Open access • Book Chapter • DOI:10.1007/3-540-36136-7_20

Funnel Heap - A Cache Oblivious Priority Queue — Source link

Gerth Stølting Brodal, Rolf Fagerberg

Institutions: Aarhus University

Published on: 21 Nov 2002 - International Symposium on Algorithms and Computation

Topics: Cache algorithms, Cache-oblivious algorithm, Cache pollution, Cache coloring and Cache

Funnel Heap - A Cache Oblivious Priority Queue

Gerth Stølting Brodal[1,⋆,⋆⋆] and Rolf Fagerberg[1,⋆]

BRICS[⋆⋆⋆], Department of Computer Science, University of Aarhus, Ny Munkegade, DK-8000 Århus C, Denmark. E-mail: {gerth,rolf}@brics.dk

Abstract The cache oblivious model of computation is a two-level memory model with the assumption that the parameters of the model are unknown to the algorithms. A consequence of this assumption is that an algorithm efficient in the cache oblivious model is automatically efficient in a multi-level memory model. Arge et al. recently presented the first optimal cache oblivious priority queue, and demonstrated the importance of this result by providing the first cache oblivious algorithms for graph problems. Their structure uses cache oblivious sorting and selection as subroutines. In this paper, we devise an alternative optimal cache oblivious priority queue based only on binary merging. We also show that our structure can be made adaptive to different usage profiles.

1 Introduction

External memory models are formal models for analyzing the memory access patterns of algorithms on modern computer architectures with several levels of memory and caches. The cache oblivious model, recently introduced by Frigo et al. [13], is based on the I/O model of Aggarwal and Vitter [1], which has been the most widely used external memory model—see the surveys by Arge [2] and Vitter [14]. Both models assume a two-level memory hierarchy where the lower level has size M and data is transfered between the two levels in blocks of B elements. The difference is that in the I/O model the algorithms are aware of B and M, whereas in the cache oblivious model these parameters are unknown to the algorithms and I/Os are handled automatically by an optimal off-line cache replacement strategy.

Frigo et al. [13] showed that an efficient algorithm in the cache oblivious model is automatically efficient on each level of a multi-level memory model. They also presented optimal cache oblivious algorithms for matrix transposition, FFT, and sorting. Cache oblivious search trees which match the search cost of the standard (cache aware) B-trees [4] were presented in [6,8,9,11]. Cache oblivious algorithms have also been given for problems in computational geometry [6,10], for scanning dynamic sets [5], and for layout of static trees [7]. Recently, the first

⋆ Partially supported by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

⋆⋆ Supported by the Carlsberg Foundation (contract number ANS-0257/20).

⋆⋆⋆ Basic Research in Computer Science, www.brics.dk, funded by the Danish National Research Foundation.

cache oblivious priority queue was developed by Arge et al. [3], who also showed how this result leads to several cache oblivious graph algorithms. The structure of Arge et al. uses existing cache oblivious sorting and selection algorithms as subroutines.

In this paper, we present an alternative cache oblivious priority queue, Funnel Heap, based only on binary merging. Essentially, our structure is a single heap-ordered tree with binary mergers in the nodes and buffers on the edges. It was inspired by the cache oblivious merge sort algorithm Funnelsort presented in [13] and simplified in [10]. Like the priority queue of Arge et al., our data structure supports the operations Insert and DeleteMin using amortized $O(\frac{1}{B}\log_{M/B}\frac{N}{B})$ I/Os per operation, under the so-called tall cache assumption $M \geq B^2$. Here, N is the total number of elements inserted.

For a slightly different algorithm we give a refined analysis, showing that the priority queue adapts to different profiles of usage. More precisely, we show that the ith insertion uses amortized $O(\frac{1}{B}\log_{M/B}\frac{N_i}{B})$ I/Os, where N_i can be defined in any of the following three ways: (a) N_i is the number of elements present in the priority queue when performing the ith insert operation, (b) if the ith inserted element is removed by a DeleteMin operation prior to the jth insertion then $N_i = j - i$, or (c) N_i is the maximum rank that the ith inserted element has during its lifetime in the priority queue, where rank denotes the number of smaller elements present in the priority queue. DeleteMin is amortized for free since the work is charged to the insertions. These results extend the line of research taken in [12], where (a) and (c) are called size profile and max depth profile, respectively.

