Online Algorithms

CS 3AC3

Ryszard Janicki

Department of Computing and Software, McMaster University, Hamilton, Ontario, Canada

Acknowledgments: Material based on *An Introduction to the Analysis of Algorithms* by Michael Soltys (Chapter 5)

Online vs Offline Algorithms

- The algorithms discussed so far were *offline* algorithms, in the sense that entire input was given at the beginning.
- In this lecture we change our paradigm and consider *online* algorithms, where the input is presented piecemeal, in a serial fashion, and the algorithms has to make decisions based on incomplete information, without knowledge of future events.
- *Typical example*: A caching discipline.

Caching Discipline

- A hard disk from which data is read into *much smaller* random access memory.
- It must be decided which data has to be overwritten with new data.
- New requests arrive continuously, and it is hard to predict future requests.
- We *must* overwrite parts of random access memory with new requests, but we have to try to minimize future *misses*.
- *Misses*: data that is required but not present in random access memory, and so it has to be brought from the hard disk.
- Minimizing the number of misses is difficult when the future requests are unknown.

List Accessing Problem

- A filing cabinet contains l labeled but unsorted files.
- We receive a sequence of requests to access files; each request is a file label.
- After receiving a request for a file, we must *locate* it, *process* it, and *return* it to the cabinet.
- Since the files are unordered, we must flip through the files starting at the beginning, until the requested file is located.
- If a file is in position i, we incur a search cost of i in locating it.
- If the file is not in the cabinet, the cost is l, which is the total number of files.
- After taking out the file, we must return it to the cabinet, but we may choose to reorganize the cabinet, for instance, we might put it closer to the front.

List Accessing Problem

- After taking out the file, we must return it to the cabinet, but we may choose to reorganize the cabinet, for instance, we might put it closer to the front.
- It may save us some search time in the future: if a certain file is requested frequently, it is wise to insert it closer to the front.
- Our goal is to find a reorganization rule that minimize the search time.

List Accessing Problem

- Let $\sigma = \sigma_1, \sigma_2, \ldots, \sigma_n$ be a finite sequence of n requests.
- To service request σ_i, an algorithm must search for the item labeled σ_i by traversing the list from the begiining.
- If item σ_i is in position j, the cost of search is j (up to a constant).
- Furthermore, the algorithm may reorganize the list any time.
- Each transposition has a cost 1 (up to a constant), however, immediately after accessing an item, we allow it to be move free of charge to any location closer to the front of this list.
- Why free? As we scan the list we keep a pointer at a given location along the way and then insert σ_i in that location almost free of additional search or reorganization costs (our estimations are up to some constant).
- All other transportations are paid.

Static List Accessing Model: Move To Front Algorithm

- Static List Accessing Model: Only requests are to access an item on the list, there are no insertions or deletions.
- Move To Front (MTF) algorithm: After accessing an intem, we move it to the front of the list, without changing the relative order of the other items.
- $MTF(\sigma)$ - for $\sigma = \sigma_1, \sigma_2, \ldots, \sigma_n$, for each i, it finds σ_i and reallocates it to the front.
- The cost of $MTF(\sigma)$, traditionally also denoted by $MTF(\sigma)$, is the sum of the costs of all the searches, as moving to the front is free.

Move To Front Algorithm

Example

Let $l = x_1, x_2, x_3, x_4, x_5$ and $\sigma = x_5, x_1, x_2, x_5$

Notation: $\overset{x_j,(k)}{\rightarrow}$ means x_j *found and moved to the front at the cost* k.

$$x_1, x_2, x_3, x_4, x_5 \overset{x_5,(5)}{\rightarrow} x_5, x_1, x_2, x_3, x_4 \overset{x_1,(2)}{\rightarrow} x_1, x_5, x_2, x_3, x_4 \overset{x_2,(3)}{\rightarrow}$$
$$x_2, x_1, x_5, x_3, x_4 \overset{x_5,(3)}{\rightarrow} x_5, x_2, x_1, x_5, x_3, x_4$$

- Total cost $MTF(\sigma) = 5 + 2 + 3 + 3 = 13$.
- The list l has been changed from *old* $l = x_1, x_2, x_3, x_4, x_5$ to *new* $l = x_5, x_2, x_1, x_5, x_3, x_4$.

