Karlsruher Institut für Technologie

Algorithmen II
Vorlesung am 30.01.2014

Online Algorithmen

INSTITUT FÜR THEORETISCHE INFORMATIK · PROF. DR. DOROTHEA WAGNER

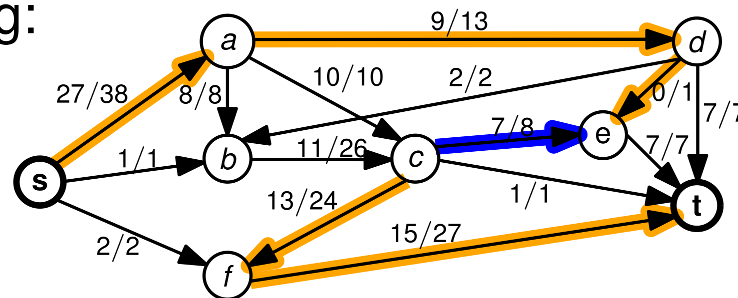
Einführung

Bisher in der Vorlesung betrachtet: **Offline-Algorithmen**

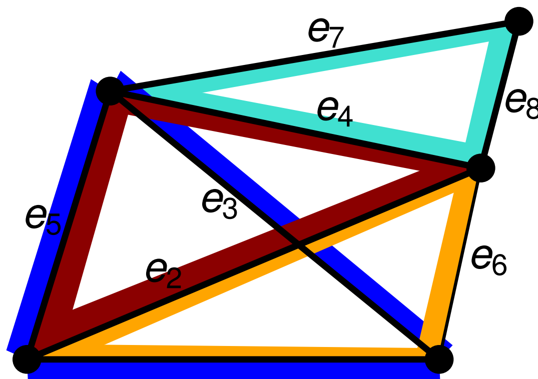
Charakteristische Merkmale:

- Eingabe muss vor Beginn der Ausführung bekannt sein.
- Berechnungen geschehen auf der gesamten Eingabe.

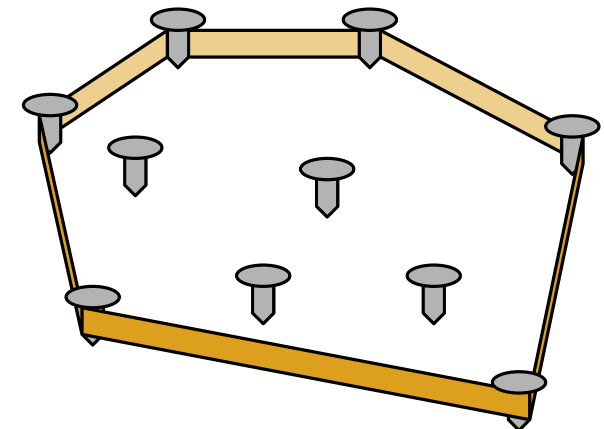
Beispiele aus der Vorlesung:



Flussalgorithmen



Algorithmen für Kreisbasen


Algorithmus für konvexe Hülle

Einführung

Online-Algorithmus: Ein Algorithmus heißt *Online-Algorithmus*, wenn er eine Eingabesequenz sequentiell stückweise abarbeitet, ohne die gesamte Eingabesequenz zu betrachten.

Charakteristische Merkmale:

- Algorithmus liefert nach Abarbeitung einzelner Blöcke der Eingabesequenz bereits eine Lösung.
- Entscheidungen beruhen nur auf bereits vergangenen Ereignissen, aber nicht auf zukünftigen.

Beispiel: Job Scheudling eines Prozessors

Gesucht: Zuweisung für Prozessor, sodass durchschnittliche Wartezeit minimiert wird.

Kompetitive Analyse

Karlsruher Institut für Technologie

Idee: Bestimme Qualität eines Online-Algorithmus mithilfe eines optimalen Offline-Algorithmus.

Definition 35: Ein Online-Algorithmus $\mathcal{A}$ heißt *c-kompetitiv*, falls es eine Konstante α gibt, sodass für alle Instanzen I gilt

$$\mathcal{A}(I) \leq c \cdot \mathrm{OPT}(I) + \alpha.$$

Dabei bezeichnet $\mathcal{A}(I)$ den Wert der Lösung von $\mathcal{A}$ auf I angewendet und $\mathrm{OPT}(I)$ den Wert einer optimalen Lösung für I.
Falls $\alpha \leq 0$, dann heißt $\mathcal{A}$ *strikt c-kompetitiv*.

Bemerkung:

- Formulierung für Minimierungsprob., Maximierungsprob. kann analog formuliert werden.
- $\mathcal{A}$ heißt *kompetitiv*, falls es Konstante c gibt, sodass $\mathcal{A}$ c-kompetitiv ist.
- c wird die *relative Güte* genannt.
- Strikter c-kompetitiver Online-Algorithmus ist Approximationsalgorithmus mit relativer Güte c.

Beobachtung: Wenn $\mathcal{A}$ c-kompetitiv ist, dann ist er auch c'-kompetitiv für alle $c' \geq c$.

Ziel: Finde für ein gegebenes Problem einen Online-Algorithmus mit minimaler relativer Güte.

Beispiel: Ski-Verleih

Szenario: Man möchte mehrmals in die Alpen, um dort Ski zu fahren. Allerdings weiß man noch nicht wie häufig und man besitzt auch noch keine eigenen Ski.
Man steht nun vor der Entscheidung: Entweder man leiht sich im Skigebiet Ski für 50 € für den jeweiligen Ausflug, oder man kauft sich eigene Ski für 300 €.

Frage: Wie verhält man sich am besten?

Beispiel: Ski-Verleih

Szenario: Man möchte mehrmals in die Alpen, um dort Ski zu fahren. Allerdings weiß man noch nicht wie häufig und man besitzt auch noch keine eigenen Ski.
Man steht nun vor der Entscheidung: Entweder man leiht sich im Skigebiet Ski für 50 € für den jeweiligen Ausflug, oder man kauft sich eigene Ski für 300 €.

Frage: Wie verhält man sich am besten?

Einfacher Online-Algorithmus $\mathcal{A}$: Leihe für die ersten fünf Ausflüge Ski und kaufe vor dem sechsten eigene.

Beispiel: Ski-Verleih

Karlsruher Institut für Technologie

Szenario: Man möchte mehrmals in die Alpen, um dort Ski zu fahren. Allerdings weiß man noch nicht wie häufig und man besitzt auch noch keine eigenen Ski.
Man steht nun vor der Entscheidung: Entweder man leiht sich im Skigebiet Ski für 50 € für den jeweiligen Ausflug, oder man kauft sich eigene Ski für 300 €.

Frage: Wie verhält man sich am besten?

Einfacher Online-Algorithmus $\mathcal{A}$: Leihe für die ersten fünf Ausflüge Ski und kaufe vor dem sechsten eigene.

Analyse: Sei k die Anzahl der Ausflüge, die man tatsächlich unternimmt.

Sei $k \leq 5$: Man zahlt $k \cdot 50$. Die Lösung eines optimalen Offline-Algorithmus ist ebenfalls $k \cdot 50$ für $k \leq 5$. Es ergibt sich eine relative Güte von $\frac{k \cdot 50}{k \cdot 50} = 1$.

Sei $k > 5$: Man zahlt $5 \cdot 50 + 300$. Die Lösung eines optimalen Offline-Algorithmus ist 300. Es ergibt sich eine relative Güte von $\frac{550}{300} = \frac{11}{6}$.

Algorithmus ist $\frac{11}{6}$-kompetitiv.

Institut für Theoretische Informatik
Prof. Dr. Dorothea Wagner

Beispiel: Ski-Verleih

Karlsruher Institut für Technologie

Szenario: Man möchte mehrmals in die Alpen, um dort Ski zu fahren. Allerdings weiß man noch nicht wie häufig und man besitzt auch noch keine eigenen Ski.
Man steht nun vor der Entscheidung: Entweder man leiht sich im Skigebiet Ski für 50 € für den jeweiligen Ausflug, oder man kauft sich eigene Ski für 300 €.

Gibt es einen besseren Online-Algorithmus?

Annahme: Man kauft für den x-ten Ausflug eigene Ski und unternimmt tatsächlich k Ausflüge.

Beispiel: Ski-Verleih

Karlsruher Institut für Technologie

Szenario: Man möchte mehrmals in die Alpen, um dort Ski zu fahren. Allerdings weiß man noch nicht wie häufig und man besitzt auch noch keine eigenen Ski.
Man steht nun vor der Entscheidung: Entweder man leiht sich im Skigebiet Ski für 50 € für den jeweiligen Ausflug, oder man kauft sich eigene Ski für 300 €.

Gibt es einen besseren Online-Algorithmus?

Annahme: Man kauft für den x-ten Ausflug eigene Ski und unternimmt tatsächlich k Ausflüge.

1. Fall: $x < 6$

Für $k < x$ bleibt die relative Güte 1.
Für $x = k$ liefert der optimale Algorithmus $50 \cdot x$ und der Online-Algorithmus $50 \cdot (x - 1) + 300$:

$$\frac{50 \cdot x + 250}{50 \cdot x} = 1 + \frac{5}{x} \geq 2$$

Beispiel: Ski-Verleih

Karlsruher Institut für Technologie

Szenario: Man möchte mehrmals in die Alpen, um dort Ski zu fahren. Allerdings weiß man noch nicht wie häufig und man besitzt auch noch keine eigenen Ski.
Man steht nun vor der Entscheidung: Entweder man leiht sich im Skigebiet Ski für 50 € für den jeweiligen Ausflug, oder man kauft sich eigene Ski für 300 €.

Gibt es einen besseren Online-Algorithmus?

Annahme: Man kauft für den x-ten Ausflug eigene Ski und unternimmt tatsächlich k Ausflüge.

1. Fall: $x < 6$

Für $k < x$ bleibt die relative Güte 1.
Für $x = k$ liefert der optimale Algorithmus $50 \cdot x$ und der Online-Algorithmus $50 \cdot (x - 1) + 300$:

$$\frac{50 \cdot x + 250}{50 \cdot x} = 1 + \frac{5}{x} \geq 2$$

2. Fall: $x > 6$

Für $x = k$ liefert der optimale Algorithmus 300 während der Online-Algorithmus $50 \cdot (x - 1) + 300$:

$$\frac{50 \cdot x + 250}{300} = \frac{5 + x}{6}$$

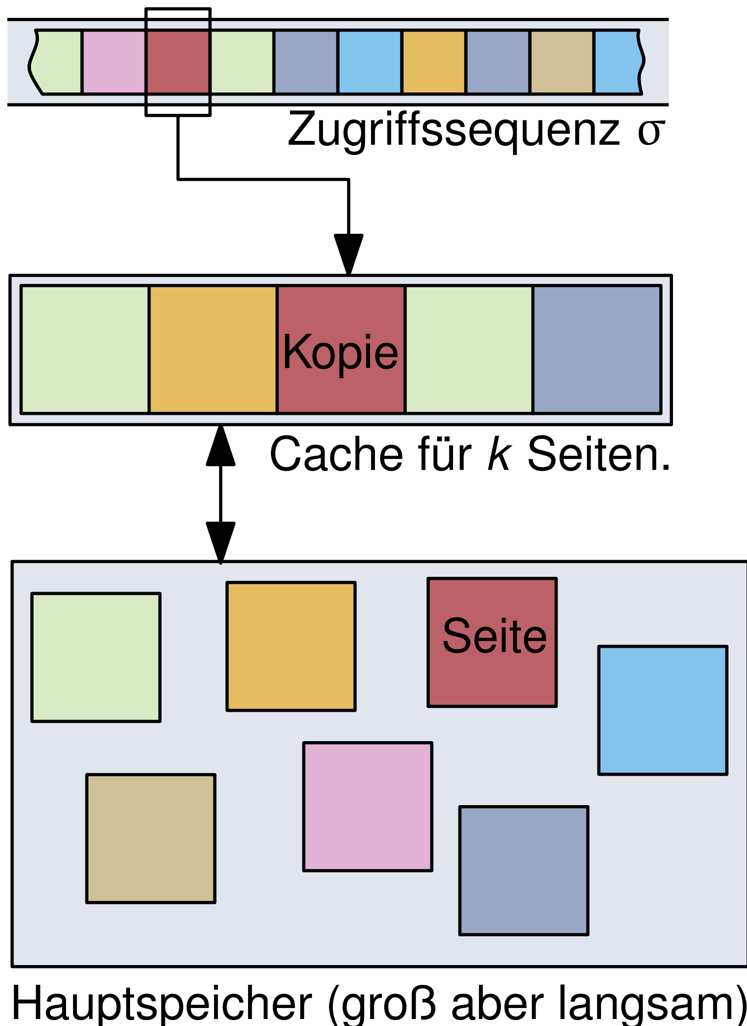
Bester Online-Algorithmus ist $\frac{11}{6}$-kompetitiv (mit $x = 6$)

Institut für Theoretische Informatik
Prof. Dr. Dorothea Wagner

Online-Algorithmen für Paging

Institut für Theoretische Informatik
Prof. Dr. Dorothea Wagner

Paging



- Hauptspeicher enthält N Seiten: $P = \{p_1, \ldots, p_N\}$
- Cache kann k Kopien von Seiten aus dem Hauptspeicher enthalten ($k < N$).
- n sequentielle Seitenzugriffe eines Programms werden beschrieben durch die Sequenz

$$\sigma \colon \{1, \ldots, n\} \to \{1, \ldots, N\}$$

Ablauf:

1. Programm fragt die i-te Seite $p_{\sigma(i)}$ an.
2. Falls $p_{\sigma(i)}$ noch nicht im Cache enthalten ist (**Fehlzugriff**), dann wird $p_{\sigma(i)}$ in den Cache geladen.
3. Programm greift auf $p_{\sigma(i)}$ im Cache zu.

Wie Seiten im Cache ersetzen, damit möglichst wenige Fehlzugriffe auftreten?

Institut für Theoretische Informatik
Prof. Dr. Dorothea Wagner

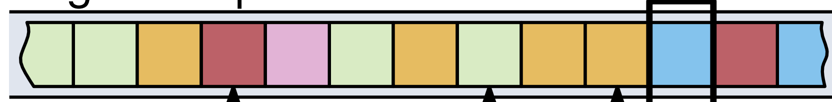
Strategien

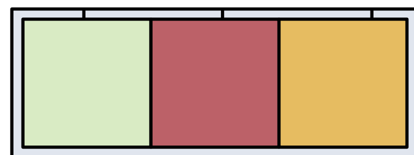
Name	Ersetze Seite im Cache,...
FIFO: First In/First Out	die bereits am längsten im Cache ist.
LIFO: Last In/First Out	die am neusten im Cache ist.
LFU: Least Frequently Used	die bisher am wenigsten häufig angefordert wurde.
LRU: Least Recently Used	deren Anfrage am weitesten in der Vergangenheit liegt.
FWF: Flush When Full	(Gebe alle Seiten frei, wenn Cache voll ist.)
LFD: Longest Forward Distance	deren Anfrage am weitesten in der Zukunft liegt.

Alle bis auf LFD sind Online-Algorithmen. LFD muss die komplette Anfragesequenz kennen und ist somit ein Offline-Algorithmus.

Beispiel LRU:

Zugriffssequenz σ





Cache

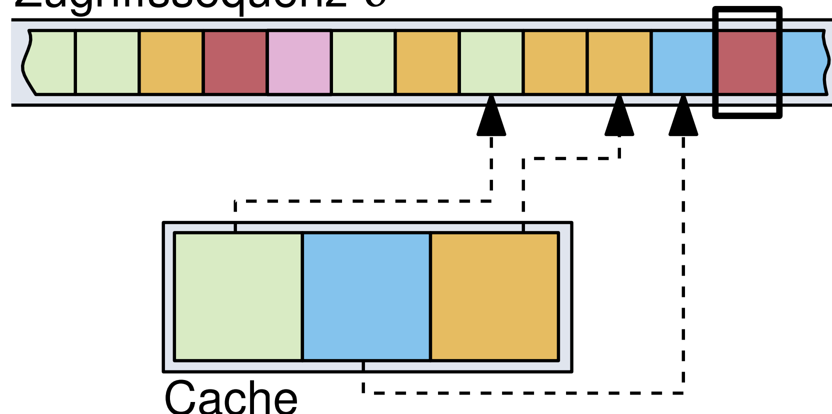
1. Speichere für jede Seite im Cache, wann sie zuletzt verwendet wurde.
2. Ersetze die Seite im Cache, deren Verwendung am weitesten in der Vergangenheit liegt.

Strategien

Name	Ersetze Seite im Cache,...
FIFO: First In/First Out	die bereits am längsten im Cache ist.
LIFO: Last In/First Out	die am neusten im Cache ist.
LFU: Least Frequently Used	die bisher am wenigsten häufig angefordert wurde.
LRU: Least Recently Used	deren Anfrage am weitesten in der Vergangenheit liegt.
FWF: Flush When Full	(Gebe alle Seiten frei, wenn Cache voll ist.)
LFD: Longest Forward Distance	deren Anfrage am weitesten in der Zukunft liegt.

Alle bis auf LFD sind Online-Algorithmen. LFD muss die komplette Anfragesequenz kennen und ist somit ein Offline-Algorithmus.

Beispiel LRU:



1. Speichere für jede Seite im Cache, wann sie zuletzt verwendet wurde.
2. Ersetze die Seite im Cache, deren Verwendung am weitesten in der Vergangenheit liegt.

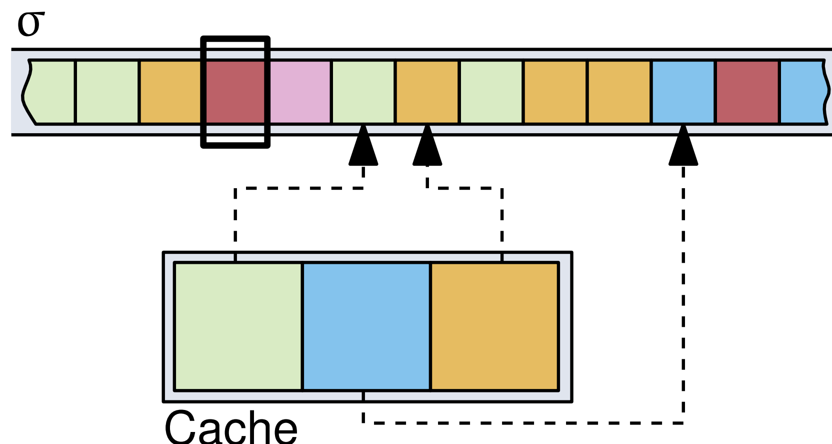
Institut für Theoretische Informatik
Prof. Dr. Dorothea Wagner

Strategien

Name	Ersetze Seite im Cache,...
FIFO: First In/First Out	die bereits am längsten im Cache ist.
LIFO: Last In/First Out	die am neusten im Cache ist.
LFU: Least Frequently Used	die bisher am wenigsten häufig angefordert wurde.
LRU: Least Recently Used	deren Anfrage am weitesten in der Vergangenheit liegt.
FWF: Flush When Full	(Gebe alle Seiten frei, wenn Cache voll ist.)
LFD: Longest Forward Distance	deren Anfrage am weitesten in der Zukunft liegt.

Alle bis auf LFD sind Online-Algorithmen. LFD muss die komplette Anfragesequenz kennen und ist somit ein Offline-Algorithmus.

Beispiel LFD:



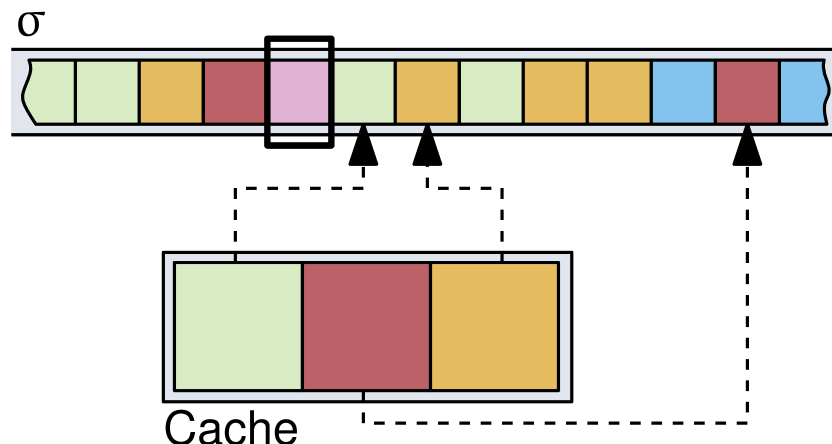
1. Speichere für jede Seite im Cache, wann sie als nächstes verwendet wird.
2. Ersetze die Seite im Cache, deren Verwendung am weitesten in der Zukunft liegt.

