

## Das Paging-Problem

Der Hauptspeicher eines von-Neumann-Rechners besteht typischerweise aus dem Cache-Speicher und dem Arbeitsspeicher.

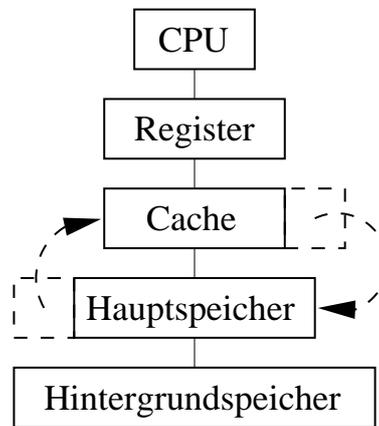


Abbildung 1: Speicherhierarchie

- (90, 10)-Regel: 90 % der Zugriffe auf Daten benötigen lediglich 10 % der Daten, auf denen das Programm arbeitet.
- Speicherzugriffe auf den Cache sind schnell, Speicherzugriffe auf den Hauptspeicher sind langsam.
- Daten in Cache und Hauptspeicher sind in Seiten zusammengefaßt  $\Rightarrow$  Zugriff auf Daten in Hauptspeicher teuer, da ein Seitenaustausch nötig ist (Cache ist immer vollständig mit Seiten belegt).
- Der Cache habe eine Kapazität von  $k$  Seiten.

Sei  $\sigma = (\sigma_1, \dots, \sigma_m)$  die Folge von Zugriffen auf Seiten, wobei  $\sigma_i$  die Aufforderung „Lies Datum aus Seite  $\sigma_i$ “ bedeute. Die Kosten eines Seitenzugriffs  $\sigma_i$  sind:

$$c(\sigma_i) = \begin{cases} 0, & \text{falls } \sigma_i \text{ im Cache} \\ 1 & \text{sonst} \end{cases}$$

Falls  $c(\sigma_i) = 1$ , so liegt ein *Seitenfehler* (*page fault*) vor.

**Ziel (Paging-Problem):** Führe die Swap-Operationen für  $\sigma$  so aus, daß

$$C(\sigma) = \sum_{i=1}^m c(\sigma_i)$$

minimal wird.

**Ein optimaler Offline-Algorithmus für das Paging-Problem**

**Strategie:** LFD (*longest forward distance*) „Lagere bei Seitenfehler eine solche Seite aus, die erst am weitesten in der Zukunft nachgefragt wird.“

**Satz (Belady 1966):** LFD ist ein optimaler (Offline-)Algorithmus für das Paging-Problem.

**Beweis:** (durch Widerspruch)

*Annahme:* Es existiere ein Algorithmus MIN, der besser ist als LFD, d.h. für alle Anfragefolgen  $\sigma$  gilt  $C_{\text{MIN}}(\sigma) \leq C_{\text{LFD}}(\sigma)$  und es existiert eine kürzeste Folge  $\sigma = (\sigma_1, \dots, \sigma_m)$  mit  $C_{\text{MIN}}(\sigma) < C_{\text{LFD}}(\sigma)$ . Sei  $\sigma_i$  diejenige Anfrage aus  $\sigma$ , für die sich MIN und LFD erstmals unterschiedlich verhalten, d.h. vor der Anfrage  $\sigma_i$  haben beide Verfahren denselben Cache-Inhalt, aber bei  $\sigma_i$  entfernt LFD die Seite  $q$ , während MIN die Seite  $p$  aus dem Cache entfernt, mit  $p \neq q$ . Sei  $t > i$  der erste Zeitpunkt, wo MIN die Seite  $q$  aus dem Cache entfernt.

**Behauptung:** Das Verhalten von MIN zwischen  $\sigma_i$  und  $\sigma_t$  läßt sich derart ändern, daß ein neuer Algorithmus MIN\* entsteht, der sich bei  $\sigma_i$  ebenfalls wie LFD verhält und für den gilt:  $C_{\text{MIN}^*}(\sigma) \leq C_{\text{MIN}}(\sigma)$  (d.h. MIN\* arbeitet einen Schritt mehr nach der Strategie LFD als MIN).

