OnlineMin: A Fast Strongly Competitive Randomized Paging Algorithm

Gerth Stølting Brodal[*,†] Gabriel Moruz[‡,§]
Andrei Negoescu[‡,¶]

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 a new randomized paging algorithm OnlineMin that has optimal competitiveness and allows fast implementations. In fact, if k pages fit in internal memory the best previous solution required $O(k^2)$ time per request and $O(k)$ space. We present two implementations of OnlineMin which use $O(k)$ space, but only $O(\log k)$ worst case time and $O(\log k/\log\log k)$ worst case time per page request respectively.

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* [15, 19], states that an online algorithm A has competitive ratio c if for any input sequence 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.

[*]MADALGO – Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation.

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

[‡]Institut für Informatik, Goethe-Universität Frankfurt am Main, Robert-Mayer-Str. 11-15, 60325 Frankfurt am Main, Germany. Email: {gabi,negoescu}@cs.uni-frankfurt.de.

[§]Partially supported by the DFG grants ME 3250/1-3 and MO 2057/1-1, and by MADALGO.

[¶]Partially supported by DFG grant ME 3250/1-3 and by MADALGO.

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* 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 [7] an optimal offline algorithm, denoted MIN, was given. In [19] 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. [10] 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 has a competitive ratio of $(2H_k - 1)$. The first strongly competitive randomized algorithm being H_k-competitive was PARTITION [18]. 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 has very fast running time ($O(1)$ dictionary operations) per request, but it is not strongly competitive. Furthermore, in [9] 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 [6] a modification of EQUITABLE, denoted EQUITABLE2[1], improved the space complexity to $O(k)$. Both EQUITABLE algorithms are based on a characterization in [16] 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, 8, 11].

Our contributions. In this paper we propose a strongly competitive randomized paging algorithm, denoted ONLINEMIN. We first propose an implementation for it which handles a page request in $O(\log k)$ worst case time, and then

[1] In [6] EQUITABLE2 is denoted A_k. Due to its similarity to EQUITABLE we use its original name of EQUITABLE2.

we improve this implementation to achieve $O(\log k / \log\log k)$ time in the worst case for processing a page request. This is a significant improvement over the fastest known strongly competitive algorithm, EQUITABLE, which needs $O(k^2)$ time per request[2]. The space requirement of both our implementations is $O(k)$, due to the forgiveness technique used in 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 [16] as the EQUITABLE algorithms. The analysis of this process yields a simple cache update rule which is different from the one in [1, 6], 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 two more efficient implementations: the first implementation uses simple pointer-based data structures to achieve $O(\log k)$ worst case time per page request, whereas the second implementation exploits the power of the RAM model to achieve $O(\log k / \log\log k)$ worst case time per request.

2 Randomized Selection Process

In this section we recall the notions of *offset functions* for paging algorithms introduced in [16]. 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, 6], 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 for σ 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 after processing σ iff $\omega(C) = 0$. In [16] it was shown that the class of valid configurations $\mathcal{V}$ determines the value of ω on any configuration C by $\omega(C) = \min_{X \in \mathcal{V}}\{|C \setminus X|\}$.

Koutsoupias and Papadimitriou [16] showed that ω can be represented by a sequence of $k+1$ disjoint page sets $(L_0, L_1, \ldots, L_k)$, denoted *layers*, which can be constructed as follows[3]. Initially each layer L_i, where $i > 0$, consists of one

[2]Since no explicit implementation of EQUITABLE2 is provided, due to their similarity we assume it to be the same as for EQUITABLE.

[3]We use a slightly modified, yet equivalent, version of the layer representation in [16].

of the first requested k pairwise distinct pages. The layer L_0 contains all pages not in $L_1, \ldots, L_k$. Since the offset function ω depends on the input sequence it has to be updated after each request. If ω is the offset function for input σ and page p is requested next, we denote by ω^p the offset function which results for the input σp and update the layers as follows[4]:

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

In [16] the relationship in Lemma 1 between the layer representation of ω and the class of valid configurations $\mathcal{V}$ was given.

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

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 u be the smallest index such that $L_{u+1}, \ldots, L_k$ are all singletons. We distinguish two sets: the set of *revealed* pages $R(\omega) = L_{u+1} \cup \cdots \cup L_k$, and the set of *unrevealed* pages $N(\omega) = L_1 \cup \cdots \cup L_u$. A valid configuration contains all revealed pages and no page from L_0. Note that when requesting some unrevealed page p in the support, we have $R(\omega^p) = R(\omega) \cup \{p\}$ and the number of layers containing unrevealed 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 describes the same offset function.