Strategien

Name	Ersetze Seite im Cache,...
FIFO: First In/First Out	die bereits am längsten im Cache ist.
LIFO: Last In/First Out	die am neusten im Cache ist.
LFU: Least Frequently Used	die bisher am wenigsten häufig angefordert wurde.
LRU: Least Recently Used	deren Anfrage am weitesten in der Vergangenheit liegt.
FWF: Flush When Full	(Gebe alle Seiten frei, wenn Cache voll ist.)
LFD: Longest Forward Distance	deren Anfrage am weitesten in der Zukunft liegt.

Alle bis auf LFD sind Online-Algorithmen. LFD muss die komplette Anfragesequenz kennen und ist somit ein Offline-Algorithmus.

Beispiel LFD:



1. Speichere für jede Seite im Cache, wann sie als nächstes verwendet wird.
2. Ersetze die Seite im Cache, deren Verwendung am weitesten in der Zukunft liegt.

Longest Forward Distance (LFD)

Theorem 50: LFD ist ein optimaler Offline-Algorithmus für Paging.

Beweis:

Vergleiche LFD-Algorithmus mit einem beliebigen Paging-Algorithmus $\mathcal{A}$.

Sei hierzu σ eine beliebige Sequenz mit $|\sigma|$ Anfragen.

Behauptung: Für jedes $i = 0, \ldots, |\sigma|$ gibt es einen Paging Algorithmus $\mathcal{A}_i$, sodass

1. $\mathcal{A}_i$ die ersten i Anfragen wie der LFD-Algorithmus bearbeitet, und
2. $\mathcal{A}_i$ nicht mehr Fehlzugriffe als $\mathcal{A}$ auf σ erzeugt.

⟶ Mit $i = |\sigma|$ folgt damit das Theorem.

Beweis der Behauptung mit Induktion über i.

Longest Forward Distance (LFD)

Karlsruher Institut für Technologie

Theorem 50: LFD ist ein optimaler Offline-Algorithmus für Paging.

Beweis:

Vergleiche LFD-Algorithmus mit einem beliebigen Paging-Algorithmus $\mathcal{A}$.

Sei hierzu σ eine beliebige Sequenz mit $|\sigma|$ Anfragen.

Behauptung: Für jedes $i = 0, \ldots, |\sigma|$ gibt es einen Paging Algorithmus $\mathcal{A}_i$, sodass

1. $\mathcal{A}_i$ die ersten i Anfragen wie der LFD-Algorithmus bearbeitet, und
2. $\mathcal{A}_i$ nicht mehr Fehlzugriffe als $\mathcal{A}$ auf σ erzeugt.

→ Mit $i = |\sigma|$ folgt damit das Theorem.

Beweis der Behauptung mit Induktion über i.

Induktionsanfang:

Wähle $\mathcal{A}_0$ so, dass alle Anfragen wie von $\mathcal{A}$ abgearbeitet werden.

→ $\mathcal{A}_0$ hat damit gleich viele Fehlzugriffe wie $\mathcal{A}$ auf σ.

Longest Forward Distance (LFD)

Karlsruher Institut für Technologie

Theorem 50: LFD ist ein optimaler Offline-Algorithmus für Paging.

Beweis:

Vergleiche LFD-Algorithmus mit einem beliebigen Paging-Algorithmus $\mathcal{A}$.

Sei hierzu σ eine beliebige Sequenz mit $|\sigma|$ Anfragen.

Behauptung: Für jedes $i = 0, \ldots, |\sigma|$ gibt es einen Paging Algorithmus $\mathcal{A}_i$, sodass

1. $\mathcal{A}_i$ die ersten i Anfragen wie der LFD-Algorithmus bearbeitet, und
2. $\mathcal{A}_i$ nicht mehr Fehlzugriffe als $\mathcal{A}$ auf σ erzeugt.

→ Mit $i = |\sigma|$ folgt damit das Theorem.

Beweis der Behauptung mit Induktion über i.

Induktionsschritt:

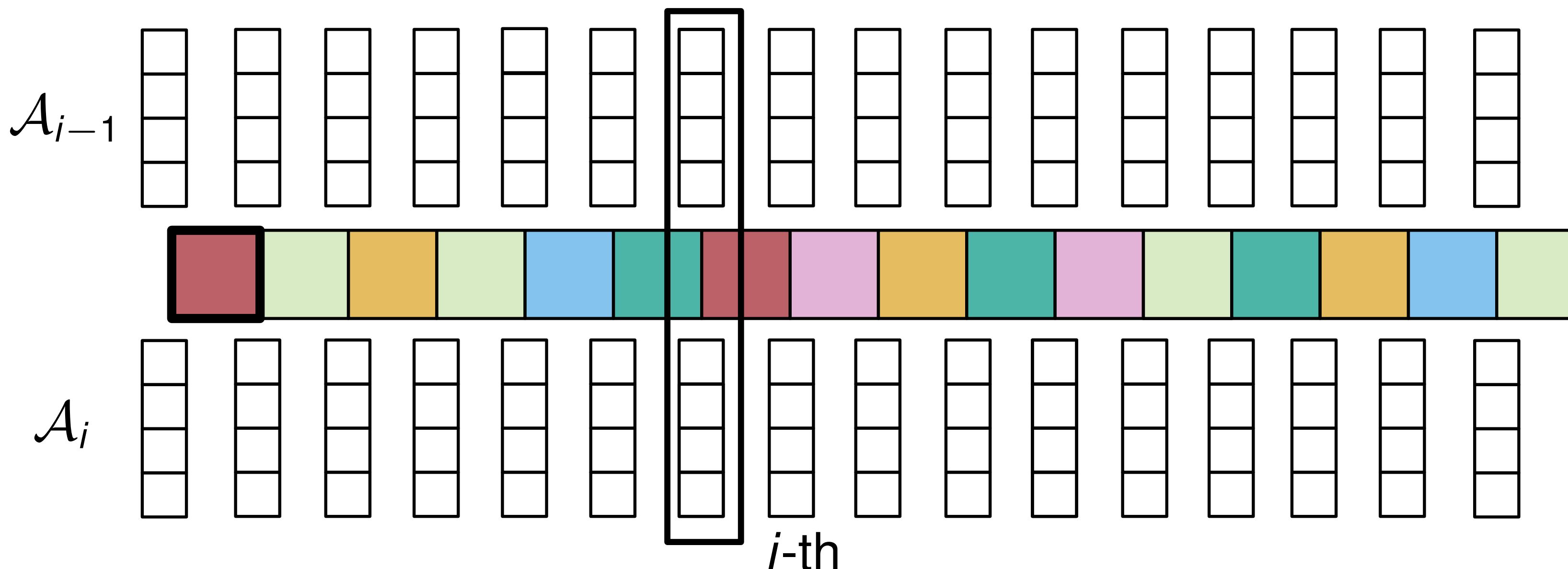
1. Nehme an, dass Behauptung für $i - 1$ gilt.
2. Konstruiere aufbauend auf $\mathcal{A}_{i-1}$ den Algorithmus $\mathcal{A}_i$.

   → Insbesondere definiere, wie $\mathcal{A}_i$ die Anfragen i bis $|\sigma|$ abarbeitet.
3. Zeige: $\mathcal{A}_i$ hat nicht mehr Fehlzugriffe auf σ als $\mathcal{A}_{i-1}$.

Longest Forward Distance (LFD)

Konstruktion von $\mathcal{A}_i$ aufbauend auf $\mathcal{A}_{i-1}$

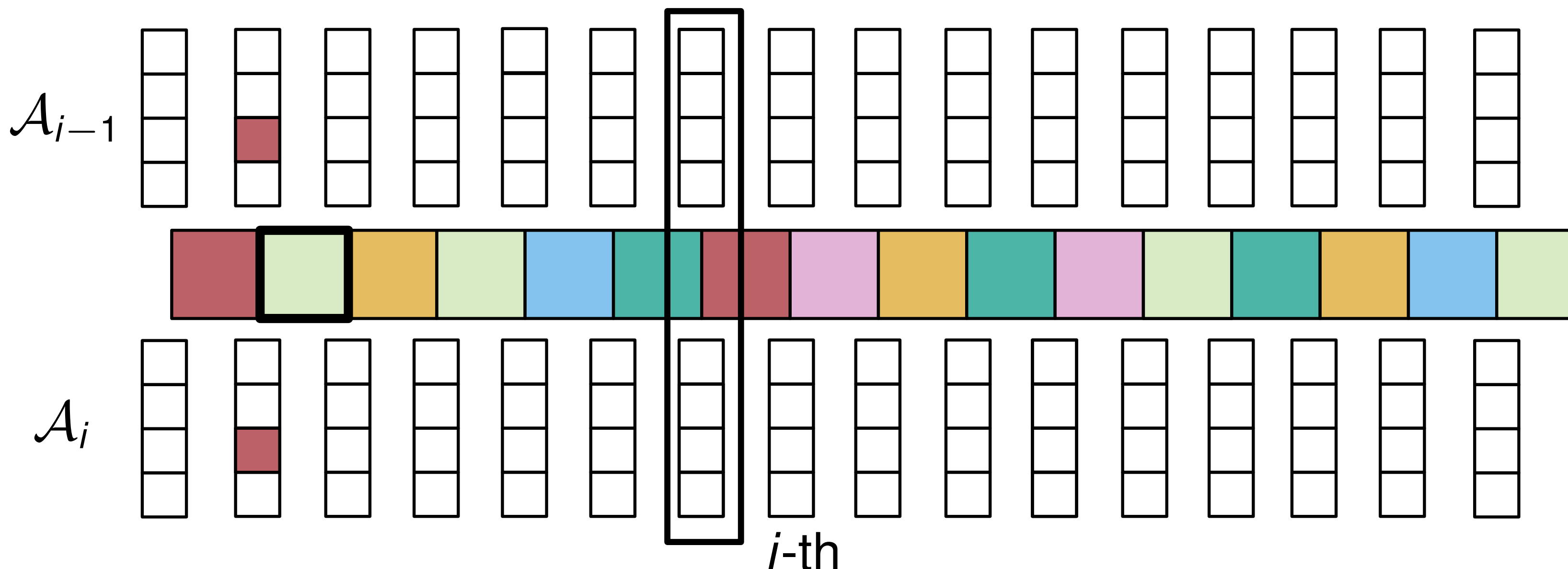
1. $\mathcal{A}_i$ bearbeitet die ersten $i-1$ Anfragen wie $\mathcal{A}_{i-1}$ und damit wie LFD.



Longest Forward Distance (LFD)

Konstruktion von $\mathcal{A}_i$ aufbauend auf $\mathcal{A}_{i-1}$

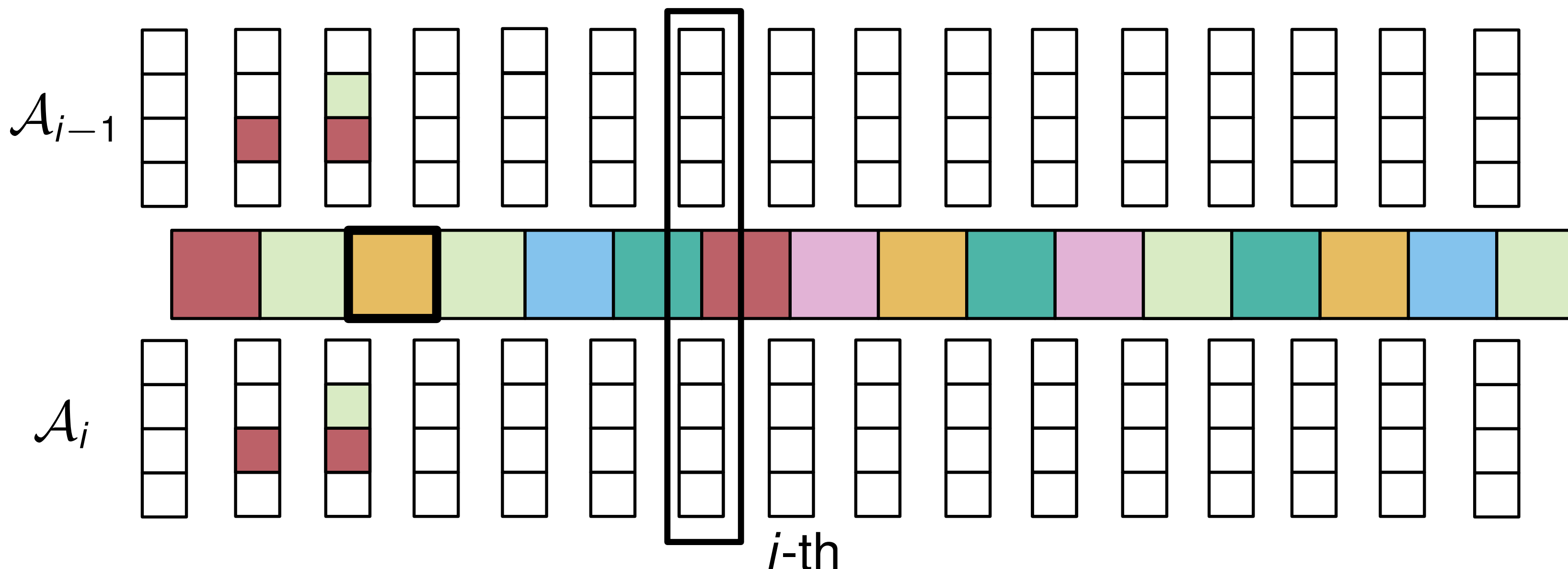
1. $\mathcal{A}_i$ bearbeitet die ersten $i-1$ Anfragen wie $\mathcal{A}_{i-1}$ und damit wie LFD.



Longest Forward Distance (LFD)

Konstruktion von $\mathcal{A}_i$ aufbauend auf $\mathcal{A}_{i-1}$

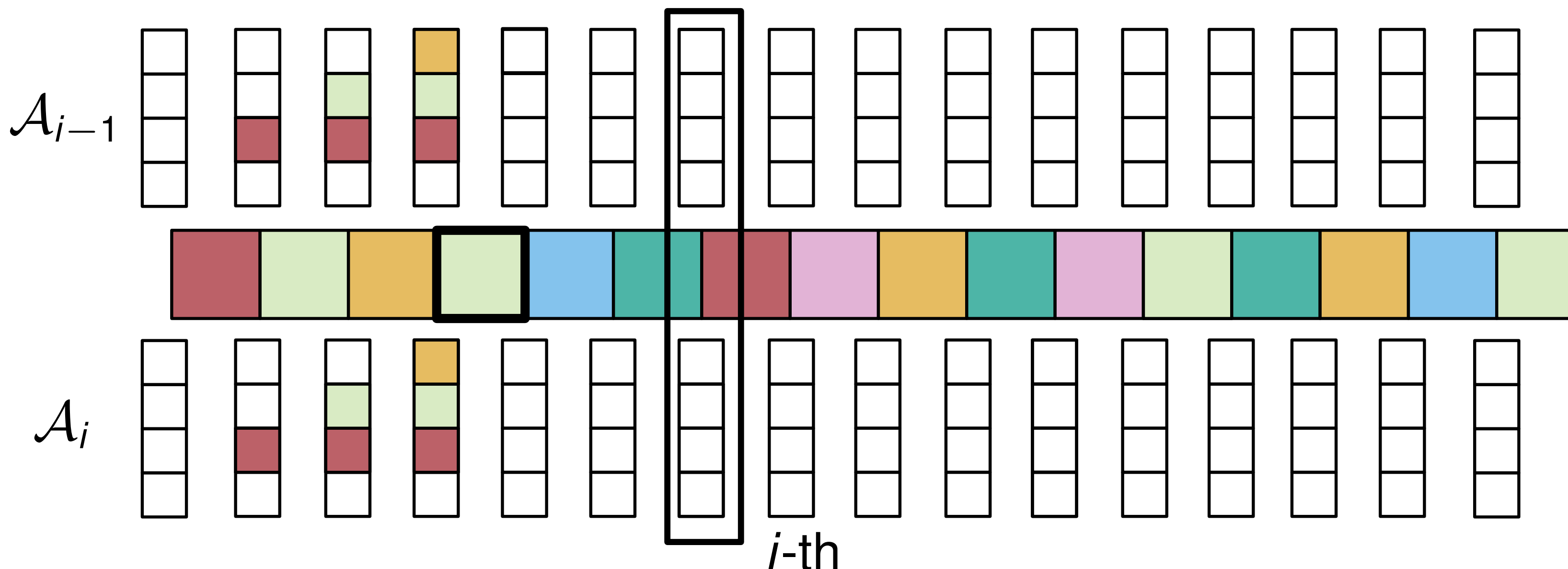
1. $\mathcal{A}_i$ bearbeitet die ersten $i-1$ Anfragen wie $\mathcal{A}_{i-1}$ und damit wie LFD.



Longest Forward Distance (LFD)

Konstruktion von $\mathcal{A}_i$ aufbauend auf $\mathcal{A}_{i-1}$

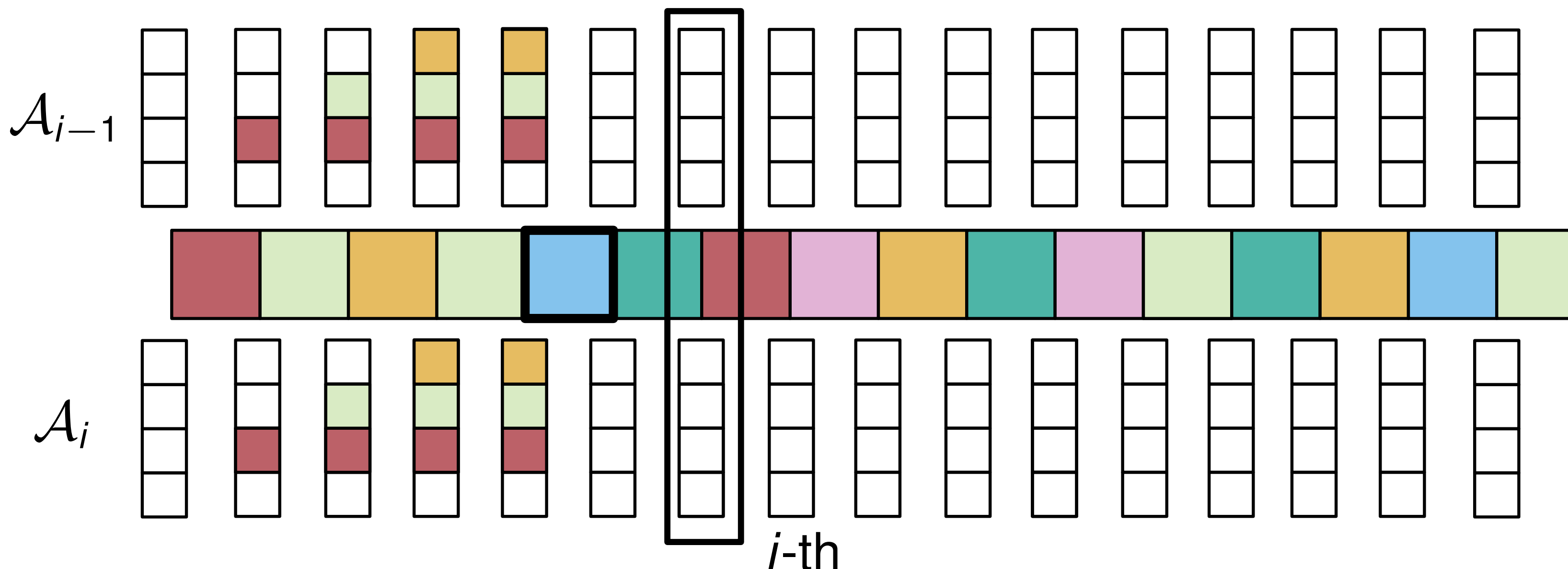
1. $\mathcal{A}_i$ bearbeitet die ersten $i-1$ Anfragen wie $\mathcal{A}_{i-1}$ und damit wie LFD.



Longest Forward Distance (LFD)

Konstruktion von $\mathcal{A}_i$ aufbauend auf $\mathcal{A}_{i-1}$

1. $\mathcal{A}_i$ bearbeitet die ersten $i-1$ Anfragen wie $\mathcal{A}_{i-1}$ und damit wie LFD.