We note that as in [10], we can relax the tall cache assumption by changing parameters in the construction. More precisely, for any $\varepsilon > 0$ a data structure only assuming $M \geq B^{1+\varepsilon}$ can be made, at the expense of $\log_{M/B}(x)$ becoming $\frac{1}{\varepsilon}\log_M(x)$ in the expressions above for the running times. We leave the details to the full paper.

This paper is organized as follows. In Section 2 we introduce the concept of mergers and in Section 3 we describe our priority queue. Section 4 gives the analysis of the presented data structure. Finally, Section 5 gives the analysis based on different profiles of usage.

2 Mergers

Our data structure is based on *binary mergers*. A binary merger takes as input two sorted streams of elements and delivers as output the sorted stream formed by merging of these. A *merge step* moves one element from the head of an input stream to the tail of the output stream. The heads of the input streams and the tail of the output stream reside in *buffers* holding a limited number of elements. A buffer is simply an array of elements, plus fields storing its capacity and pointers to its first and last elements. Figure 1 shows a binary merger.

Binary mergers may be combined to *binary merge trees* by letting the output buffer of one merger be an input buffer of another. In other words, a binary merge

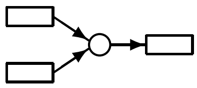

Figure 1. A binary merger.

tree is a binary tree with mergers at the internal nodes and buffers at the edges. The leaves of the tree are buffers containing the streams to be merged. See Figure 3 for an example of a merge tree. Note that we describe a merger and its output buffer as separate entities mainly in order to visualize the binary merge process. In an actual implementation, the two will probably be identified, and merge trees will simply be binary trees of buffers.

Invoking a binary merger in a merge tree means performing merge steps until its output buffer is full (or both input streams are exhausted). If an input buffer gets empty during the process (but the corresponding stream is not exhausted), it is filled by invoking the merger having this buffer as output buffer. If both input streams of a merger get exhausted, its output stream is marked as exhausted. The resulting recursive procedure, except for the issue of exhaustion, is shown in Figure 2 as $\textsc{Fill}(v)$. An invocation $\textsc{Fill}(r)$ of the root r of the merge tree produces the next part of a stream which is the merge of the streams at the leaves of the tree.

```
Procedure Fill(v)
    while v's output buffer is not full
        if left input buffer empty
            Fill(left child of v)
        if right input buffer empty
            Fill(right child of v)
        perform one merge step
```

Figure 2. Invoking a merger.

One particular merge tree is the *k-merger*. In this paper, we only consider $k = 2^i$ for i a positive integer. A k-merger is a perfectly balanced binary merge tree with $k-1$ binary mergers, k input streams, and buffers of specific sizes. The size of the output buffer of the root is k^3. The sizes of the remaining buffers are defined recursively: Let the *top tree* be the subtree consisting of all nodes of depth at most $\lceil i/2 \rceil$, and let the subtrees rooted by nodes at depth $\lceil i/2 \rceil + 1$ be the *bottom trees*. The buffers at edges between nodes at depth $\lceil i/2 \rceil$ and depth $\lceil i/2 \rceil + 1$ all have size $\lceil k^{3/2} \rceil$, and the sizes of the remaining buffers are defined by recursion on the top tree and the bottom trees. A 16-merger is illustrated in Figure 3.