Equitable, Equitable2 and the forgiveness technique. Given the layer representation of ω by the sequence requested so far, a probability distribution over all possible actual cache configurations was proposed in [1]. 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. A page replacement strategy that maintains this probability distribution under the constraints that 1) it replaces one page only upon a cache fault and 2) the cache content does not change upon a cache hit, was shown to be H_k-competitive [1] . The authors also provide the randomized algorithm EQUITABLE which handles a page request in $O(k^2)$ time and achieves the desired distribution under both constraints.

Note that by repeatedly requesting pages from L_0 the amount of pages in the support increases. In order to reduce the space requirements *forgiveness*

[4]For easiness of exposition we refer by ω to both the offset function and its corresponding layer representation.

techniques can be applied, which use an approximation of the offset function in order to cap the support size. The intuition behind these techniques is that a large support implies that the adversary did not play optimally and there is a large gap between the actual ratio and the worst case ratio of H_k. This gap cannot be closed by the adversary with future requests, and thus allows the online algorithm to deviate from the original layer update rule when tracking the offset function, while still preserving the H_k-competitiveness.

Given an offset function ω, both EQUITABLE and EQUITABLE2 have identical cache distributions. The difference consists in the forgiveness steps, more precisely they have different update rules for the offset function ω, when the support becomes too large. If the support size reaches a threshold EQUITABLE uses an approximation of the current offset function in order to bound the support size by $O(k^2 \log k)$. EQUITABLE2 uses an improved forgiveness step leading to space requirements of $O(k)$. More precisely whenever the support contains $3k$ pages and a page p is requested from L_0, EQUITABLE2 adds p to L_1 and handles the update of ω as p would have been requested from L_1.

Definition 1 *Given the current offset function $\omega = (L_0, \ldots, L_k)$ and the page request p, the update rule for ω including the forgiveness step of* EQUITABLE2 *is as follows*

$$\omega^p = \begin{cases} (L_0 \setminus \{p\}, L_1, \ldots, L_{k-2}, L_{k-1} \cup L_k, \{p\}) & \text{if } p \in L_0, |S(\omega)| < 3k, \\ (L_0 \setminus \{p\} \cup L_1, \ldots, L_{k-2}, L_{k-1}, L_k, \{p\}) & \text{if } p \in L_0, |S(\omega)| = 3k, \\ (L_0, \ldots, L_{i-2}, L_{i-1} \cup L_i \setminus \{p\}, L_{i+1}, \ldots, L_k, \{p\}) & \text{if } p \in L_i, i > 0. \end{cases}$$

Note that by the given update rule the layers $L_1, \ldots, L_k$ contain each at least one element. Thus in the case $|S(\omega)| = 3k$ the support size decreases by $|L_1| \geq 1$ and increases by 1 (the requested page p), and therefore we always have $|S(\omega)| \leq 3k$. In [6] it was shown that applying this update rule for ω still leads to a competitive ratio of H_k.

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.

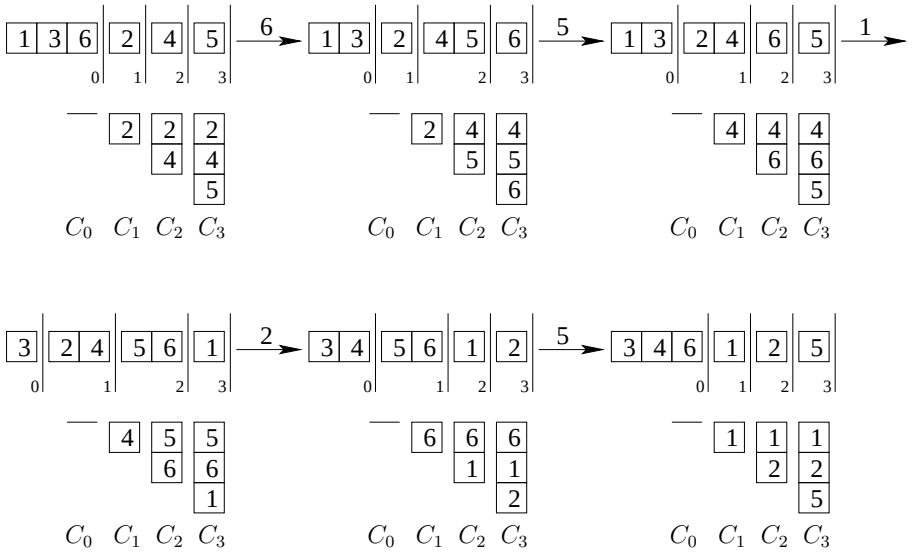

Figure 1: Example for updating the layers $L_0, \ldots, L_k$ and the selection sets $C_0, \ldots, C_k$ for $k = 3$. The initial cache configuration is $\{2, 4, 5\}$. The request sequence is $(6, 5, 1, 2, 5)$ and the priority of a page is its number.