Longest Forward Distance (LFD)

Konstruktion von $\mathcal{A}_i$ aufbauend auf $\mathcal{A}_{i-1}$

1. $\mathcal{A}_i$ bearbeitet die ersten $i - 1$ Anfragen wie $\mathcal{A}_{i-1}$ und damit wie LFD.

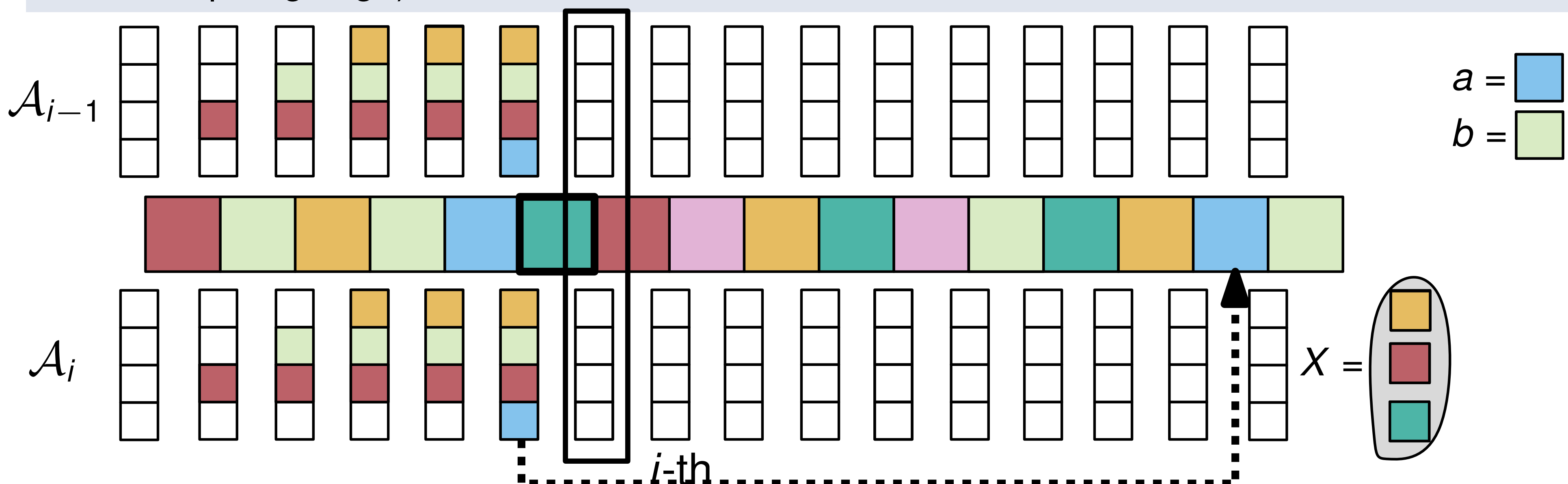
2. Wenn bei der i-ten Anfrage ein Fehlzugriff auftritt, dann wendet $\mathcal{A}_i$ LFD-Prinzip an.

→ Nach dem i-ten Schritt können die Caches verschieden sein.

- → $\mathcal{A}_i$ verdrängt Seite a nach LFD-Prinzip.
- → $\mathcal{A}_{i-1}$ verdrängt Seite b.

Nach der i-ten Anfrage sei X die Menge der gemeinsamen Seiten.

(Wenn kein Fehlzugriff auftritt, dann können restliche Anfragen gleich behandelt werden und die Behauptung folgt.)



Longest Forward Distance (LFD)

Konstruktion von $\mathcal{A}_i$ aufbauend auf $\mathcal{A}_{i-1}$

1. $\mathcal{A}_i$ bearbeitet die ersten $i-1$ Anfragen wie $\mathcal{A}_{i-1}$ und damit wie LFD.

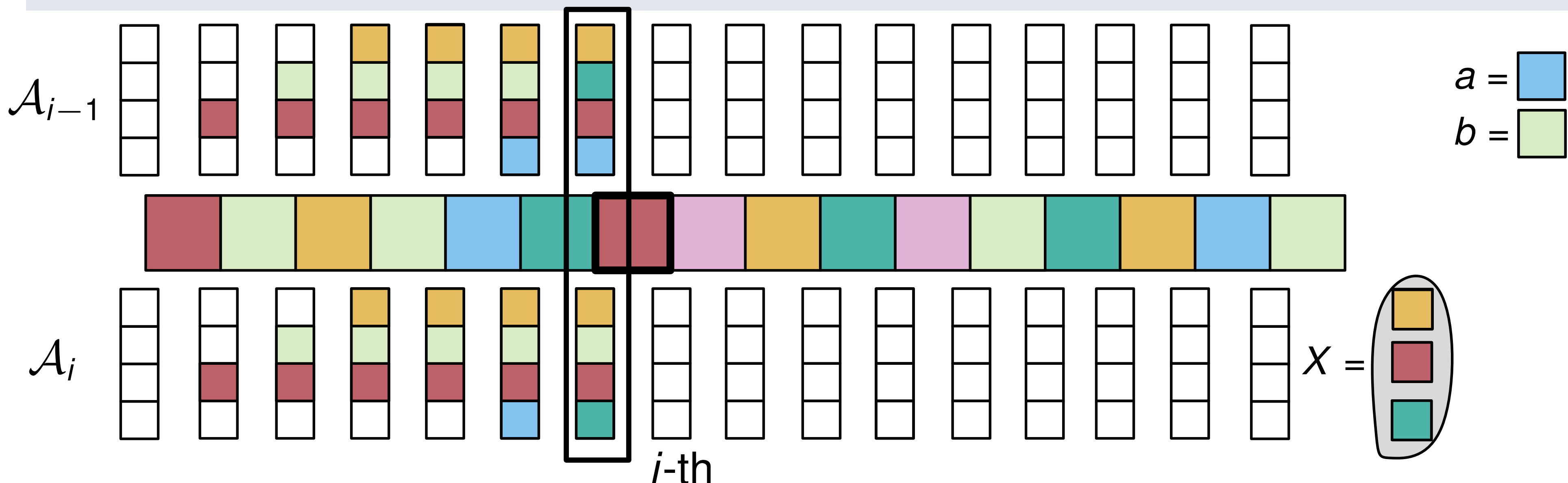
2. Wenn bei der i-ten Anfrage ein Fehlzugriff auftritt, dann wendet $\mathcal{A}_i$ LFD-Prinzip an.

→ Nach dem i-ten Schritt können die Caches verschieden sein.

- → $\mathcal{A}_i$ verdrängt Seite a nach LFD-Prinzip.
- → $\mathcal{A}_{i-1}$ verdrängt Seite b.

Nach der i-ten Anfrage sei X die Menge der gemeinsamen Seiten.

(Wenn kein Fehlzugriff auftritt, dann können restliche Anfragen gleich behandelt werden und die Behauptung folgt.)

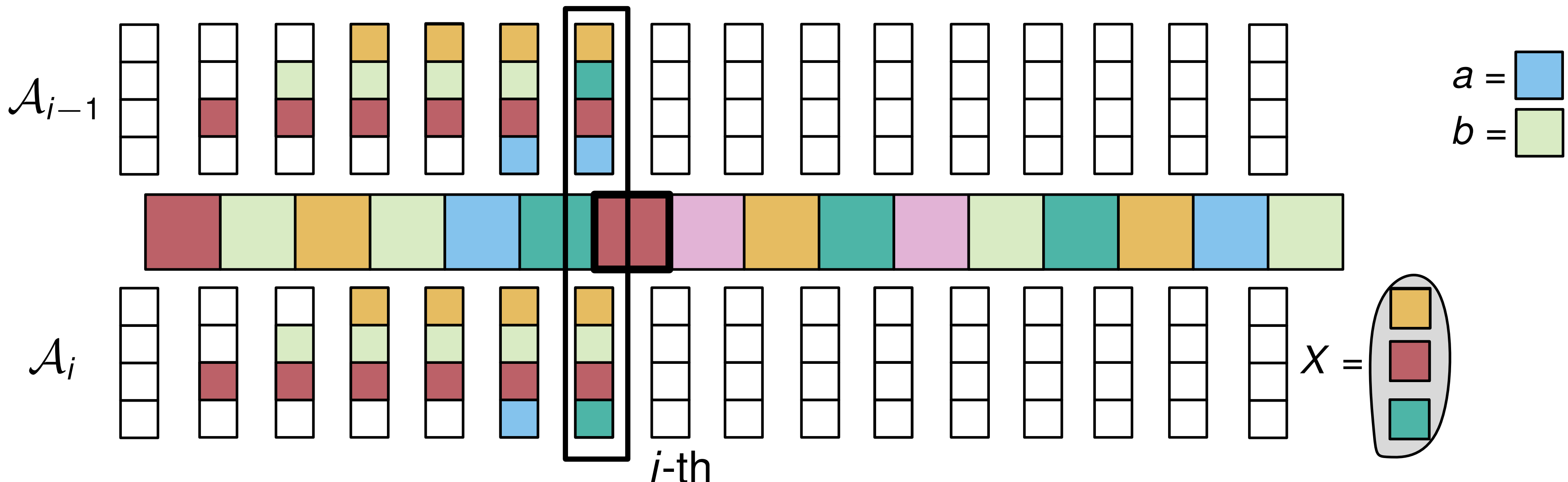


Longest Forward Distance (LFD)

Konstruktion von $\mathcal{A}_i$ aufbauend auf $\mathcal{A}_{i-1}$

1. $\mathcal{A}_i$ bearbeitet die ersten $i-1$ Anfragen wie $\mathcal{A}_{i-1}$ und damit wie LFD.
2. Wenn bei der i-ten Anfrage ein Fehlzugriff auftritt, dann wendet $\mathcal{A}_i$ LFD-Prinzip an.

Beobachtung: Bisher hatte $\mathcal{A}_i$ nicht mehr Fehlzugriffe als $\mathcal{A}_{i-1}$.



Longest Forward Distance (LFD)

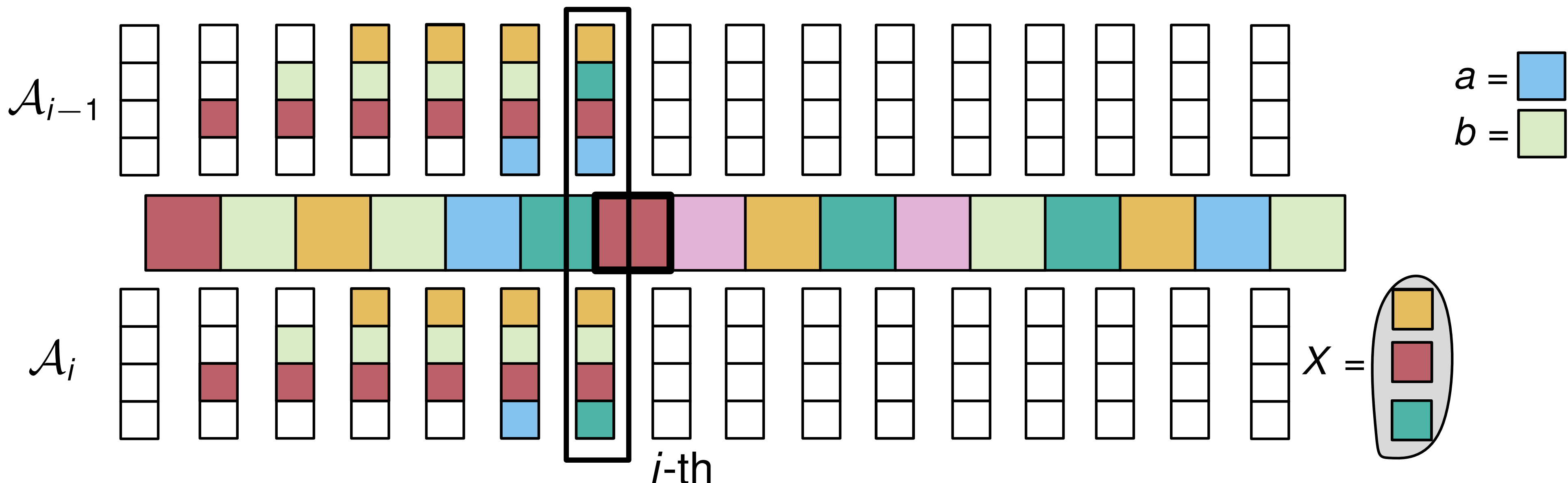
Karlsruher Institut für Technologie

Konstruktion von $\mathcal{A}_i$ aufbauend auf $\mathcal{A}_{i-1}$

3. Verhalten von $\mathcal{A}_i$ nach der i-ten Anfragen.

Idee: Definiere Verhalten von $\mathcal{A}_i$ so, dass

1. sich die Caches von $\mathcal{A}_{i-1}$ und $\mathcal{A}_i$ aneinander angleichen, und
2. $\mathcal{A}_i$ insgesamt nicht mehr Fehlzugriffe als $\mathcal{A}_{i-1}$ haben kann .



Longest Forward Distance (LFD)

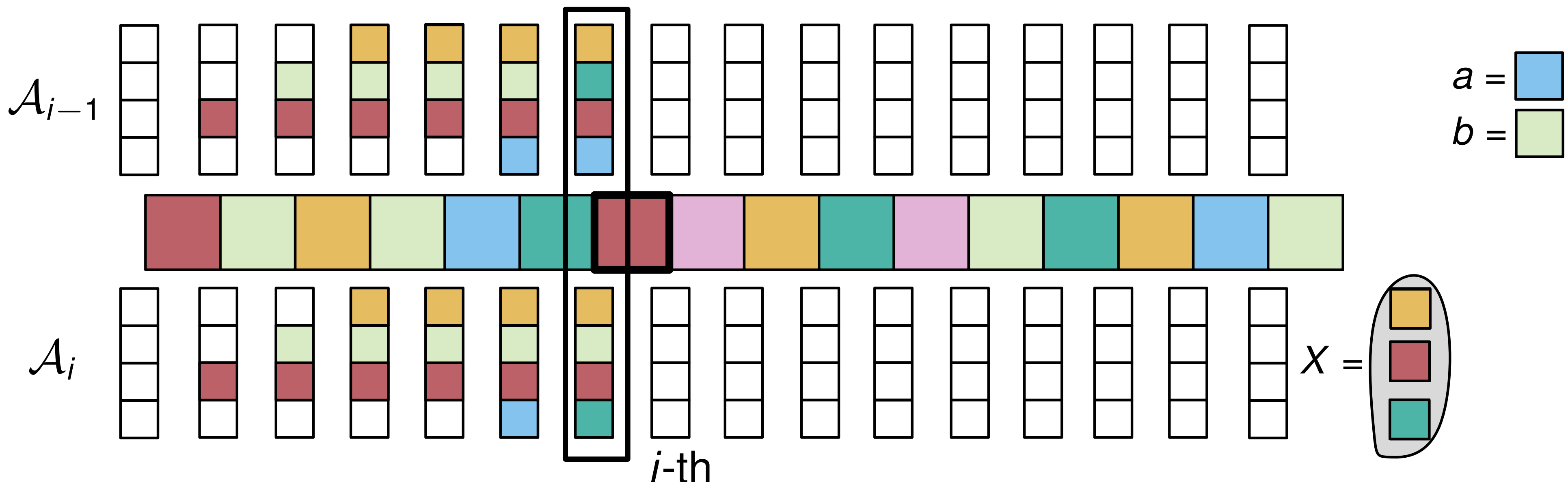
Konstruktion von $\mathcal{A}_i$ aufbauend auf $\mathcal{A}_{i-1}$

3. Verhalten von $\mathcal{A}_i$ nach der i-ten Anfragen.

Fall 1: $a = b$ Caches sind bereits gleich.

→ $\mathcal{A}_i$ behandelt restliche Anfragen wie $\mathcal{A}_{i-1}$.

→ $\mathcal{A}_i$ hat nicht mehr Fehlzugriffe als $\mathcal{A}_{i-1}$.



Longest Forward Distance (LFD)

Konstruktion von $\mathcal{A}_i$ aufbauend auf $\mathcal{A}_{i-1}$

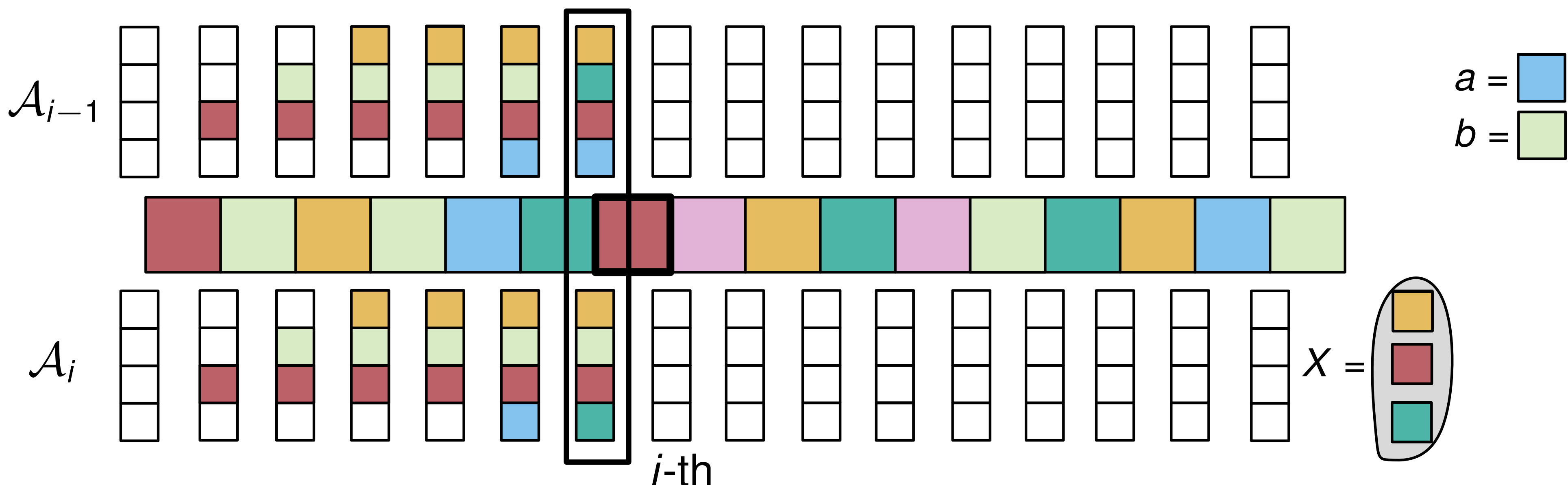
3. Verhalten von $\mathcal{A}_i$ nach der i-ten Anfragen.

Fall 2: $a \neq b$

Beobachtung: Da sich die Seiten aus X sowohl im Cache von $\mathcal{A}_{i-1}$ als auch im Cache von $\mathcal{A}_i$ befinden, kann $\mathcal{A}_i$ den Algorithmus $\mathcal{A}_{i-1}$ mit gleich vielen Fehlzugriffen imitieren bis

1. $\mathcal{A}_{i-1}$ die Seite a verdrängt, oder
2. a angefragt wird, oder
3. b angefragt wird.

→ Lasse $\mathcal{A}_i$ die Anfragen so wie $\mathcal{A}_{i-1}$ behandeln, bis einer der drei Fälle auftritt.