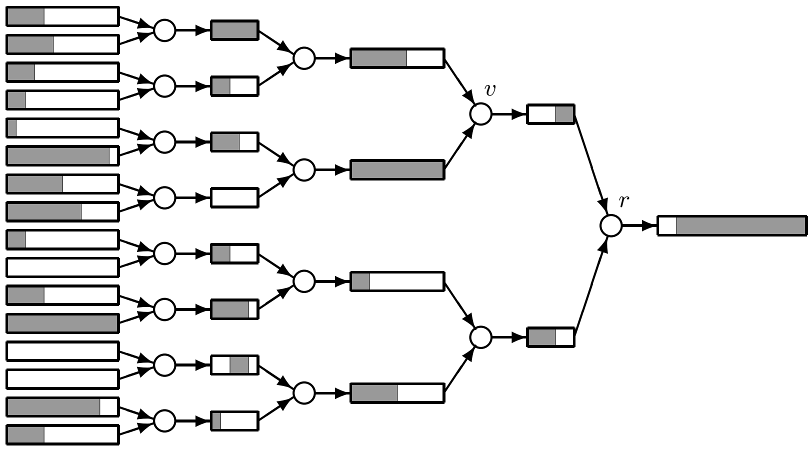

Figure 3. A 16-merger consisting of 15 binary mergers. Shaded regions are the occupied parts of the buffers. The procedure $\textsc{Fill}(r)$ has been called on the root r, and is currently performing merge steps at its left child v.

To achieve I/O efficiency in the cache oblivious model, the memory layout of a k-merger is also defined recursively. The entire k-merger is laid out in contiguous memory locations, first the top tree, then the middle buffers, and finally the bottom trees, and this layout is applied recursively within the top tree and each of the bottom trees.

The k-merger structure was defined by Frigo et al. [13] for use in their cache oblivious mergesort algorithm Funnelsort. The algorithm described above for invoking a k-merger appeared in [10], and is a simplification of the original one. For both algorithms, the following lemma holds [10,13].

Lemma 1. *The invocation of the root of a k-merger uses $O(k + \frac{k^3}{B} \log_{M/B} k^3)$ I/Os, if $M \geq B^2$. The space required for a k-merger is $O(k^2)$, not counting the space for the input and output streams.*

3 The Priority Queue

Our data structure consists of a sequence of k-mergers with double-exponentially increasing k, linked together in a list as depicted in Figure 4, where circles are binary mergers, rectangles are buffers, and triangles are k-mergers. The entire structure constitutes a single binary merge tree. Roughly, the growth of k is given by $k_{i+1} = k_i^{4/3}$.

More precisely, let k_i and s_i be values defined inductively as follows,

$$\begin{aligned} (k_1, s_1) &= \quad (2, 8)\ , \\ s_{i+1} &= s_i(k_i + 1)\ , \\ k_{i+1} &= \lceil\lceil s_{i+1}{}^{1/3} \rceil\rceil\ , \end{aligned} \tag{1}$$

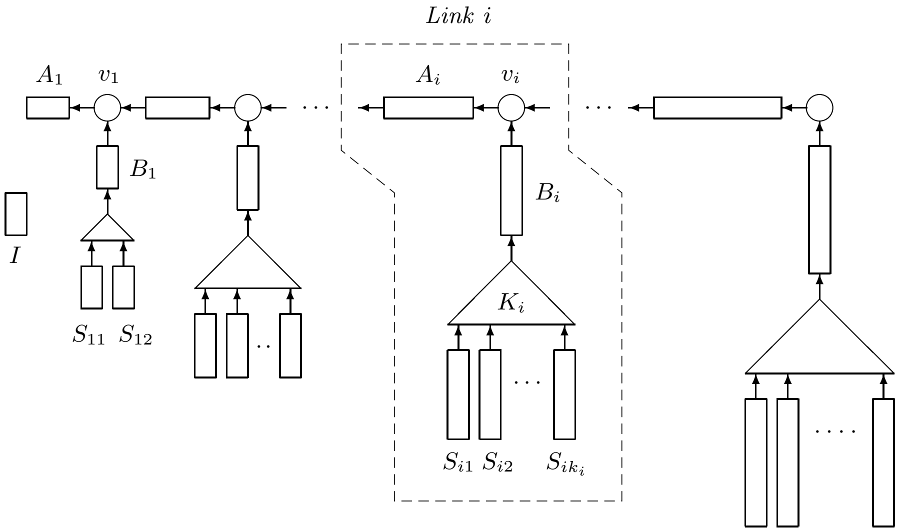

Figure 4. The priority queue based on binary mergers.

where $[\lceil x \rceil]$ denotes the smallest power of two above x, i.e. $[\lceil x \rceil] = 2^{\lceil \log x \rceil}$. *Link i* in the linked list consists of a binary merger v_i, two buffers A_i and B_i, and a k_i-merger K_i with k_i input buffers $S_{i1}, \ldots, S_{ik_i}$. We refer to B_i, K_i, and $S_{i1}, \ldots, S_{ik_i}$ as the *lower part* of the link. The size of both A_i and B_i is k_i^3, and the size of each S_{ij} is s_i. Link i has an associated counter c_i for which $1 \leq c_i \leq k_i + 1$. Its initial value is one. It will be an invariant that $S_{ic_i}, \ldots, S_{ik_i}$ are empty.