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 2 *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 set* $C_0(\omega) = \emptyset$ *and 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* $j \in \{0, \ldots, k\}$. *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 $i \leq j < k$ 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:

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

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

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 $j \geq i$ is the smallest index with $|C_j \cap C_k| = j$.*

Before the proof, we note for the third case that the constraint $|C_j \cap C_k| = j$ means that all pages in C_j are also in C_k. While in general this constraint does not hold for all j, it is satisfied for all layers containing revealed pages (and the rightmost layer containing unrevealed pages) and thus such a j always exists. Moreover, $|C_j \cap C_k| = j$ is equivalent to $|(L_1 \cup \cdots \cup L_j) \cap C_k| = j$, since C_j has elements only in $L_1 \cup \cdots \cup L_j$ 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, the fact that $C_k = \max_{k-1}(C_{k-2} \cup L_{k-1}) \cup L_k$, and 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 $C_i, \ldots, C_k$ 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. If $i > 1$ 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 $|C_1| = 1$. If $i > 1$ 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 that $C'_k = C_k \setminus \min(C_j) \cup \{p\}$. □

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 possess 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 timestamp of the last request for any page in L_i is smaller than the timestamp of the last request for any page in L_{i+1}.

Page replacement. The algorithm maintains the invariant that $M = C_k$ after each request. To do so, it keeps track of the layer partition $\omega = (L_0, \ldots, L_k)$ according to Definition 1 (including the forgiveness step of EQUITABLE2), where it suffices to store only the support layers $(L_1, \ldots, L_k)$. The cache update is performed according to Theorem 1[5]. 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 condition that the layer index of p_j is j, i.e. $p_j \in L_j$, is satisfied (index j is guaranteed to exist, since the condition holds for all revealed pages and the rightmost unrevealed page). We evict the page with minimal priority from $p_1, \ldots, p_j$. The layers are updated after the cache update.

Priorities. To assign priorities, we develop a data structure which maintains a dynamic random *ordered* set P of integers. We require at all times that the ranks of numbers in P correspond to an (equally distributed) random permutation of

[5]Theorem 1 does not explicitly take into account the forgiveness step. According to Definition 1, if $p \in L_0$ and forgiveness is applied we treat p as if it was requested in L_1.

$\{1, \ldots, |P|\}$. Under the assumption that the size of P is bounded by a number u, we require $|P|$ to support two operations: *expand*, which adds a new element to P, and *delete*(x) which removes element x from P.

We use the universe $U = \{1, \ldots, u\}$ for numbers in P. We start with $P = \emptyset$. Upon an expand operation we choose an element uniformly at random from $U \setminus P$ and insert it in P. Upon a delete operation, the element to delete is simply removed from P. In particular, the expand operation corresponds to a step in the Fisher-Yates shuffle algorithm (original method) [12]. They showed that applying u expand operation results in a random permutation of U. In Lemma 3 we show that the two operations can be implemented efficiently and that the ranks of elements in P form a random permutation.

Lemma 3 *The ranks of the elements in P represent a random permutation of $\{1, \ldots, |P|\}$. The data structure can be implemented in $O(1)$ time per request and uses $O(u)$ space.*

Proof. Let $e_x = (e_1, \ldots, e_x)$ be a random variable describing the ranks in the sequence after x consecutive expand operations. Since after u expand operation the Fisher-Yates shuffle yields a random permutation. Thus, e_u is a random permutation of U and it follows that e_x is a random permutation of $\{1, \ldots, x\}$. If e_x describes the ranks in P, we get upon an expand operation the distribution e_{x+1} and upon a delete operation the distribution e_{x-1}. We conclude that ranks in P correspond to $e_{|P|}$ and thus a random permutation of $\{1, \ldots, |P|\}$.

