Merge-Buffer zusammen mindestens $k \cdot B$ Elemente enthalten, dann kann mindestens ein weiteres Element vom Merging-Thread verarbeitet werden, ohne dass ein neuer Block aus dem externen Speicher geladen werden muss.

Beweis: Annahme Merge- und Overlap-Buffer enthalten zusammen kB Elemente, aber ein neuer Block muss geladen werden.

→ Overlap-Buffer ist leer, da ansonsten ein Laden aus dem externen Speicher nicht nötig ist.

→ Damit enthält Merge-Buffer kB Elemente.

→ Es können Elemente vom Merging-Thread verarbeitet werden.

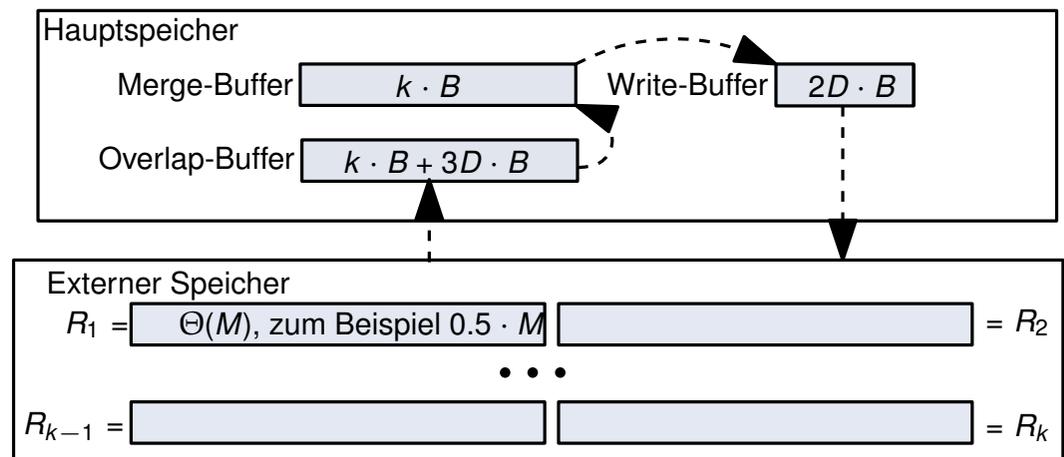


1. Fall: I/O-beschränkt

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 $2L \geq DB\ell$, blockiert der I/O-Thread erst, wenn alle Eingabeblöcke gelesen worden sind.

→ I/O-Thread ist langsamer als Merging-Thread



1. Fall: I/O-beschränkt

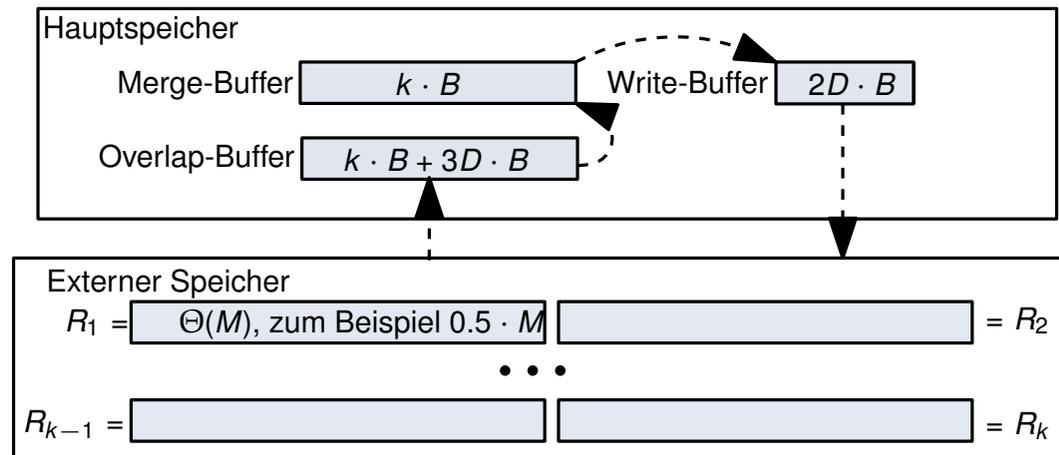
Lemma 6: Sei ℓ die Zeit, die der Merging-Thread benötigt um ein Element der Ausgabe zu erzeugen und sei L die Zeit, die gebraucht wird um D beliebige Blöcke zu laden/speichern.

Falls $2L \geq DB\ell$, blockiert der I/O-Thread erst, wenn alle Eingabeblöcke gelesen worden sind.

$y = \frac{L}{\ell}$: Anzahl Elemente, die während eines I/O-Schritts vermengt werden können.

w = Anzahl Elemente im Write-Buffer. r = Anzahl Elemente im Overlap- und Merge-Buffer.

Aus $2L \geq DB\ell$ folgt: $y \geq \frac{DB}{2}$



1. Fall: I/O-beschränkt

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 $2L \geq DB\ell$, blockiert der I/O-Thread erst, wenn alle Eingabeblöcke gelesen worden sind.

$y = \frac{L}{\ell}$: Anzahl Elemente, die während eines I/O-Schritts vermengt werden können.

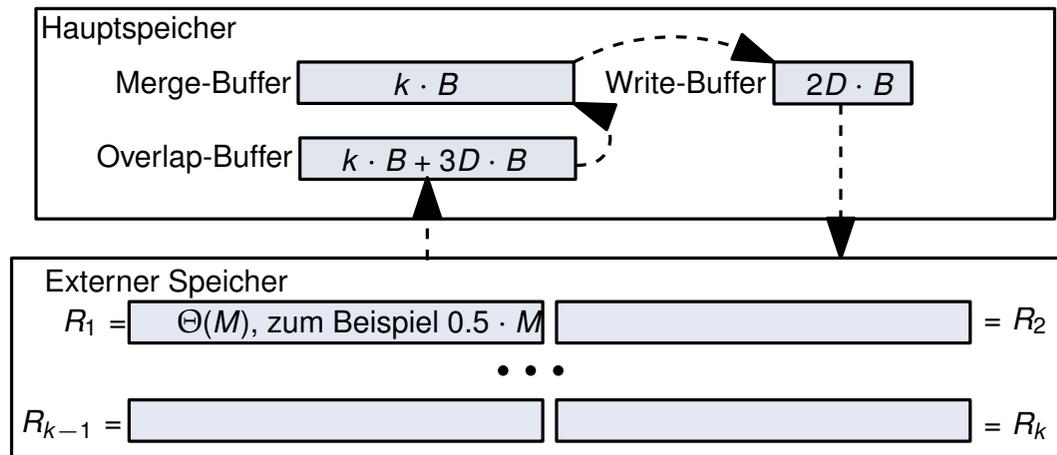
w = Anzahl Elemente im Write-Buffer. r = Anzahl Elemente im Overlap- und Merge-Buffer.

Aus $2L \geq DB\ell$ folgt: $y \geq \frac{DB}{2}$

1. Fall: $y \geq DB$



In einem I/O-Schritt werden mindestens so viele Elemente vom Merging-Thread verarbeitet wie vom I/O-Thread.



1. Fall: I/O-beschränkt

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 $2L \geq DB\ell$, blockiert der I/O-Thread erst, wenn alle Eingabeblocke gelesen worden sind.

$y = \frac{L}{\ell}$: Anzahl Elemente, die während eines I/O-Schritts vermengt werden können.

w = Anzahl Elemente im Write-Buffer. r = Anzahl Elemente im Overlap- und Merge-Buffer.