Move To Front Algorithm

Theorem

Let OPT be an optimal offline algorithm for the static list accessing model. Suppose that OPT and MTF both start with the same list configuration. Then, for any sequence of requests σ, where $|\sigma| = n$, we have:

$$MTF(\sigma) \leq 2OPT_S(\sigma) + OPT_P(\sigma) - OPT_F(\sigma) - n,$$

where $OPT_S(\sigma)$, $OPT_P(\sigma)$,$OPT_F(\sigma)$ are the total cost of searches, *the total cost of* paid transpositions *and the total cost of* free transpositions, *of OPT on σ, respectively.*

Proof.

- Assume *MTF* and *OPT* work in parallel, starting from the same list, and neither starts to process σ_i until the other is ready to do so.
- $$a_i = t_i + (\Phi_i - \Phi_{i-1})$$

  where t_i is the actual cost that *MTF* incurs for processing that request,

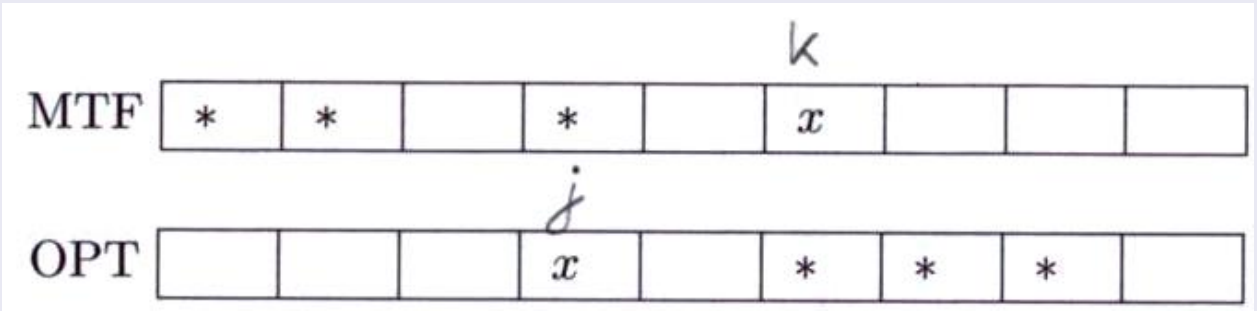
  Φ_i is a *potential function*, defined as the number of *inversions* in *MTF*'s list with respect to *OPT*'s list.
- a_i is called the *amortized cost*, and $\Phi_i - \Phi_{i-1}$ is a measure of the "distance" between *MTF*'s list and *OPT*'s list after processing σ_i.
- *An inversion*: an ordered pair (x_j, x_k), where x_j precedes x_k in *MTF*'s list, but x_k precedes x_j in *OPT*'s list.

Example

The list of *MTF*: x_1, x_2, x_3, the list of *OPT*: x_3, x_2, x_1,
inversions $(x_1, x_2), (x_1, x_3), (x_2, x_3)$, so $\Phi = 3$

