Open access • Journal Article • DOI:10.1007/S00224-012-9427-Y

OnlineMin: A Fast Strongly Competitive Randomized Paging Algorithm

— Source link

Gerth Stølting Brodal, Gabriel Moruz, Andrei Negoescu

Institutions: National Research Foundation of South Africa, Goethe University Frankfurt

Published on: 01 Jan 2015 - Theory of Computing Systems \/ Mathematical Systems Theory (Springer US)

Topics: Page replacement algorithm, Competitive analysis, Paging, Online algorithm and Randomized algorithm

OnlineMin: A Fast Strongly Competitive Randomized Paging Algorithm

Gerth Stølting Brodal[1], Gabriel Moruz[2,⋆], and Andrei Negoescu[2]

[1] MADALGO[⋆⋆], Department of Computer Science, Aarhus University. Åbogade 34, 8200 Aarhus N, Denmark. Email: `gerth@cs.au.dk`.

[2] Goethe University Frankfurt am Main. Robert-Mayer-Str. 11-15, 60325 Frankfurt am Main, Germany. Email: {`gabi,negoescu`}`@cs.uni-frankfurt.de`.

Abstract. In the field of online algorithms paging is one of the most studied problems. For randomized paging algorithms a tight bound of H_k on the competitive ratio has been known for decades, yet existing algorithms matching this bound have high running times. We present the first randomized paging approach that both has optimal competitiveness and selects victim pages in subquadratic time. In fact, if k pages fit in internal memory the best previous solution required $O(k^2)$ time per request and $O(k)$ space, whereas our approach takes also $O(k)$ space, but only $O(\log k)$ time in the worst case per page request.

1 Introduction

Online algorithms are algorithms for which the input is not provided beforehand, but is instead revealed item by item. The input is to be processed sequentially, without assuming any knowledge of future requests. The performance of an online algorithm is usually measured by comparing its cost against the cost of an optimal offline algorithm, i.e. an algorithm that is provided all the input beforehand and processes it optimally. This measure, denoted *competitive ratio* [9, 12], states that an online algorithm A has competitive ratio c if its cost satisfies $cost(A) \leq c \cdot cost(OPT) + b$, where $cost(OPT)$ is the cost of an optimal offline algorithm and b is a constant. If A is a randomized algorithm, $cost(A)$ denotes the expected cost. In particular, an online algorithm is denoted *strongly competitive* if its competitive ratio is optimal. While the competitive ratio is a quality guarantee for the cost of the solution computed by an online algorithm, factors such as space complexity, running time, or simplicity are also important.

In this paper we study *paging algorithms*, a prominent and well studied example of online algorithms. We are provided with a two-level memory hierarchy, consisting of a cache and a disk, where the cache can hold up to k pages and the disk size is infinite. When a page is requested, if it is in the cache a *cache hit* occurs and the algorithm proceeds to the next page. Otherwise, a *cache miss*

⋆ Partially supported by the DFG grant ME 3250/1-2, and by MADALGO.

⋆⋆ Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation.

occurs and the algorithm has to load the page from the disk; if the cache was full, a page must be evicted to accommodate the new one. The cost is given by the number of cache misses performed.

Related work. Paging has been extensively studied over the last decades. In [4] an optimal offline algorithm, denoted MIN, was given. In [12] a lower bound of k on the competitive ratio for deterministic paging algorithms was shown. Several algorithms, such as LRU and FIFO, meet this bound and are thus strongly competitive. For randomized algorithms, Fiat et al. [7] proved a lower bound of H_k on the competitive ratio, where $H_k = \sum_{i=1}^{k} 1/i$ is the k-th harmonic number. They also gave an algorithm, named Mark, which is $(2H_k - 1)$-competitive. The first strongly competitive randomized algorithm being H_k-competitive was Partition [11]. For Partition, the memory requirement and runtime per request can reach $\Theta(n)$, where n is the number of page requests, and n can be far greater than k. Partition was characterized in [1] as counter-intuitive and difficult to understand. The natural question arises if there exist simpler and more efficient strongly competitive randomized algorithms. The Mark algorithm can be easily implemented using $O(k)$ memory and very fast running time ($O(1)$ dictionary operations) per request, but it is not strongly competitive. Furthermore, in [6] it was shown that no Mark-like algorithm can be better than $(2H_k - 1)$-competitive. The strongly competitive randomized algorithm Equitable [1] was a first breakthrough towards efficiency, improving the memory complexity to $O(k^2 \log k)$ and the running time to $O(k^2)$ per page request. In [3] a modification of Equitable, denoted Equitable2, improved the space complexity to $O(k)$. Both Equitable algorithms are based on a characterization in [10] in the context of work functions. The main idea is to define a probability distribution on the set of all possible configurations of the cache and ensure that the cache configuration obeys this distribution. For each request, it requires k probability computations, each taking $O(k)$ time. For a detailed view on paging algorithms, we refer the interested reader to the comprehensive surveys [2, 5, 8].