Aus $2L \geq DB\ell$ folgt: $y \geq \frac{DB}{2}$

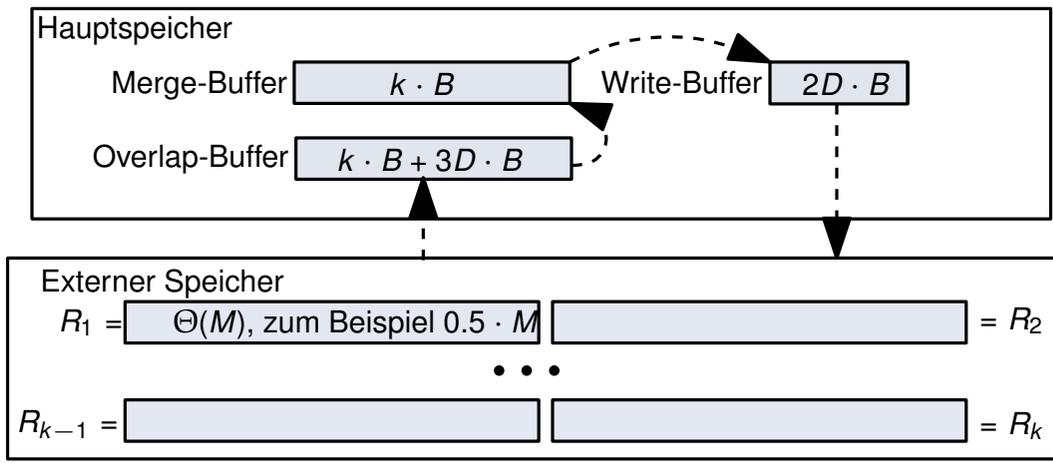
1. Fall: $y \geq DB$

Aus Lemma 5 folgt:

r kann nicht $kB + DB$ überschreiten.

Der Overlap-Buffer hat immer genug Platz für D weitere Blöcke.

→ I/O-Thread blockiert nicht.



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.

1. Fall: I/O-beschränkt

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 $2L \geq DB\ell$, blockiert der I/O-Thread erst, wenn alle Eingabeblocke gelesen worden sind.

$y = \frac{L}{\ell}$: Anzahl Elemente, die während eines I/O-Schritts vermengt werden können.

w = Anzahl Elemente im Write-Buffer. r = Anzahl Elemente im Overlap- und Merge-Buffer.

Aus $2L \geq DB\ell$ folgt: $y \geq \frac{DB}{2}$

2. Fall: $\frac{DB}{2} \leq y < DB$



In einem I/O-Schritt werden weniger Elemente vom Merging-Thread verarbeitet, als vom I/O-Thread.

1. Fall: I/O-beschränkt

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 $2L \geq DB\ell$, blockiert der I/O-Thread erst, wenn alle Eingabeblocke gelesen worden sind.

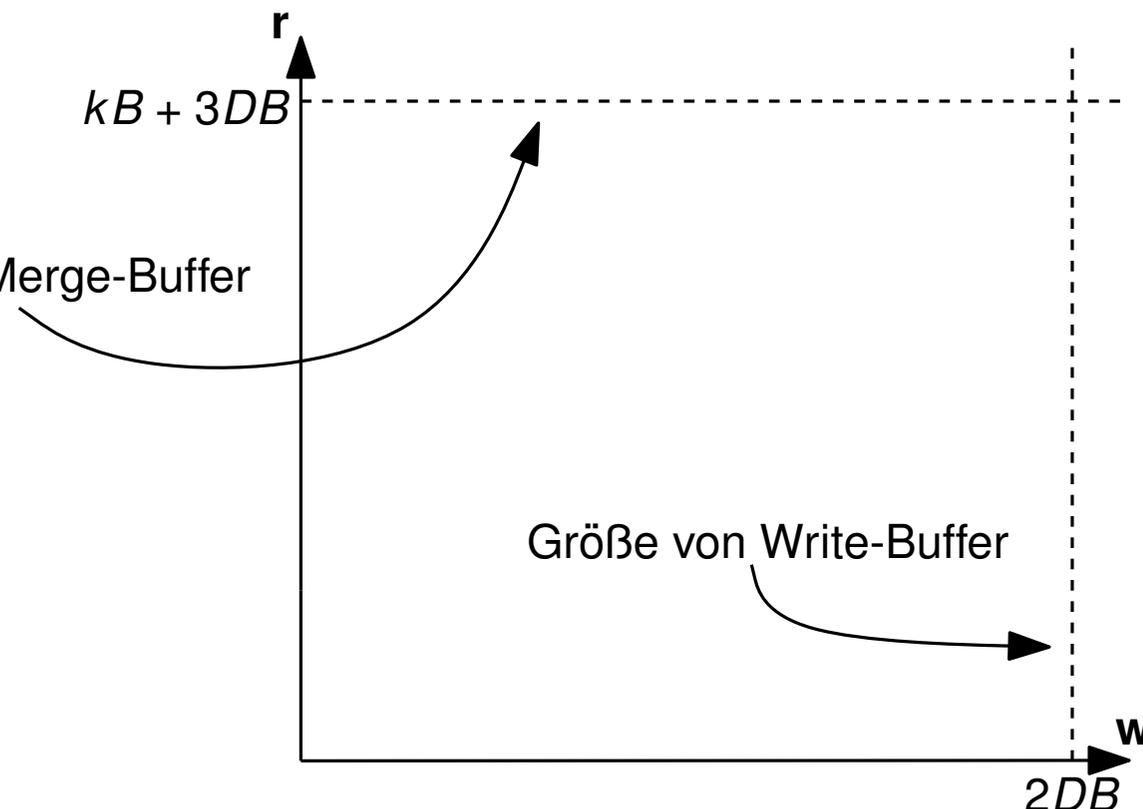
$y = \frac{L}{\ell}$: Anzahl Elemente, die während eines I/O-Schritts vermengt werden können.

w = Anzahl Elemente im Write-Buffer. r = Anzahl Elemente im Overlap- und Merge-Buffer.

Aus $2L \geq DB\ell$ folgt: $y \geq \frac{DB}{2}$

2. Fall: $\frac{DB}{2} \leq y < DB$

Größe von Overlap- und Merge-Buffer



1. Fall: I/O-beschränkt

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 $2L \geq DB\ell$, blockiert der I/O-Thread erst, wenn alle Eingabeblocke gelesen worden sind.

$y = \frac{L}{\ell}$: Anzahl Elemente, die während eines I/O-Schritts vermengt werden können.

w = Anzahl Elemente im Write-Buffer. r = Anzahl Elemente im Overlap- und Merge-Buffer.

Aus $2L \geq DB\ell$ folgt: $y \geq \frac{DB}{2}$

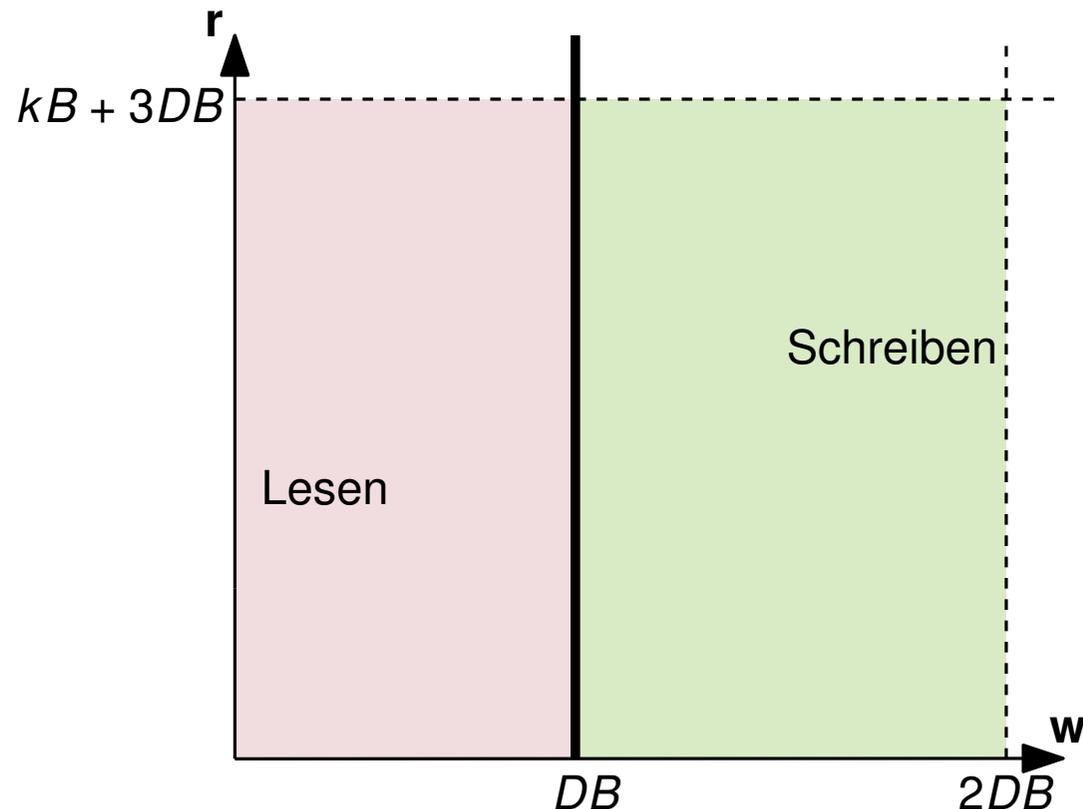
2. Fall: $\frac{DB}{2} \leq y < DB$

I/O-Thread liest nur wenn

$$w < DB$$

I/O-Thread schreibt nur, wenn

$$w \geq DB$$



1. Fall: I/O-beschränkt

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 $2L \geq DB\ell$, blockiert der I/O-Thread erst, wenn alle Eingabeblocke gelesen worden sind.