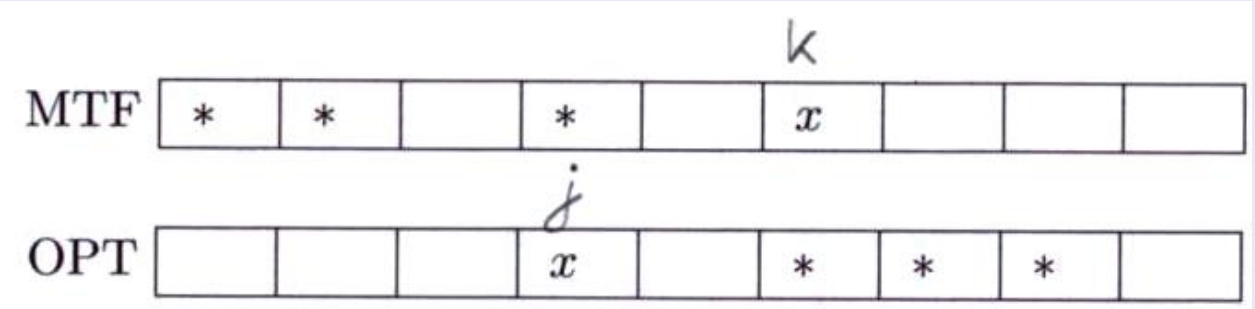
- Since the lists are initially identical: $\Phi_0 = 0$.

- $MTF(\sigma) = \sum_{i=1}^{n} t_i = \Phi_0 - \Phi_n + \sum_{i=1}^{n} a_i \leq \sum_{i=1}^{n} a_i.$
- We will show that $a_i = t_i + (\Phi_i - \Phi_{i-1}) \leq (2s_i - 1) + p_i - f_i$, where s_i is the search cost of OPT for accessing request σ_i, and p_i and f_i are the paid and free transportations, respectively, incurred by OPT when servicing σ_i
- Consider the case below (in this case $j < k$ but we do not make such assumption in general)



$*$ denotes items located before x in MTF but after x in OPT, i.e. *inversions* wrt x.

- Let MFT make a move, i.e. it moves x to the beginning of the list. Suppose that there are v inversions ($*$ in the picture).



- The number of elements before x in MTF is $k-1$, and of these $k-1$ elements v are $*$.
- Both list contains the same elements, from all non-$*$ before x in MTF, $k-1-v$ must be before x in OPT (if an element is before x in MTF and after x in OPT it is $*$).
- There are at least $k-1-v$ items that precede x in both lists.
- $k-1-v \leq j-1$ since x is in the j-th position in OPT, i.e. $k-v \leq j$.
- When MTF moves x to the front we have:
  1. $k-1-v$ new inversions are created with respect to OPT's list,
  2. v inversions are eliminated with respect to OPT's list.
- Hence, the contribution to a_i is:
  $k+((k-1-v)-v) = 2(k-v)-1 \leq 2j-1 = 2s-1$
  as for search s in OPT we have $s=j$.

- What about p_i and f_i?
- So far we have considered the moves of MTF, now we have to consider moves of OPT.
- For each paid transposition, the only change can come from the two transposed items as the relative order for the rest remain the same. Hence one paid transposition contributes 1, and servicing σ_i by OPT requires p_i paid transpositions.
- For each free transposition, MTF already put the transposed item x at the from of its list, and free transpositions can only move forward, so the number of item before x in OPT decreases by 1. Since servicing σ_i by OPT requires f_i free transpositions, we have to subtract f_i.
- So we have proved $a_i \leq (2s_i - 1) + p_i - f_i$.
- This shows that:

$$\sum_{i=1}^{n} a_i \leq \sum_{i=1}^{n}((2s_i - 1) + p_i - f_i) = 2(\sum_{i=1}^{n} s_i) + (\sum_{i=1}^{n} p_i) - (\sum_{i=1}^{n} f_i) - n =$$

$$2OPT_S(\sigma) + OPT_P(\sigma) - OPT_F(\sigma) - n$$

□

Dynamic List Accessing Model

- We also have *insertions*, where the cost of an insertion is $l + 1$, where l is the length of the list, and
- *deletions*, where the cost of a deletion is the same as the cost of an access, i.e. the position of the item on the list.