The data structure can be implemented using an array A of size u and a list L of size $|P|$. We store the elements from P in the first $|P|$ locations of A and the rest of A contains all elements from $U \setminus P$. The element order is given by the list L. We further assume direct access to the position of any element $i \in U$ in A and in L using additional $O(u)$ space. Initially we set $A[i] = i$ and $L = \emptyset$. Upon an expand operation we choose a random index r in the range $[|P| + 1, u]$ and append $A[r]$ to L. To reflect this in A we switch the contents of $A[|P| + 1]$ and $A[r]$. The deletion of element x is similar, we first look up the index i_x of x in A and swap $A[i_x]$ and $A[|P|]$. We further delete x from L. □

By the forgiveness mechanism the support size is at all times $O(k)$ and thus priorities can be maintained using the data structure previously introduced. When a page enters the support we assign it a priority using the expand operation, and when it leaves the support we use the delete operation. This takes $O(k)$ space and $O(1)$ time per page request (at most one page is assigned a new priority at each request) by setting u to the maximal support size which is guaranteed by the forgiveness technique to be at most $3k$.

Time and space complexity. Storing the layer partition together with the page priorities needs $O(k)$ space by applying the forgiveness mechanism of EQUITABLE2 [6]. A naive implementation storing the layers in an array processes a page request in $O(k)$ time. In the remainder of the paper we will improve this

naive bound first to $O(\log k)$ worst case time per request using simple pointer-based data structures and then to $O(\log k / \log\log k)$ time per request using data structures in the RAM model.

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 4.

Lemma 4 OnlineMin *is H_k-competitive.*

3.2 Algorithm Implementation

In this section we show that OnlineMin can be implemented using a sorted list augmented with a series of specific operations. We will later focus only on giving data structures supporting these operations.

Basic structure. In the following we represent each page in the support by the timestamp of its last request. 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-prio(`l_p`)`. Find the cache element $l_q \in (l_1, \ldots, l_p)$ with minimum priority.

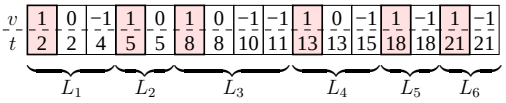

Figure 2: Example for list L: representing pages by timestamps of last requests, we have $L_1 = \{2, 4\}$, $L_2 = \{5\}$, $L_3 = \{8, 10, 11\}$, $L_4 = \{13, 15\}$, $L_5 = \{18\}$, and $L_6 = \{21\}$. Layer delimiters are emphasized and the memory content is $M = \{4, 10, 11, 15, 18, 21\}$.

- `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.

We note that the prefix sum cannot be negative, and thus for the `find-zero` operation it suffices to find the first element to the right having the minimum prefix sum. For this reason, we refer to `find-zero` also as `find-pref-min`.

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-prio` applied to the value returned by `find-zero`. If $p \in L_0$ and 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-prio` 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 in Definition 1. 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.

We note that while the priority of each page takes $O(\log k)$ space, the timestamp for its last request takes $O(\log n)$ space. We reduce the space requirement to $O(\log k)$ by simply resetting the timestamps for pages and layers in support after $O(k)$ operations, setting the new timestamps to $1, \ldots, |S|$, where $|S|$ is the support size. The new timestamps are assigned in a left-to-right manner, thus ensuring that the relative order of the new timestamps reflect the old order. Since by the forgiveness mechanism we have at all times $|S| = O(k)$, it follows that $O(1)$ amortized time per page request is required. We further deamortize this by resetting only c timestamps per page request, in a left-to-right manner, where c is a constant with $c > 2$; it is necessary to have $c > 2$ because at each request two elements (i.e. the rightmost set delimiter and page) receive new timestamps according to the current (large) timestamp, and we need to ensure that more pages receive updated (small) timestamps. At the end of the day, in $O(1)$ worst case time per request we ensure that the timestamps take also $O(\log k)$ space.

3.3 Pointer-based 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 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 $\mathcal{T}(u)$. To deal with the priorities, at each node u we store the minimum priority $u.min\_p$ of the cache pages in $\mathcal{L}(u)$ and a pointer $u.idx\_min\_p$ to the leaf having this priority; if no cache pages exist, $u.min\_p$ is set to infinity and $u.idx\_min\_p$ to null. For handling the v-values we store at each node u the sum $u.sum$ of the v-values stored in $\mathcal{L}(u)$ and the minimum prefix sum $u.pref\_min$ of the v-values