Bei  $\sigma_i$  lagert MIN\* die Seite  $q$  aus (wie LFD). Wegen der Strategie LFD wird die Seite  $p$  durch eine Anfrage  $\sigma_a$  ( $a > i$ ) vor der Anfrage nach Seite  $q$  durch  $\sigma_b$  in den Cache geladen.

$$\rightsquigarrow \sigma = (\sigma_1, \dots, \sigma_{i-1}, \sigma_i, \dots, \underbrace{\sigma_a}_p, \dots, \underbrace{\sigma_b}_q, \dots, \sigma_m)$$

Wie LFD bedient MIN\* bei  $\sigma_i$  die Anfrage durch Entfernen von  $q$  aus dem Cache. Nach Verarbeitung von  $\sigma_i$  haben MIN und MIN\*  $k - 1$  gleiche Seiten im Cache.

*Zeige:* Bis zu dem Zeitpunkt, wo MIN und MIN\* sich wieder gleich verhalten, haben MIN und MIN\* immer  $k - 1$  gleiche Seiten im Cache.

*Verhalten von MIN\*:*

$t \hat{=}$  Zeitpunkt, wo  $q$  unter MIN aus Cache entfernt wird. Unterscheide zwei Fälle:

1.  $i < t < b$

Beschreibung, wie MIN\* Seitenanfragen  $\sigma_l$ ,  $i < l < t$ , bedient. Durch Induktion über  $l$  zeige: MIN und MIN\* haben  $k - 1$  gleiche Seiten im Cache ( $\rightsquigarrow$  es gibt genau eine Seite  $e$  in MIN\*s Cache mit  $e \notin$  MINs Cache).

*Induktionsanfang:*  $l = i + 1 \rightsquigarrow e = p$

$\sigma_l \neq e$ : MIN hat Seitenfehler  $\Leftrightarrow$  MIN\* hat Seitenfehler. Bei Seitenfehler entfernt MIN\* dieselbe Seite aus dem Cache wie MIN (da MIN  $q$  erst zum Zeitpunkt  $t$  aus Cache entfernt, kann MIN\* sich hier genauso wie MIN verhalten), d.h. MIN und MIN\* enthalten jeweils nach Bedienung von  $\sigma_l$  wiederum  $k - 1$  gleiche Seiten im Cache.

$\sigma_l = e$ : Hier hat MIN Seitenfehler, MIN\* jedoch nicht. MIN holt  $e$  in den Cache und lagert eine Seite aus. Nach dieser Aktion haben MIN und MIN\* wieder  $k - 1$  gleiche Cacheseiten.

Bei Bedienung von  $\sigma_t$  haben beide einen Seitenfehler: MIN\* entfernt  $e$  aus dem Cache und MIN entfernt  $q$  aus dem Cache. Ab jetzt verhalten sich MIN und MIN\* identisch und es gilt:  $C_{\text{MIN}^*}(\sigma) \leq C_{\text{MIN}}(\sigma)$ .

2.  $t \geq b$

MIN\* verwendet hier bei  $\sigma_l - i < l < b$  - dieselbe Strategie wie in Fall 1. Mit derselben

Argumentation wie oben folgt bei  $\sigma_b = q$ : MIN und MIN\* haben  $k - 1$  gemeinsame Seiten im Cache. Da die Seite  $p$  vor  $q$  angefragt wird, ist der Fall „ $\sigma_l = e$ “ wenigstens einmal eingetreten  $\rightsquigarrow C_{\text{MIN}^*}(\sigma_1 \dots \sigma_{t-1}) < C_{\text{MIN}}(\sigma_1 \dots \sigma_{t-1})$ .