Our contributions. In this paper we propose a strongly competitive randomized paging algorithm, denoted OnlineMin, that handles each page request in $O(\log k)$ time in the worst case. This is a significant improvement over the fastest known algorithm, Equitable[3], which needs $O(k^2)$ time per request. The space requirements of our algorithm are $O(k)$, like Equitable2.

The main building block of our algorithm is a priority based incremental selection process starting from the same characterization of an optimal solution in [10] as the Equitable algorithms. The analysis of this process yields a simple cache update rule which is different from the one in [1, 3], but leads to the same probability distribution of the cache content. A straightforward implementation of our update rule requires $O(k)$ time per request. Additionally we design appropriate data structures that result in an implementation which processes a page request in $O(\log k)$ time in the worst case.

[3] Since no explicit implementation of Equitable2 is provided, due to their similarity we assume it to be the same as for Equitable.

2 Randomized Selection Process

In this section we first give some preliminary notions about *offset functions* for paging algorithms introduced in [10]. We then describe in Section 2.2 a new priority based selection process which is the basis of our algorithm OnlineMin. We analyze the selection process in order to obtain a simple page replacement rule which remains at all times consistent with the outcome of the selection process. Finally, in Section 2.3 we prove equivalences between the cache distribution of our selection process and the Equitable algorithms [1, 3], which implies that OnlineMin is H_k-competitive.

2.1 Preliminaries

Let σ be the request sequence so far. For the construction of a competitive paging algorithm it is of interest to know the possible cache configurations if σ has been processed with minimal cost. We call these configurations *valid*.

For fixed σ and an arbitrary cache configuration C (a set of k pages), the *offset function* ω assigns C the difference between the minimal cost of processing σ ending in configuration C and the minimal cost of processing σ. Thus C is a valid configuration iff $\omega(C) = 0$. Koutsoupias and Papadimitriou [10] showed that ω can be represented by a sequence of $k + 1$ disjoint page sets, denoted layers, and proved the following[4].

Lemma 1. *If $(L_0, \ldots, L_k)$ is a layer representation of ω, then a set C of k pages is a valid configuration, i.e. $\omega(C) = 0$, iff $|C \cap (\cup_{i \le j} L_i)| \le j$ for all $0 \le j \le k$.*

The layer representation is defined as follows. Initially each layer L_i, where $i > 0$, consists of one of the first requested k pairwise distinct pages. The layer L_0 contains all pages not in $L_1, \ldots, L_k$. We denote by ω^p the offset function which results from ω by requesting p. We have the following update rule.

$$\omega^p = \begin{cases} (L_0 \setminus \{p\}, L_1, \ldots, L_{k-2}, L_{k-1} \cup L_k, \{p\}), & \text{if } p \in L_0 \\ (L_0, \ldots, L_{i-2}, L_{i-1} \cup L_i \setminus \{p\}, L_{i+1}, \ldots, L_k, \{p\}), & \text{if } p \in L_i, i > 0 \end{cases}$$

We give an example of an offset function for $k = 3$ in Figure 1. The support of ω is defined as $S(\omega) = L_1 \cup \cdots \cup L_k$. In the remainder of the paper, we call a set with a single element *singleton*. Also, let i be the smallest index such that $L_i, \ldots, L_k$ are singletons. We distinguish the set of *revealed* pages $R(\omega) = L_i \cup \cdots \cup L_k$, and the set of *non-revealed* pages $N(\omega) = L_1 \cup \cdots \cup L_{i-1}$. A valid configuration contains all revealed pages and no page from L_0. Note that when requesting some non-revealed page p in the support, we have $R(\omega^p) = R(\omega) \cup \{p\}$ and the number of layers containing non-revealed items decreases by one. Moreover, if $p \notin L_1$ then $N(\omega^p) = N(\omega) \setminus \{p\}$ and otherwise $N(\omega^p) = N(\omega) \setminus L_1$. Also, the layer representation is not unique and especially each permutation of the layers containing revealed items describe the same offset function.

[4] We use a slightly modified, yet equivalent, version of the layer representation in [10].

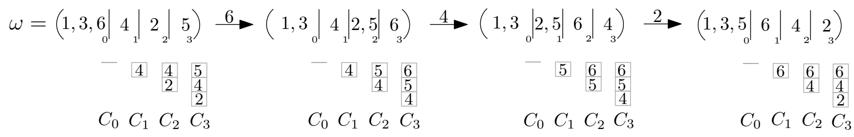

Fig. 1. The update of ω and the selection sets. The initial cache configuration is $\{4, 2, 5\}$ for $k = 3$ and request the pages $6, 4, 2$. The priority of a page is its number.