Theorem

Let OPT be an optimal offline algorithm for the dynamic list accessing model. Suppose that OPT and MTF both start with the same list configuration. Then, for any sequence of requests σ, where $|\sigma| = n$, we have:

$$MTF(\sigma) \leq 2OPT_S(\sigma) + OPT_P(\sigma) - OPT_F(\sigma) - n,$$

where $OPT_S(\sigma)$, $OPT_P(\sigma)$,$OPT_F(\sigma)$ are the total cost of searches, *the total cost of* paid transpositions *and the total cost of* free transpositions, *of OPT on σ, respectively.*

Competitiveness/Approximation

- **Competitive Analysis**: the payoff of an **online algorithm** is measured by comparing its performance to that of an **optimal offline algorithm**.
- We say that an online algorithm ALG is *c-competitive* if there is a constant α such that for all finite input sequences σ:

$$ALG(\sigma) \leq c \cdot OPT(\sigma) + \alpha$$

- The *infimum* of a subset $S \subseteq Reals$ is the largest element r, not necessarily in S, such that for all $s \in S$, $r \leq s$. For example 0 is infimum of $S = (0, 1)$.
- The *competitive ratio* of ALG, denoted $\mathcal{R}(ALG)$, is the infimum over all values c such that ALG is c-competitive.

Upper Bound for *OPT* for List Accessing Model

Proposition

For every sequence of requests σ, $OPT(\sigma) \leq n \cdot l$, where l is the length of the list and n is $|\sigma|$

Proof.

OPT is the optimal offline algorithm, and hence it must do as well as any algorithm *ALG*. Suppose we service all requests one-by-one in the naive way, without making any rearrangements. The cost of this scheme is bounded by $n \cdot l$, so $OPT(\sigma) \leq n \cdot l$. □

Competitiveness of *MTF*

Proposition

MTF is a 2-competitive algorithm.

Proof.

$MTF(\sigma) \leq 2 \cdot OPT_S(\sigma) + OPT_P(\sigma) - OPT_F(\sigma) - n$
$\leq 2 \cdot OPT_S(\sigma) + OPT_P(\sigma) \leq 2 \cdot (OPT_S(\sigma) + OPT_P(\sigma)) =$
$2 \cdot OPT(\sigma)$. □

Competitive Ratio of *MTF*

Proposition

$\mathcal{R}(MTF) \leq 2 - \frac{1}{l}$.

Proof.

$MTF(\sigma) \leq 2 \cdot OPT_S(\sigma) + OPT_P(\sigma) - OPT_F(\sigma) - n$
$\leq 2 \cdot OPT_S(\sigma) + OPT_P(\sigma) - n \leq$
$2 \cdot (OPT_S(\sigma) + OPT_P(\sigma)) - n = 2 \cdot OPT(\sigma) - n.$
On the other hand, $OPT(\sigma) \leq n \cdot l$, i.e. $n \geq \frac{OPT(\sigma)}{l}$. Hence
$MTF(\sigma) \leq 2 \cdot OPT(\sigma) - n \leq 2 \cdot OPT(\sigma) - \frac{OPT(\sigma)}{l} =$
$(2 - \frac{1}{l}) \cdot OPT(\sigma)$. □

Paging

- Consider a two-level *virtual memory system*: each level, slow and fast, can store a number of fixed-size memory units called *pages*.
- The slow memory stores N pages, and the fast memory stores k pages, where $k < N$. The k is usually much smaller than N.
- Given a request for page p_i, the system must make page p_i available in the fast memory.
- If p_i is already in the fast memory, called a *hit*, the system need not do anything.
- Otherwise, on a *miss*, the system incurs a *page fault*, and must copy the page p_i from the slow memory to the fast memory.
- **Problem**: which page to evict from the fast memory to make space for p_i.
- In order to *minimize* the number of page faults, the choice of which page to evict must be made wisely.

Paging

- Typical examples of fast and slow memory pair are a RAM and hard disk, respectively, or a processor-cache and RAM, respectively.
- In general, we shall refer to the fast memory as "the cache".
- Paging disciplines:

LRU	*Least Recently Used*
CLOCK	*Clock Replacement*
FIFO	*First-In/First-Out*
LIFO	*Last-In/First-Out*
LFU	*Least Frequently Used*
LFD	*Longest Forward Distance*

The top five are online algorithms; the last one, LFD, is an offline algorithm. In Lecture Notes 3, pages 33-39, we have shown that LFD is the optimal offline algorithm for paging.