$y = \frac{L}{\ell}$: Anzahl Elemente, die während eines I/O-Schritts vermengt werden können.

w = Anzahl Elemente im Write-Buffer. r = Anzahl Elemente im Overlap- und Merge-Buffer.

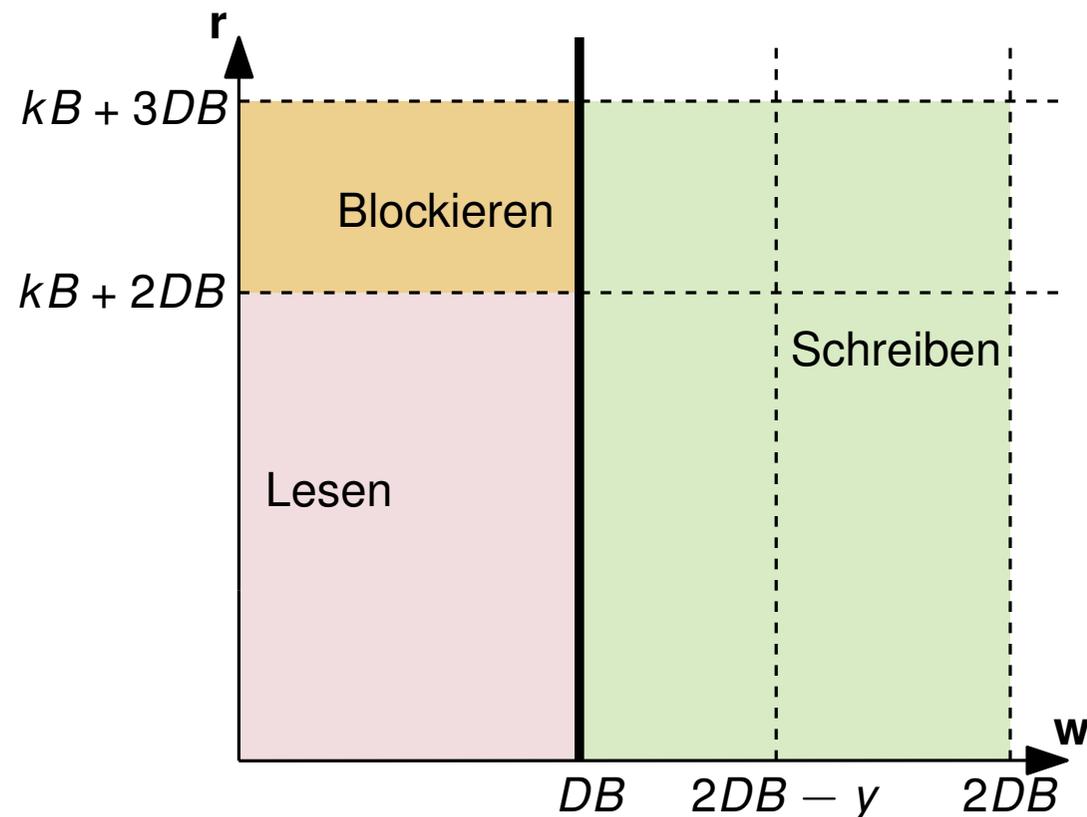
Aus $2L \geq DB\ell$ folgt: $y \geq \frac{DB}{2}$

2. Fall: $\frac{DB}{2} \leq y < DB$

I/O-Thread liest nur wenn
 $w < DB$

I/O-Thread schreibt nur, wenn
 $w \geq DB$

I/O-Thread blockiert gdw.
 $w < DB$ und $r > kB + 2DB$



1. Fall: I/O-beschränkt

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 $2L \geq DB\ell$, blockiert der I/O-Thread erst, wenn alle Eingabeblocke gelesen worden sind.

$y = \frac{L}{\ell}$: Anzahl Elemente, die während eines I/O-Schritts vermengt werden können.

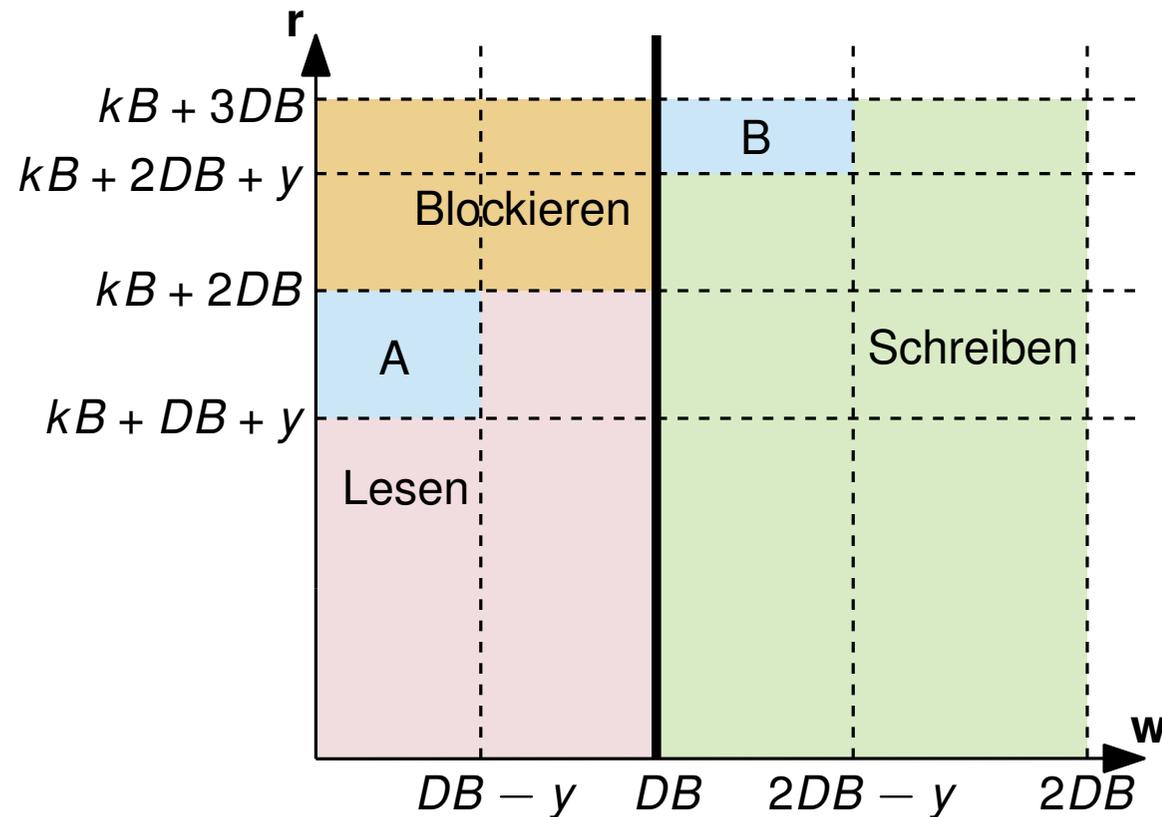
w = Anzahl Elemente im Write-Buffer. r = Anzahl Elemente im Overlap- und Merge-Buffer.