Additionally, the structure contains one insertion buffer I of size s_1. All buffers contain a (possibly empty) sorted sequence of elements. The structure is laid out in memory in the order I, link 1, link 2, ..., and within link i the layout order is c_i, A_i, v_i, B_i, K_i, S_{i1}, S_{i2}, ..., S_{ik_i}.

The linked list of buffers and mergers constitute one binary tree T with root v_1 and with sorted sequences of elements on the edges. We maintain the invariant that this tree is heap-ordered, i.e. when traversing any path towards the root, elements will be passed in decreasing order. Note that the invocation of a binary merger maintains this invariant. The invariant implies that if buffer A_1 is non-empty, the minimum element in the queue will be in A_1 or in I.

To perform a DeleteMin operation, we first call Fill(v_1) if buffer A_1 is empty. We then remove the smallest of the elements in A_1 and I from its buffer, and return it.

To perform an Insert operation, the new element is inserted into I while maintaining the sorted order of the buffer. If the number of elements in I is less than s_1, we stop. If the number of elements in I becomes s_1, we perform the following Sweep(i) operation, with i being the lowest index for which $c_i \leq k_i$. We find i by examining the links in increasing order.

The purpose of SWEEP(i) is to move the content of links $1, \ldots, i-1$ to the buffer S_{ic_i}. It may be seen as a carry ending at digit i during addition of one, if we view the sequence $c_1, c_2, c_3, \ldots$ as the digits of a number. More precisely, SWEEP(i) traverses the path p from A_1 to S_{ic_i} in the tree T and records how many elements each buffer on this path currently contains. During the traversal, it also forms a sorted stream σ_1 of the elements in the buffers on the part of p from A_i to S_{ic_i}. This is done by moving the elements to an auxiliary buffer. In another auxiliary buffer, it forms a sorted stream σ_2 of all elements in links $1, \ldots, i-1$ and in buffer I by marking A_i as exhausted and calling DELETEMIN repeatedly. It then merges σ_1 and σ_2 into a single stream σ, traverses p again while inserting the front elements of σ in the buffers on p in such a way that these buffers contain the same numbers of elements as before the insertion, and then inserts the remaining part of σ in S_{ic_i}. Finally, it resets c_ℓ to one for $\ell = 1, 2, \ldots, i-1$ and increments c_i by one.

4 Analysis

4.1 Correctness

By the discussion above, correctness of DELETEMIN is immediate. For INSERT, we must show that the two invariants are maintained and that S_{ic_i} does not overflow when calling SWEEP(i).

After an INSERT, the new contents in the buffers on the path p are the smallest elements in σ, distributed exactly as the old contents. Hence, an element on this path can only be smaller than the element occupying the same location before the operation. It follows that the heap-order invariant is maintained.

The lower part of link i is emptied each time c_i is reset to one. This implies that the invariant requiring $S_{ic_i}, \ldots, S_{ik_i}$ to be empty is maintained. It also implies that the lower part of link i never contains more than the number of elements inserted into $S_{i1}, S_{i2}, \ldots, S_{ik_i}$ by the k_i SWEEP(i) operations occurring since last time c_i was reset. From the definition (1) we by induction on i get $s_i = s_1 + \sum_{j=1}^{i-1} k_j s_j$ for all i. If follows by induction on time that the number of elements inserted into S_{ic_i} during SWEEP(i) is at most s_i.

4.2 Complexity

Most of the work performed is the movement of elements upwards in the tree T during invocations of binary mergers in T. We account for the I/Os incurred during the filling of a buffer by charging them evenly to the elements filled into the buffer, except when an A_i or B_i buffer is not filled completely due to exhaustion, where we account for the I/Os by other means.