Equitable *and* Equitable2*.* Based on the layer partition above both Equitable algorithms are described using a probability distribution over all configurations where the probability that C is the cache content is defined as the probability of being obtained at the end of the following random process. Starting with $C = R(\omega)$ a page p is selected uniformly at random from $N(\omega)$, p is added to C, and ω is set to ω^p. This process is iterated until C has k pages. The probability for each configuration reachable by one page replacement is computed from its actual configuration such that the distribution remains consistent with the random process. The request is handled according to the computed probabilities.

2.2 Selection process for OnlineMin

If ω is the offset function for the input requested so far an online algorithm should have a configuration similar to the cache C_{OPT} of an optimal strategy. We know that C_{OPT} contains all revealed items and no item from L_0. Which non-revealed items are in the cache depends on future requests. To guess the order of future requests of non-revealed items OnlineMin assigns priorities to pages when they are requested. It maintains the cache content of an optimal offline algorithm under the assumption that the priorities reflect the order of future requests. We introduce a priority based selection process for the layer representation of ω. Assuming that each order of priorities has equal probability, we prove that the outcome of the selection process has the same probability distribution as the Equitable algorithms. Our approach allows an efficient and easy-to-implement update method for the cache of OnlineMin, which is consistent with our selection process.

In the following we assume that pages from $L_1, \ldots, L_k$ have pairwise distinct priorities. For some set S we denote by $\min_j(S)$ and $\max_j(S)$ the subset of S of size j having the smallest and largest priorities respectively. Furthermore, $\min(S) = \min_1(S)$ and $\max(S) = \max_1(S)$.

Definition 1. *We construct iteratively* $k+1$ selection sets $C_0(\omega), \ldots, C_k(\omega)$ *from the layer partition* $\omega = (L_0, \ldots, L_k)$ *as follows. We first set* $C_0(\omega) = \emptyset$ *and then for* $j = 1, \ldots, k$ *we set* $C_j(\omega) = \max_j(C_{j-1}(\omega) \cup L_j)$.

When ω is clear from the context, we let $C_i = C_i(\omega)$. For a page request p and offset function $\omega = (L_0, \ldots, L_k)$, denote $\omega^p = (L'_0, \ldots, L'_k)$ and let C'_k be the result of the selection process on ω^p. By the layer update rule each layer contains at least one element and the following result follows immediately.

Fact 1 *$|C_j| = j$ for all $j \in \{0, \dots, k\}$. If $|L_j|$ is singleton then $C_j = C_{j-1} \cup L_j$. Moreover, all revealed pages are in C_k.*

Updating C_k. We analyze how C_k changes upon a request. First we give an auxiliary result in Lemma 2 and then show in Theorem 1 that C'_k can be obtained from C_k by at most one page replacement. We get how C'_k can be directly constructed from C_k and the layers, without executing the whole selection process.

Lemma 2. *Let p be the requested page from layer L_i, where $0 < i < k$. If for some j, with $i \le j < k$ we have $q \in C_j$ and $C'_{j-1} = C_j \setminus \{q\}$, then we get:*

$$C'_j = \begin{cases} C_{j+1} \setminus \{q\}, & \text{if } q \in C_{j+1} \\ C_{j+1} \setminus \min\{C_{j+1}\}, & \text{otherwise} \end{cases}$$

Proof. We have:

$$C'_j = \max_j (L'_j \cup C'_{j-1}) = \max_j (L_{j+1} \cup C_j \setminus \{q\}) = C_{j+1} \setminus \{q\} \text{ (case: } q \in C_{j+1})$$

$$C'_j = \max_j (L'_j \cup C'_{j-1}) = \max_j (L_{j+1} \cup C_j \setminus \{q\}) = \max_j (C_{j+1}) \text{ (case: } q \notin C_{j+1})$$

In both cases, we first use the assumption $C'_{j-1} = C_j \setminus \{q\}$ and the partition update rule, $L'_j = L_{j+1}$. In the case $q \in C_{j+1}$ we use $C_{j+1} = \max_{j+1} (L_{j+1} \cup C_j) = \max_j (L_{j+1} \cup C_j \setminus \{q\}) \cup \{q\}$, which holds as $q \in C_j$ implies $q \notin L_{j+1}$. If $q \notin C_{j+1}$, we use $C_{j+1} = \max_{j+1} (L_{j+1} \cup C_j) = \max_{j+1} (L_{j+1} \cup C_j \setminus \{q\})$. We have $q \in C_j$, $q \notin C_{j+1}$ and $|C_{j+1}| = j + 1$, which leads to $C'_j = \max_j (C_{j+1}) = C_{j+1} \setminus \min\{C_{j+1}\}$. □