Aus $2L \geq DB\ell$ folgt: $y \geq \frac{DB}{2}$

2. Fall: $\frac{DB}{2} \leq y < DB$

Idee: Zeige, dass

1. *Blockieren*-Zustände nur von Regionen A und B erreichbar sind.
2. A und B sind nicht erreichbar.



1. Fall: I/O-beschränkt

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 $2L \geq DB\ell$, blockiert der I/O-Thread erst, wenn alle Eingabeblocke gelesen worden sind.

$y = \frac{L}{\ell}$: Anzahl Elemente, die während eines I/O-Schritts vermengt werden können.

w = Anzahl Elemente im Write-Buffer. r = Anzahl Elemente im Overlap- und Merge-Buffer.

Aus $2L \geq DB\ell$ folgt: $y \geq \frac{DB}{2}$

2. Fall: $\frac{DB}{2} \leq y < DB$

Es gilt:

1. *Blockieren*-Zustände sind ausschließlich von blauen Regionen *A* und *B* erreichbar.

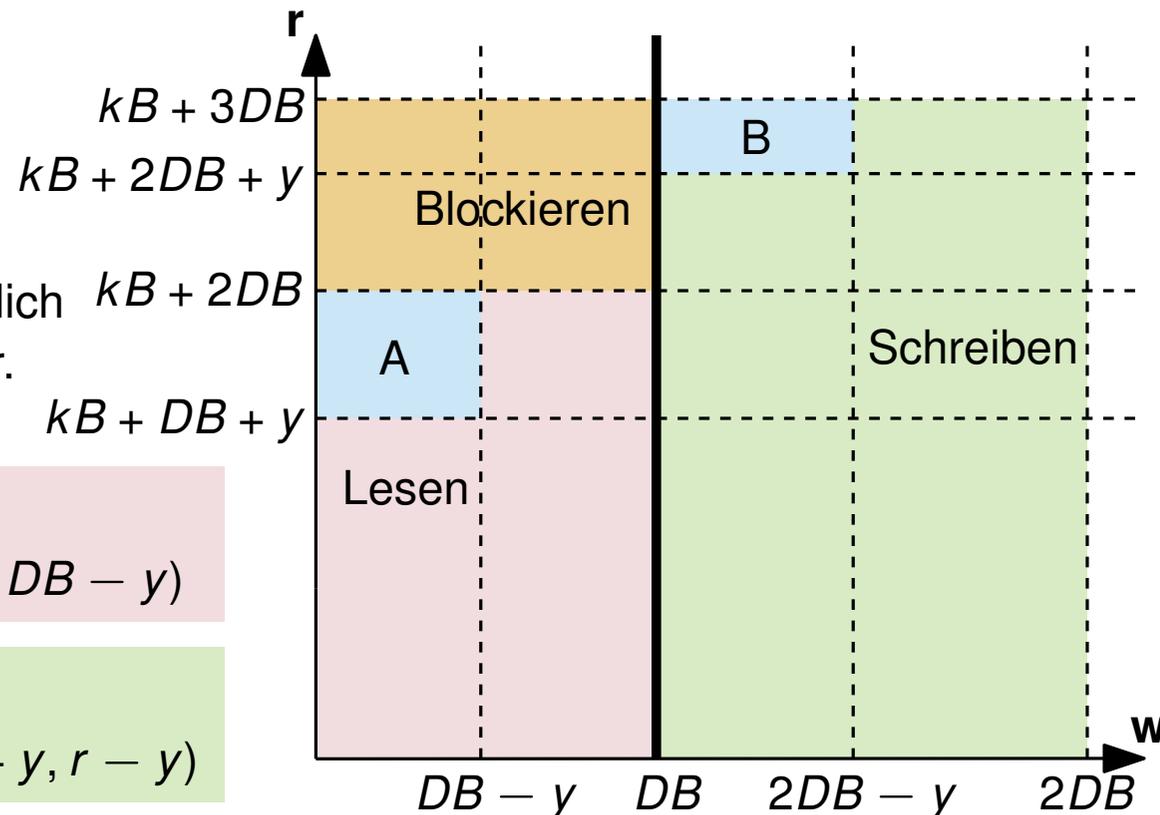
denn

Fall $w < DB$

Zustandsübergang: $(w, r) \rightsquigarrow (w + y, r + DB - y)$

Fall $w \geq DB$

Zustandsübergang: $(w, r) \rightsquigarrow (w - DB + y, r - y)$



1. Fall: I/O-beschränkt

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 $2L \geq DB\ell$, blockiert der I/O-Thread erst, wenn alle Eingabeblocke gelesen worden sind.

$y = \frac{L}{\ell}$: Anzahl Elemente, die während eines I/O-Schritts vermengt werden können.

w = Anzahl Elemente im Write-Buffer. r = Anzahl Elemente im Overlap- und Merge-Buffer.

Aus $2L \geq DB\ell$ folgt: $y \geq \frac{DB}{2}$

2. Fall: $\frac{DB}{2} \leq y < DB$

Es gilt:

2. Regionen A und B sind nicht erreichbar.

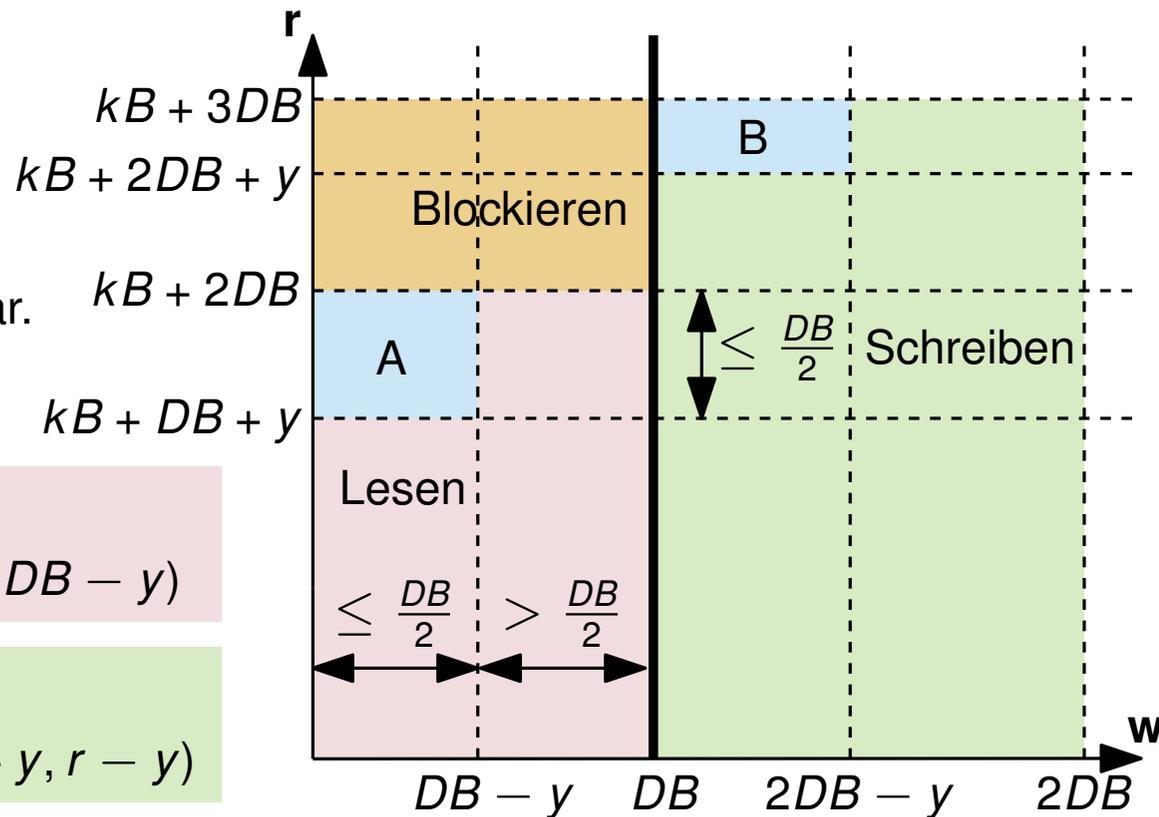
denn $\frac{DB}{2} \leq y$ und

Fall $w < DB$

Zustandsübergang: $(w, r) \rightsquigarrow (w + y, r + DB - y)$

Fall $w \geq DB$

Zustandsübergang: $(w, r) \rightsquigarrow (w - DB + y, r - y)$



2. Phase: Merging

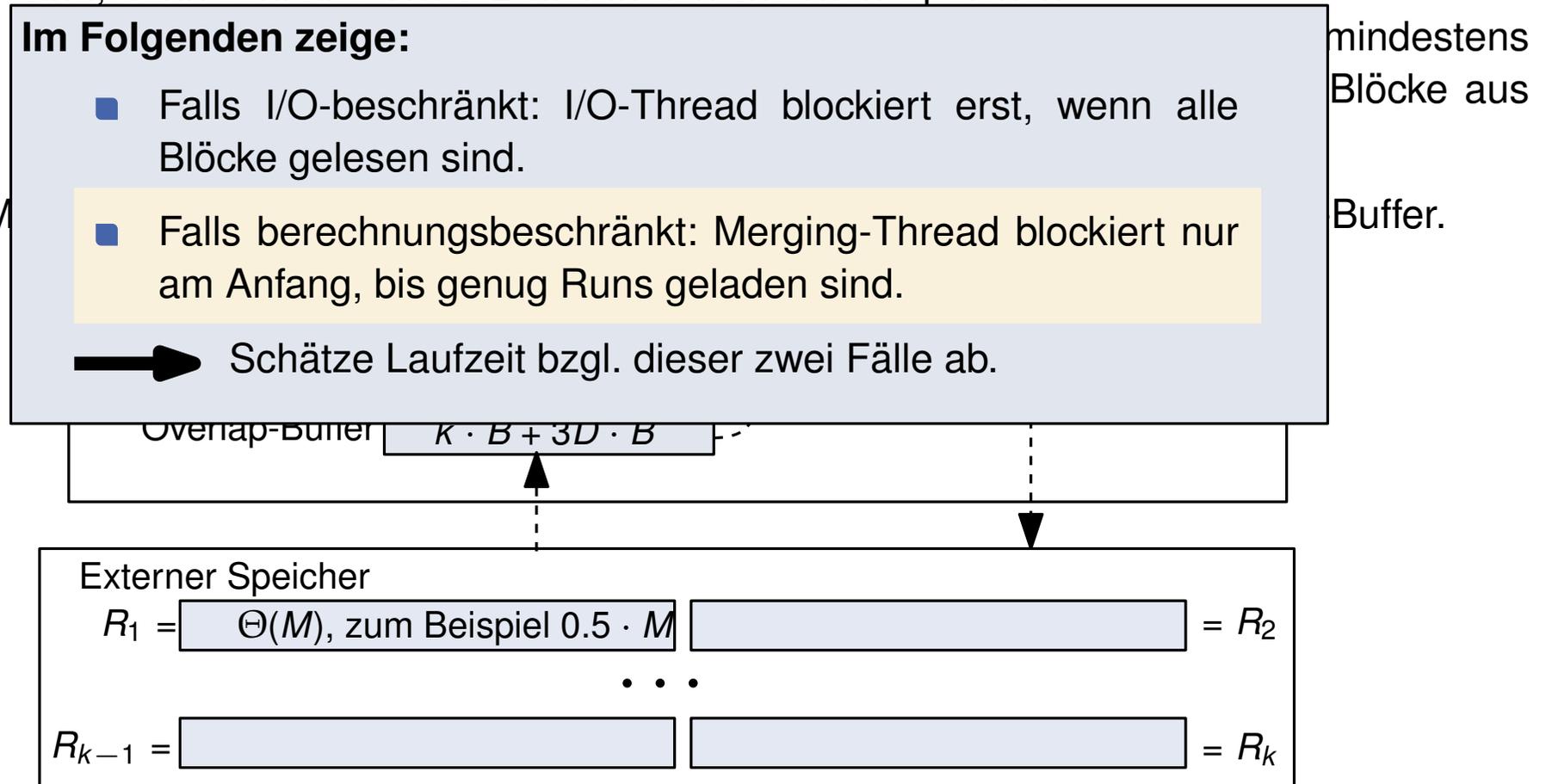
Anpassungen:

- 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. Im Folgenden zeige:

- Falls I/O-beschränkt: I/O-Thread blockiert erst, wenn alle Blöcke gelesen sind.
- Falls berechnungsbeschränkt: Merging-Thread blockiert nur am Anfang, bis genug Runs geladen sind.

➔ Schätze Laufzeit bzgl. dieser zwei Fälle ab.



2. Fall: Berechnungsbeschränkt

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 initial $\frac{k}{D} + 1$ I/O-Schritten erst dann wieder blockieren, wenn alle Elemente vermengt worden sind.

→ Merging-Thread ist langsamer als I/O-Thread

2. Fall: Berechnungsbeschränkt

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 initial $\frac{k}{D} + 1$ I/O-Schritten erst dann wieder blockieren, wenn alle Elemente vermengt worden sind.

$y = \frac{L}{\ell}$: Anzahl Elemente, die während eines I/O-Schritts vermengt werden können.

w = Anzahl Elemente im Write-Buffer. r = Anzahl Elemente im Overlap- und Merge-Buffer.

Aus $2L < DB\ell$ folgt: $y < \frac{DB}{2}$

2. Fall: Berechnungsbeschränkt

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 initial $\frac{k}{D} + 1$ I/O-Schritten erst dann wieder blockieren, wenn alle Elemente vermengt worden sind.

$y = \frac{L}{\ell}$: Anzahl Elemente, die während eines I/O-Schritts vermengt werden können.

w = Anzahl Elemente im Write-Buffer. r = Anzahl Elemente im Overlap- und Merge-Buffer.

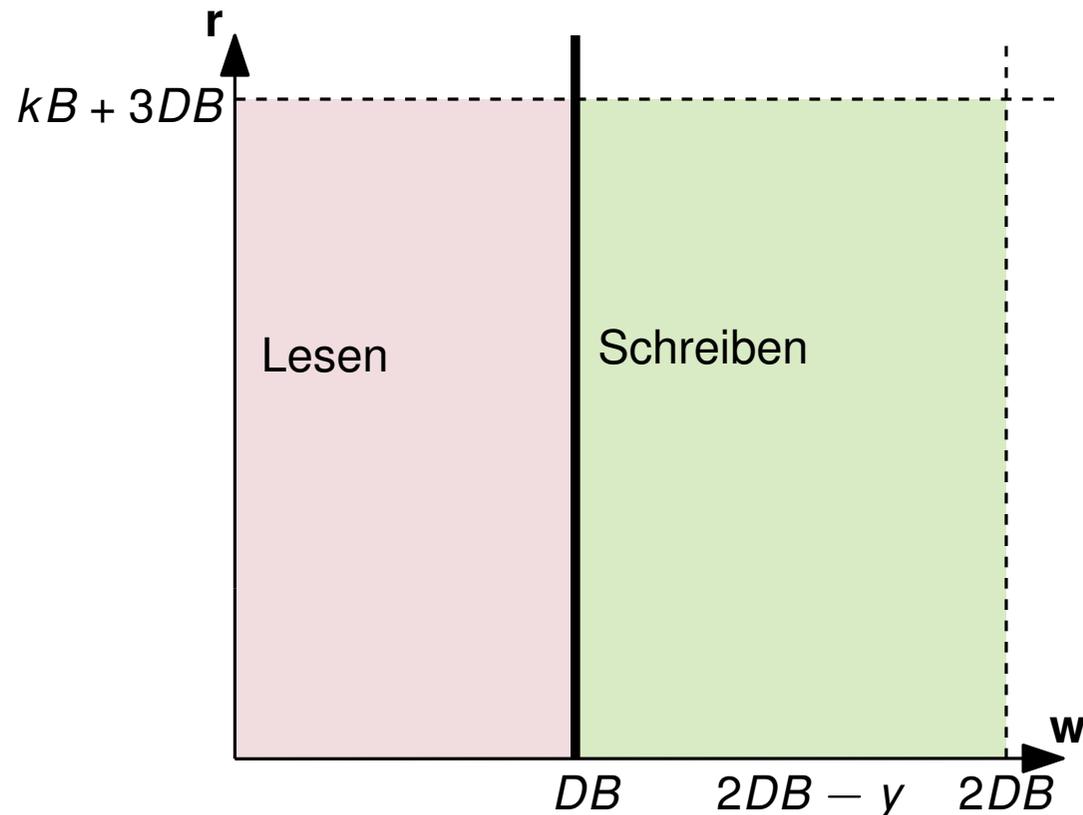
Aus $2L < DB\ell$ folgt: $y < \frac{DB}{2}$

I/O-Thread liest nur wenn

$$w < DB$$

I/O-Thread schreibt nur, wenn

$$w \geq DB$$



2. Fall: Berechnungsbeschränkt

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 initial $\frac{k}{D} + 1$ I/O-Schritten erst dann wieder blockieren, wenn alle Elemente vermengt worden sind.