We claim that the number of I/Os charged to an element during its ascent in T from an input stream of K_i to the buffer A_1 is $O(\frac{1}{B} \log_{M/B} s_i)$, if we identify elements residing in buffers on the path p at the beginning of SWEEP(i) with those residing at the same positions in these buffers at the end of SWEEP(i).

To prove the claim, we assume that the maximal number of small links are kept in cache always—the optimal cache replacement strategy of the cache oblivious model can only incur fewer I/Os. More precisely, let Δ_i be the space occupied by links 1 to i. From (1) we have ${s_i}^{1/3} \le k_i < 2{s_i}^{1/3}$, so the $\Theta(s_i k_i)$ space usage of $S_{i1}, \ldots, S_{ik_i}$ is $\Theta({k_i}^4)$, which by Lemma 1 dominates the space usage of link i. Also from (1) we have ${s_i}^{4/3} < s_{i+1} < 3{s_i}^{4/3}$, so s_i and k_i grows doubly-exponentially with i. Hence, Δ_i is dominated by the space usage of link i, implying $\Delta_i = \Theta({k_i}^4)$. We let i_M be the largest i for which $\Delta_i \le M$ and assume that links 1 to i_M are kept in cache always.

Consider the ascent of an element from K_i to B_i for $i > i_M$. By Lemma 1, each invocation of the root of K_i incurs $O(k_i + \frac{{k_i}^3}{B} \log_{M/B} {k_i}^3)$ I/Os. From $M < \Delta_{i_M+1}$ and the above discussion, we have $M = O({k_i}^4)$. The tall cache assumption $B^2 \le M$ gives $B = O({k_i}^2)$, which implies $k_i = O({k_i}^3/B)$. As we are not counting invocations of the root of K_i where B_i is not filled completely, i.e. where the root is exhausted, it follows that each element is charged $O(\frac{1}{B} \log_{M/B} {k_i}^3) = O(\frac{1}{B} \log_{M/B} s_i)$ I/Os to ascend through K_i and into B_i.

The element can also be charged during insertion into A_j for $j = i_M, \ldots, i$. The filling of A_j incurs $O(1 + |A_j|/B)$ I/Os. From $B = O({k_{i_M+1}}^2) = O({k_{i_M}}^{8/3})$ and $|A_j| = {k_j}^3$, we see that the last term dominates. Therefore an element is charged $O(1/B)$ per buffer A_j, as we only charge when the buffer is filled completely. From $M = O({k_{i_M+1}}^4) = O({s_{i_M}}^{16/9}) = O(s_{i_M})$, we by the doubly-exponentially growth of s_j get that $i - i_M = O(\log \log_M s_i) = O(\log_M s_i) = O(\log_{M/B} s_i)$. Hence, the ascent through K_i dominates over insertions into A_j for $j = i_M, \ldots, i$, and the claim is proved.

To prove the I/O complexity of our structure stated in the introduction, we note that by induction on i, at least s_i insertions take place between each call to Sweep(i). A call to Sweep(i) inserts at most s_i elements in S_{ic_i}. We let the last s_i insertions preceding the call to Sweep(i) pay for the I/Os charged to these elements during their later ascent through T. By the claim above, this cost is $O(\frac{1}{B} \log_{M/B} s_i)$ I/Os per insertion. We also let these insertions pay for the I/Os incurred by Sweep(i) during the formation and placement of streams σ_1, σ_2, and σ, and for I/Os incurred by filling buffers which become exhausted. We claim that these can be covered without altering the $O(\frac{1}{B} \log_{M/B} s_i)$ cost per insertion.

The claim is proved as follows. The formation of σ_1 is done by a traversal of the path p. By the specified layout of the data structure (including the layout of k-mergers), this traversal is part of a linear scan of the part of memory between A_1 and the end of K_i. Such a scan takes $O((\Delta_{i-1} + |A_i| + |B_i| + |K_i|)/B) = O({k_i}^3/B) = O(s_i/B)$ I/Os. The formation of σ_2 has already been accounted for by charging ascending elements. The merge of σ_1 and σ_2 into σ and the placement of σ are not more costly than a traversal of p and S_{ic_i}, and hence also incur $O(s_i/B)$ I/Os. To account for the I/Os incurred when filling buffers which become exhausted, we note that B_i, and therefore also A_i, can only become exhausted once between each call to Sweep(i). From $|A_i| = |B_i| = {k_i}^3 = \Theta(s_i)$ it follows that charging each call to Sweep(i) an additional cost of $O(\frac{s_i}{B}\log_{M/B} s_i)$ I/Os will cover all such fillings, and the claim is proved.