Theorem 1. *Let p be the requested page. Given C_k, we obtain C'_k as follows:*

1. *$p \in C_k$: $C'_k = C_k$*
2. *$p \notin C_k$ and $p \in L_0$: $C'_k = C_k \setminus \min(C_k) \cup \{p\}$*
3. *$p \notin C_k$ and $p \in L_i$, $i > 0$: $C'_k = C_k \setminus \min(C_j) \cup \{p\}$, and $j \ge i$ is the smallest index with $|C_j \cap C_k| = j$.*

Before the proof, note that for the third case $|C_j \cap C_k| = j$ is equivalent to $|(L_1 \cup \dots \cup L_j) \cap C_k| = j$ since C_j has elements only in $L_1 \cup \dots \cup L_j$ and $C_j \subseteq C_k$.

Proof. First assume that $p \in L_0$. In this case, by construction p is not in C_k. The only layers that change are L_{k-1} and L_k: $L'_{k-1} = L_{k-1} \cup L_k$ and $L'_k = \{p\}$. Applying the definition of C'_k and the fact that $C_k = \max_{k-1} (C_{k-2} \cup L_{k-1}) \cup L_k$, since L_k is singleton, we get:

$$C'_k = C'_{k-1} \cup \{p\} = \max_{k-1} (C_{k-2} \cup L_{k-1} \cup L_k) \cup \{p\} = C_k \setminus \min (C_k) \cup \{p\};$$

Now we consider the case when $p \in L_i$. We distinguish two cases: $p \in C_k$ and $p \notin C_k$. If $p \in C_k$, we have by construction that p is in all sets $C_i, \dots, C_k$ and we get $C_i = \max_i (C_{i-1} \cup L_i) = \max_{i-1} (C_{i-1} \cup L_i \setminus \{p\}) \cup \{p\}$. Based on this observation we show that $C'_{i-1} = C_i \setminus \{p\}$. It obviously holds for $i = 1$ since C'_0 is empty. For $i > 1$ we get:

$$C'_{i-1} = \max_{i-1} (C_{i-2} \cup L_{i-1} \cup L_i \setminus \{p\}) = \max_{i-1} (C_{i-1} \cup L_i \setminus \{p\}) = C_i \setminus \{p\} \ .$$

Using $C'_{i-1} = C_i \setminus \{p\}$ and $p \in C_i$, applying Lemma 2 we get $C'_i = C_{i+1} \setminus \{p\}$. Furthermore, using that p is in all sets $C_{i+1}, \ldots, C_k$, we apply Lemma 2 for all these sets which leads to $C'_{k-1} = C_k \setminus \{p\}$ and we obtain $C'_k = C'_{k-1} \cup \{p\} = C_k$.

Now we assume that $p \notin C_k$. This implies that p is a non-revealed page. First we analyze the structure of C'_{i-1} which will serve as starting point for applying Lemma 2. If $p \in C_i$ we argued before that $C'_{i-1} = C_i \setminus \{p\}$. Otherwise, we show that $C'_{i-1} = C_i \setminus \min(C_i)$. It holds for $i = 1$ since C_0 is always empty and by Fact 1 we have $|C_1| = 1$. For $i > 1$ we get:

$$C'_{i-1} = \max_{i-1} (C_{i-2} \cup L_{i-1} \cup L_i \setminus \{p\}) = \max_{i-1} (C_{i-1} \cup L_i \setminus \{p\}) = C_i \setminus \min(C_i) \ .$$

Let $j \geq i$ be the smallest index such that $|C_j \cap C_k| = j$. By construction, we have $C_j \subseteq C_k$. Applying Lemma 2 for sets $C'_{i-1}, \ldots, C'_{j-1}$ we get $C'_{j-1} = C_j \setminus \{s\}$, where $s \in C_j$ and either $s = p$, $s = \min C_j$, or s is a page with minimal priority from a set C_l, with $i \leq l < j$. Note that page s is also in C_k by the definition of C_j and thus $s = p$ can be excluded since p is not in C_k. If s is a page with minimal priority from some set C_l then all the other pages in C_l are also in C_j and thus in C_k because all of them have higher priorities than s. This leads to $C_l \subset C_k$ which contradicts the minimality of j. Thus we have $s = \min C_j$. Since the page $s = \min(C_j)$ is in all sets $C_j, \ldots, C_k$ by Lemma 2 we get $C'_{k-1} = C_k \setminus \min(C_j)$ and it follows $C'_k = C_k \setminus \min(C_j) \cup \{p\}$. ☐

2.3 Probability distribution of C_k

Theorem 2. *Assume that non-revealed pages are assigned priorities such that the order of the priorities is distributed uniformly at random. For any offset function ω, the distribution of C_k over all possible cache configurations is the same as the distribution of the cache configurations for the* Equitable *algorithms.*

Proof. Let u be the index of the last non-revealed layer, more precisely $|L_u| > 1$ and $|L_i| = 1$ for all $i > u$. The set of non-revealed items is $N(\omega) = L_1 \cup \cdots \cup L_u$ and the singletons $L_{u+1}, \ldots, L_k$ contain the revealed items $R(\omega)$.