Nach Definition entsteht bei der Anfrage  $\sigma_b = q$  bei MIN *kein* Seitenfehler (bei  $t$  lagert MIN erstmals  $q$  aus,  $t \geq b$ !), jedoch bei MIN\*. MIN\* ersetzt dann die Seite  $e$  durch die Seite  $q$ . Danach haben MIN und MIN\* wieder denselben Cache-Inhalt. Ab hier verhält sich MIN\* wieder wie MIN. Wegen  $C_{\text{MIN}^*}(\sigma_1 \dots \sigma_{t-1}) < C_{\text{MIN}}(\sigma_1 \dots \sigma_{t-1})$  folgt:

$$C_{\text{MIN}^*}(\sigma) \leq C_{\text{MIN}}(\sigma).$$

Hiermit haben wir gezeigt: Ein optimaler Algorithmus für das Paging-Problem, der  $i - 1$  Schritte die Strategie LFD verfolgt, kann zu einem optimalen Algorithmus für das Problem modifiziert werden, der  $i$  Schritte lang LFD anwendet. Mit Induktion über  $i$  folgt:

$$C_{\text{LFD}}(\sigma) \leq C_{\text{MIN}}(\sigma) \rightsquigarrow \text{LFD ist eine optimale Strategie.}$$

### Arbeitsweise von Online-Algorithmen:

Ein Online-Algorithmus  $A$  muß eine Folge  $\sigma$  von Anfragen  $\sigma = (\sigma_1, \dots, \sigma_m)$  von Aufgaben (*requests*) bedienen, wobei request  $\sigma_i$  sofort beantwortet (*answered*) werden muß, sobald  $\sigma_i$  eingelesen worden ist, *ohne* Kenntnis der weiteren requests  $\sigma_{i+1} \dots \sigma_m$ . Auch  $m$  ist im allgemeinen nicht bekannt.

**Bemerkung:** Online-Algorithmen sind i. a. nicht kostenoptimal.

Die Bewertung der Güte von Online-Algorithmen erfolgt im Vergleich mit den Kosten eines besten *Offline*-Algorithmus für die einzelnen request-Folgen (bei vollständiger Information).

**Bemerkung:** Online-Algorithmen arbeiten mit unvollständiger Information.

### Beispiel: Radlerproblem.

Herr K. besitzt eine Ferienwohnung in der Eifel, die er am Wochenende nutzt. Herr K. ist Hobby-Radler und überlegt, ob er für 1000 € ein Trekking-Rad kaufen oder ob er stattdessen wöchentlich für jeweils 10 € ein Trekking-Bike mieten soll.

Aus gesundheitlichen Gründen ist unbekannt, wie oft Herr K. das Trekking-Rad nutzen kann. Herr K. möchte nun die Kosten für die Trekking-Rad-Nutzung minimieren. Bei vollständiger Information kann eine optimale Entscheidung getroffen werden (*Offline-Verfahren*).

Sei  $\sigma = (\sigma_1, \dots, \sigma_m)$  die Folge der von K. durchgeführten Radwanderwochenenden, wobei  $\sigma_i$  jeweils die Frage „Kaufen oder Mieten“ repräsentiert. Je nach getroffener Entscheidung verursacht  $\sigma_i$  unterschiedliche Kosten:

$$c(\sigma_i) = \begin{cases} 1000 \text{ €}, & \text{falls } \sigma_i = \text{Kaufen und } \forall 1 \leq j < i : \sigma_j = \text{Mieten} \\ 10 \text{ €}, & \text{falls } \sigma_i = \text{Mieten und } \forall 1 \leq j < i : \sigma_j = \text{Mieten} \\ 0 \text{ €}, & \text{falls } \exists j : 1 \leq j < i : \sigma_j = \text{Kaufen} \end{cases}$$

Wie sollen nun die Entscheidungen „Mieten oder Kaufen“ für die Folge  $\sigma = (\sigma_1, \dots, \sigma_m)$  gefällt werden, damit

$$C(\sigma) = \sum_{i=1}^m c(\sigma_i)$$

minimiert wird?