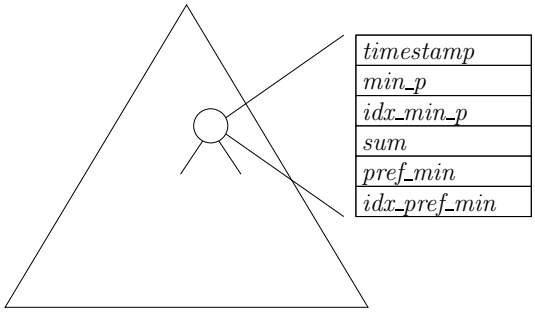

Figure 3: The additional information stored at each node in the page-set structure.

restricted on the elements of $\mathcal{L}(u)$. More precisely, if $\mathcal{L}(u) = (p_1, \ldots, p_m)$, we have $u.pref\_min = \min_{l=1}^{m}(\sum_{j=1}^{l} p_j.v)$. Also, we store a pointer $u.idx\_pref\_min$ to the leaf having the prefix sum $u.pref\_min$. All the data stored at each node is shown in Figure 3.

In the following fact it is shown that all values stored at each node can be computed bottom up in $O(1)$ time per node.

Fact 2 *For each node u with children $u.left$ and $u.right$, the following hold:*

- $u.min\_p = \min\{u.left.min\_p, u.right.min\_p\}$ *and $u.idx\_min\_p$ is either $u.left.idx\_min\_p$ or $u.right.idx\_min\_p$ depending on the origin of $u.min\_p$.*
- $u.pref\_min = \min\{u.left.pref\_min, u.left.sum + u.right.pref\_min\}$ *and $u.sum = u.left.sum + u.right.sum$. We also have that $u.idx\_pref\_min$ is either $u.left.idx\_pref\_min$ or $u.right.idx\_pref\_min$ depending on the origin of the minimum prefix sum computed.*

Updates. We discuss how to perform insertions and deletions in the page-set structure. To insert an element, we create a leaf for the new element and an internal node, and update the information stored at each internal node on the path to the root as described in Fact 2. For the $O(1)$ nodes per level involved in rotations due to rebalancing we also recompute these values. 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.

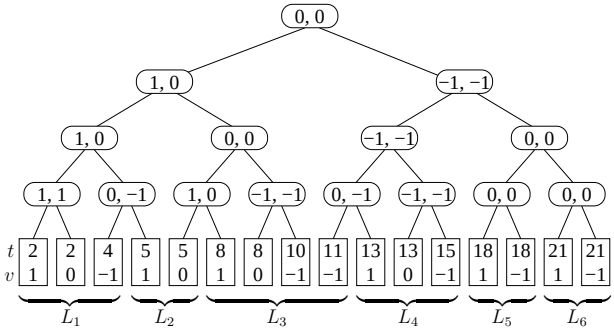

Figure 4: The page-set data structure for $L_1 = \{2, 4\}$, $L_2 = \{5\}$, $L_3 = \{8, 10, 11\}$, $L_4 = \{13, 15\}$, $L_5 = \{18\}$, and $L_6 = \{21\}$, and the memory image $M = \{4, 10, 11, 15, 18, 21\}$. For each internal node u the $(u.sum, u.pref\_min)$ values are shown.

For `find-min-prio` we find the page element having the minimum priority in $l_1, \ldots, l_p$ by traversing the path from l_p to the root in the page-set structure. For each node u on the path where l_p is in the right-subtree, consider its left child w. For all such nodes w identified and the leaf l_p, take the minimum over all $w.min\_p$, and return the corresponding $w.idx\_min\_p$ index. Since it does a bottom-up traversal, this operation takes $O(\log k)$ time in the worst case.

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 do so by traversing the path from l_p to the root. For each visited node u let s denote the sum of v-values of all elements in $\mathcal{L}(u)$ to the right of l_p. Also, let ps be the best prefix sum found so far and s_idx the corresponding leaf. Initially $s\_idx = l_p$ and $s = ps = l_p.v$. When advancing to the parent from the right children no action is required. When advancing to the parent node u from the left child we first check if $s + u.right.pref\_min < ps$ and if so we have found a better prefix sum, and update $ps = s + u.right.pref\_min$ and $s\_idx = u.right.idx\_pref\_min$. Finally we update $s = s + u.right.sum$. We return the identified leaf s_idx. This operation requires a bottom-up 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 by Definition 2 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 request 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. Theorem 3 summarizes the time and space complexities for this implementation of OnlineMin.

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

3.4 RAM model structures

In this section we provide an alternative implementation for OnlineMin which handles a page request in $O(\log k/\log\log k)$ time. In particular,we describe how the page-set structure operations can be implemented in $O(\log k/\log\log k)$ time in the RAM model and argue that this is the best possible for an approach using the page-set interface defined in Section 3.2.