In summary, charging the last s_i insertions preceding a call to Sweep(i) a cost of $O(\frac{1}{B}\log_{M/B} s_i)$ I/Os each will cover all I/Os incurred by the data structure. Given a sequence of operation on an initial empty priority queue, let $i_{\max}$ be the largest i for which Sweep(i) takes place. We have $s_{i_{\max}} \le N$, where N is the number of insertions in the sequence. An insertion can be charged by at most one call to Sweep(i) for $i = 1, \ldots, i_{\max}$, so by the doubly-exponentially growth of s_i, the number of I/Os charged to an insertion is

$$O\left(\sum_{k=0}^{\infty} \frac{1}{B} \log_{M/B} N^{(3/4)^k}\right) = O\left(\frac{1}{B}\log_{M/B} N\right) .$$

The amortized number of I/Os for a DeleteMin is actually zero, as all occurring I/Os have been charged to insertions.

5 Profile Adaptive Performance

To make the complexity bound depend on N_ℓ, we make the following changes to our priority queue. Let r_i denote the number of elements residing in the lower part of link i. The value of r_i is stored at v_i and will only need to be updated when removing an element from B_i and when a call to Sweep(i) creates a new S_{ij} list (in the later case $r_1, \ldots, r_{i-1}$ are reset to zero).

The only other modification is the following change of the call to Sweep(i). Instead of finding the lowest index i where $c_i \le k_i$, we find the lowest index i where either $c_i \le k_i$ or $r_i \le k_i s_i/2$. If $c_i \le k_i$, Sweep(i) proceeds as described Section 3, and c_i is incremented by one. Otherwise $c_i = k_i + 1$ and $r_i \le k_i s_i/2$, in which case we will recycle one of the S_{ij} buffers. If there exists an input buffer S_{ij} which is empty, we use S_{ij} as the destination buffer for Sweep(i). If all S_{ij} are nonempty, the two input buffers S_{ij_1} and S_{ij_2} with the smallest number of elements contain at most s_i elements in total. Assume without loss of generality $\min S_{ij_1} \ge \min S_{ij_2}$, where $\min S$ denotes the smallest element in stream S. We merge the content of S_{ij_1} and S_{ij_2} into S_{ij_2}. Since $\min S_{ij_1} \ge \min S_{ij_2}$ the heap order remains satisfied. Finally we apply Sweep(i) with S_{ij_1} as the destination buffer.

5.1 Analysis

The correctness follows as in Section 4.1, except that the last induction on time is slightly extended. We must now use that $k_i \ge 2$ implies $k_i s_i/2 + s_i \le k_i s_i$ to argue that Sweep(i) will not make the lower part of link i contain more than $k_i s_i$ elements in the case where $c_i = k_i + 1$ and $r_i \le k_i s_i/2$.

For the complexity, we as in Section 4.2 only have to consider the case where $i > i_M$. We note that in the modified algorithm, the additional number of I/Os required by Sweep(i) for locating and merging S_{ij_1} and S_{ij_2} is $O(k_i + s_i/B)$

I/Os. As seen in Section 4.2, this is dominated by $O(\frac{s_i}{B} \log_{M/B} s_i)$, which is the number of I/Os already charged to Sweep(i) in the analysis.

We will argue that Sweep(i) collects $\Omega(s_i)$ elements from links $1, \ldots, i-1$ that have been inserted since the last call to Sweep(j) with $j \geq i$, and that for half of these elements the value N_ℓ is $\Omega(s_i)$. The claimed amortized complexity $O(\frac{1}{B} \log_{M/B} N_\ell)$ then follows as in Section 4.2, except that we now charge the cost of Sweep(i) to these $\Omega(s_i)$ elements.