Institut für Theoretische Informatik
Prof. Dr. Dorothea Wagner

Longest Forward Distance (LFD)

Konstruktion von $\mathcal{A}_i$ aufbauend auf $\mathcal{A}_{i-1}$

3. Verhalten von $\mathcal{A}_i$ nach der i-ten Anfragen.

Fall 2: $a \neq b$

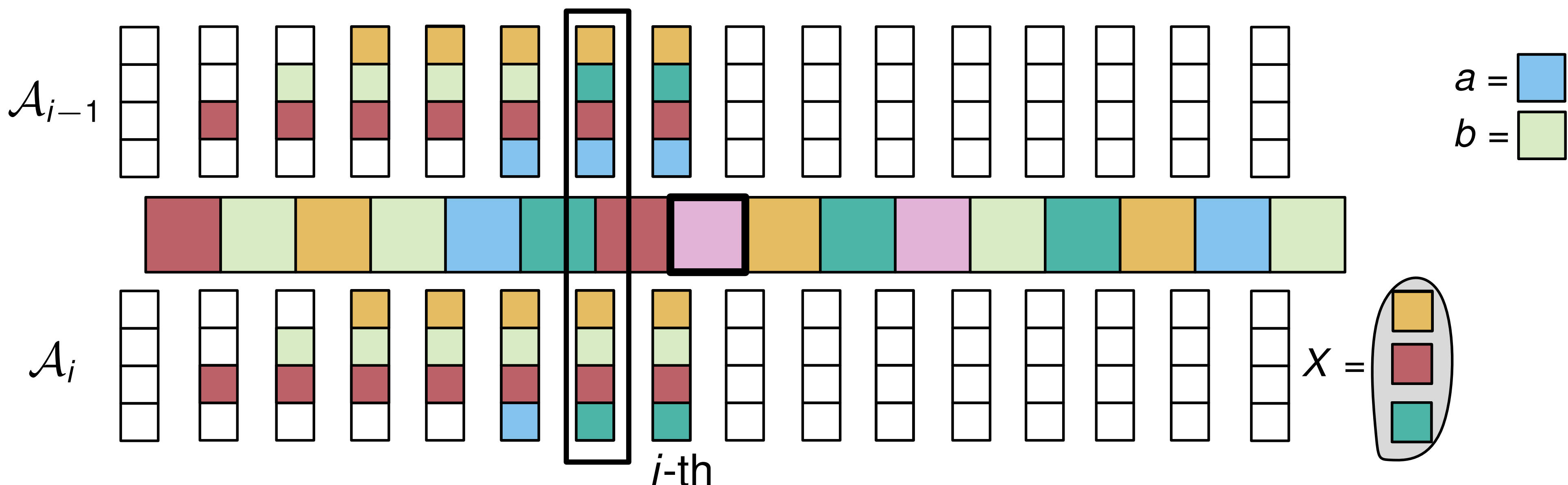
+1 Fehlzugriff für $\mathcal{A}_{i-1}$ und $\mathcal{A}_i$

Fall 2.1: $\mathcal{A}_{i-1}$ verdrängt Seite a.

⟶ Lasse $\mathcal{A}_i$ die Seite b aus dem Cache verdrängen.

Im Beispiel: $\mathcal{A}_{i-1}$ verdrängt ▢ für ▢

$\mathcal{A}_i$ verdrängt ▢ für ▢



Longest Forward Distance (LFD)

Konstruktion von $\mathcal{A}_i$ aufbauend auf $\mathcal{A}_{i-1}$

3. Verhalten von $\mathcal{A}_i$ nach der i-ten Anfragen.

Fall 2: $a \neq b$

+1 Fehlzugriff für $\mathcal{A}_{i-1}$ und $\mathcal{A}_i$

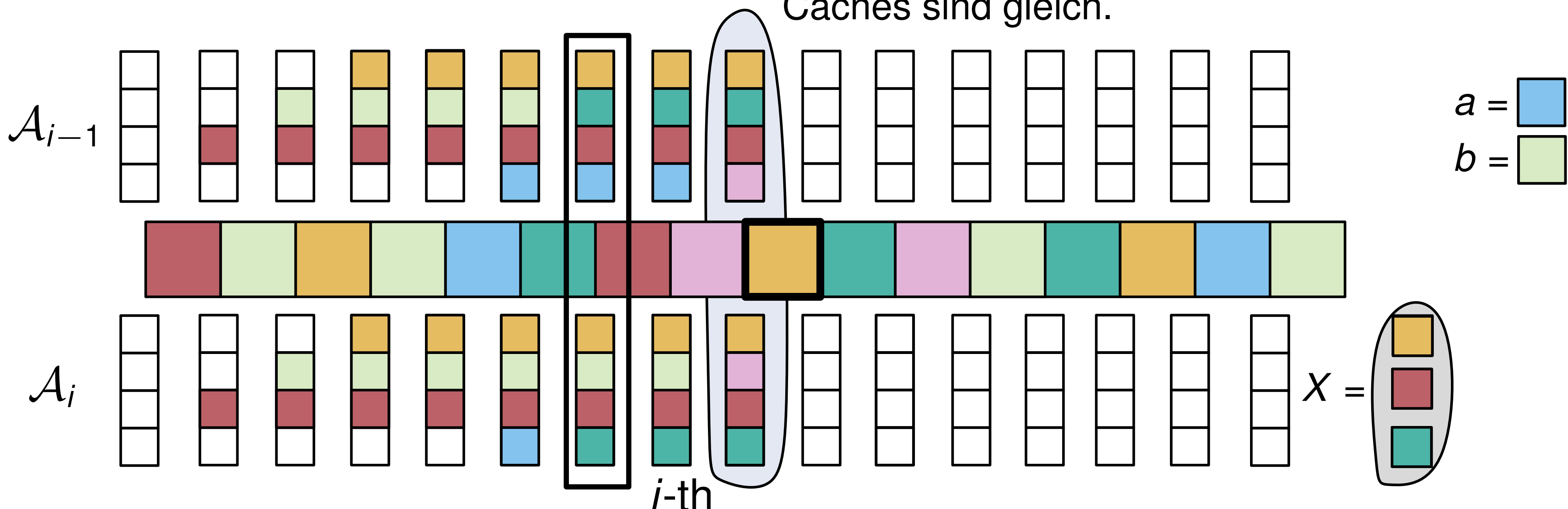
Fall 2.1: $\mathcal{A}_{i-1}$ verdrängt Seite a.

⟶ Lasse $\mathcal{A}_i$ die Seite b aus dem Cache verdrängen.

⟶ Caches von $\mathcal{A}_{i-1}$ und $\mathcal{A}_i$ sind wieder gleich.

Im Beispiel: $\mathcal{A}_{i-1}$ verdrängt ■ für ■

$\mathcal{A}_i$ verdrängt ■ für ■



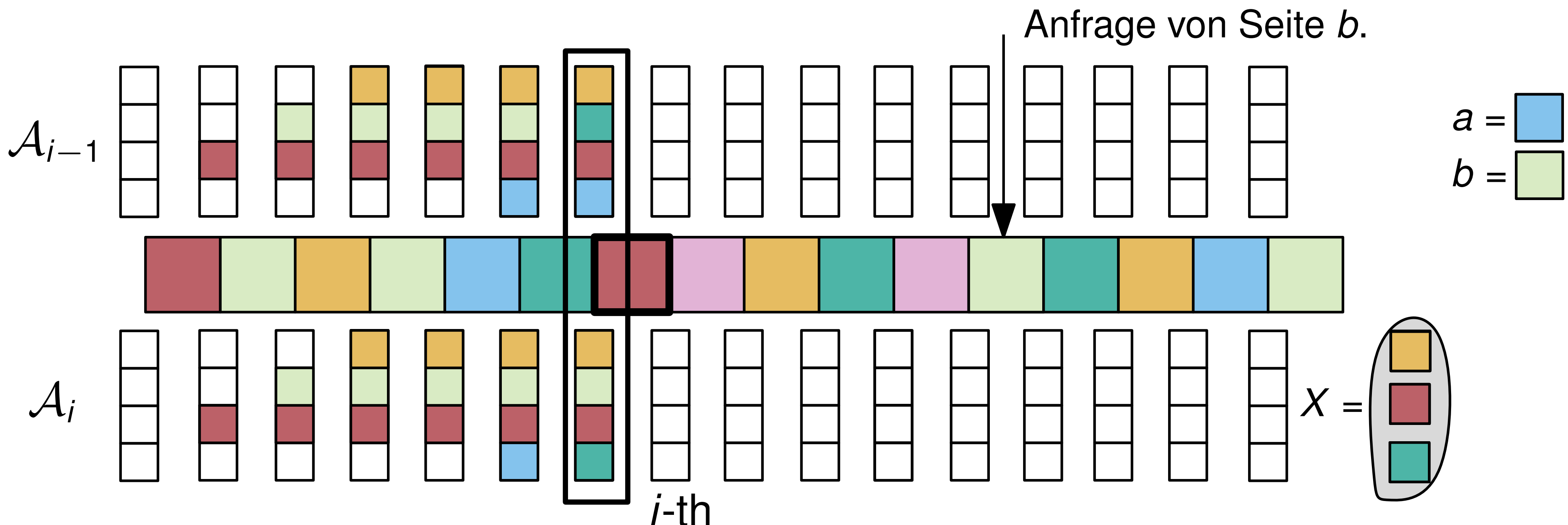
Longest Forward Distance (LFD)

Konstruktion von $\mathcal{A}_i$ aufbauend auf $\mathcal{A}_{i-1}$

3. Verhalten von $\mathcal{A}_i$ nach der i-ten Anfragen.

Fall 2: $a \neq b$

Fall 2.2: Seite b wird angefragt.



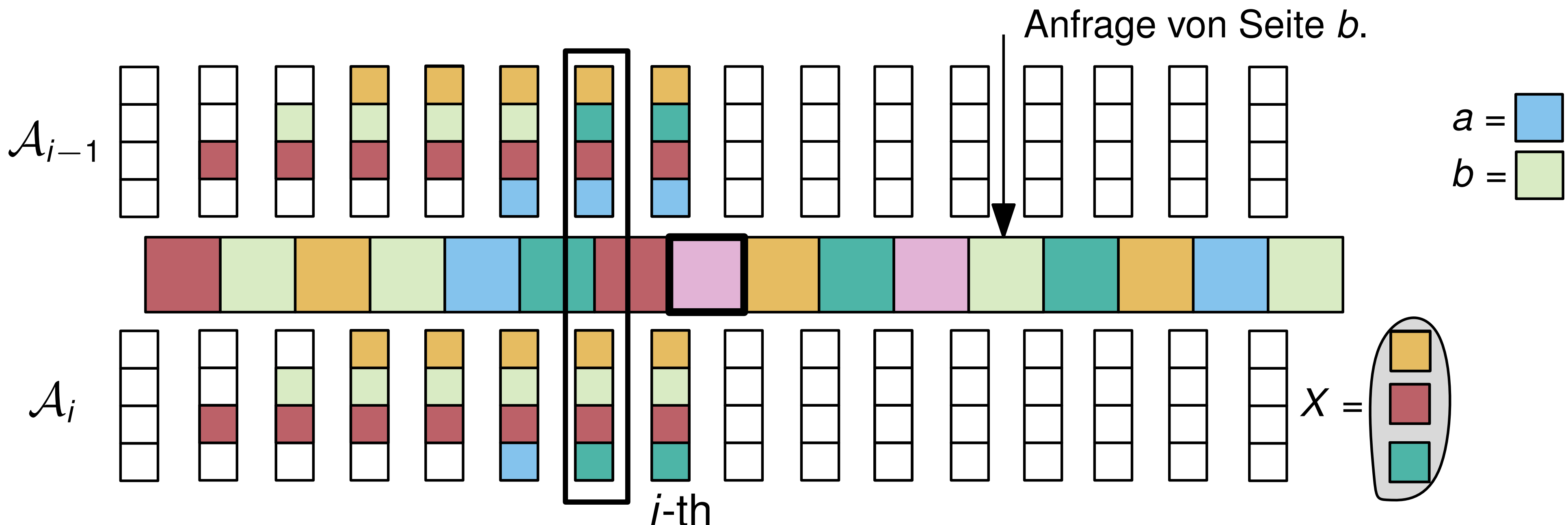
Longest Forward Distance (LFD)

Konstruktion von $\mathcal{A}_i$ aufbauend auf $\mathcal{A}_{i-1}$

3. Verhalten von $\mathcal{A}_i$ nach der i-ten Anfragen.

Fall 2: $a \neq b$

Fall 2.2: Seite b wird angefragt.



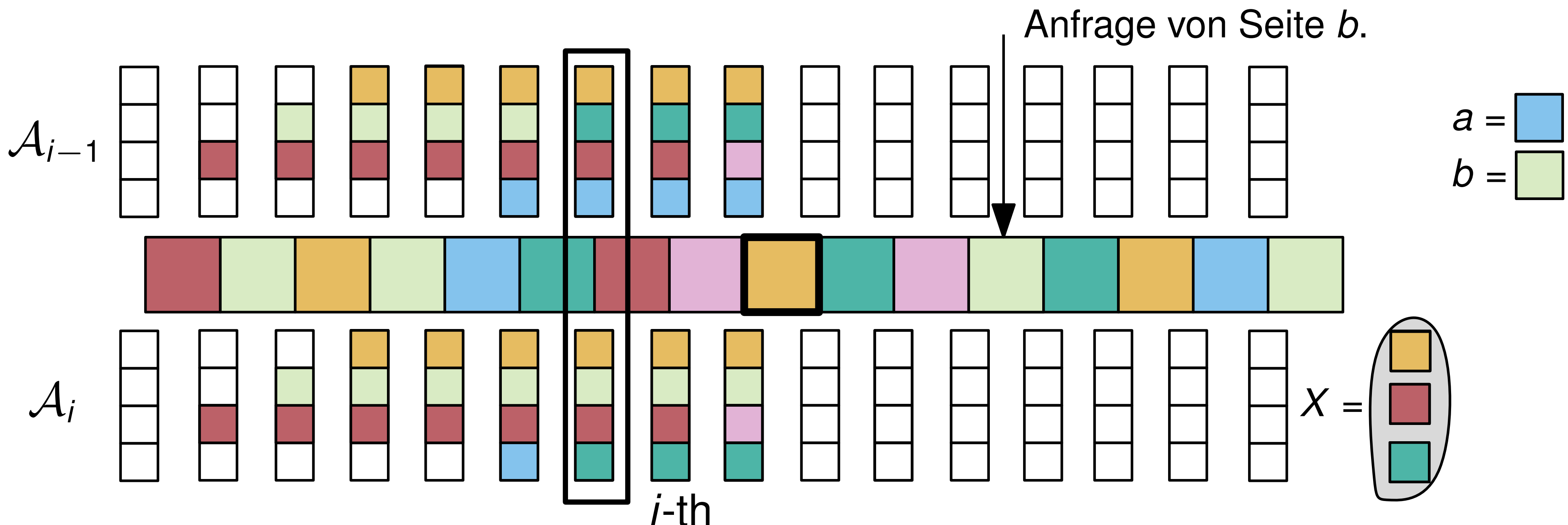
Longest Forward Distance (LFD)

Konstruktion von $\mathcal{A}_i$ aufbauend auf $\mathcal{A}_{i-1}$

3. Verhalten von $\mathcal{A}_i$ nach der i-ten Anfragen.

Fall 2: $a \neq b$

Fall 2.2: Seite b wird angefragt.



Longest Forward Distance (LFD)

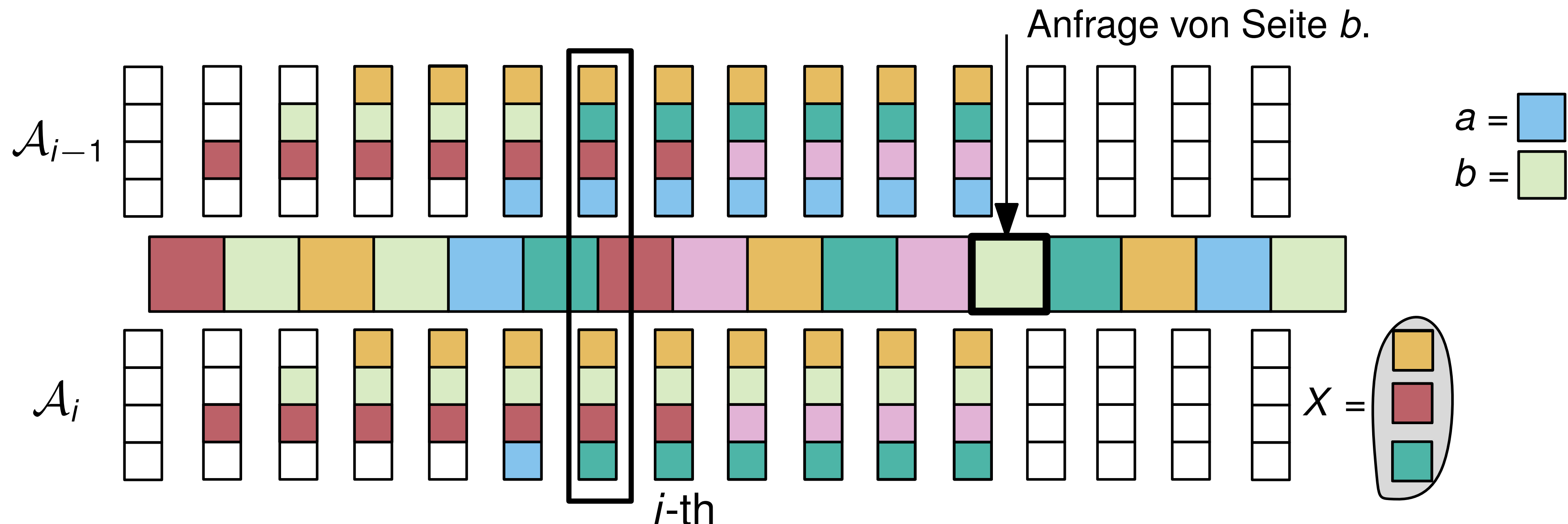
Karlsruher Institut für Technologie

Konstruktion von $\mathcal{A}_i$ aufbauend auf $\mathcal{A}_{i-1}$

3. Verhalten von $\mathcal{A}_i$ nach der i-ten Anfragen.

Fall 2: $a \neq b$

Fall 2.2: Seite b wird angefragt.



Longest Forward Distance (LFD)

Karlsruher Institut für Technologie

Konstruktion von $\mathcal{A}_i$ aufbauend auf $\mathcal{A}_{i-1}$

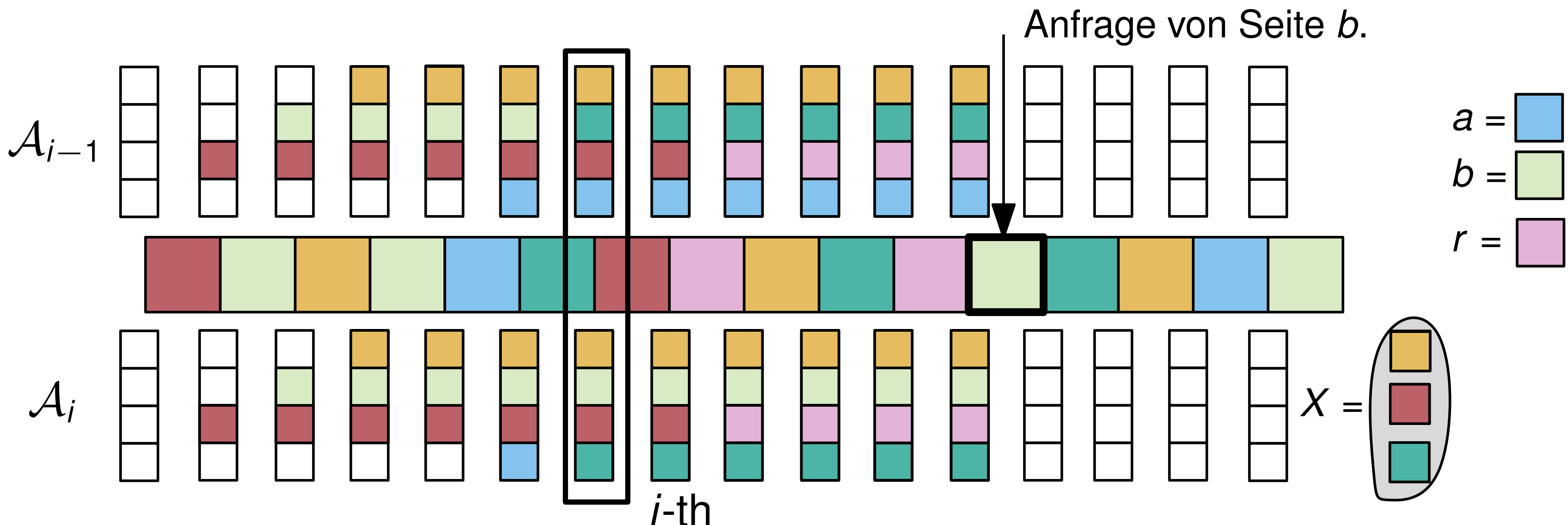
3. Verhalten von $\mathcal{A}_i$ nach der i-ten Anfragen.

Fall 2: $a \neq b$

+1 Fehlzugriff für $\mathcal{A}_{i-1}$

Fall 2.2: Seite b wird angefragt.

⟶ $\mathcal{A}_{i-1}$ muss Seite r aus Cache verdrängen, um Platz für b zu schaffen.