Demand paging

- **Demand** paging algorithms never evict a page from the cache unless there is a page fault.
- All the paging disciplines from page 20 are demand paging.
- **Simple Page Fault Model**: we charge 1 for bringing a page into the fast memory, and we charge 0 for accessing a page which is already there - in practice there are other costs. involved.

Theorem

Any page replacement algorithm, online or offline, can be modified to be demand paging without increasing the overall cost on any request sequence.

Proof.

- In a demand paging algorithm a page fault causes exactly one eviction (once the cache is full, that is), and there are no evictions between misses.
- So let ALG be any paging algorithm.
- We show how to modify it to make it a demand paging algorithm ALG', in such a way that on any input sequence ALG' incurs at most the cost (makes at most as many page moves from slow to fast memory) as ALG, i.e., $\forall\sigma.ALG'(\sigma) \leq ALG(\sigma)$.
- Suppose that ALG has a cache of size k.
- Define ALG' as follows: ALG' also has a cache of size k, plus k registers.

- ALG' runs a simulation of ALG, keeping in its k registers the page numbers of the pages that ALG would have had in its cache.
- Based on the behavior of ALG, ALG' makes decisions to evict pages. We assume that ALG does not re-arrange its slots - i.e., it never permutes the contents of its cache.
- Suppose page p is requested. If p is in the cache of ALG', then just service the request.
- Otherwise, if a page fault occurs, ALG' behaves according to the following two cases:

Case 1. If ALG also has a page fault (that is, the number of p is *not* in the registers), and ALG evicts a page from register i to make room for p, then ALG' evicts a page from slot i in its cache, to make room for p.

Case 2. If ALG does not have a page fault, then the number of p must be in, say, register i. In that case, ALG' evicts the contents of slot i in its cache, and moves p in there.

- Thus ALG' is a demand paging algorithm.
- It can be shown that $\forall\sigma.ALG'(\sigma) \leq ALG(\sigma)$ (see pages 108-110 of Soltys' book)

Demand paging

- The last theorem allows for us to restrict our attention to demand paging algorithms, and thus use the terms “page faults” and “page moves” interchangeably, in the sense that in the context of demand paging, we have a page move if and only if we have a page fault.

FIFO Algorithm

- When a page must be replaced, the *oldest* page is chosen.
- It is not necessary to record the time when a page was brought in; all we need to do is create a FIFO (First-In/First-Out) queue to hold all pages in memory.
- The FIFO algorithm is easy to understand and program, but its performance is not good in general.
- FIFO also suffers from the so called *Belady's anomaly*.
- Suppose that we have the following sequence of page requests: $1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5$.
- Then, we have more page faults when $k = 4$ than when $k = 3$.
- That is, FIFO has more page faults with a bigger cache!

Least Recently Used (LRU) Algorithm

- The optimal offline algorithm, LFD (Longest Forward Distance), evict the page whose next request is the latest, and if some pages are never requested again, then anyone of them is evicted.
- This is an impractical algorithm from the point of view of online algorithms as we do not know the future.
- If we use the recent past as an approximation of the near future, then we will replace the page that *has not been used for the longest period of time.*
- This approach is the **Least Recently Used (LRU) algorithm**.

Least Recently Used (LRU) Algorithm

- LRU replacement associates with each page the time of that page's last use.
- When a page must be replaced, LRU chooses that page that has not been used for the longest period of time.
- The LRU algorithm is considered to be good, and is often implemented—the major problem is *how* to implement it.
- Two typical solutions are counters and stacks.