The following selection process is used by both Equitable and Equitable2 to obtain the probability distribution of the cache M. Initially M contains all $k-u$ revealed items $R(\omega)$. Then u elements $x_1, \ldots, x_u$ are added to M, where x_i is chosen uniformly at random from the set of non-revealed items of $\omega^{x_1,\ldots,x_{i-1}}$, the offset function obtained from ω after requesting the sequence $x_1, \ldots, x_{i-1}$.

We define an auxiliary selection $C^*_k(\omega)$ which is a priority based version of Equitable's random process and then prove for every fixed priority assignment that $C_k(\omega) = C^*_k(\omega)$ holds.

Assume that pages in $N(\omega)$ have pairwise distinct priorities, with a uniformly distributed priority order. Initialize $C^*_k(\omega)$ to $R(\omega)$ and add elements $x^*_1, \ldots, x^*_u$

to $C^*_k(\omega)$, where x^*_i is the page with maximal priority from the non-revealed items of $\omega^{x^*_1,\ldots,x^*_{i-1}}$. Obviously all pages from $N(\omega)$ have the same probability to posses the maximal priority and thus x^*_1 and x_1 have the same distribution. Since x^*_1 is a revealed item in $\omega^{x^*_1}$, the priority order of pages in $N(\omega^{x^*_1})$ remains uniformly distributed. This implies inductively that $C^*_k(\omega)$ has the same distribution as EQUITABLE. Note that by the definition of C^*_k we have $C^*_k(\omega) = C^*_k(\omega^{x^*_1})$ because x^*_1 becomes a revealed item in $\omega^{x^*_1}$.

Now we prove for each fixed priority assignment that $C_k(\omega) = C^*_k(\omega)$ by induction on u. For $u = 0$ both C^*_k and C_k contain all k revealed items. For $u \geq 1$, let x^*_1 be the non-revealed page with the largest priority in ω. For the auxiliary process, we have already shown that $C^*_k(\omega) = C^*_k(\omega^{x^*_1})$. Also, the index u for $\omega^{x^*_1}$ is smaller by one than for ω, which by inductive hypothesis leads to $C^*_k(\omega) = C^*_k(\omega^{x^*_1}) = C_k(\omega^{x^*_1})$. It remains to prove that $C_k(\omega^{x^*_1}) = C_k(\omega)$. By the definition of the selection process for $C_1, \ldots, C_k$ we have $C_k(\omega) = C_u(\omega) \cup R(\omega)$. Page x^*_1 has the highest priority from $N(\omega) = L_1 \cup \cdots \cup L_u$ and thus it is a member of $C_u(\omega)$ and hence also in $C_k(\omega)$. Applying the update rule from Theorem 1 we get $C_k(\omega) = C_k(\omega^{x^*_1})$, and this concludes the proof. □

3 Algorithm OnlineMin

3.1 Algorithm

ONLINEMIN initially holds in its cache M the first k pairwise distinct pages. Note that the last requests for all pages in L_i are smaller than the last requests for all pages in L_{i+1}.

Page replacement. The algorithm maintains as invariant that $M = C_k$ after each request. To do so, it keeps track of the layer partition $\omega = (L_0, \ldots, L_k)$, where it suffices to store only the support layers $(L_1, \ldots, L_k)$. The cache update is performed according to Theorem 1. More precisely, if the requested page p is in the cache, M remains unchanged. If a cache miss occurs and p is from L_0 the page with minimal priority from M is replaced by p. If p is from L_i with $i > 0$, and $p \notin M$ we first identify the set C_j in Theorem 1 satisfying $|C_j \cap M| = j$. This can be done as follows. Let $p_1, \ldots, p_k$ be the pages in M sorted in increasing order by their layer index. We search the minimal index $j \geq i$, such that the layer index of p_j is j, i.e. $p_j \in L_j$. We evict the page with minimal priority from $p_1, \ldots, p_j$. The layers are updated after the cache update is done.

Forgiveness. If the amount of pages in $(L_1, \ldots, L_k)$ is $3k$ and a page in L_0 is requested we apply the *forgiveness mechanism* in [3]. More precisely, we perform the partition and cache update as if the requested page was from L_1. Doing this all pages in L_1 are moved to L_0, i.e. they are removed from the support, and the support size never exceeds $3k$.

Priorities. If page p is requested from L_0, we select for p a rank within the support chosen uniformly at random, i.e. a number in $\{0, \ldots, |S_w|\}$, and we assign it a priority such that it reflects its rank.