Longest Forward Distance (LFD)

Konstruktion von $\mathcal{A}_i$ aufbauend auf $\mathcal{A}_{i-1}$

3. Verhalten von $\mathcal{A}_i$ nach der i-ten Anfragen.

Fall 2: $a \neq b$

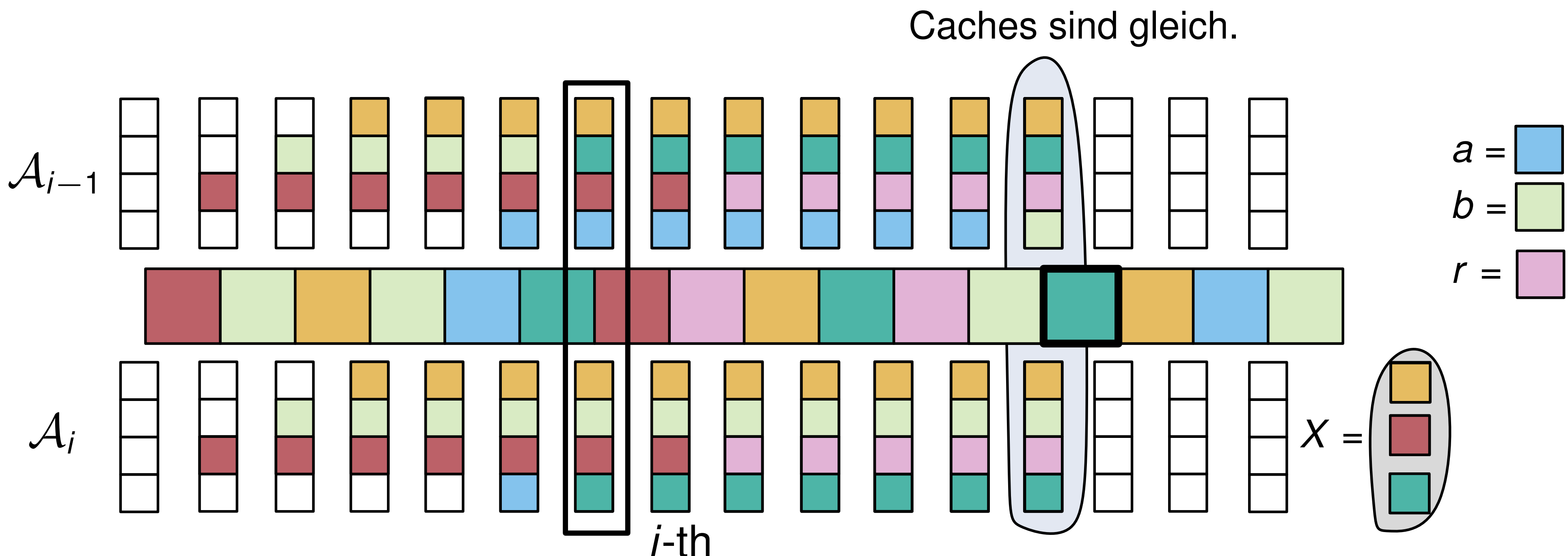
+1 Fehlzugriff für $\mathcal{A}_{i-1}$

Fall 2.2: Seite b wird angefragt.

⟶ $\mathcal{A}_{i-1}$ muss Seite r aus Cache verdrängen, um Platz für b zu schaffen.

Fall 2.2.1: $r = a$

⟶ Caches von $\mathcal{A}_{i-1}$ und $\mathcal{A}_i$ sind wieder gleich.



Longest Forward Distance (LFD)

Konstruktion von $\mathcal{A}_i$ aufbauend auf $\mathcal{A}_{i-1}$

3. Verhalten von $\mathcal{A}_i$ nach der *i*-ten Anfragen.

Fall 2: $a \neq b$

+1 Fehlzugriff für $\mathcal{A}_{i-1}$

Fall 2.2: Seite *b* wird angefragt.

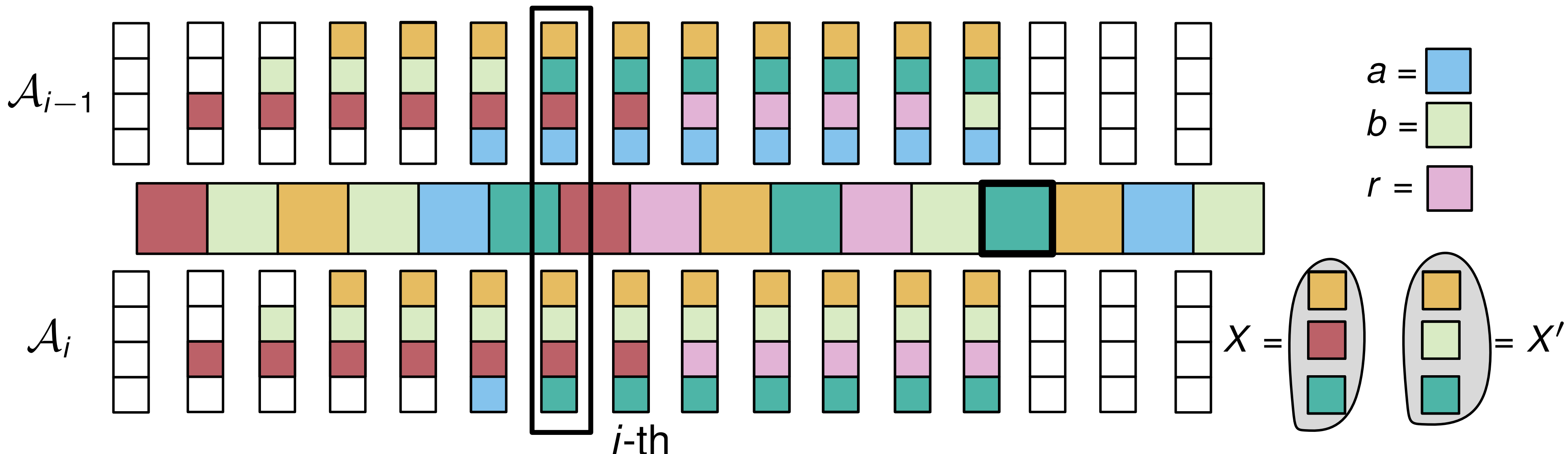
→ $\mathcal{A}_{i-1}$ muss Seite *r* aus Cache verdrängen, um Platz für *b* zu schaffen.

Fall 2.2.2: $r \neq a$

→ $\mathcal{A}_{i-1}$ hat Seiten $X' \cup \{a\}$ im Cache.

$\mathcal{A}_i$ hat Seiten $X' \cup \{r\}$ im Cache.

X' = Menge der Seiten, die in beiden Caches vorkommen.



Institut für Theoretische Informatik
Prof. Dr. Dorothea Wagner

Longest Forward Distance (LFD)

Konstruktion von $\mathcal{A}_i$ aufbauend auf $\mathcal{A}_{i-1}$

3. Verhalten von $\mathcal{A}_i$ nach der i-ten Anfragen.

Fall 2: $a \neq b$

+1 Fehlzugriff für $\mathcal{A}_{i-1}$

Fall 2.2: Seite b wird angefragt.

⟶ $\mathcal{A}_{i-1}$ muss Seite r aus Cache verdrängen, um Platz für b zu schaffen.

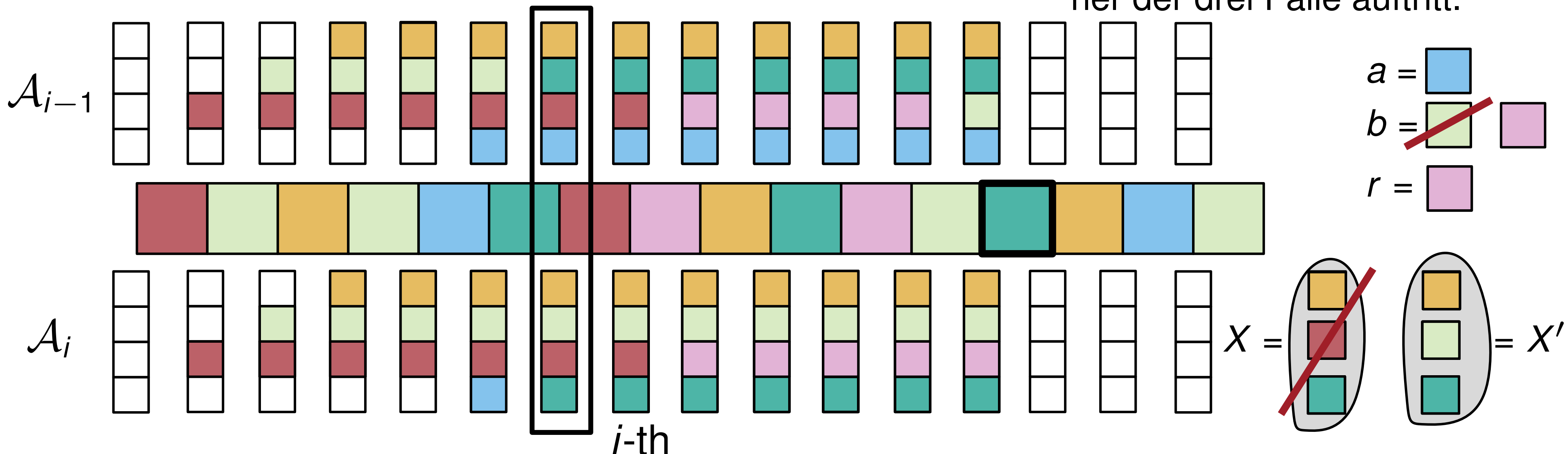
Fall 2.2.2: $r \neq a$

⟶ $\mathcal{A}_{i-1}$ hat Seiten $X' \cup \{a\}$ im Cache. ⟶ r übernimmt Rolle von b. X' übernimmt Rolle von X.

$\mathcal{A}_i$ hat Seiten $X' \cup \{r\}$ im Cache.

⟶ $\mathcal{A}_i$ imitiert $\mathcal{A}_{i-1}$ bis wieder einer der drei Fälle auftritt.

X' = Menge der Seiten, die in beiden Caches vorkommen.



Longest Forward Distance (LFD)

Karlsruher Institut für Technologie

Konstruktion von $\mathcal{A}_i$ aufbauend auf $\mathcal{A}_{i-1}$

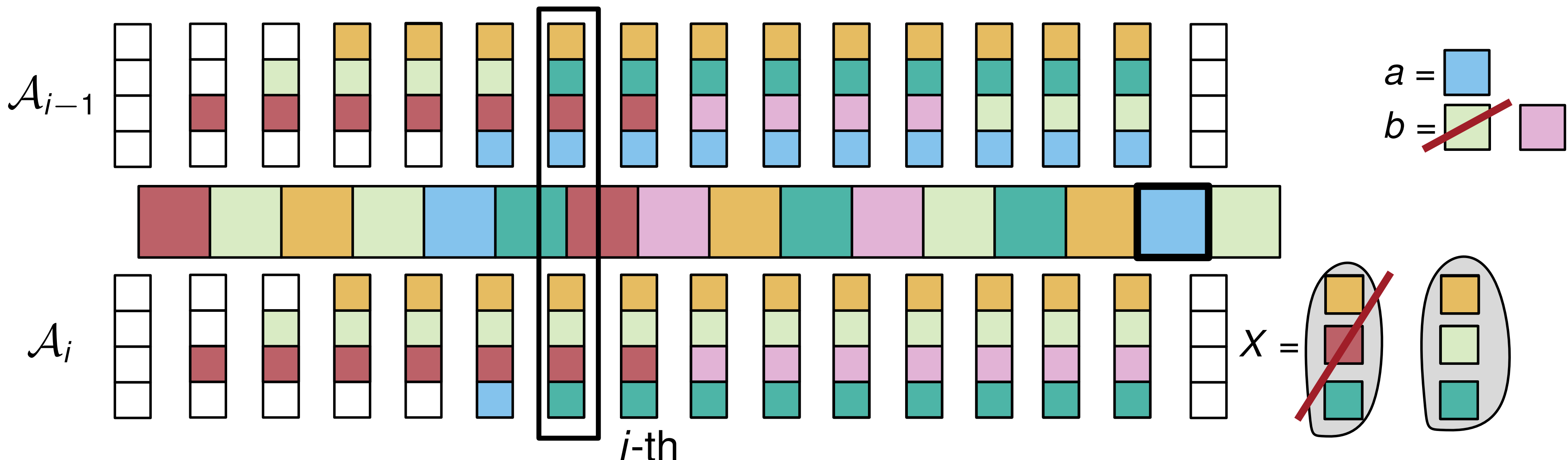
3. Verhalten von $\mathcal{A}_i$ nach der i-ten Anfragen.

Fall 2: $a \neq b$

+1 Fehlzugriff für $\mathcal{A}_i$

Fall 2.3: Seite a wird angefragt.

⟶ $\mathcal{A}_i$ verdrängt Seite b, um für a Platz zu schaffen.



Longest Forward Distance (LFD)

Konstruktion von $\mathcal{A}_i$ aufbauend auf $\mathcal{A}_{i-1}$

3. Verhalten von $\mathcal{A}_i$ nach der i-ten Anfragen.

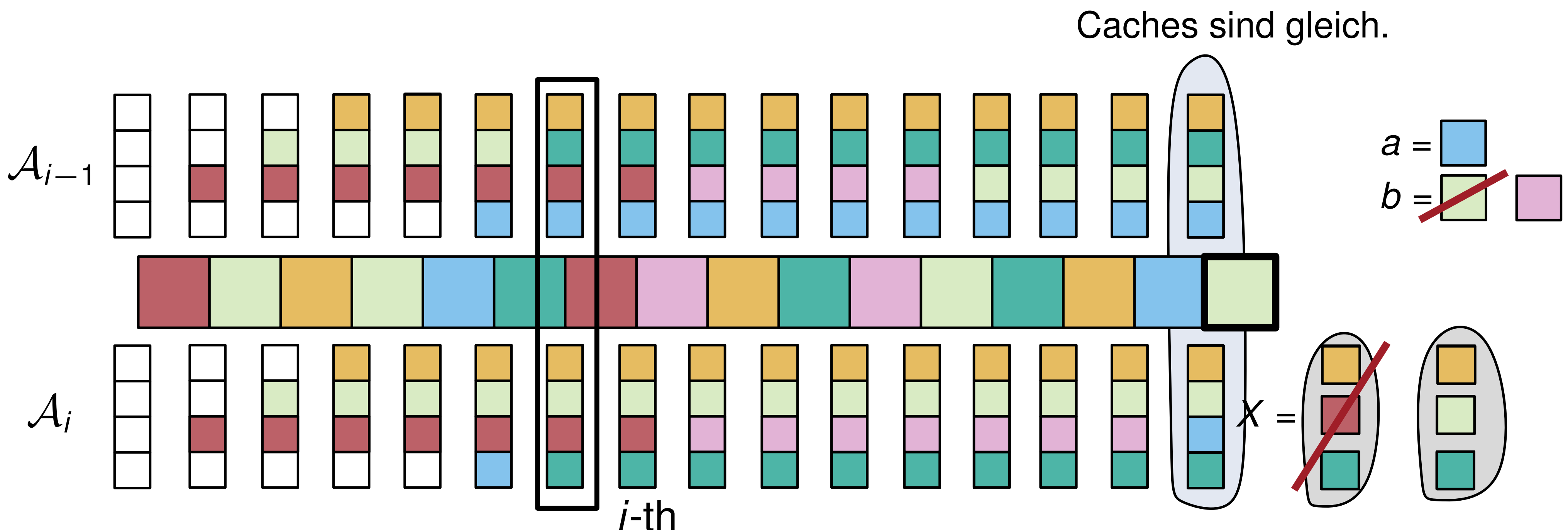
Fall 2: $a \neq b$

+1 Fehlzugriff für $\mathcal{A}_i$

Fall 2.3: Seite a wird angefragt.

→ $\mathcal{A}_i$ verdrängt Seite b, um für a Platz zu schaffen.

→ Caches von $\mathcal{A}_{i-1}$ und $\mathcal{A}_i$ sind wieder gleich.



Longest Forward Distance (LFD)

Karlsruher Institut für Technologie

Konstruktion von $\mathcal{A}_i$ aufbauend auf $\mathcal{A}_{i-1}$

3. Verhalten von $\mathcal{A}_i$ nach der i-ten Anfragen.

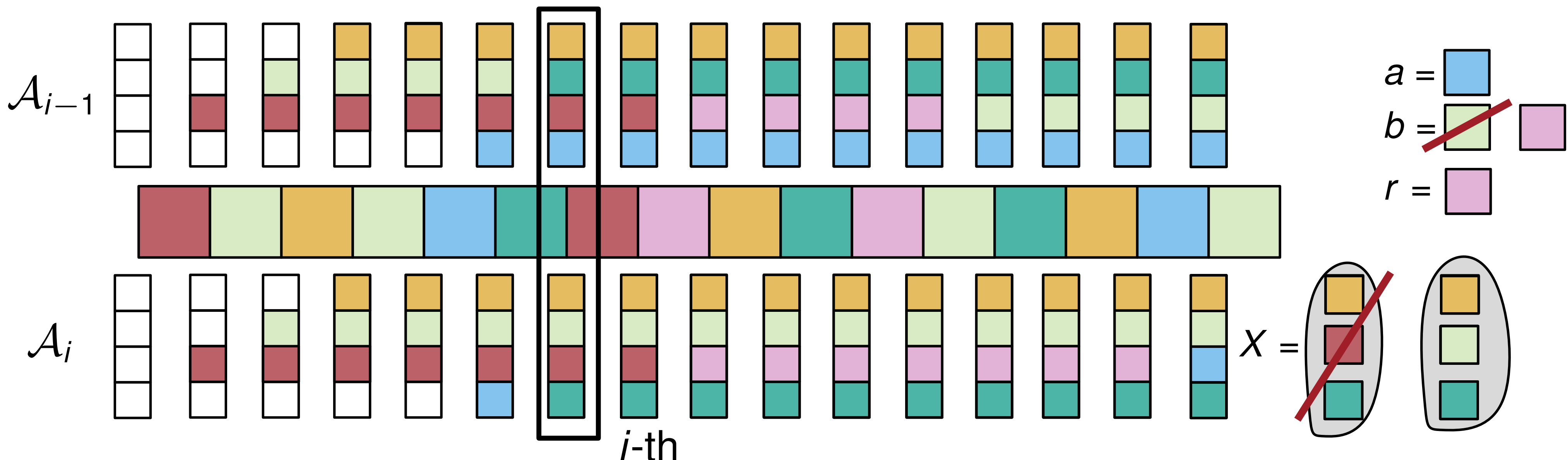
Fall 2: $a \neq b$

Beobachtung: Da sich die Seiten aus X sowohl im Cache von $\mathcal{A}_{i-1}$ als auch im Cache von $\mathcal{A}_i$ befinden, kann $\mathcal{A}_i$ den Algorithmus $\mathcal{A}_{i-1}$ mit gleich vielen Fehlzugriffen imitieren bis

1. $\mathcal{A}_{i-1}$ die Seite a verdrängt, oder → Danach sind Caches gleich.
2. a angefragt wird, oder → Danach sind Caches gleich. +1 Fehlz. für $\mathcal{A}_i$
3. b angefragt wird. → +1 Fehlzugriff für $\mathcal{A}_{i-1}$

Da a nach LFD-Prinzip gewählt, wird b mind. einmal vor a angefragt.

→ Fehlzugriff von $\mathcal{A}_i$ kann immer mit Fehlzugriff von $\mathcal{A}_{i-1}$ verrechnet werden.



Longest Forward Distance (LFD)

Karlsruher Institut für Technologie

Theorem 50: LFD ist ein optimaler Offline-Algorithmus für Paging.

Beweis:

Vergleiche LFD-Algorithmus mit einem beliebigen Paging-Algorithmus $\mathcal{A}$.

Sei hierzu σ eine beliebige Sequenz mit $|\sigma|$ Anfragen.

Behauptung: Für jedes $i = 0, \ldots, |\sigma|$ gibt es einen Paging Algorithmus $\mathcal{A}_i$, sodass

1. $\mathcal{A}_i$ die ersten i Anfragen wie der LFD-Algorithmus bearbeitet, und
2. $\mathcal{A}_i$ nicht mehr Fehlzugriffe als $\mathcal{A}$ auf σ erzeugt.

⟶ Mit $i = |\sigma|$ folgt damit das Theorem.