For the set structure, we use the data structure in [4] which supports updates and predecessor queries in $O(\sqrt{\log k/\log\log k})$ time while using $O(k)$ space. A data structure which supports the `search-page`, `insert`, `delete`, and `find-min-prio` operations in $O(\log k/\log\log k)$ time can be found in [20, Section 5], which adapts fusion tree ideas [13] to priority search trees [17].

In the following we modify the page-set structure previously introduced to support both `find-min-prio` and `find-zero` operations in $O(\log k/\log\log k)$ time. In particular, for the `find-min-prio` operation we borrow ideas from [20]. Instead of the balanced binary search tree in Section 3.3 we use a B-tree [5], where each node has degree at most $\Delta = \log^{\varepsilon} k$ for some $0 < \varepsilon \leq 1/4$.

Again, each node u stores $u.min\_p$, $u.idx\_min\_p$, $u.sum$ and $u.idx\_pref\_min$. Furthermore, to support the `find-min-prio` operation and to be able to update $u.min\_p$, each node u stores additionally the following:

- A Q-heap $u.Q$ over the $w.min\_p$ priorities at the children of u.
- A word $u.\pi$ storing for all the Δ children the rank of $u.child[i].min\_p$ among $u.child[1].min\_p, ..., u.child[\Delta].min\_p$. Therefore the number of bits required by $u.\pi$ is $\Delta \log \Delta = o(\log k)$.

The Q-heap of Fredman and Willard [14, Theorem, page 550], supports in $O(1)$ time insertions, deletions, predecessor queries (in particular min-queries), and rank queries (how many elements are smaller than a query element) on sets having at most $(\log k)^{1/4}$ elements. The data structure requires word size at least $\log k$ and time $O(k)$ to preprocess some global tables.

The Q-heap $u.Q$ allows us to update $u.min\_p$ in $O(1)$ time when the i'th child w gets a new $w.min\_p$ value, by deleting the old $w.min\_p$ value from $u.Q$, inserting the new $w.min\_p$ into $u.Q$, and querying $u.Q$ for the new minimum. When updating $u.min\_p$ we can also compute the new rank of $w.min\_p$ among the children of node u using $u.Q$, and update $u.\pi$ using a precomputed table:

$T[old\_\pi, i, new\_rank] = new\_\pi$. Note that old_π, i, and new_rank in total only require $\Delta \log \Delta + \log \Delta + \log \Delta = o(\log k)$ bits, i.e. the table needs $k^{o(1)}$ precomputed entries. To perform `find-min-prio` we similarly traverse a leaf-to-root path. At each node u where we now come from the i'th child w, we identify the minimum of $u.child[1].min\_p, \ldots, u.child[i-1].min\_p$ by considering π only, which again can be answered using a precomputed table T, where $T[\pi, i]$ stores the index of the child having the minimum min_p value.

To support the `find-zero` operation we need to efficiently update and query the prefix sum fields. To this end, we consider the updates in $\mathcal{T}(u)$ in *phases*, where each phase consists of Δ updates below u. At each node u we store the following:

- A counter $u.count$ in the range $0 \ldots \Delta - 1$, counting the number of updates below u since the start of the phase.
- A word $u.\tau$ storing an array of size Δ with $M[i] \in [0, \ldots, \Delta(\Delta + 2)]$ for all i; this means that $u.\tau$ can be stored in $\Delta(1 + 2 \log \Delta) = o(\log k)$ bits.
- Two arrays m and PS, each of size Δ.
- A word $u.\varsigma$ storing an array dm of size Δ with $dm[i] \in [-\Delta, \ldots, \Delta]$, i.e. $u.\varsigma$ can be stored in $\Delta \log(2\Delta + 1) = o(\log k)$ bits.

At the beginning of a phase for some node u we reset the information stored at u as follows. The arrays PS and m store the prefix sum and the minimum prefix sum for the children of u respectively, that is $PS[i] = \sum_{j=1}^{i-1} u.child[j].sum$ and $m[i] = u.child[i].pref\_min + PS[i]$. Also, we set $dm[i] = 0$ for all i. After a number of updates in a phase for $\mathcal{T}(u)$, $dm[i]$ is the value to be added to $PS[i]$ to get the correct values for $PS[i]$ and $m[i]$.

We let M be an approximation of m which maintains at all times during the phase the following invariant: for all i, if $m[j] + dm[j] = \min(m[i] + dm[i], \ldots, m[\Delta] + dm[\Delta])$ then we have that $M[j] = \min(M[i], \ldots, M[\Delta])$.