Time and space complexity. Storing the layer partition together with the page priorities needs $O(k)$ space by applying the forgiveness mechanism. A naive implementation storing the layers in an array processes a page request in $O(k)$ time. In the remainder of the paper we show how to improve this complexity to $O(\log k)$ time per request in the worst case.

Competitive ratio. We showed in Theorem 2 that the probability distribution over the cache configurations for OnlineMin and Equitable2 are the same. This holds also when using the forgiveness step, and thus the two algorithms have the same expected cost. This leads to the result in Lemma 3.

Lemma 3. OnlineMin *is* H_k*-competitive.*

3.2 Algorithm Implementation

We show how to implement OnlineMin efficiently, such that a page request is processed in $O(\log k)$ worst case time while using $O(k)$ space. In the following we represent each page in the support by the timestamp of its last request.

Basic structure. Consider a list $L = (l_1, \ldots, l_t)$, with $t \leq 4k$, where L has two types of elements: k *layer delimiters* and at most $3k$ *page elements.* Furthermore, we distinguish two types of page elements: *cache elements* which are the pages in the cache and *support elements* which are pages in the support but not in the cache. We store in L the layers $L_1, \ldots, L_k$ from left to right, separated by k layer delimiters. For each layer L_i we store its layer delimiter, followed by the pages in L_i. For each list element l_i, be it page element or layer delimiter, we store a timestamp t_i and a *v-value* v_i with $v_i \in \{-1, 0, 1\}$; for page elements we also store the priority. For some element l_i, if it is a layer delimiter for some layer L_j, we set $v_i = 1$ and t_i to the minimum of all page timestamps in L_j. If l_i is a page element, then t_i is set to the timestamp corresponding to the last request of the page; we set $v_i = -1$ for cache elements and $v_i = 0$ for support elements. Note that the layer delimiters always have t_i values matching the first page in their layer. As described before, layer delimiters always precede page elements. An example is given in Figure 2.

Note that the v-values have the property that $|C_k \cap (L_1 \cup \cdots \cup L_i)| = i$ iff the prefix sum of the v-values for the last element in L_i is zero. Furthermore, since $|C_k \cap (L_1 \cup \cdots \cup L_i)| \leq i$ the prefix sum cannot be negative. This property will be used when dealing with a cache miss caused by a page from L_i, with $i > 0$.

We show how to implement OnlineMin using the following operations on L:

- *find-layer(l_p).* For some page l_p, find its layer delimiter.
- *search-page(l_p).* Check whether l_p is a page in L.
- *insert(l_p), delete(l_p).* The item l_p is inserted (or deleted) in L.
- *find-min(l_p).* Find the cache element $l_q \in (l_1, \ldots, l_p)$ with minimum priority.
- *find-zero(l_p).* Find the smallest j, with $p \leq j$ such that $\sum_{l=1}^{j} v_l = 0$, and return l_j.

v	1	0	−1	1	0	1	0	−1	−1	1	0	−1	1	−1	1	−1
t	2	2	4	5	5	8	8	10	11	13	13	15	18	18	21	21

Fig. 2. Example for list L: representing pages by timestamps of last requests, we have $L_1 = \{2, 4\}$, $L_2 = \{5\}$, $L_3 = \{8, 10, 11\}$, $L_4 = \{13, 15\}$, $L_5 = \{18\}$, and $L_6 = \{21\}$. Layer delimiters are emphasized and the memory is $M = \{4, 10, 11, 15, 18, 21\}$.

We describe how to update the list L upon a request for some page p. OnlineMin keeps in memory at all times the elements in L having the v-value equal to -1.

If $p \notin M$, we must identify a page to be evicted from M. To evict a page we set its v-value to zero and to load a page we set its v-value to -1. We first find the layer delimiter for p. We can have $p \in L_i$ with $0 < i \leq k$ or $p \in L_0$. If $p \in L_i$, the page to be evicted is the cache element in $L_1 \cup \cdots \cup L_j$ having the minimum priority, where $j \geq i$ is the minimal index satisfying $|M \cap (L_1 \cup \cdots \cup L_j)| = j$. This is done using `find-zero`(l_{L_i}), where l_{L_i} is the layer delimiter of L_i, and the page to be evicted is identified using `find-min` applied to the value returned by `find-zero`. If $p \in L_0$, if the forgiveness need not be applied, the page having the smallest priority in M is to be evicted. We identify this page in L using `find-min` on the last element in L. If we must apply forgiveness, we treat p as being a support page in L_1.