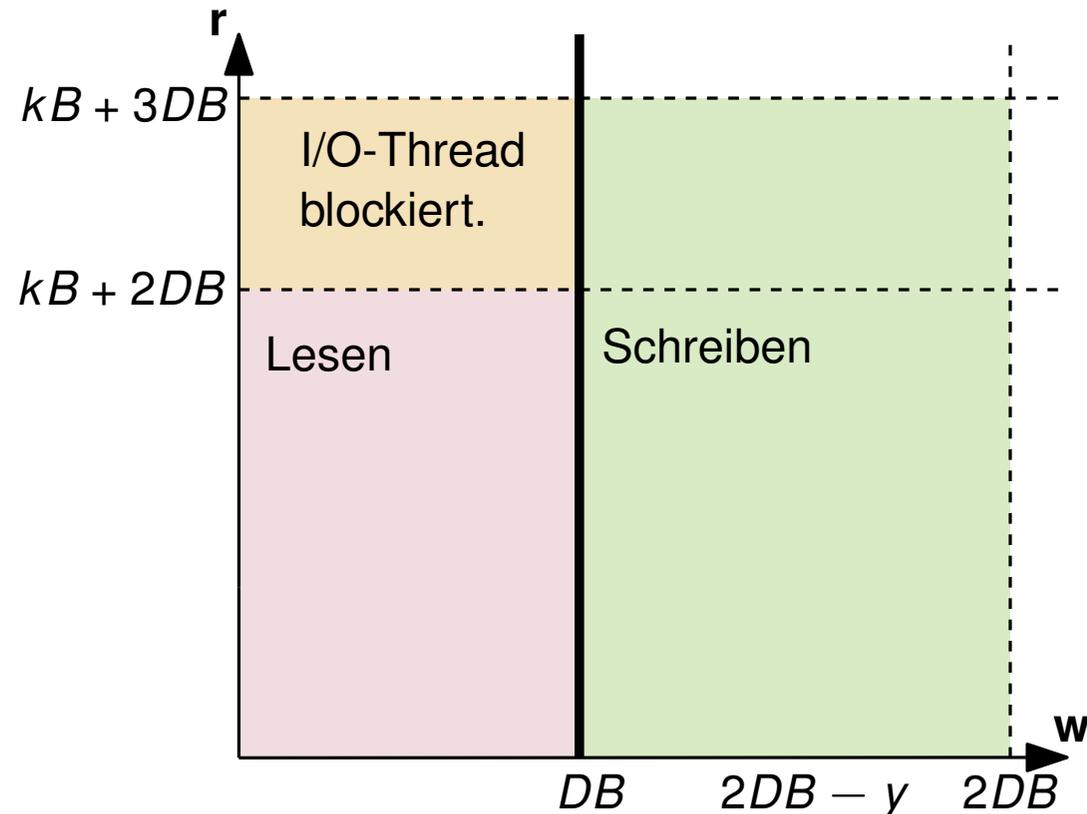
$y = \frac{L}{\ell}$: Anzahl Elemente, die während eines I/O-Schritts vermengt werden können.

w = Anzahl Elemente im Write-Buffer. r = Anzahl Elemente im Overlap- und Merge-Buffer.

Aus $2L < DB\ell$ folgt: $y < \frac{DB}{2}$

I/O-Thread liest nur wenn
 $w < DB$

I/O-Thread schreibt nur, wenn
 $w \geq DB$



2. Fall: Berechnungsbeschränkt

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 initial $\frac{k}{D} + 1$ I/O-Schritten erst dann wieder blockieren, wenn alle Elemente vermengt worden sind.

$y = \frac{L}{\ell}$: Anzahl Elemente, die während eines I/O-Schritts vermengt werden können.

w = Anzahl Elemente im Write-Buffer. r = Anzahl Elemente im Overlap- und Merge-Buffer.

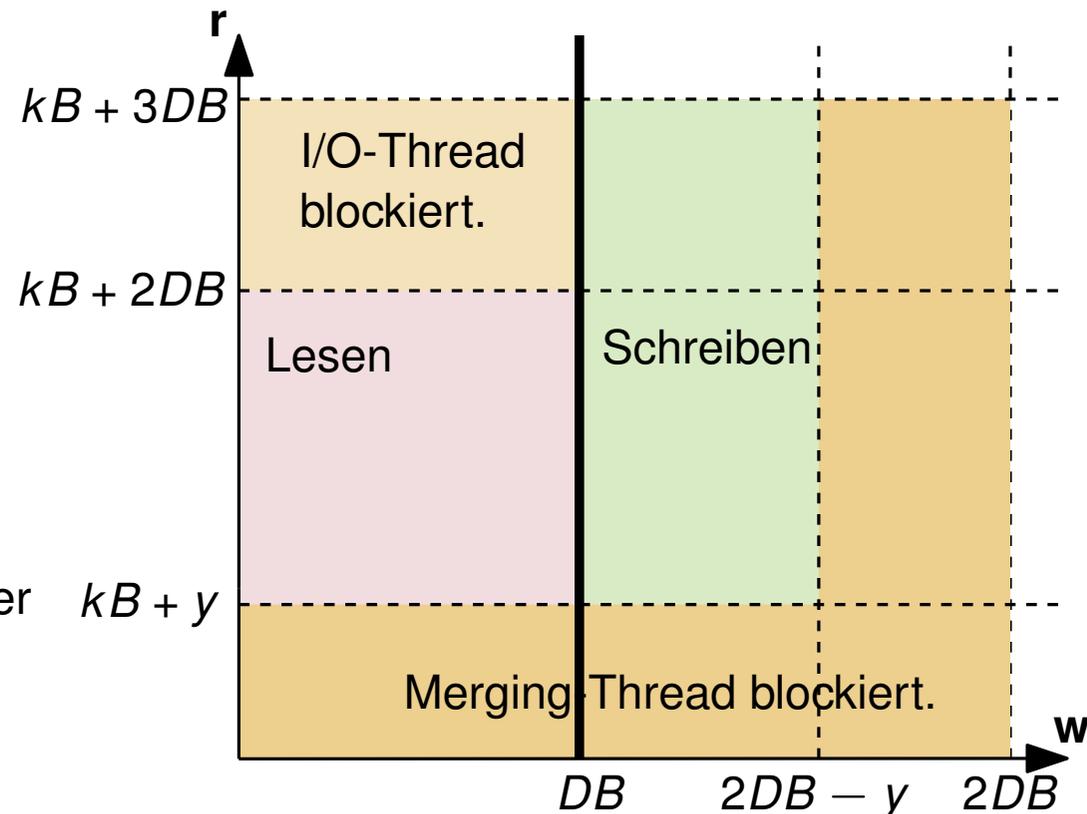
Aus $2L < DB\ell$ folgt: $y < \frac{DB}{2}$

Merging-Thread blockiert gdw.

$$w > 2DB - y \text{ oder } r < kB + y$$

Writer-Buffer ist voll

Overlap- und Merge-Buffer sind leer $kB + y$



2. Fall: Berechnungsbeschränkt

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 initial $\frac{k}{D} + 1$ I/O-Schritten erst dann wieder blockieren, wenn alle Elemente vermengt worden sind.

$y = \frac{L}{\ell}$: Anzahl Elemente, die während eines I/O-Schritts vermengt werden können.

w = Anzahl Elemente im Write-Buffer. r = Anzahl Elemente im Overlap- und Merge-Buffer.