Kompetitive Paging-Algorithmen

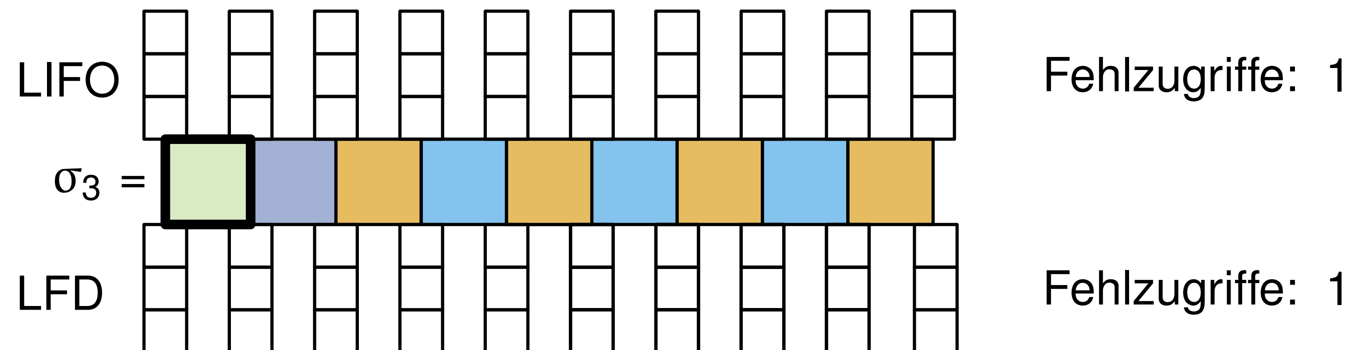
Karlsruher Institut für Technologie

LIFO (Last In/First out) ist nicht kompetitiv, d.h. die relative Güte c kann beliebig groß werden:

Sei k Cache-Größe und $P = \{p_1, \ldots, p_k, p_{k+1}\}$ Seiten im Hauptspeicher. Betrachte für beliebiges $m \in \mathbb{N}$ die Sequenz

$$\sigma_m = p_1, \ldots, p_k, (p_{k+1}, p_k)^m$$

- LIFO wird zuerst die Seiten $p_1, \ldots, p_k$ in den Cache aufnehmen (k Fehlzugriffe).
- Für die $k + 1$-te Anfrage wird LIFO die Seite p_k durch die Seite p_{k+1} ersetzen.
- Für die $k + 2$-te Anfrage wird LIFO die Seite p_{k+1} durch die Seite p_k ersetzen und usw.
- LIFO hat deshalb k+2m Fehlzugriffe.
- LFD hat nur k+1 Fehlzugriffe.
- Folglich: Für jedes c gibt es ein m, sodass $\mathcal{A}(\sigma_m) > c \cdot \mathrm{OPT}(\sigma_m)$



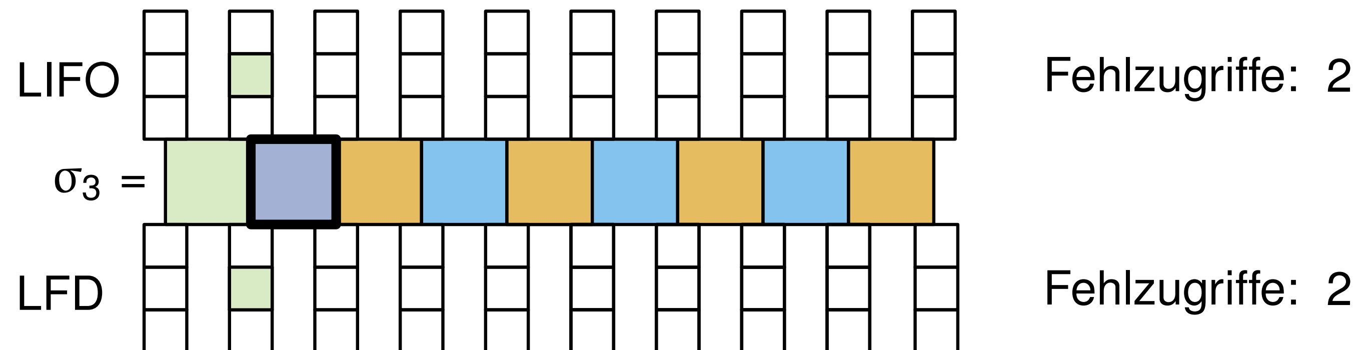
Kompetitive Paging-Algorithmen

LIFO (Last In/First out) ist nicht kompetitiv, d.h. die relative Güte c kann beliebig groß werden:

Sei k Cache-Größe und $P = \{p_1, \ldots, p_k, p_{k+1}\}$ Seiten im Hauptspeicher. Betrachte für beliebiges $m \in \mathbb{N}$ die Sequenz

$$\sigma_m = p_1, \ldots, p_k, (p_{k+1}, p_k)^m$$

- LIFO wird zuerst die Seiten $p_1, \ldots, p_k$ in den Cache aufnehmen (k Fehlzugriffe).
- Für die $k+1$-te Anfrage wird LIFO die Seite p_k durch die Seite p_{k+1} ersetzen.
- Für die $k+2$-te Anfrage wird LIFO die Seite p_{k+1} durch die Seite p_k ersetzen und usw.
- LIFO hat deshalb k+2m Fehlzugriffe.
- LFD hat nur k+1 Fehlzugriffe.
- Folglich: Für jedes c gibt es ein m, sodass $\mathcal{A}(\sigma_m) > c \cdot \text{OPT}(\sigma_m)$



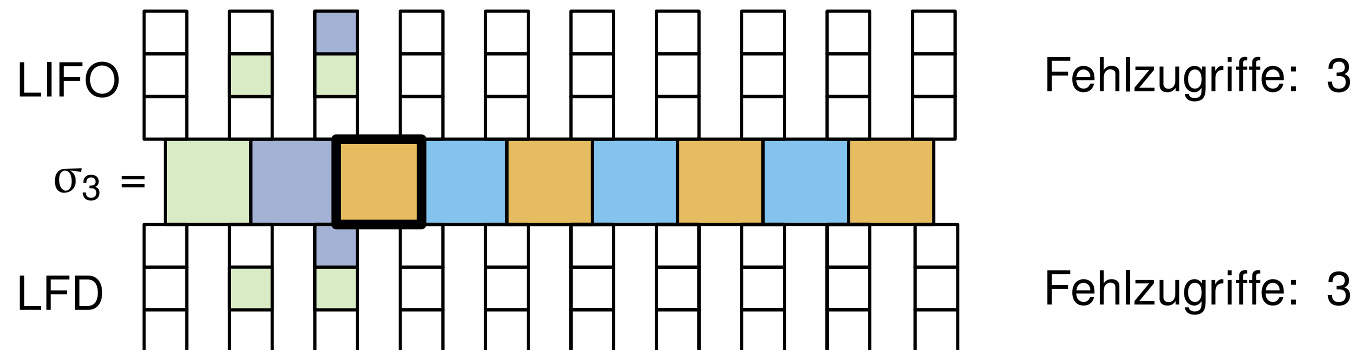
Kompetitive Paging-Algorithmen

LIFO (Last In/First out) ist nicht kompetitiv, d.h. die relative Güte c kann beliebig groß werden:

Sei k Cache-Größe und $P = \{p_1, \ldots, p_k, p_{k+1}\}$ Seiten im Hauptspeicher. Betrachte für beliebiges $m \in \mathbb{N}$ die Sequenz

$$\sigma_m = p_1, \ldots, p_k, (p_{k+1}, p_k)^m$$

- LIFO wird zuerst die Seiten $p_1, \ldots, p_k$ in den Cache aufnehmen (k Fehlzugriffe).
- Für die $k+1$-te Anfrage wird LIFO die Seite p_k durch die Seite p_{k+1} ersetzen.
- Für die $k+2$-te Anfrage wird LIFO die Seite p_{k+1} durch die Seite p_k ersetzen und usw.
- LIFO hat deshalb k+2m Fehlzugriffe.
- LFD hat nur k+1 Fehlzugriffe.
- Folglich: Für jedes c gibt es ein m, sodass $\mathcal{A}(\sigma_m) > c \cdot \mathrm{OPT}(\sigma_m)$



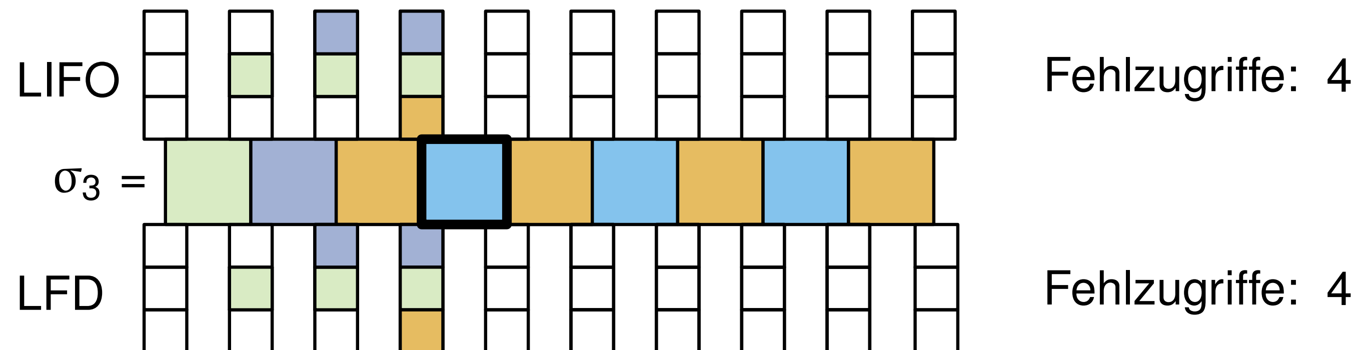
Kompetitive Paging-Algorithmen

LIFO (Last In/First out) ist nicht kompetitiv, d.h. die relative Güte c kann beliebig groß werden:

Sei k Cache-Größe und $P = \{p_1, \ldots, p_k, p_{k+1}\}$ Seiten im Hauptspeicher. Betrachte für beliebiges $m \in \mathbb{N}$ die Sequenz

$$\sigma_m = p_1, \ldots, p_k, (p_{k+1}, p_k)^m$$

- LIFO wird zuerst die Seiten $p_1, \ldots, p_k$ in den Cache aufnehmen (k Fehlzugriffe).
- Für die $k+1$-te Anfrage wird LIFO die Seite p_k durch die Seite p_{k+1} ersetzen.
- Für die $k+2$-te Anfrage wird LIFO die Seite p_{k+1} durch die Seite p_k ersetzen und usw.
- LIFO hat deshalb k+2m Fehlzugriffe.
- LFD hat nur k+1 Fehlzugriffe.
- Folglich: Für jedes c gibt es ein m, sodass $\mathcal{A}(\sigma_m) > c \cdot \text{OPT}(\sigma_m)$



Institut für Theoretische Informatik
Prof. Dr. Dorothea Wagner

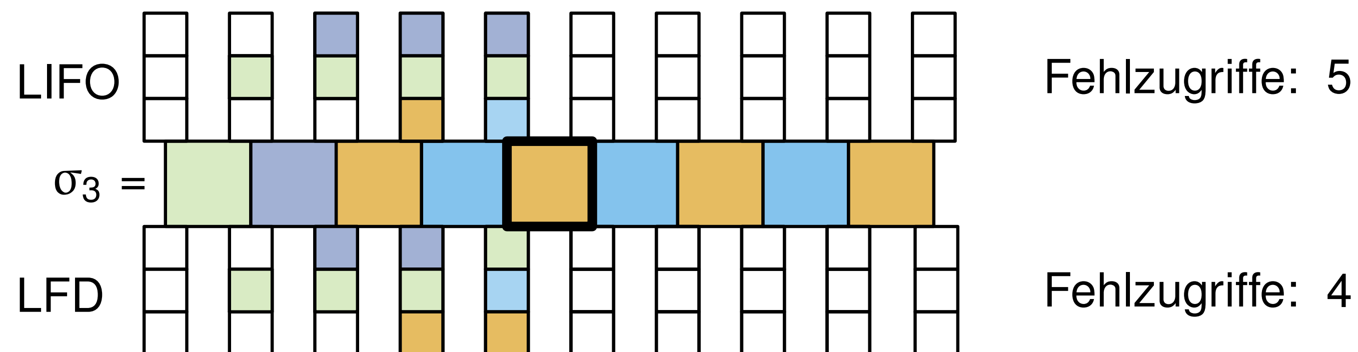
Kompetitive Paging-Algorithmen

LIFO (Last In/First out) ist nicht kompetitiv, d.h. die relative Güte c kann beliebig groß werden:

Sei k Cache-Größe und $P = \{p_1, \dots, p_k, p_{k+1}\}$ Seiten im Hauptspeicher. Betrachte für beliebiges $m \in \mathbb{N}$ die Sequenz

$$\sigma_m = p_1, \dots, p_k, (p_{k+1}, p_k)^m$$

- LIFO wird zuerst die Seiten $p_1, \dots, p_k$ in den Cache aufnehmen (k Fehlzugriffe).
- Für die $k+1$-te Anfrage wird LIFO die Seite p_k durch die Seite p_{k+1} ersetzen.
- Für die $k+2$-te Anfrage wird LIFO die Seite p_{k+1} durch die Seite p_k ersetzen und usw.
- LIFO hat deshalb k+2m Fehlzugriffe.
- LFD hat nur k+1 Fehlzugriffe.
- Folglich: Für jedes c gibt es ein m, sodass $\mathcal{A}(\sigma_m) > c \cdot \mathrm{OPT}(\sigma_m)$



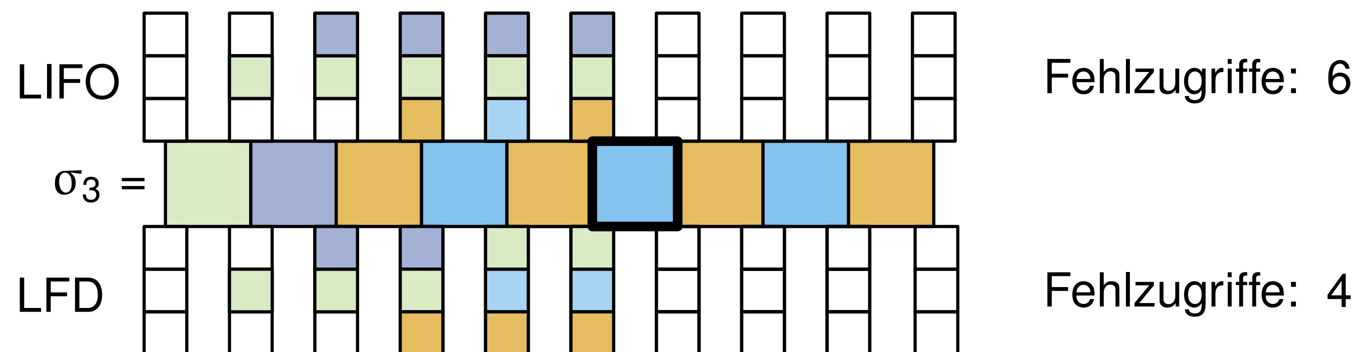
Kompetitive Paging-Algorithmen

LIFO (Last In/First out) ist nicht kompetitiv, d.h. die relative Güte c kann beliebig groß werden:

Sei k Cache-Größe und $P = \{p_1, \ldots, p_k, p_{k+1}\}$ Seiten im Hauptspeicher. Betrachte für beliebiges $m \in \mathbb{N}$ die Sequenz

$$\sigma_m = p_1, \ldots, p_k, (p_{k+1}, p_k)^m$$

- LIFO wird zuerst die Seiten $p_1, \ldots, p_k$ in den Cache aufnehmen (k Fehlzugriffe).
- Für die $k+1$-te Anfrage wird LIFO die Seite p_k durch die Seite p_{k+1} ersetzen.
- Für die $k+2$-te Anfrage wird LIFO die Seite p_{k+1} durch die Seite p_k ersetzen und usw.
- LIFO hat deshalb k+2m Fehlzugriffe.
- LFD hat nur k+1 Fehlzugriffe.
- Folglich: Für jedes c gibt es ein m, sodass $\mathcal{A}(\sigma_m) > c \cdot \mathrm{OPT}(\sigma_m)$



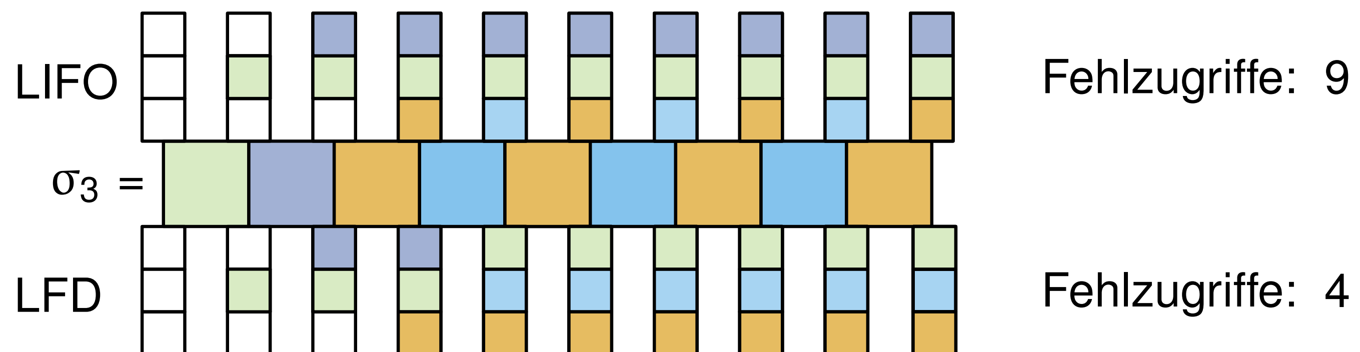
Kompetitive Paging-Algorithmen

LIFO (Last In/First out) ist nicht kompetitiv, d.h. die relative Güte c kann beliebig groß werden:

Sei k Cache-Größe und $P = \{p_1, \ldots, p_k, p_{k+1}\}$ Seiten im Hauptspeicher. Betrachte für beliebiges $m \in \mathbb{N}$ die Sequenz

$$\sigma_m = p_1, \ldots, p_k, (p_{k+1}, p_k)^m$$

- LIFO wird zuerst die Seiten $p_1, \ldots, p_k$ in den Cache aufnehmen (k Fehlzugriffe).
- Für die $k+1$-te Anfrage wird LIFO die Seite p_k durch die Seite p_{k+1} ersetzen.
- Für die $k+2$-te Anfrage wird LIFO die Seite p_{k+1} durch die Seite p_k ersetzen und usw.
- LIFO hat deshalb k+2m Fehlzugriffe.
- LFD hat nur k+1 Fehlzugriffe.
- Folglich: Für jedes c gibt es ein m, sodass $\mathcal{A}(\sigma_m) > c \cdot \mathrm{OPT}(\sigma_m)$



Institut für Theoretische Informatik
Prof. Dr. Dorothea Wagner

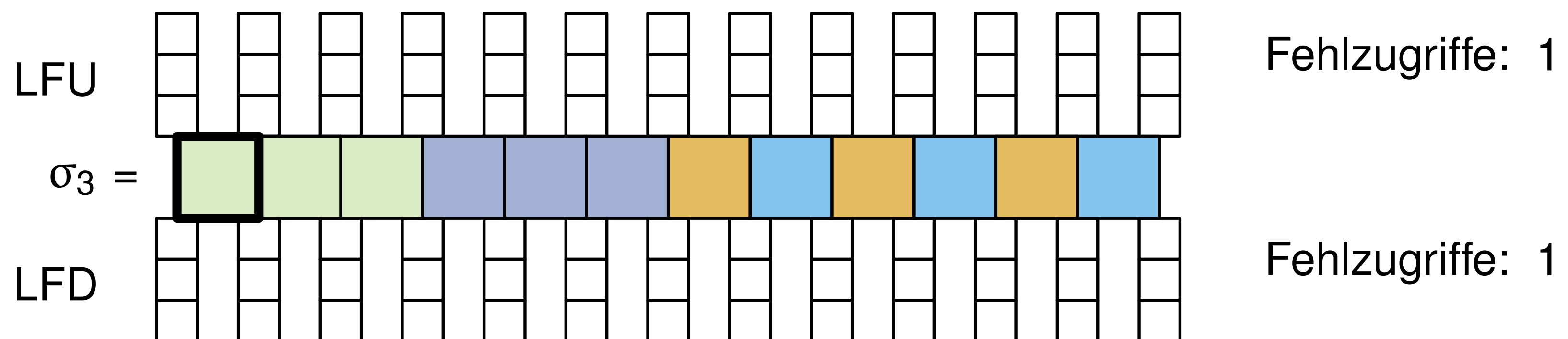
Kompetitive Paging-Algorithmen

LFU (Least Frequently Used) ist nicht kompetitiv, d.h. relative Güte c ist nicht beschränkt:

Sei k Cache-Größe und $P = \{p_1, \ldots, p_k, p_{k+1}\}$ Seiten im Hauptspeicher. Betrachte für beliebiges $m \in \mathbb{N}$ die Sequenz

$$\sigma_m = p_1^m, \ldots, p_{k-1}^m, (p_k, p_{k+1})^m$$

- LFU wird zuerst die Seiten $p_1, \ldots, p_{k-1}$ in den Cache aufnehmen (k-1 Fehlzugriffe).
- Nach $m \cdot (k-1)$ Anfragen, wird LFU für jedes Anfragepaar (p_k, p_{k+1}) einen Fehlzugriff haben. Folglich m weitere Fehlzugriffe.
- Da LFD nur k+1 Fehlzugriffe hat, ist die relative Güte von LFU nicht beschränkt.