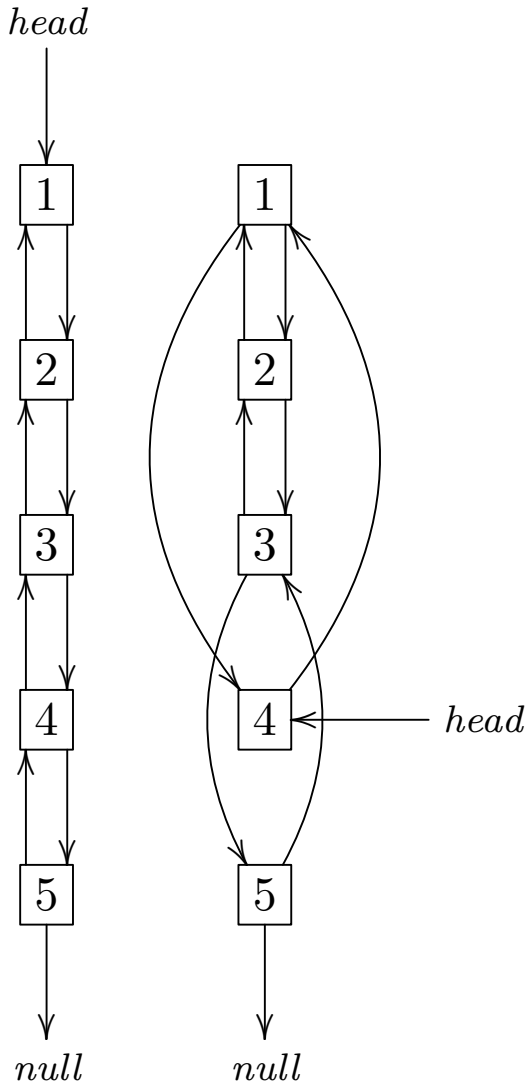
Counters: Keep track of the time when a given page was last referenced, updating the counter every time we request it.

- This scheme requires a search of the page table to find the LRU page, and a write to memory for each request.
- An obvious problem might be clock overflow.

Stack: Keep a stack of page numbers.

- Whenever a page is referenced, it is removed from the stack and put on the top.
- In this way, the top of the stack is always the most recently used page, and the bottom is the LRU page.
- Because entries are removed from the middle of the stack, it is best implemented by a doubly-linked list.

- LRU stack implementation with a doubly-linked list. The requested page is page 4; the left list shows the state *before* page 4 is requested, and the right list shows the state *after* the request has been serviced.



Least Recently Used (LRU) Algorithm

Lemma

LRU does not incur Belady's anomaly (on any cache size and any request sequence).

Proof.

- Let $\sigma = p_1, p_2, \ldots, p_3$ be a request sequence, and let $LRU_i(\sigma)$ be the number of faults that LRU incurs on σ with a cache of size i.
- It can be shown that $LRU_i(\sigma) \geq LRU_{i+1}(\sigma)$.
- From the above it follows that for any pair $i < j$ and any request sequence σ, $LRU_i(\sigma) \geq LRU_j(\sigma)$.
- LRU does not incur Belady's anomaly.

Marking algorithms

- Consider a cache of size k and a request sequence σ.
- We divide the request sequence into phases as follows: phase 0 is the empty sequence.
- For every $i \geq 1$, phase i is the maximal sequence following phase $i-1$ that contains at most k distinct page requests; that is, if it exists, phase $i+1$ begins on the request that constitutes the $k+1$ distinct page request since the start of the i-th phase.
- Such a partition is called a **k-phase partition**.
- This partition is independent of any particular algorithm processing σ.
- Examples of 3-phase and 2-phase partitions:

$$\underbrace{1,2,1,2,1,2,3}_{\text{3-phase \#1}},\underbrace{4,5,6,6,6,6,6,6,6,4,5,4}_{\text{3-phase \#2}},\underbrace{7,7,7,7,1,2}_{\text{3-phase \#3}}.$$

$$\sigma = \underbrace{1,1,3,1}_{\text{2-phase \#1}},\underbrace{5,1,5,1,5,1}_{\text{2-phase \#2}},\underbrace{3,4,4,4}_{\text{2-phase \#3}},\underbrace{2,2,2,2}_{\text{2-phase \#4}}.$$

Marking algorithms

- Let σ be any request sequence and consider its k-phase partition. Associate with each page a bit called the *mark*.
- The marking is done for the sake of analysis (this is not implemented by the algorithm, but "by us" to keep track of the doings of the algorithm).
- For each page, when its mark bit is set we say that the page is *marked*, and otherwise, *unmarked*.
- Suppose that at the beginning of each k-phase we unmark all the pages, and we mark a page when it is first requested during the k-phase.
- A *marking algorithm* never evicts a marked page from its fast memory.