Aus $2L < DB\ell$ folgt: $y < \frac{DB}{2}$

Merging-Thread blockiert gdw.

$$w > 2DB - y \text{ oder } r < kB + y$$

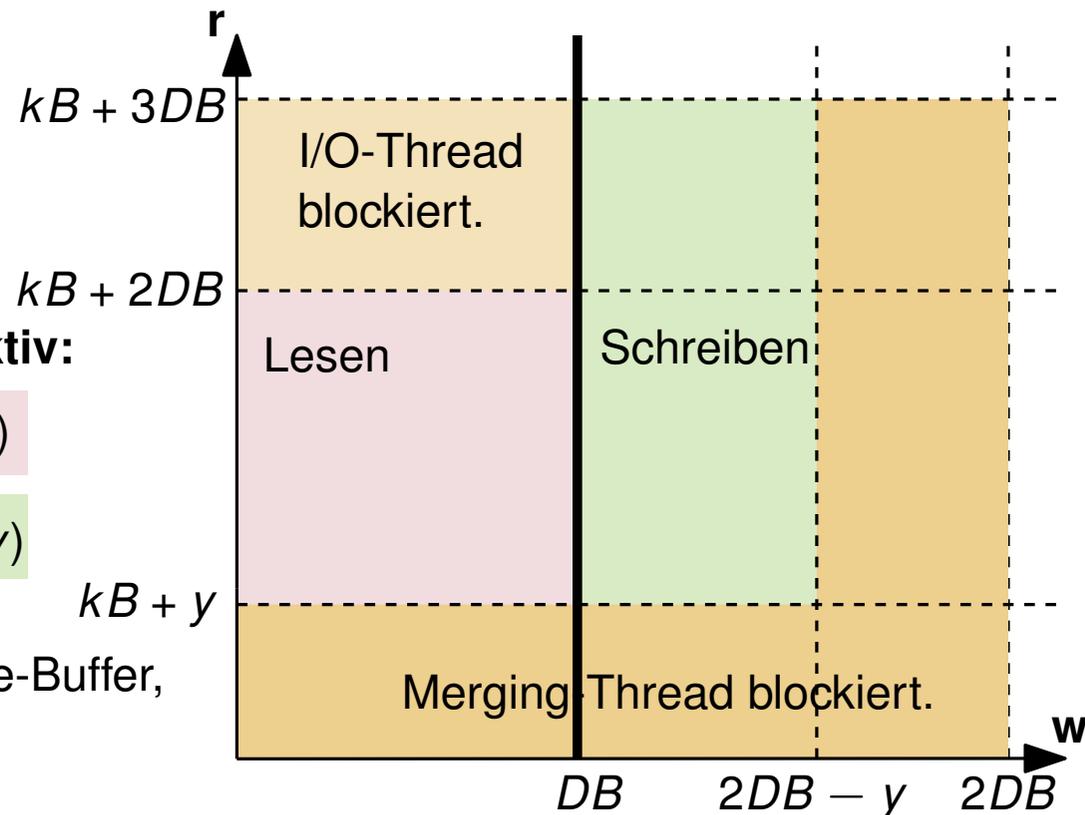
Folgende Übergänge, wenn I/O-Thread aktiv:

Fall $w < DB$: $(w, r) \rightsquigarrow (w + y, r + DB - y)$

Fall $w \geq DB$: $(w, r) \rightsquigarrow (w - DB + y, r - y)$

Ansonsten:

Merging-Thread bewegt Element in den Write-Buffer, bis I/O-Thread wieder aktiv ist.



2. Fall: Berechnungsbeschränkt

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 initial $\frac{k}{D} + 1$ I/O-Schritten erst dann wieder blockieren, wenn alle Elemente vermengt worden sind.

$y = \frac{L}{\ell}$: Anzahl Elemente, die während eines I/O-Schritts vermengt werden können.

w = Anzahl Elemente im Write-Buffer. r = Anzahl Elemente im Overlap- und Merge-Buffer.

Aus $2L < DB\ell$ folgt: $y < \frac{DB}{2}$

Merging-Thread blockiert gdw.

$$w > 2DB - y \text{ oder } r < kB + y$$

Folgende Übergänge, wenn I/O-Thread aktiv:

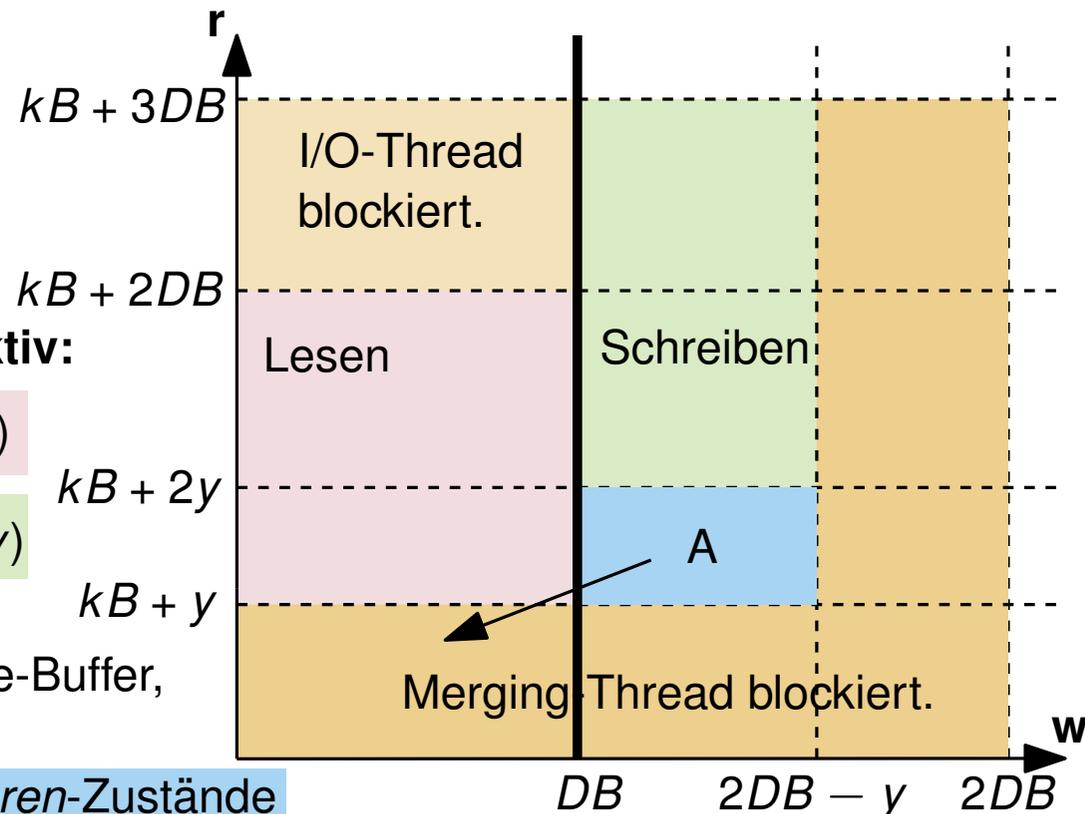
Fall $w < DB$: $(w, r) \rightsquigarrow (w + y, r + DB - y)$

Fall $w \geq DB$: $(w, r) \rightsquigarrow (w - DB + y, r - y)$

Ansonsten:

Merging-Thread bewegt Element in den Write-Buffer, bis I/O-Thread wieder aktiv ist.

➔ Region A einziger Zugang für *Blockieren*-Zustände



2. Fall: Berechnungsbeschränkt

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 initial $\frac{k}{D} + 1$ I/O-Schritten erst dann wieder blockieren, wenn alle Elemente vermengt worden sind.

$y = \frac{L}{\ell}$: Anzahl Elemente, die während eines I/O-Schritts vermengt werden können.

w = Anzahl Elemente im Write-Buffer. r = Anzahl Elemente im Overlap- und Merge-Buffer.