Ein einfacher *Offline*-Algorithmus  $A_{\text{off}}$  löst das Problem:

```

KP:=1000€; MP:=10€;
if  $m > \frac{KP}{MP}$  then  $\sigma_i = \text{Kaufen}$ 
      else for all  $1 \leq i \leq m$  do  $\sigma_i = \text{Mieten}$ ;

```

**Behauptung:** Diese Offline-Strategie ist optimal!

Ein Online-Algorithmus kennt nun  $m$  nicht. Wie soll also  $A_{\text{on}}$  die Entscheidungen  $\sigma_i$  treffen?

**Behauptung:** Zu jeder Strategie eines Online-Algorithmus  $A_{\text{on}}$  gibt es ein  $\sigma$ , so daß

$$C_{A_{\text{on}}}(\sigma) > C_{A_{\text{off}}}(\sigma).$$

**Behauptung:** Es gibt Online-Algorithmus  $A_{\text{on}}$  für das Radlerproblem, der Kosten

$$C_{A_{\text{on}}}(\sigma) \leq 2C_{A_{\text{off}}}(\sigma)$$

verursacht.

Online-Algorithmus  $A_{\text{on}}$  für das Radlerproblem:

```

Input:  $\sigma = (\sigma_1, \dots, \sigma_m)$  online
for all  $1 \leq i \leq m$  do
  if  $i \leq \frac{KP}{MP}$  then
     $\sigma_i = \text{Mieten}$ 
  else
    if  $\frac{KP}{MP} < i \leq \frac{KP}{MP} + 1$  then
       $\sigma_i = \text{Kaufen}$ 
    else
      „schon entschieden“

```

Es gilt  $C_{A_{\text{on}}}(\sigma) \leq 2C_{A_{\text{off}}}(\sigma)$ , denn falls  $m \leq \frac{KP}{MP}$ , so ist  $C_{A_{\text{on}}}(\sigma)$  gerade gleich  $C_{A_{\text{off}}}(\sigma)$ , sonst gilt  $C_{A_{\text{on}}}(\sigma) = 2C_{A_{\text{off}}}(\sigma)$ .

**Berechnung der Güte eines Online-Algorithmus**  $A_{\text{on}}$  gemäß Sleator/Tarjan mittels *competitive analysis*, d.h. vergleiche  $C_{A_{\text{on}}}(\sigma)$  mit  $C_{\text{optimal}}(\sigma)$  ( $\hat{=}$  Kosten eines optimalen Offline-Algorithmus). Sei  $c > 0$ . Wir sagen,  $A_{\text{on}}$  heißt *c-competitive* (*c-konkurrierend*), falls ein  $a \geq 0$  existiert mit  $C_{A_{\text{on}}}(\sigma) \leq c \cdot C_{\text{optimal}}(\sigma) + a$  für jede Anfragefolge  $\sigma$ .  $c$  heißt *competitive ratio*.

**Bemerkung:** Der Online-Algorithmus aus dem Radlerbeispiel ist 2-competitive.

## Online-Strategien für das Paging-Problem

- LIFO: Entferne aus Cache bei Seitenfehler die zuletzt geladene Seite.
- FIFO: Entferne aus Cache bei Seitenfehler die zuerst geladene Seite.
- LRU (least recently used): Entferne aus Cache bei Seitenfehler die Seite, auf die am längsten nicht mehr zugegriffen worden ist.
- LFU (least frequently used): Entferne aus Cache bei Seitenfehler die Seite, auf die am wenigsten zugegriffen wurde.

**Bemerkung:** LIFO und LFU sind nicht  $c$ -competitive für beliebiges  $c > 0$ .

**Satz:** LRU und FIFO sind  $k$ -competitive ( $k = \text{Anzahl der Seiten im Cache}$ ).

**Beweis:** (hier nur für LRU, FIFO analog)

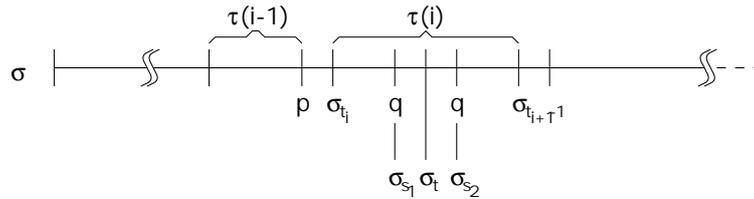
Sei  $\sigma = (\sigma_1, \dots, \sigma_m)$  beliebige Anfrage-Folge. Zeige:  $C_{\text{LRU}}(\sigma) \leq k \cdot C_{\text{OPT}}(\sigma)$ .