Anzahl Anfragen: 1 0 0 0

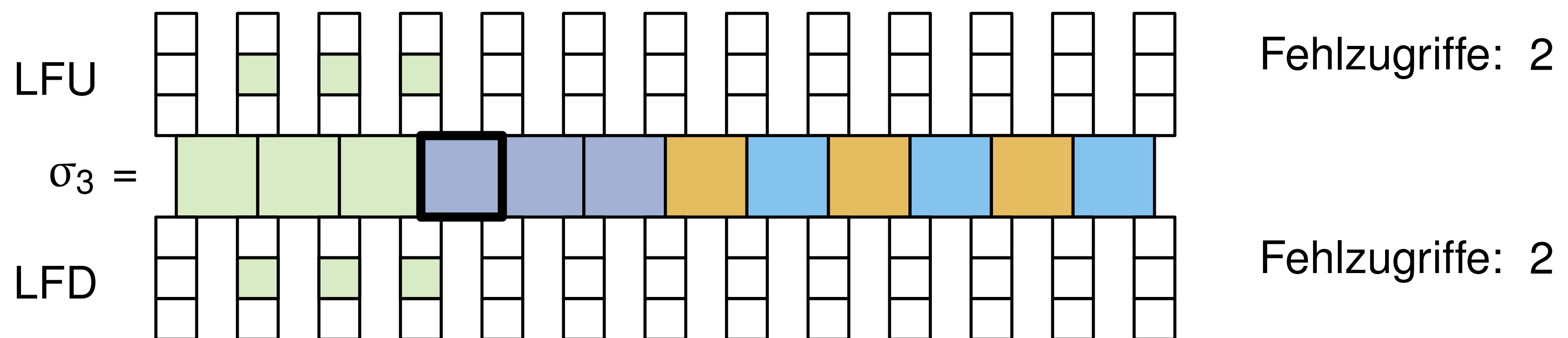
Kompetitive Paging-Algorithmen

LFU (Least Frequently Used) ist nicht kompetitiv, d.h. relative Güte c ist nicht beschränkt:

Sei k Cache-Größe und $P = \{p_1, \ldots, p_k, p_{k+1}\}$ Seiten im Hauptspeicher. Betrachte für beliebiges $m \in \mathbb{N}$ die Sequenz

$$\sigma_m = p_1^m, \ldots, p_{k-1}^m, (p_k, p_{k+1})^m$$

- LFU wird zuerst die Seiten $p_1, \ldots, p_{k-1}$ in den Cache aufnehmen (k-1 Fehlzugriffe).
- Nach $m \cdot (k-1)$ Anfragen, wird LFU für jedes Anfragepaar (p_k, p_{k+1}) einen Fehlzugriff haben. Folglich m weitere Fehlzugriffe.
- Da LFD nur k+1 Fehlzugriffe hat, ist die relative Güte von LFU nicht beschränkt.



Anzahl Anfragen: 3 1 0 0

Institut für Theoretische Informatik
Prof. Dr. Dorothea Wagner

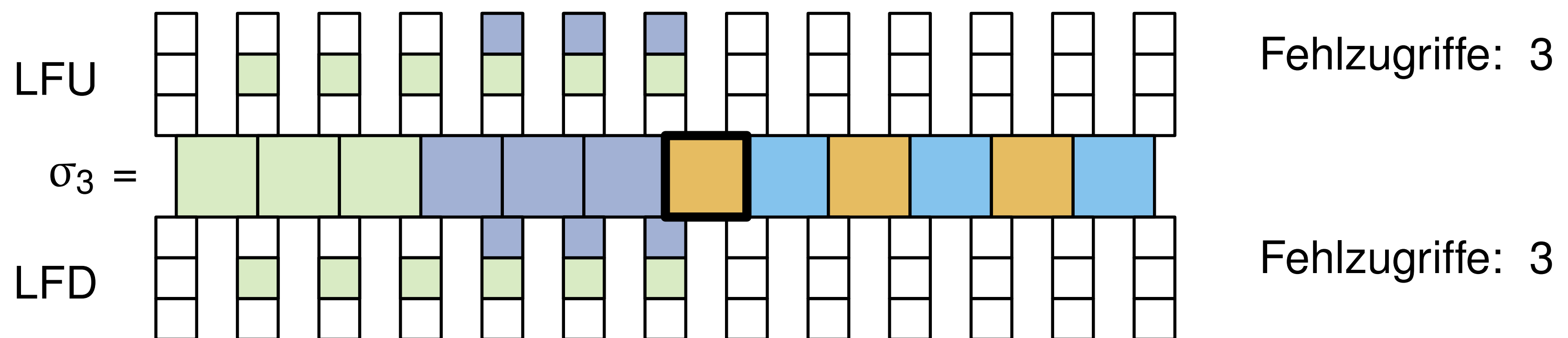
Kompetitive Paging-Algorithmen

LFU (Least Frequently Used) ist nicht kompetitiv, d.h. relative Güte c ist nicht beschränkt:

Sei k Cache-Größe und $P = \{p_1, \ldots, p_k, p_{k+1}\}$ Seiten im Hauptspeicher. Betrachte für beliebiges $m \in \mathbb{N}$ die Sequenz

$$\sigma_m = p_1^m, \ldots, p_{k-1}^m, (p_k, p_{k+1})^m$$

- LFU wird zuerst die Seiten $p_1, \ldots, p_{k-1}$ in den Cache aufnehmen (k-1 Fehlzugriffe).
- Nach $m \cdot (k-1)$ Anfragen, wird LFU für jedes Anfragepaar (p_k, p_{k+1}) einen Fehlzugriff haben. Folglich m weitere Fehlzugriffe.
- Da LFD nur k+1 Fehlzugriffe hat, ist die relative Güte von LFU nicht beschränkt.



Anzahl Anfragen: 3 3 1 0

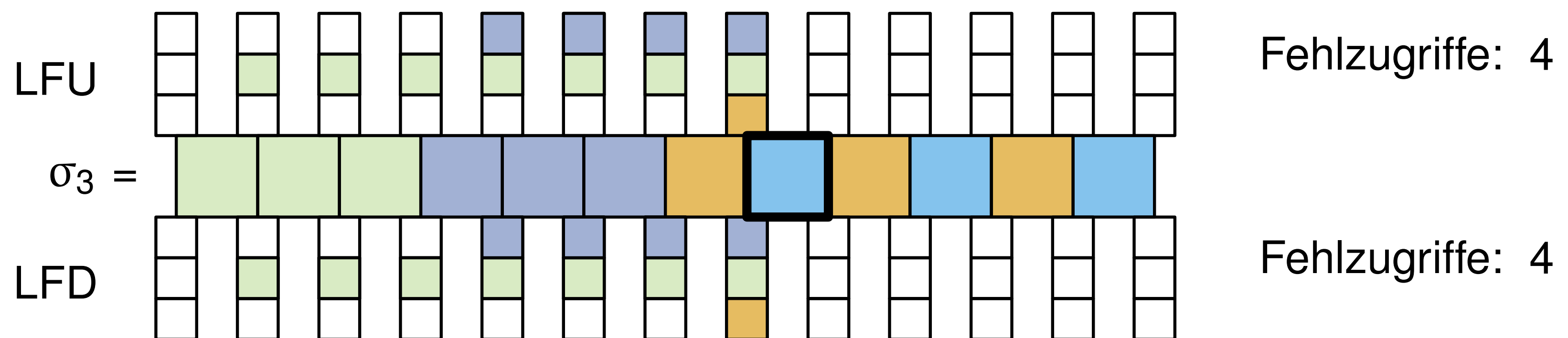
Kompetitive Paging-Algorithmen

LFU (Least Frequently Used) ist nicht kompetitiv, d.h. relative Güte c ist nicht beschränkt:

Sei k Cache-Größe und $P = \{p_1, \ldots, p_k, p_{k+1}\}$ Seiten im Hauptspeicher. Betrachte für beliebiges $m \in \mathbb{N}$ die Sequenz

$$\sigma_m = p_1^m, \ldots, p_{k-1}^m, (p_k, p_{k+1})^m$$

- LFU wird zuerst die Seiten $p_1, \ldots, p_{k-1}$ in den Cache aufnehmen (k-1 Fehlzugriffe).
- Nach $m \cdot (k-1)$ Anfragen, wird LFU für jedes Anfragepaar (p_k, p_{k+1}) einen Fehlzugriff haben. Folglich m weitere Fehlzugriffe.
- Da LFD nur k+1 Fehlzugriffe hat, ist die relative Güte von LFU nicht beschränkt.



Anzahl Anfragen: 3 3 1 1

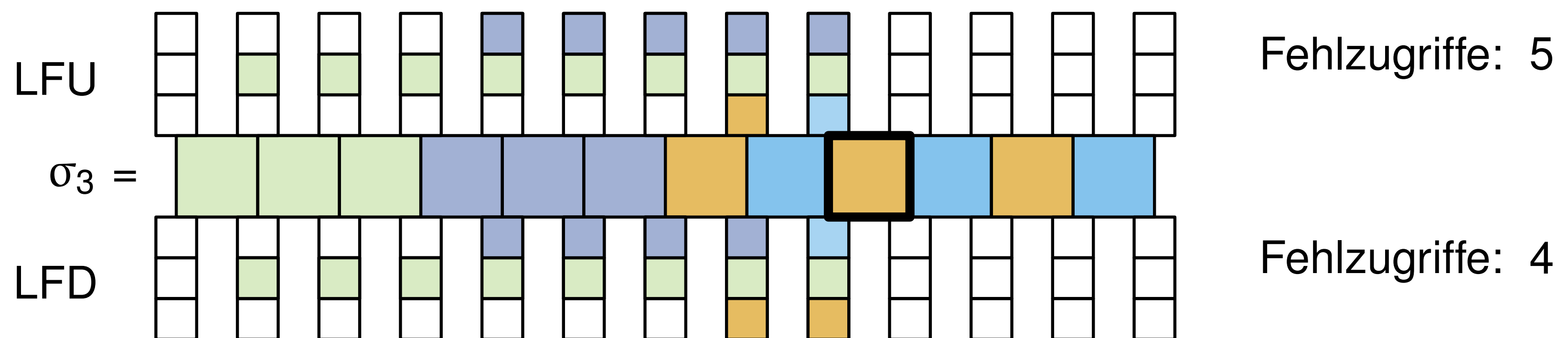
Kompetitive Paging-Algorithmen

LFU (Least Frequently Used) ist nicht kompetitiv, d.h. relative Güte c ist nicht beschränkt:

Sei k Cache-Größe und $P = \{p_1, \dots, p_k, p_{k+1}\}$ Seiten im Hauptspeicher. Betrachte für beliebiges $m \in \mathbb{N}$ die Sequenz

$$\sigma_m = p_1^m, \dots, p_{k-1}^m, (p_k, p_{k+1})^m$$

- LFU wird zuerst die Seiten $p_1, \dots, p_{k-1}$ in den Cache aufnehmen (k-1 Fehlzugriffe).
- Nach $m \cdot (k-1)$ Anfragen, wird LFU für jedes Anfragepaar (p_k, p_{k+1}) einen Fehlzugriff haben. Folglich m weitere Fehlzugriffe.
- Da LFD nur k+1 Fehlzugriffe hat, ist die relative Güte von LFU nicht beschränkt.



Anzahl Anfragen: 3 3 2 1

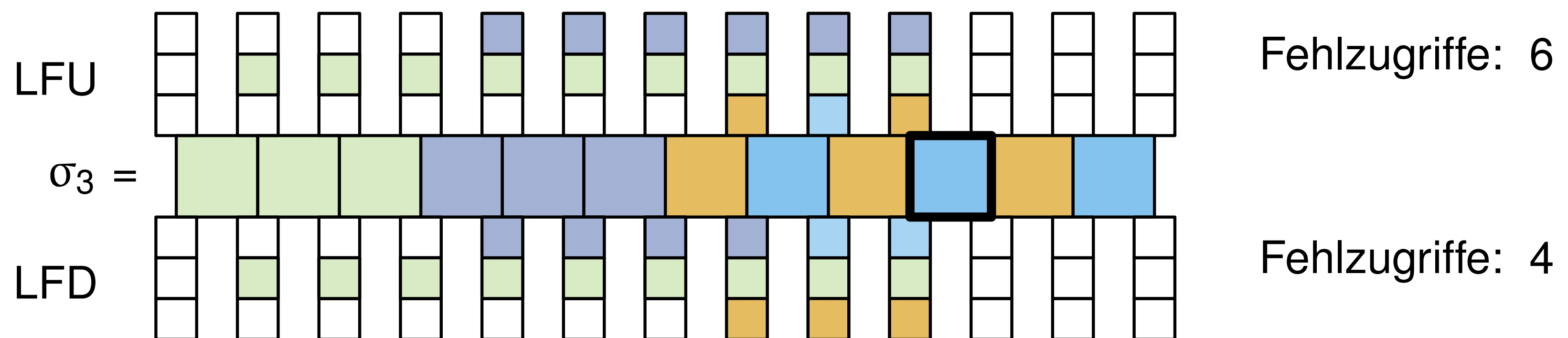
Kompetitive Paging-Algorithmen

LFU (Least Frequently Used) ist nicht kompetitiv, d.h. relative Güte c ist nicht beschränkt:

Sei k Cache-Größe und $P = \{p_1, \ldots, p_k, p_{k+1}\}$ Seiten im Hauptspeicher. Betrachte für beliebiges $m \in \mathbb{N}$ die Sequenz

$$\sigma_m = p_1^m, \ldots, p_{k-1}^m, (p_k, p_{k+1})^m$$

- LFU wird zuerst die Seiten $p_1, \ldots, p_{k-1}$ in den Cache aufnehmen (k-1 Fehlzugriffe).
- Nach $m \cdot (k-1)$ Anfragen, wird LFU für jedes Anfragepaar (p_k, p_{k+1}) einen Fehlzugriff haben. Folglich m weitere Fehlzugriffe.
- Da LFD nur k+1 Fehlzugriffe hat, ist die relative Güte von LFU nicht beschränkt.



Anzahl Anfragen: 3 3 2 2

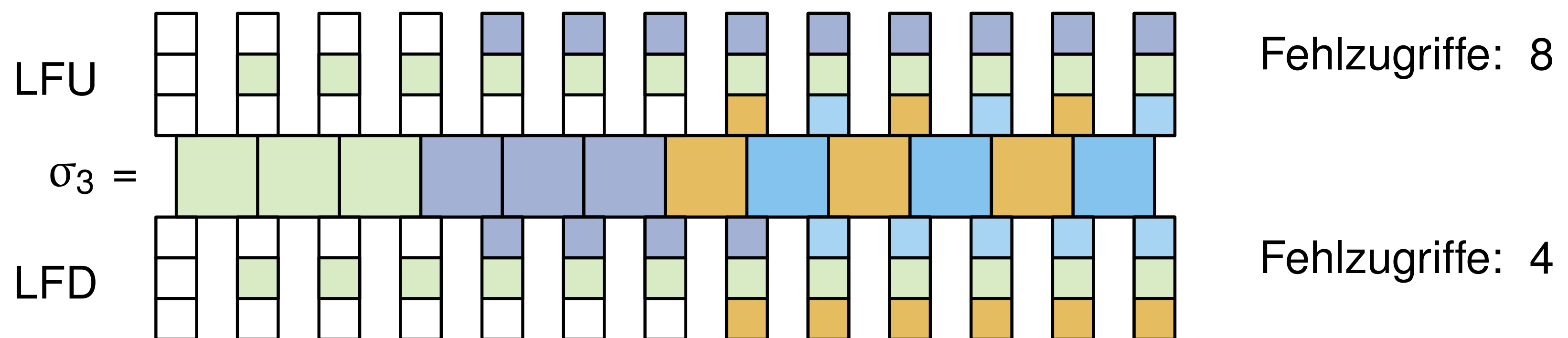
Kompetitive Paging-Algorithmen

LFU (Least Frequently Used) ist nicht kompetitiv, d.h. relative Güte c ist nicht beschränkt:

Sei k Cache-Größe und $P = \{p_1, \ldots, p_k, p_{k+1}\}$ Seiten im Hauptspeicher. Betrachte für beliebiges $m \in \mathbb{N}$ die Sequenz

$$\sigma_m = p_1^m, \ldots, p_{k-1}^m, (p_k, p_{k+1})^m$$

- LFU wird zuerst die Seiten $p_1, \ldots, p_{k-1}$ in den Cache aufnehmen (k-1 Fehlzugriffe).
- Nach $m \cdot (k-1)$ Anfragen, wird LFU für jedes Anfragepaar (p_k, p_{k+1}) einen Fehlzugriff haben. Folglich m weitere Fehlzugriffe.
- Da LFD nur k+1 Fehlzugriffe hat, ist die relative Güte von LFU nicht beschränkt.

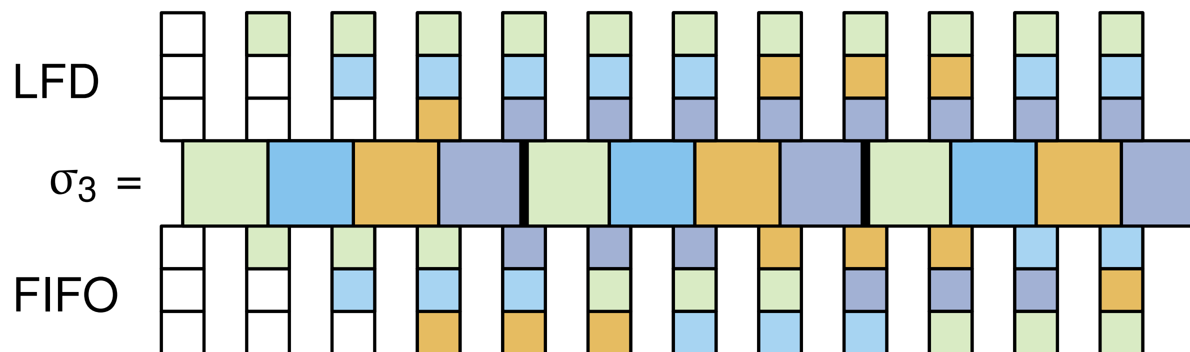


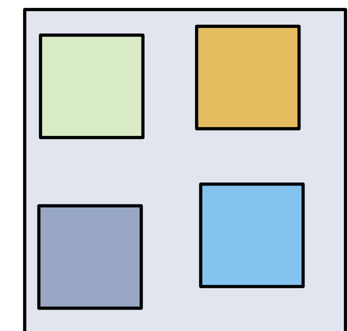
Anzahl Anfragen: 3 3 3 3

Kompetitive Online-Algorithmen

Theorem 51: Es gibt keinen deterministischen Online-Algorithmus für Paging, der eine bessere relative Güte als k erreicht. Dabei gibt k die Größe des Caches an.