At the beginning of a phase we construct M for decreasing index i, while keeping track of the minimum $m[min]$ where $min > i$. Initially $min = u.degree$ and $M[u.degree] = \Delta^2$. For $i = u.degree - 1$ `downto` 1, we compute $M[i]$ as follows: if $m[i] \geq m[min] + \Delta$ then $M[i] = M[min] + \Delta$, and if $m[i] \leq m[min] - \Delta$, then $M[i] = M[min] - \Delta$; otherwise, $M[i] = M[min] + m[i] - m[min]$. If after computing $M[i]$ we have $M[i] < M[min]$ then we set $min = i$, see e.g. Figure 5. The key idea is that any update can only change the $pref\_min$ value of a node by at most one, since the v-values are in $\{-1, 0, 1\}$. Therefore, if for some i and j it holds that $|u.child[i].pref\_min - u.child[j].pref\_min| > \Delta$ at the beginning of the phase, then their relative order does not change during the Δ updates in the phase.

To analyze the range of the $M[i]$ values, we note that since $M[j]$ is decreasing during the construction any assigned value can be at most $M[u.degree] + \Delta$, i.e. $\Delta(\Delta + 1)$. Similarly, each $M[i] \geq M[i + 1] - \Delta$, i.e. all $M[i] \geq M[u.degree] - (\Delta - 1)\Delta = \Delta^2 - \Delta^2 + \Delta = \Delta$. Since each update below u during a phase can

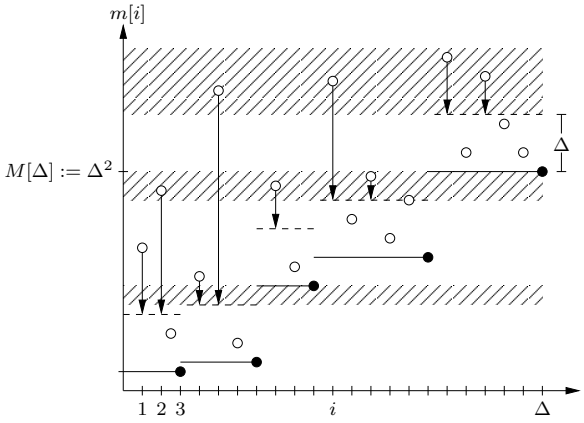

Figure 5: Illustration of the construction of M from m. Circles are the elements of m. For each i the solid line/circle shows the minimum m to the right of i. Elements that cannot become the answer during the next Δ updates are replaced by the elements pointed to by an arrow. The shaded area are the ranges of the domain removed in the transformation from m to M.

change $M[i]$ by at most ± 1, it follows that during the Δ operations in a phase all $M[i]$ values are within the range $[0 \dots \Delta(\Delta+2)]$.

During an update where $u.child[i].sum$ is incremented we increment all $M[j]$ and $dm[j]$ for $i < j \leq u.degree$; also, we increment $M[i]$ and $m[i]$ whenever $u.child[i].pref\_min$ is incremented. The case when $u.child[i].sum$ is decremented is treated similarly. The updating of M and dm can be done in $O(1)$ time using table lookup, i.e. $T1[old\_\tau, i] = new\_\tau$ and $T2[old\_\varsigma, i] = new\_\varsigma$. Since M is reconstructed only at the beginning of a phase, and M and dm can be constructed in $O(\Delta)$ time, it follows that the amortized cost to update M and dm for an update below u is amortized $O(1)$.

Updating the $u.sum$ values bottom-up during updates is done by adding (subtracting) the inserted (deleted) value along the leaf-to-root path. The child i storing the new $u.idx\_pref\_min = u.child[i].idx\_pref\_min$ can be extracted from $u.M$ (i.e. $u.\tau$) using a table lookup, and $v.pref\_min = m[i] + dm[i]$ (we compute the index of the minimum child before actually knowing the exact value).

A query, i.e. `find-zero`, is performed bottom-up as in Section 3.3, where we keep track of the minimum prefix sum s so far, except that when reaching node u from child i, we need to find the child j with minimum prefix sum among children $i+1, \dots, degree(u)$. This can be extracted from $u.M$ using table lookup in $O(1)$ time. The minimum prefix sum for $u.child[j]$ is $PS[j] = u.m[j] + u.dm[j]$, which is compared to the prefix sum from $u.child[i]$ which is $PS[i] = s + u.PS[i] + v.dm[i]$; s becomes the minimum of $PS[i]$ and $PS[j]$. If this minimum is $PS[j]$, then idx_pref_min is updated to $u.child[j].idx\_pref\_min$.