LRU und OPT starten mit demselben Cache-Inhalt. Zerlege nun  $\sigma$  in Teilfolgen  $\tau(0), \tau(1), \tau(2), \dots$ , wobei LRU in der Teilfolge  $\tau(0)$  höchstens  $k$  Seitenfehler besitzt und in den Teilfolgen  $\tau(i), i \geq 1$ , genau  $k$  Seitenfehler (Die Zerlegung von  $\sigma$  in  $\tau(0) \dots$  kann konstruktiv aus  $\sigma$  durch Scannen von rechts nach links gefunden werden. Zähle hierbei die Seitenfehler. Immer wenn  $k$  Seitenfehler erreicht worden sind, ist eine neue Teilfolge gefunden.).

**Behauptung:** In jeder Teilfolge  $\tau(i), i \geq 0$ , macht OPT mindestens einen Seitenfehler ( $\Rightarrow$  Satz).

Die Behauptung gilt für  $\tau(0)$ , da LRU und OPT mit derselben Cache-Belegung starten und somit der erste Seitenfehler sowohl bei LRU als auch bei OPT entsteht.

Sei  $\sigma_{t_i}$  die erste Anfrage und  $\sigma_{t_{i+1}-1}$  die letzte Anfrage im Block  $\tau(i), i \geq 1$ . Sei  $p$  die zuletzt angefragte Seite in  $\tau(i-1)$ .



**Lemma:**  $\tau(i)$  enthält Anfragen nach  $k$  paarweise verschiedenen Seiten, die alle von  $p$  verschieden sind.

(Mit Lemma folgt, daß OPT in  $\tau(i)$  mindestens einen Seitenfehler begeht, da bei Betreten von  $\tau(i)$   $p$  im Cache ist und in  $\tau(i)$  noch  $k$  verschiedene Seiten  $\neq p$  benötigt werden. Da die Cache-Größe durch  $k$  beschränkt ist, muß jeder Paging-Algorithmus mindestens eine Seite in  $\tau(i)$  auslagern.)

**Beweis (Lemma):** Das Lemma gilt offensichtlich, falls Anfragen, für die unter LRU in  $\tau(i)$  ein Seitenfehler auftritt, alle paarweise verschieden und auch verschieden von  $p$  sind. Es bleiben zwei weitere Fälle:

1. LRU begehe zweimal den Seitenfehler bei  $q$  in  $\tau(i)$ , d.h.  $\exists t_i \leq s_1 < s_2 \leq t_{i+1} - 1$ . LRU hat Seitenfehler bei  $\sigma_{s_1} = q = \sigma_{s_2}$  in  $\tau(i)$ .  $q$  wurde also in den Cache gebracht zum Zeitpunkt  $s_1$ , aus dem Cache wieder entfernt zum Zeitpunkt  $t$  und wiederum in den Cache geholt zum Zeitpunkt  $s_2$ :  $s_1 < t < s_2$ .  
Beim Entfernen von  $q$  aus dem Cache zum Zeitpunkt  $t$  muß  $q$  diejenige Seite im Cache gewesen sein, die *least recently used* war. Dann besitzt die Teilfolge  $\sigma_{s_1}, \dots, \sigma_{s_2}$  Anfragen nach  $k+1$  verschiedenen Seiten, denn bevor  $q$  bei  $\sigma_t$  aus dem Cache entfernt wurde, müssen alle zum Zeitpunkt  $\sigma_{s_1}$  im Cache befindlichen Seiten wegen der LRU-Regel aus dem Cache entfernt worden sein. Von diesen  $k+1$  verschiedenen Seiten sind  $k$  verschieden von  $p$ .
2. LRU begeht keine zwei Seitenfehler bei  $q$  in  $\tau(i)$ , aber bei einem Seitenfehler in  $\tau(i)$  wird Seite  $p$  angefordert (Argumentation wie oben).