Marking algorithms

- Consider the case:

$$\sigma = \underbrace{1,1,3,1}_{\text{2-phase \#1}}, \underbrace{5,1,5,1,5,1}_{\text{2-phase \#2}}, \underbrace{3,4,4,4}_{\text{2-phase \#3}}, \underbrace{2,2,2,2}_{\text{2-phase \#4}}.$$

- Marking for the case above:

step	1	2	3	4	5	step	1	2	3	4	5
1	x					10	x				x
2	x					11			x		
3	x		x			12			x	x	
4	x		x			13			x	x	
5					x	14			x	x	
6	x				x	15		x			
7	x				x	16		x			
8	x				x	17		x			
9	x				x	18		x			

Theorem

LRU is a marking algorithm.

Proof.

- Suppose that LRU on a cache of size k is not a marking algorithm.
- Let σ be a request sequence where there exists a k-phase partition, during which some marked page p is evicted.
- Consider the first request for p during this k-phase:

$$\sigma = p_1, p_2, p_3, \ldots, \ldots, \underbrace{\ldots, p, \ldots, \ldots, \ldots}_{k\text{-phase}}, \ldots, \ldots$$

- Immediately after p is serviced, it is marked as *most recently used* page in the cache (i.e., it is put at the top of the doubly-linked list).
- In order for p to leave the cache, LRU must incur a page fault while p is the least recently used page.
- It follows that during the k-phase in question, $k+1$ distinct pages were requested: there are the $k-1$ pages that pushed p to the end of the list, there is p, and the page that got p evicted.
- Contradiction; a k-phase has at most k distinct pages. □

Flush When Full (FWF) Algoritm

- **Flush When Full (FWF)**: whenever there is a page fault and there is no space left in the cache, evict all pages currently in the cache—call this action a "flush".
- Each slot in the cache has a single bit associated with it. At the beginning, all these bits are set to zero.
- When a page p is requested, FWF checks only the slots with a marked bit.
- If p is found, it is serviced.
- If p is not found, then it has to be brought in from the slow memory (even if it actually is in the cache, in an unmarked slot).
- FWF looks for a slot with a zero bit, and one of the following happens:
  1. a slot with a zero bit (an unmarked page) is found, in which case FWF replaces that page with p.
  2. a slot with a zero bit is not found (all pages are marked), in which case FWF unmarks all the slots, and replaces any page with p, and it marks p's bit.

FWF and FIFO

Proposition

1. *FWF is a marking algorithm.*
2. *FIFO is not a marking algorithm.*

Proof.

1. FWF really implements the marking bit, so it is almost a marking algorithm by definition.
2. FIFO is not a marking algorithm because with $k = 3$, and the request sequence $1, 2, 3, 4, 2, 1$ it will evict 2 in the second phase even though it is marked.

Longest Forward Distance (LFD)

Theorem

LFD is the optimal (offline) page replacement algorithm, i.e., OPT = LFD.

Proof.

see Lecture Notes 8, pages 15-21. □

- LFD evicts the page that will not be used for the longest period of time, and as such, it cannot be implemented online because it requires knowledge of the future.
- However, it is very useful for comparison studies, i.e., for example: competitive analysis.

Theorem

Any marking algorithm ALG is $\left(\frac{k}{k-h+1}\right)$*-competitive, where* k *is the size of its cache, and* h *is the size of the cache of OPT.*