After updating the cache, we perform in L the layer updates as follows. If $p \in L_i$ with $i > 0$, the layers are updated as follows: $L_{i-1} = L_{i-1} \cup L_i \setminus \{p\}$, $L_j = L_{j+1}$ for all $j \in \{i, \ldots, k-1\}$, and $L_k = \{p\}$. We first delete the layer delimiter for L_i and the page element for p, which triggers not only the merge of L_{i-1} and $L_i \setminus \{p\}$, but also shifts all the remaining layers, i.e. $L_j = L_{j+1}$ for all $j \geq i$. If we deleted the layer delimiter for L_1, we also delete all pages in L_1 because in this case L_1 is merged with L_0. To create $L_k = \{p\}$, we simply insert at the end a new layer delimiter followed by p, both items having as timestamp the current timestamp.

If $p \in L_0$, we first check whether we must apply the forgiveness step, and if so we apply it by simulating the insertion of p in L_1 and then requesting it, as described above. If forgiveness need not be applied, we update the layers $L_{k-1} = L_{k-1} \cup L_k$ and $L_k = \{p\}$ as follows. We first delete the layer delimiter of L_k, which translates into merging L_{k-1} and L_k. Then, we insert a new layer delimiter having the timestamp of the current request, i.e. create L_k, and insert p with the same timestamp.

3.3 Data Structures

We implement all the operations previously introduced using two data structures: a *set structure* and a *page-set* structure. The set structure focuses only on the `find-layer` operation, and the page-set data structure deals with the remaining operations. While most operations can be implemented using standard data

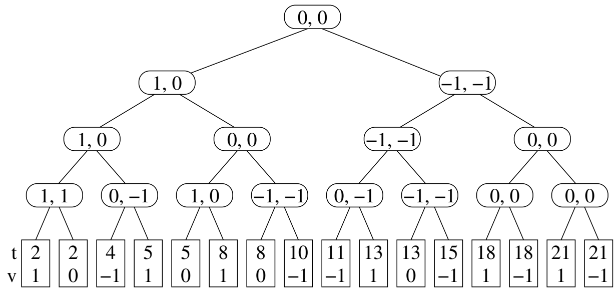

Fig. 3. The page-set data structure for $L_1 = \{2, 4\}$, $L_2 = \{5\}$, $L_3 = \{8, 10, 11\}$, $L_4 = \{13, 15\}$, $L_5 = \{18\}$, and $L_6 = \{21\}$, and the memory $M = \{4, 10, 11, 15, 18, 21\}$. For each internal node u we show the (s_u, m_u) values.

structures, i.e. balanced binary search trees, the key operation for the page-set structure is `find-zero`. That is because we need to find in sublinear time the first item to the right of an arbitrary given element having the prefix sum zero in the presence of updates, and the item that is to be returned can be as far as $\Theta(k)$ positions in L.

Set structure. The set structure is in charge only for the `find-layer` operation. To do so, it must also support updating the layers. It is a classical balanced binary search tree, e.g. an AVL tree, built on top of the layer delimiters in L having as keys the timestamps of the delimiters. Whenever a layer delimiter is inserted or deleted from L, the set structure is updated accordingly. Each operation takes $O(\log k)$ time in the worst case.

Page-set structure. The page-set structure contains all elements of L and supports all the remaining operations required on L. We store the elements of L, i.e. both page elements and layer delimiters, in the leaves of a regular leaf oriented balanced binary search tree indexed by the timestamps. For some node u, denote by $\mathcal{T}(u)$ the subtree rooted at u and by $\mathcal{L}(u)$ the leaves of $\mathcal{T}(u)$. For each node u we store the sum s_u of the v-values in $\mathcal{L}(u)$. We also store the minimum prefix sum value m_u among all the prefix sums restricted on the elements within $\mathcal{L}(u)$. More precisely, if $\mathcal{L}(u) = (p_1, \ldots, p_m)$, we have $m_u = \min_{l=1}^{m}(\sum_{j=1}^{l} p_j)$. Finally, in each node u we also store the minimum priority of a cache page in the subtree rooted at u. Note that if the subtree rooted at u has no cache elements the priority field is set to infinity.

Fact 2 *For each internal node u we have that $m_u = \min(m_{u_l}, s_{u_l} + m_{u_r})$, where u_l and u_r denote the left and right child of u respectively.*

Updates. We discuss how to perform insertions and deletions in the page-set structure. To insert an element, we first identify its location and then insert it.

It remains to update the information at the internal nodes, i.e. the sum of the v-values, the minimum prefix-sum values and the minimum priorities. The sums of the elements of the subtrees are easily updated in a bottom up traversal, together with the minimum priorities, even if rotations need to be done. The minimum prefix sum values can also be updated in a bottom up traversal using the observation stated in Fact 2. Deleting an element in the page-set structure is done analogously to insertion. We note however that when requesting a page in L_1 we must delete both the layer delimiter and all page elements in L_1 from the data structure which leads to $O(\log k)$ amortized time. We will show later how to improve this bound to $O(\log k)$ worst case time for deletions as well.