**Satz:** Sei  $A$  ein (determinierter) Online-Algorithmus für das Paging-Problem. Wenn der Algorithmus  $A$   $c$ -competitive ist, so gilt:

$$c \geq k.$$

**Beweis:** Sei  $S = \{p_1, \dots, p_{k+1}\}$  eine Folge von Seiten ( $k \hat{=}$  Cachegröße), A beliebiger Online-Algorithmus und OPT optimaler Offline-Algorithmus. Betrachte Anfragefolge, für die jede Anfrage nach einer Seite gerade diejenige Seite fordert, die nicht im Cache ist, d.h. A macht bei jeder Anfrage einen Seitenfehler. OPT habe einen Seitenfehler bei Anfrage  $\sigma_t$ . Beim Bedienen von  $\sigma_t$  kann OPT eine Seite auslagern, die während der nächsten  $k - 1$  Anfragen  $\sigma_{t+1} \dots \sigma_{t+k-1}$  nicht nachgefragt wird  $\rightsquigarrow$  für je  $k$  aufeinanderfolgende Anfragen begeht OPT höchstens einen Fehler.

## Randomisierung von Online-Algorithmen

Bei der Bestimmung des Faktors  $c$  eines  $c$ -competitive Online-Algorithmus A wird immer ein Gegner (*adversary*) definiert, der eine Anfragefolge  $\sigma$  erzeugt, die er und A bedienen müssen. Daher kennt der Gegner (bisher ein optimaler Offline-Algorithmus) die von A verfolgte Strategie. Bei (randomisierten) Online-Algorithmen spielt es eine Rolle, ob es dem Gegner erlaubt ist, die von A in der Vergangenheit getroffenen Entscheidungen zu sehen.

Unterscheide nun drei Typen von Gegnern:

- *Oblivious adversary* (vergeßlicher Gegner)  
Der Oblivious-Gegner erzeugt vor Beginn der Rechnung von A eine vollständige Anfragefolge  $\sigma$ . Der oblivious Gegner erhält Kosten von  $C_{\text{OPT}}(\sigma)$ .
- *Adaptiver Online-Gegner* (adaptiv  $\hat{\approx}$  „situationsangepaßt“)  
Der Gegner kennt die Aktionen des Online-Algorithmus und erzeugt die nächste Anfrage  $\sigma_i$  auf Grund der Antworten des (randomisierten) Online-Algorithmus A bei allen früheren Anfragen. Der Gegner bedient alle Anfragen *online*, d.h. ohne Kenntnis der (Zufalls-)Aktionen von A in Gegenwart und Zukunft. Kosten des Gegners:  $C_{\text{OPT adapt. Online-Gegner}}(\sigma)$ .
- *Adaptiver Offline-Gegner*  
Dieser Gegner erzeugt situationsangepaßt (adaptiv) Anfragefolgen  $\sigma$ . Die Kosten des adaptiven Offline-Gegners sind:  $C_{\text{OPT}}(\sigma)$ .

**Definition:** Der randomisierte Online-Algorithmus A ist  $c$ -competitive gegenüber dem Oblivious-Gegner, wenn gilt:

$$\underbrace{E(C_A(\sigma))}_{\substack{\text{Erwartungswert über} \\ \text{alle zufällig gesteuerten} \\ \text{Rechnungen von A bei } \sigma}} \leq c \cdot C_{\text{OPT}}(\sigma) + a.$$

Sei A ein randomisierter Online-Algorithmus und ADV ein adaptiver Online-(Offline-)Gegner. Dann ist A  $c$ -competitive gegenüber einem beliebigen solchen ADV, wenn gilt:

$$E(C_A(\sigma)) \leq c \cdot E(C_{\text{ADV}}(\sigma)) + a.$$

**Satz (Ben-David et al. 1994):** Wenn es einen  $c$ -competitive *randomisierten* Online-Algorithmus gegen einen beliebigen adaptiven *Offline*-Gegner gibt, so existiert bereits ein  $c$ -competitive *deterministischer* Online-Algorithmus (d.h. Randomisierung ist kein Mittel gegen adaptive *Offline*-Gegner).