Beispiel:





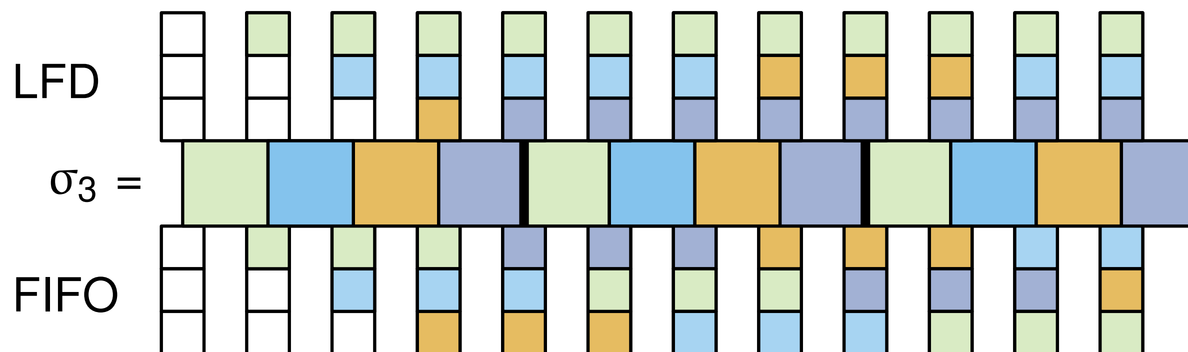
Kompetitive Online-Algorithmen

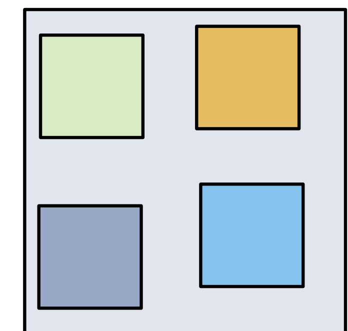
Theorem 51: Es gibt keinen deterministischen Online-Algorithmus für Paging, der eine bessere relative Güte als k erreicht. Dabei gibt k die Größe des Caches an.

Sei $\mathcal{A}$ Online-Algorithmus für Paging mit Cache-Größe k

Idee: Konstruiere beliebig lange Sequenz σ, sodass $|\sigma| = \mathcal{A}(\sigma) \geq k \cdot \text{LFD}(\sigma)$

Beispiel:





Kompetitive Online-Algorithmen

Theorem 51: Es gibt keinen deterministischen Online-Algorithmus für Paging, der eine bessere relative Güte als k erreicht. Dabei gibt k die Größe des Caches an.

Sei $\mathcal{A}$ Online-Algorithmus für Paging mit Cache-Größe k

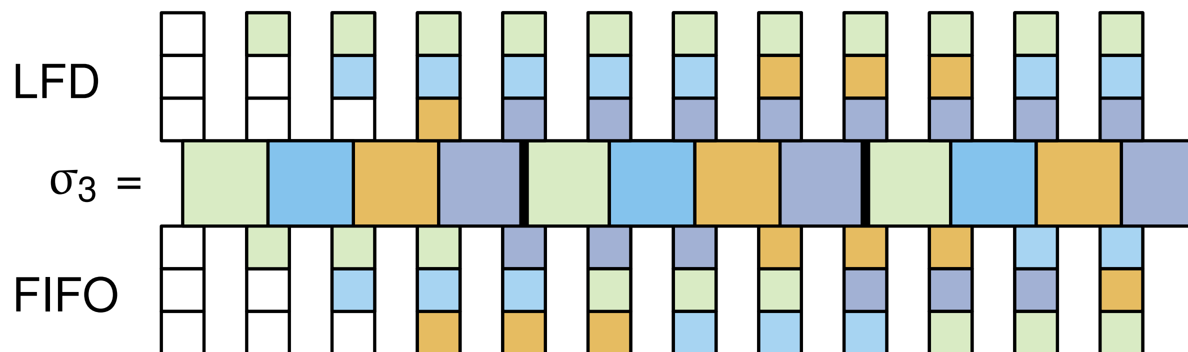
Idee: Konstruiere beliebig lange Sequenz σ, sodass $|\sigma| = \mathcal{A}(\sigma) \geq k \cdot \text{LFD}(\sigma)$

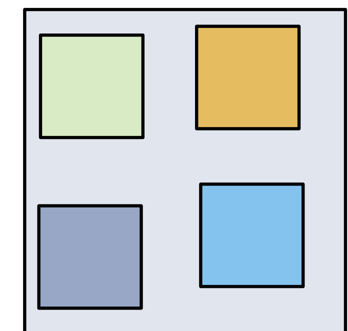
Annahme: Der Hauptspeicher enthält $k + 1$ Seiten

→ Es gibt immer eine Seite, die nicht im Cache ist.

Da $\mathcal{A}$ deterministisch ist, kann immer eine Sequenz σ konstruiert werden, sodass in jedem Schritt die angefragte Seite nicht im Cache ist.

Beispiel:





Kompetitive Online-Algorithmen

Theorem 51: Es gibt keinen deterministischen Online-Algorithmus für Paging, der eine bessere relative Güte als k erreicht. Dabei gibt k die Größe des Caches an.

Sei $\mathcal{A}$ Online-Algorithmus für Paging mit Cache-Größe k

Idee: Konstruiere beliebig lange Sequenz σ, sodass $|\sigma| = \mathcal{A}(\sigma) \geq k \cdot \text{LFD}(\sigma)$

Annahme: Der Hauptspeicher enthält $k+1$ Seiten

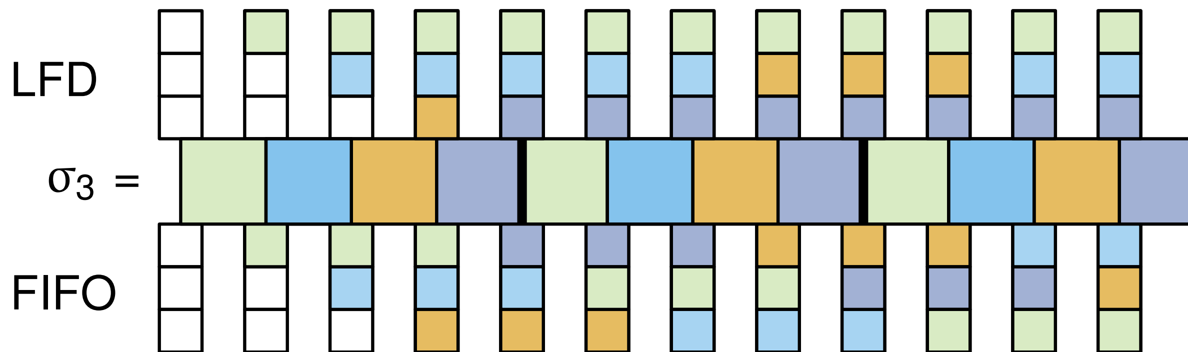
⟶ Es gibt immer eine Seite, die nicht im Cache ist.

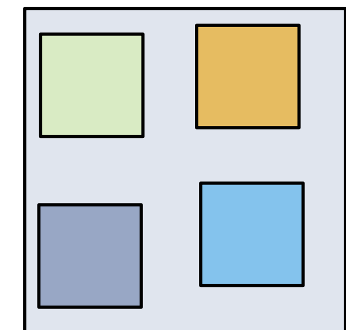
Da $\mathcal{A}$ deterministisch ist, kann immer eine Sequenz σ konstruiert werden, sodass in jedem Schritt die angefragte Seite nicht im Cache ist.

⟶ $\mathcal{A}(\sigma) = |\sigma|$

⟶ $\text{LFD}(\sigma) \leq \frac{|\sigma|}{k}$ denn für jede Stelle von σ hat LFD die folgenden k Anfragen im Cache.

Beispiel:



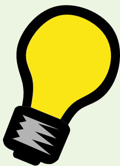


Institut für Theoretische Informatik
Prof. Dr. Dorothea Wagner

Konservative Paging-Algorithmen

(h,k)-Paging-Problem:

- Optimaler Offline-Algorithmus für Paging arbeitet auf Cache der Größe h.
- Online-Algorithmus für Paging arbeitet auf Cache der Größe k mit $k \geq h$.
- Online-Algorithmus bekommt größeren Cache um dessen Unwissenheit auszugleichen.



Béládys Anomalie: Für manche der Paging-Algorithmen kann man Zugriffssequenzen finden, sodass sie mit einem kleineren Cache weniger Fehlzugriffe liefern als mit einem größeren. FIFO ist ein solcher Algorithmus (siehe Übung).

Institut für Theoretische Informatik
Prof. Dr. Dorothea Wagner

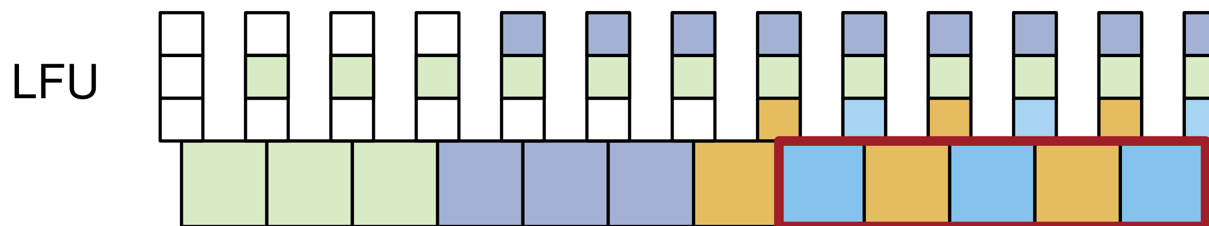
Konservative Paging-Algorithmen

(h,k)-Paging-Problem:

- Optimaler Offline-Algorithmus für Paging arbeitet auf Cache der Größe h.
- Online-Algorithmus für Paging arbeitet auf Cache der Größe k mit $k \geq h$.
- Online-Algorithmus bekommt größeren Cache um dessen Unwissenheit auszugleichen.

Definition 36: Ein Paging-Algorithmus $\mathcal{A}$ mit Cache der Größe k heißt *konservativ*, falls für jede Anfragesequenz σ folgende Aussage gilt:
Jede Teilsequenz σ' von σ, die maximal k verschiedene Seiten enthält, erzeugt maximal k Fehlzugriffe während $\mathcal{A}$ die Sequenz σ abarbeitet.

Beispiel: LFU ist nicht konservativ.



Sequenz enthält 2 verschiedene Seiten, aber erzeugt 5 Fehlzugriffe.

Institut für Theoretische Informatik
Prof. Dr. Dorothea Wagner

Konservative Paging-Algorithmen

Karlsruher Institut für Technologie

(h,k)-Paging-Problem:

- Optimaler Offline-Algorithmus für Paging arbeitet auf Cache der Größe h.
- Online-Algorithmus für Paging arbeitet auf Cache der Größe k mit $k \geq h$.
- Online-Algorithmus bekommt größeren Cache um dessen Unwissenheit auszugleichen.

Definition 36: Ein Paging-Algorithmus $\mathcal{A}$ mit Cache der Größe k heißt *konservativ*, falls für jede Anfragesequenz σ folgende Aussage gilt:
Jede Teilsequenz σ' von σ, die maximal k verschiedene Seiten enthält, erzeugt maximal k Fehlzugriffe während $\mathcal{A}$ die Sequenz σ abarbeitet.

Name	Ersetze Seite im Cache,...
FIFO: First In/First Out	die bereits am längsten im Cache ist.
LIFO: Last In/First Out	die am neusten im Cache ist.
LFU: Least Frequently Used	die bisher am wenigsten häufig angefordert wurde.
LRU: Least Recently Used	deren Anfrage am weitesten in der Vergangenheit liegt.
FWF: Flush When Full	(Gebe alle Seiten frei, wenn Cache voll ist.)

konservative Paging-Algorithmen (FIFO, LRU)

Konservative Paging-Algorithmen

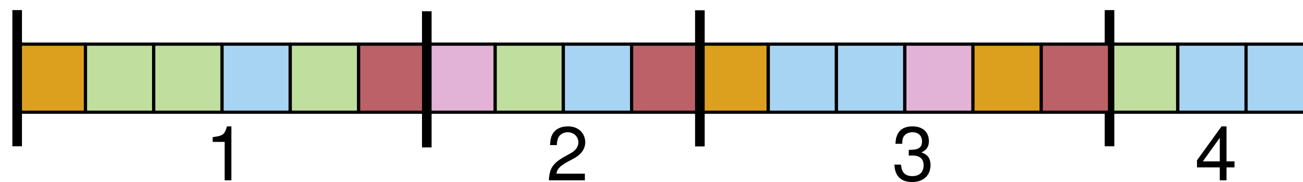
Theorem 52: Jeder konservative Paging-Algorithmus mit Cache-Größe k ist $\frac{k}{k-h+1}$-kompetitiv bezüglich jedes optimalen Offline-Algorithmus mit Cache-Größe h.

Beweis: Sei $\mathcal{A}$ konservativer Algorithmus mit Cache-Größe k.

Definiere *k-Phasen-Partition* einer Sequenz σ:

- Phase 0 ist die leere Sequenz.
- Für $i > 0$ sei Phase i längste Teilsequenz von σ, die sich direkt an Phase $i-1$ anschließt und maximal k verschiedene Seiten enthält.

Beispiel: 4-Phasen-Partition



Konservative Paging-Algorithmen

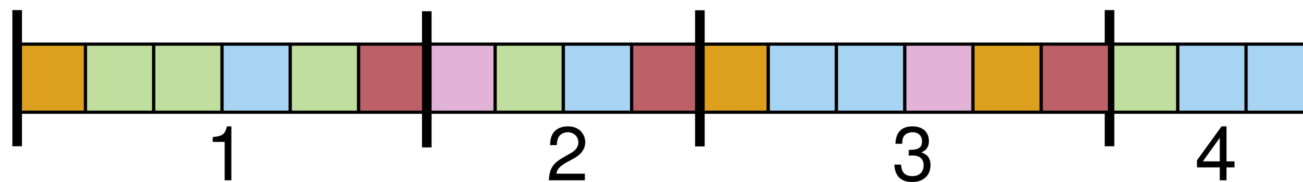
Theorem 52: Jeder konservative Paging-Algorithmus mit Cache-Größe k ist $\frac{k}{k-h+1}$-kompetitiv bezüglich jedes optimalen Offline-Algorithmus mit Cache-Größe h.

Beweis: Sei $\mathcal{A}$ konservativer Algorithmus mit Cache-Größe k.

Definiere *k-Phasen-Partition* einer Sequenz σ:

- Phase 0 ist die leere Sequenz.
- Für $i > 0$ sei Phase i längste Teilsequenz von σ, die sich direkt an Phase $i-1$ anschließt und maximal k verschiedene Seiten enthält.

Beispiel: 4-Phasen-Partition



Beobachtung: Da $\mathcal{A}$ konservativ ist, kann er pro Phase nur k Fehlzugriffe haben.

Zeige: Ein optimaler Algorithmus OPT mit Cache-Größe $h \leq k$ hat pro Phase mindestens $k - h + 1$ Fehlzugriffe. (Außer in der letzten Phase.)

→ Damit folgt die Behauptung.

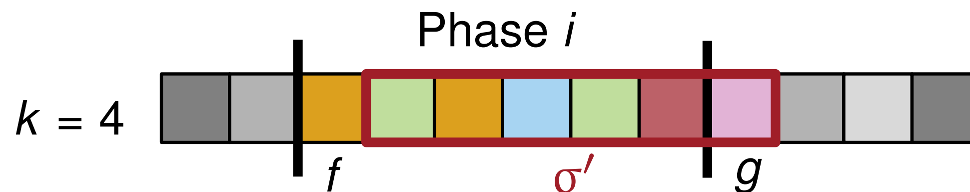
Konservative Paging-Algorithmen

Theorem 52: Jeder konservative Paging-Algorithmus mit Cache-Größe k ist $\frac{k}{k-h+1}$-kompetitiv bezüglich jedes optimalen Offline-Algorithmus mit Cache-Größe h.

Beweis: Sei $\mathcal{A}$ konservativer Algorithmus mit Cache-Größe k.

Zeige: Ein optimaler Algorithmus OPT mit Cache-Größe $h \leq k$ hat pro Phase mindestens $k-h+1$ Fehlzugriffe haben. (Außer in der letzten Phase.)

Betrachte beliebige Phase $i \geq 1$, die nicht die letzte Phase ist.



f = erste Seite der Phase i
g = erste Seite der nächsten Phase.
σ'= Sequenz beginnend nach f bis einschließlich g.

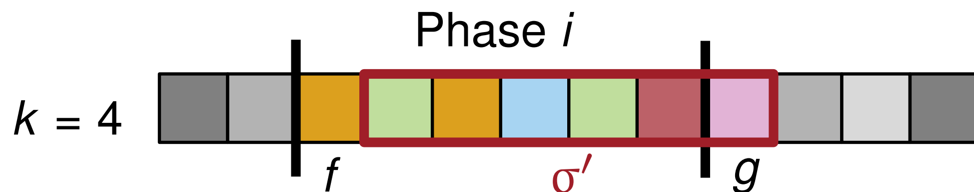
Konservative Paging-Algorithmen

Theorem 52: Jeder konservative Paging-Algorithmus mit Cache-Größe k ist $\frac{k}{k-h+1}$-kompetitiv bezüglich jedes optimalen Offline-Algorithmus mit Cache-Größe h.

Beweis: Sei $\mathcal{A}$ konservativer Algorithmus mit Cache-Größe k.

Zeige: Ein optimaler Algorithmus OPT mit Cache-Größe $h \leq k$ hat pro Phase mindestens $k - h + 1$ Fehlzugriffe haben. (Außer in der letzten Phase.)

Betrachte beliebige Phase $i \geq 1$, die nicht die letzte Phase ist.



f = erste Seite der Phase i
g = erste Seite der nächsten Phase.
σ' = Sequenz beginnend nach f bis einschließlich g.

1. Die Sequenz σ' enthält mindestens k zu f unterschiedliche Seiten (nach Def. einer Phase).
2. Wenn OPT beginnt σ' abzuarbeiten, enthält dessen Cache $h-1$ zu f unterschiedliche Seiten.

→ OPT hat mindestens $k - (h - 1) = k - h + 1$ Fehlzugriffe.

Institut für Theoretische Informatik
Prof. Dr. Dorothea Wagner

Konservative Paging-Algorithmen

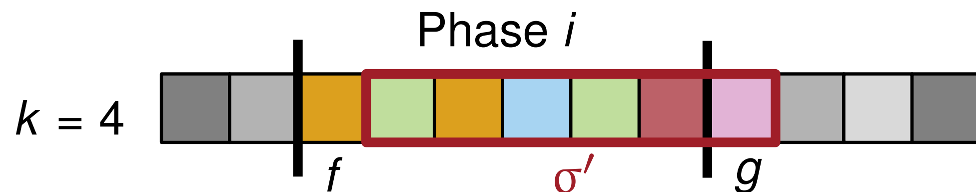
Karlsruher Institut für Technologie

Theorem 52: Jeder konservative Paging-Algorithmus mit Cache-Größe k ist $\frac{k}{k-h+1}$-kompetitiv bezüglich jedes optimalen Offline-Algorithmus mit Cache-Größe h.

Beweis: Sei $\mathcal{A}$ konservativer Algorithmus mit Cache-Größe k.

Zeige: Ein optimaler Algorithmus OPT mit Cache-Größe $h \leq k$ hat pro Phase mindestens $k - h + 1$ Fehlzugriffe haben. (Außer in der letzten Phase.)

Betrachte beliebige Phase $i \geq 1$, die nicht die letzte Phase ist.



f = erste Seite der Phase i
g = erste Seite der nächsten Phase.
σ' = Sequenz beginnend nach f bis einschließlich g.

1. Die Sequenz σ' enthält mindestens k zu f unterschiedliche Seiten (nach Def. einer Phase).
2. Wenn OPT beginnt σ' abzuarbeiten, enthält dessen Cache $h-1$ zu f unterschiedliche Seiten.

→ OPT hat mindestens $k - (h - 1) = k - h + 1$ Fehlzugriffe.

Letzte Phase: OPT kann weniger als $k - h + 1$ viele Fehlzugriffe haben.

→ Vernachlässigbar bei kompetiver Analyse, da dies nur einmal auftritt.

Institut für Theoretische Informatik
Prof. Dr. Dorothea Wagner

Konservative Paging-Algorithmen

Karlsruher Institut für Technologie

Theorem 52: Jeder konservative Paging-Algorithmus mit Cache-Größe k ist $\frac{k}{k-h+1}$-kompetitiv bezüglich jedes optimalen Offline-Algorithmus mit Cache-Größe h.

Anwendung:

- Für (k, k)-Paging ist jeder konservative Paging-Algorithmus k-kompetitiv, was nach Theorem 51 der unteren Schranke entspricht.
- Für $h = \frac{k}{2}$ sind konservative Online-Algorithmen für Paging 2-kompetitiv.