Queries. We turn to queries supported by the page-set structure, which are the queries required on L. The search-page operation is implemented using a standard search in a leaf-oriented binary search tree.

To find the page element having the minimum priority in $l_1, \ldots, l_p$, we first find the value of the priority as follows. On the path from l_p to the root, for each node u we consider the minimum priority value stored at its left child if the left child is not on the path. The priority to be returned is the smallest among these minimums. To find the page, we traverse the tree top-down and at each node we branch on the subtree matching the minimum priority value. Since it does a bottom-up and a top-down traversal, this operation takes $O(\log k)$ time.

It remains to deal with the `find-zero` operation, where we are given some leaf storing l_p and must return the first leaf to the right which has the prefix sum of the v-values zero. We note that the prefix sum cannot be negative, and thus it suffices to find the first leaf to the right having the minimum prefix sum. We do so in two steps: we first identify a subtree containing the leaf having the minimum prefix sum in bottom-up traversal and then we identify the leaf itself in a top-down traversal of this subtree. To identify the subtree containing the leaf to be returned, we traverse the path from the leaf storing l_p to the root while maintaining a sum s of the v-values of the right children not on this path, and at each node u we compute a score as follows. If the right child u_r of u is not on the path, the score of u is given by $s + m_{u_r}$ and afterwards we set $s = s + s_{u_r}$. The subtree we are looking for is the one having the minimum score; in case of several subtrees having an identical score, the leftmost one, i.e. the first one encountered on the path from the leaf to the root, is considered. To identify the leaf having the minimum prefix sum, we do a top-down traversal of the subtree previously computed and we use the observation stated in Fact 2 to decide which way to branch, i.e. we branch left if $m_{u_l} \leq s_{u_l} + m_{u_r}$ and we branch right otherwise. This operation requires a bottom-up and a top-down traversal of the tree and thus takes $O(\log k)$ time in the worst case.

Worst-case bounds. The only operation taking $\omega(\log k)$ time is page deletion, more precisely when a page in L_1 is requested all pages in L_1 are moved to L_0 and thus should be removed from the support. Instead of deleting the set delimiter and all the pages corresponding to L_1, we delete only the set delimiter. With the leading set delimiter removed, the list L no longer starts with a set delimiter, but with at most $O(k)$ elements having the v-value set to 0, since all of these pages belong to L_0 and thus cannot be cache elements. Also, these pages do not influence the prefix sums for the v-values. When we process a page, we simply start by checking if the leftmost element in the tree has a v-value of 0, and if so we delete it. Since each page requests adds at most one new element to the support, the space complexity is still $O(k)$. This way deletions can be done in $O(\log k)$ time in the worst case.

Each page request uses $O(1)$ operations in both data structures. In Theorem 3 we give the time and space complexities for OnlineMin.

Theorem 3. OnlineMin *uses $O(k)$ space and processes a request in $O(\log k)$ time in the worst case.*

Acknowledgements

We would like to thank previous anonymous reviewers for very insightful comments and suggestions. Also, we would like to thank Annamária Kovács for useful advice on improving the presentation of the paper.

References

1. D. Achlioptas, M. Chrobak, and J. Noga. Competitive analysis of randomized paging algorithms. *Theoretical Computer Science*, 234(1-2):203–218, 2000.
2. S. Albers. Online algorithms: a survey. *Mathematical Programming*, 97(1–2):3–26, 2003.
3. W. W. Bein, L. L. Larmore, J. Noga, and R. Reischuk. Knowledge state algorithms. *Algorithmica*, 60(3):653–678, 2011.
4. L. A. Belady. A study of replacement algorithms for virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.
5. A. Borodin and R. El-Yaniv. *Online computation and competitive anlysis*. Cambridge University Press, 1998.
6. M. Chrobak, E. Koutsoupias, and J. Noga. More on randomized on-line algorithms for caching. *Theoretical Computer Science*, 290(3):1997–2008, 2003.
7. A. Fiat, R. M. Karp, M. Luby, L. A. McGeoch, D. D. Sleator, and N. E. Young. Competitive paging algorithms. *Journal of Algorithms*, 12(4):685–699, 1991.
8. A. Fiat and G. J. Woeginger, editors. *Online Algorithms, The State of the Art (the book grow out of a Dagstuhl Seminar, June 1996)*, 1998.
9. A. R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator. Competitive snoopy caching. *Algorithmica*, 3:77–119, 1988.
10. E. Koutsoupias and C. H. Papadimitriou. Beyond competitive analysis. In *Proc. 35th Symposium on Foundations of Computer Science*, pages 394–400, 1994.
11. L. A. McGeoch and D. D. Sleator. A strongly competitive randomized paging algorithm. *Algorithmica*, 6(6):816–825, 1991.
12. D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.