Aus $2L < DB\ell$ folgt: $y < \frac{DB}{2}$

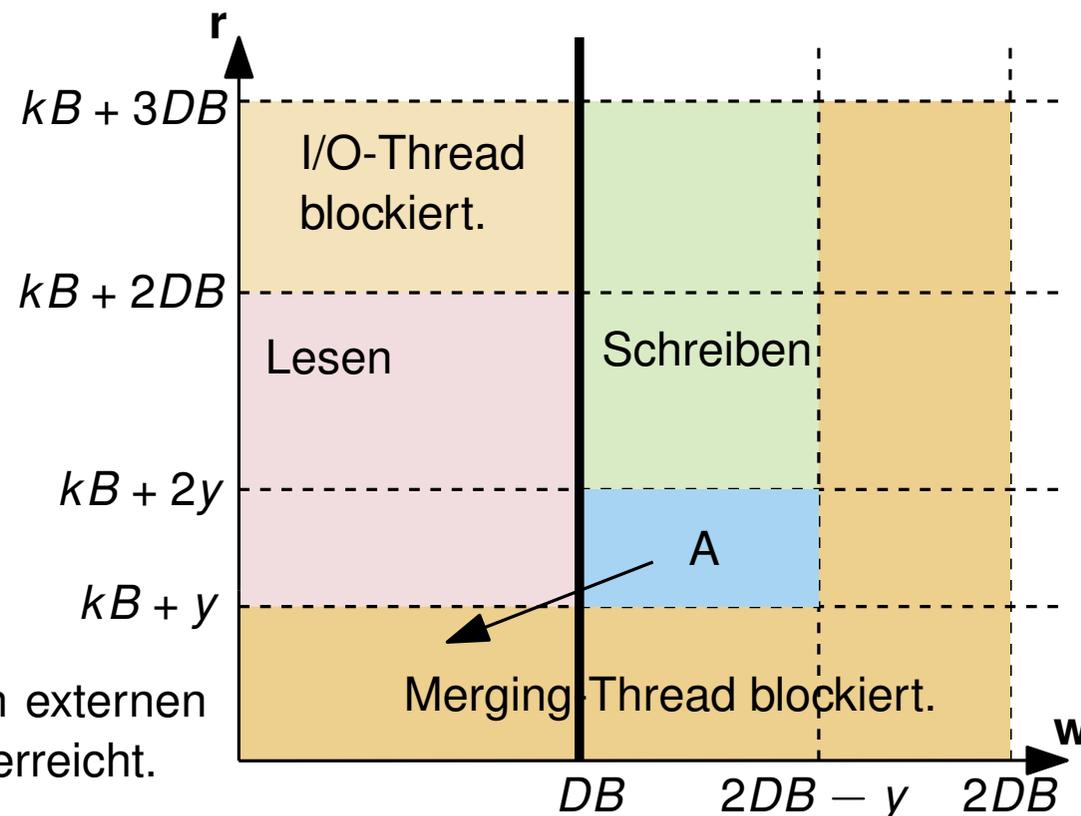
Merging-Thread blockiert gdw.

$$w > 2DB - y \text{ oder } r < kB + y$$

Region A aber nicht erreichbar, von Zuständen in denen der Merging-Thread aktiv ist.

→ $y < \frac{DB}{2}$

→ I/O-Thread schreibt Write-Buffer in externen Speicher, bevor Merging-Thread A erreicht.



2. Phase: Merging

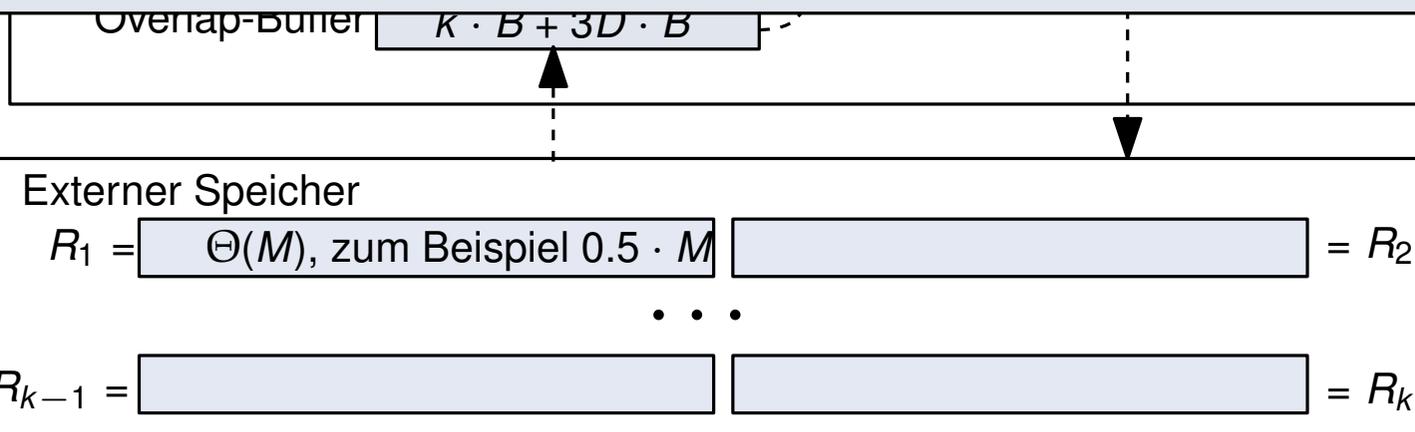
Anpassungen:

- 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. Im Folgenden zeige:

- Falls I/O-beschränkt: I/O-Thread blockiert erst, wenn alle Blöcke gelesen sind.
- Falls berechnungsbeschränkt: Merging-Thread blockiert nur am Anfang, bis genug Runs geladen sind.

➔ Schätze Laufzeit bzgl. dieser zwei Fälle ab.



mindestens
Blöcke aus
Buffer.

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

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

vereinigt werden.

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

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

vereinigt werden.

1. Fall: $2L \geq DB\ell$

Lemma 6 impliziert, dass nach $\frac{Ln}{DB}$ Zeit alle Blöcke geladen sind.

Lemma 6: Falls $2L \geq DB\ell$, blockiert der I/O-Thread erst, wenn alle Eingabeblocks gelesen worden sind.

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

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

vereinigt werden.

1. Fall: $2L \geq DB\ell$

Lemma 6 impliziert, dass nach $\frac{Ln}{DB}$ Zeit alle Blöcke geladen sind.

Es müssen noch $O((k + D)B)$ Elemente aus dem Overlap- und Merge-Buffer verarbeitet und in den externen Speicher geschrieben werden.

↳ Benötigt $O(\ell(k + D)B + L\lceil k/D\rceil) = O(L\lceil k/D\rceil)$ Zeit.

Lemma 6: Falls $2L \geq DB\ell$, blockiert der I/O-Thread erst, wenn alle Eingabeblocks gelesen worden sind.

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

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

vereinigt werden.

1. Fall: $2L \geq DB\ell$ **Laufzeit:** $O\left(2 \cdot \frac{Ln}{DB} + L\lceil k/D \rceil\right)$

Lemma 6 impliziert, dass nach $\frac{Ln}{DB}$ Zeit alle Blöcke geladen sind.

Es müssen noch $O((k + D)B)$ Elemente aus dem Overlap- und Merge-Buffer verarbeitet und in den externen Speicher geschrieben werden.

↳ Benötigt $O(\ell(k + D)B + L\lceil k/D \rceil) = O(L\lceil k/D \rceil)$ Zeit.

Insgesamt: $O\left(2 \cdot \frac{Ln}{DB} + L\lceil k/D \rceil\right)$ Jeder Block muss auch im externen Speicher wieder abgelegt werden.

Lemma 6: Falls $2L \geq DB\ell$, blockiert der I/O-Thread erst, wenn alle Eingabeblocks gelesen worden sind.

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

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

vereinigt werden.

1. Fall: $2L \geq DB\ell$ **Laufzeit:** $O\left(2 \cdot \frac{Ln}{DB} + L\lceil k/D \rceil\right)$

2. Fall: $2L < DB\ell$ **Laufzeit:** $O(\ell n)$

- Lemma 7 impliziert, dass nach $k/D + 1$ vielen I/O-Schritten, der Merging-Thread beginnt zu arbeiten und erst aufhört, wenn alle Element abgearbeitet sind.
- Für die Abarbeitung aller Elemente benötigt der Merge-Thread $O(\ell n)$ Zeit.

Lemma 7: 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.