The main property of the modified algorithm is captured by the following invariant:

> For each i, the links $1, \ldots, i$ contain in total at most $\sum_{j=1}^{i} |A_j| = \sum_{j=1}^{i} k_j^3$ elements which have been removed from A_{i+1} by the binary merger v_i since the last call to Sweep(j) with $j \geq i+1$.

Here, we after a call to Sweep($i+1$) define all elements in A_j to have been removed from A_ℓ for $1 \leq j < \ell \leq i+1$. When an element e is removed from A_{i+1} by v_i and is output to A_i, then all elements in the lower part of link i must be larger than e. All elements removed from A_{i+1} since the last call to Sweep(j) with $j \geq i+1$ were smaller than e. These elements must either be stored in A_i or have been removed from A_i by the merger in v_{i-1}. It follows that at most $\sum_{j=1}^{i-1} |A_j| + |A_i| - 1$ elements removed from A_{i+1} are present in links $1, \ldots, i$. Hence, the invariant remains valid after moving e from A_{i+1} to A_i. By definition, the invariant remains valid after a call to Sweep(i).

A call to Sweep(i) will create a stream with at least $s_1 + \sum_{j=1}^{i-1} k_j s_j/2 \geq s_i/2$ elements. By the above invariant, at least $t = s_i/2 - \sum_{j=1}^{i-1} |A_j| = s_i/2 - \sum_{j=1}^{i-1} k_j^3 = \Omega(s_i)$ elements must have been inserted since the last call to Sweep(j) with $j \geq i$. Finally, for each of the three definitions of N_ℓ in Section 1 we for at least $t/2$ of the t elements have $N_\ell \geq t/2$, because:

(a) For each of the $t/2$ most recently inserted elements, at least $t/2$ elements were already inserted when these elements where inserted.
(b) For each of the $t/2$ earliest inserted elements, at least $t/2$ other elements have been inserted before they themselves get deleted.
(c) The $t/2$ largest elements each have (maximum) rank at least $t/2$.

This proves the complexity stated in Section 1.

References

1. A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, Sept. 1988.
2. L. Arge. External memory data structures. In *Proc. 9th Annual European Symposium on Algorithms (ESA)*, volume 2161 of *LNCS*, pages 1–29. Springer, 2001.
3. L. Arge, M. A. Bender, E. D. Demaine, B. Holland-Minkley, and J. I. Munro. Cache-oblivious priority queue and graph algorithm applications. In *Proc. 34th Ann. ACM Symp. on Theory of Computing*, pages 268–276. ACM Press, 2002.

4. R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.
5. M. Bender, R. Cole, E. Demaine, and M. Farach-Colton. Scanning and traversing: Maintaining data for traversals in a memory hierarchy. In *Proc. 10th Annual European Symposium on Algorithms (ESA)*, 2002. To appear.
6. M. Bender, R. Cole, and R. Raman. Exponential structures for cache-oblivious algorithms. In *Proc. 29th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 2380 of *LNCS*, pages 195–207. Springer, 2002.
7. M. Bender, E. Demaine, and M. Farach-Colton. Efficient tree layout in a multilevel memory hierarchy. In *Proc. 10th Annual European Symposium on Algorithms (ESA)*, 2002. To appear.
8. M. A. Bender, E. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. In *Proc. 41st Ann. Symp. on Foundations of Computer Science*, pages 399–409. IEEE Computer Society Press, 2000.
9. M. A. Bender, Z. Duan, J. Iacono, and J. Wu. A locality-preserving cache-oblivious dynamic dictionary. In *Proc. 13th Ann. ACM-SIAM Symp. on Discrete Algorithms*, pages 29–39, 2002.
10. G. S. Brodal and R. Fagerberg. Cache oblivious distribution sweeping. In *Proc. 29th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 2380 of *LNCS*, pages 426–438. Springer, 2002.
11. G. S. Brodal, R. Fagerberg, and R. Jacob. Cache oblivious search trees via binary trees of small height. In *Proc. 13th Ann. ACM-SIAM Symp. on Discrete Algorithms*, pages 39–48, 2002.
12. M. J. Fischer and M. S. Paterson. Fishspear: A priority queue algorithm. *Journal of the ACM*, 41(1):3–30, 1994.
13. M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science*, pages 285–297. IEEE Computer Society Press, 1999.
14. J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, June 2001.