[Ohne Beweis]

**Satz (Ben-David et al.):** Wenn es einen  $c$ -competitive *randomisierten* Online-Algorithmus gegen einen beliebigen adaptiven *Online*-Gegner gibt, so gibt es einen  $c^2$ -competitive *deterministischen* Online-Algorithmus.

[Ohne Beweis]

## Randomisierte Paging-Algorithmen gegen Oblivious-Gegner

### Der Randomisierte Online-Algorithmus MARKIERE

MARKIERE arbeitet in *Phasen*: Zu Beginn einer Phase sind alle Seiten unmarkiert. Immer, wenn eine Seite angefordert wird, wird sie markiert. Bei einem Seitenfehler wird unter den unmarkierten Seiten im Cache (gleichverteilt) zufällig eine Seite zum Auslagern ausgewählt. Wird bei der Seitenanforderung festgestellt, daß alle Seiten im Cache markiert sind, so beginnt die nächste Phase.

**Definition:** Sei  $H_k := \sum_{i=1}^k \frac{1}{i}$  die  $k$ -te harmonische Zahl.

**Bemerkung:**  $H_k \approx \ln(k) + \underbrace{\gamma}_{\approx 0,577\dots}$  (für  $k$  groß).  
(Eulerkonstante)

**Satz:** MARKIERE ist  $2 \cdot \ln(k)$ -konkurrierend gegenüber jedem Oblivious-Gegner (mit  $k \hat{=}$  Cachegröße).

**Bemerkung:** Es gibt einen komplizierteren randomisierten Online-Algorithmus, der  $H_k$ -konkurrierend ist gegenüber beliebigen Oblivious-Gegnern.

**Bemerkung:**  $H_k$ -competitive ( $\hat{=}$   $\ln(k)$ -competitive) ist das Beste, was randomisierte Online-Algorithmen gegenüber Oblivious-Gegnern erreichen können.

**Beweis:** Sei  $\sigma = (\sigma_1, \dots, \sigma_m)$  Anfragefolge, so daß MARKIERE Seitenfehler bei  $\sigma_1$  ( $\mathbb{E}$ ) hat. Sei  $\sigma_i \dots \sigma_j$  eine Phase von MARKIERE, d. h.  $j > i$  ist die kleinste Zahl, so daß  $\sigma_i, \dots, \sigma_{j+1}$  Anfragen nach genau  $k + 1$  verschiedenen Seiten enthalten (denn am Ende einer Phase sind alle Seiten im Cache markiert).

Betrachte eine beliebige Phase. Eine Seite heißt *fad* (verbraucht, *stale*), falls sie unmarkiert ist, aber in der vorherigen Phase markiert worden ist. Eine Seite heißt *sauber* (*clean*), wenn sie weder fad noch markiert ist (d. h. saubere Seiten sind immer nur im Hauptspeicher).

Sei  $c$  die Anzahl der in der aktuellen Phase angeforderten sauberen Seiten. Zeige nun:

1. Die amortisierte Zahl der Seitenfehler, die OPT macht, beträgt mindestens  $c/2$ .
2. Die erwartete Zahl der Seitenfehler, die MARKIERE begeht, ist höchstens  $c \cdot H_k$ .

Hiermit ist dann offensichtlich auch die Aussage des Satzes gezeigt.