Inserting a new leaf into the tree might cause a node to get more than Δ

children, in which case we split the affected node into two nodes of degree at most $\Delta/2+1$ each. Whenever a node is split, gets a new child, or loses a child, we recompute all the information at the node in $O(\Delta)$ time. This way the total number of node splits is bounded by $O(\#insertions/\Delta)$ and it follows that the total cause for splitting and inserting/deleting leaves is $O(\#insertions \cdot \Delta)$, i.e. amortized $O(\Delta)$ per update. To avoid the height of the tree to exceed $O(\log_\Delta k)$, we globally rebuild the tree and all the associated information from scratch in $O(k)$ time whenever half of the leaves inserted into the tree have been deleted.

To achieve worst case bounds we use standard deamortization techniques. We perform the node splitting and the computation of the appropriate values at the beginning of each phase incrementally. We simply save the state of the node and at each update we perform $O(1)$ computations such that after Δ updates the updated values in the given node are computed. Similarly, the global rebuilding can be done incrementally as well, which yields $O(1)$ at each level of the tree for all operations. Since the tree has height $O(\log_\Delta k)$ where $\Delta = \log^\varepsilon k$, it follows that all operations can be implemented in $O(\log k/\log\log k)$ worst case time.

The following results stems from the fact that each page request uses a constant number of operations on the previously introduced data structure.

Theorem 4 *A page request can be done in $O(\log k/\log\log k)$ time while using $O(k)$ space.*

Lower bounds The following cell-probe (RAM) lower bounds (using words of $\log k$ bits) state that for the page-set structure we cannot achieve better than $\Omega(\log k/\log\log k)$ query time with polylogarithmic deletion bounds. According to [3, Proposition 2] (note after proposition about decremental priority searching) any data structure supporting `delete` and `find-min-prio` requires $\Omega(\log k/\log\log k)$ time for polylogarithmic deletion time. We note that the given lower bound applies to the page-set structure in particular and not to the paging problem in general, not even to the approach taken by OnlineMin. Nonetheless, they show that to process a page request in $o(\log k/\log\log k)$ time any implementation must exploit some particular characteristics of OnlineMin.

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] S. Alstrup, T. Husfeldt, and T. Rauhe. Marked ancestor problems. In *Proc. 39th Annual Symposium on Foundations of Computer Science*, pages 534–544. IEEE Computer Society, 1998.

[4] A. Andersson and M. Thorup. Dynamic ordered sets with exponential search trees. *Journal of the ACM*, 54(3), 2007.

[5] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

[6] W. W. Bein, L. L. Larmore, J. Noga, and R. Reischuk. Knowledge state algorithms. *Algorithmica*, 60(3):653–678, 2011.

[7] L. A. Belady. A study of replacement algorithms for virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[8] A. Borodin and R. El-Yaniv. *Online computation and competitive anlysis*. Cambridge University Press, 1998.

[9] M. Chrobak, E. Koutsoupias, and J. Noga. More on randomized on-line algorithms for caching. *Theoretical Computer Science*, 290(3):1997–2008, 2003.

[10] 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.

[11] A. Fiat and G. J. Woeginger, editors. *Online Algorithms, The State of the Art (the book grow out of a Dagstuhl Seminar, June 1996)*, volume 1442 of *Lecture Notes in Computer Science*. Springer, 1998.

[12] R. A. Fisher and F. Yates. *Statistical tables for biological, agricultural, and medical research (3rd edition)*. Oliver and Boyd, 1948.

[13] M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, 1993.

[14] M. L. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *Journal of Computer and System Sciences*, 48(3):533–551, 1994.

[15] A. R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator. Competitive snoopy caching. *Algorithmica*, 3:77–119, 1988.

[16] E. Koutsoupias and C. H. Papadimitriou. Beyond competitive analysis. In *Proc. 35th Symposium on Foundations of Computer Science*, pages 394–400. IEEE Computer Society, 1994.

[17] E. M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2):257–276, 1985.

[18] L. A. McGeoch and D. D. Sleator. A strongly competitive randomized paging algorithm. *Algorithmica*, 6(6):816–825, 1991.

[19] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[20] D. E. Willard. Examining computational geometry, van Emde Boas trees, and hashing from the perspective of the fusion tree. *SIAM Journal of Computing*, 29(3):1030–1049, 2000.