**Beweis zu 1:** Sei  $S_{\text{OPT}}$  die Menge der Seiten im Cache von OPT,  
 $S_M$  die Menge der Seiten im Cache von MARKIERE,  
 $d_I = |S_{\text{OPT}} \setminus S_M|$  zu Beginn einer Phase,  
 $d_F = |S_{\text{OPT}} \setminus S_M|$  zum Ende einer Phase.

**Behauptung:** OPT hat  $\geq c - d_I$  Seitenfehler in der Phase, denn  $c - d_I$  der sauberen Seiten, die im Cache während der Phase benötigt werden, befinden sich zu Phasenbeginn nicht in OPTs Cache.

**Behauptung:** OPT hat  $\geq d_F$  Seitenfehler in der Phase, denn  $d_F$  der während der Phase im Cache benötigten Seiten befinden sich am Phasenende nicht mehr im Cache (wurden also durch

Seiteneinlagerungen verdrängt).

↪ OPT macht während der Phase

$$\geq \max\{c - d_I, d_F\} \geq \frac{1}{2}(c - d_I + d_F) = \frac{c}{2} - \frac{d_I}{2} + \frac{d_F}{2}$$

viele Seitenfehler. Für je zwei aufeinanderfolgende Phasen  $Ph_1$  und  $Ph_2$  gilt:  $d_{F_{Ph_1}} = d_{I_{Ph_2}}$ .

↪ Aufsummieren über alle Phasen und Division durch die Phasenzahl ergibt:

$$\frac{c}{2} - \underbrace{\frac{d_{I_{\text{Anfang}}}}{2 \cdot \#Phasen}}_{=0} + \underbrace{\frac{d_{F_{\text{Ende}}}}{2 \cdot \#Phasen}}_{\geq 0} \geq \frac{c}{2}.$$

**Beweis zu 2:**  $c$  Anfragen nach sauberen Seiten verursachen Kosten von  $c$  ( $c \geq 1$  wegen der Definition einer Phase) ↪ während der aktuellen Phase werden  $s = k - c \leq k - 1$  Anfragen nach faden Seiten gestellt.

Berechne für  $1 \leq i \leq s$  die erwarteten Kosten der Anfrage nach der  $i$ -ten faden Seite (Kosten 0, wenn Seite im Cache, 1 sonst):

Sei  $c(i) = \#$  sauberer Seiten, die in der Phase vor der  $i$ -ten Anfrage nach einer faden Seite angefragt wurden. Sei  $s(i) = \#$  verbleibender fader Seiten vor der  $i$ -ten Anfrage nach fader Seite. Wenn MARKIERE die  $i$ -te Anfrage nach einer faden Seite bedient, befinden sich  $s(i) - c(i)$  der  $s(i)$  faden Seiten im Cache, jede mit gleicher Wahrscheinlichkeit. Die erwarteten Kosten dieses Seitenzugriffs sind also:

$$\underbrace{\frac{s(i) - c(i)}{s(i)}}_{\substack{\text{W'keit, daß angefragte} \\ \text{fade Seite noch} \\ \text{im Cache ist}}} \cdot 0 + \underbrace{\frac{c(i)}{s(i)}}_{\substack{\text{W'keit, daß angefragte} \\ \text{fade Seite nicht mehr} \\ \text{im Cache ist}}} \cdot 1 \leq \frac{c}{s(i)} \stackrel{s(i)=k-(i-1)}{=} \frac{c}{k-i+1}.$$

↪ Erwartete Kosten der Bedienung der faden Seiten in der aktuellen Phase:

$$\sum_{i=1}^s \frac{c}{k-i+1} \leq \sum_{i=1}^{k-1} \frac{c}{k-i+1} = \sum_{i=2}^k \frac{c}{i} = c(H_k - 1).$$

$$\Rightarrow \# \text{ Seitenfehler von MARKIERE pro Phase} \leq \underbrace{c(H_k - 1)}_{\substack{\text{SF bei Zugriff} \\ \text{auf fade Seiten}}} + \underbrace{c}_{\substack{\text{SF bei Zugriff} \\ \text{auf saubere} \\ \text{Seiten}}} = c \cdot H_k.$$