# An Optimal Cache-Oblivious Priority Queue and its Application to Graph Algorithms*

Lars Arge†
Dept. of Computer Science
Duke University
large@cs.duke.edu

Michael A. Bender‡
Dept. of Computer Science
SUNY Stony Brook
bender@cs.sunysb.edu

Erik D. Demaine§
Lab. for Computer Science
MIT
edemaine@mit.edu

Bryan Holland-Minkley ¶
Dept. of Computer Science
Duke University
bhm@cs.duke.edu

J. Ian Munro ‖
School of Computer Science
University of Waterloo
imunro@uwaterloo.ca

*Revised paper submitted to SIAM Journal on Computing*

May 5, 2005

**Abstract**

We develop an optimal cache-oblivious priority queue data structure, supporting insertion, deletion, and delete-min operations in $O(\frac{1}{B}\log_{M/B}\frac{N}{B})$ amortized memory transfers, where $M$ and $B$ are the memory and block transfer sizes of any two consecutive levels of a multilevel memory hierarchy. In a cache-oblivious data structure, $M$ and $B$ are not used in the description of the structure. Our structure is as efficient as several previously developed external memory (cache-aware) priority queue data structures, which all rely crucially on knowledge about $M$ and $B$. Priority queues are a critical component in many of the best known external memory graph algorithms, and using our cache-oblivious priority queue we develop several cache-oblivious graph algorithms.

## 1 Introduction

As the memory systems of modern computers become more complex, it is increasingly important to design algorithms that are sensitive to the structure of memory. One of the characteristic features

*An extended abstract version of this paper was presented at the 2002 ACM Symposium on Theory of Computation (STOC'02) [12].

†Supported in part by the National Science Foundation through ESS grant EIA–9870734, RI grant EIA–9972879, CAREER grant CCR–9984099, and ITR grant EIA–0112849.

‡Supported in part by the National Science Foundation through ITR grant EIA–0112849 and by HRL Laboratories and Sandia National Laboratories.

§Supported in part by the National Science Foundation through ITR grant EIA–0112849.

¶Supported in part by the National Science Foundation through ITR grant EIA–0112849.

‖Supported in part by the Natural Science and Engineering Council through grant RGPIN 8237-97 and the Canada Research Chair in Algorithm Design.

of modern memory systems is that they are made up of a hierarchy of several levels of cache, main memory, and disk. While traditional theoretical computational models have assumed a "flat" memory with uniform access time, the access times of different levels of memory can vary by several orders of magnitude in current machines. Thus algorithms for hierarchical memory has received considerable attention in recent years. Very recently, the *cache-oblivious* model was introduced as a way of achieving algorithms that are efficient in arbitrary memory hierarchies without use of complicated multilevel memory models. In this paper we develop an optimal cache-oblivious priority queue and use it to develop several cache-oblivious graph algorithms.

## 1.1 Background and previous results

Traditionally, most algorithmic work has been done in the *Random Access Machine* (RAM) model of computation, which models a "flat" memory with uniform access time. Recently, some attention has turned to the development of theoretical models and algorithms for modern complicated hierarchical memory systems; refer e.g. to [3, 4, 5, 7, 54, 60]. Developing models that are both simple and realistic is a challenging task since a memory hierarchy is described by many parameters. A typical hierarchy consists of a number of memory levels, with memory level $\ell$ having size $M_\ell$ and being composed of $M_\ell/B_\ell$ blocks of size $B_\ell$. In any memory transfer from level $\ell$ to $\ell-1$, an entire block is moved atomically. Each memory level also has an associated *replacement strategy*, which is used to decide what block to remove in order to make room for a new block being brought into that level of the hierarchy. Further complications are the limited *associativity* of some levels of the hierarchy, meaning that a given block can only be loaded into a limited number of memory positions, as well as the complicated *prefetching strategies* employed by many memory systems. In order to avoid the complications of multilevel memory models, a body of work has focused on two-level memory hierarchies. Most of this work has been done in the context of problems involving massive datasets, because the extremely long access times of disks compared to the other levels of the hierarchy means that I/O between main memory and disk often is the bottleneck in such problems.

### 1.1.1 Two-level I/O model

In the two-level *I/O model* (or *external memory model*) introduced by Aggarwal and Vitter [6], the memory hierarchy consists of an internal memory of size $M$, and an arbitrarily large external memory partitioned into blocks of size $B$. The efficiency of an algorithm in this model (a so-called *I/O* or *external memory* algorithm) is measured in terms of the number of block transfers it performs between these two levels (here called *memory transfers*). An algorithm has complete control over the placement of blocks in main memory and on disk. The simplicity of the I/O model has resulted in the development of a large number of external memory algorithms and techniques. See e.g. [10, 60] for recent surveys.

The number of memory transfers needed to read $N$ contiguous elements from disk is $scan(N) = \Theta(\frac{N}{B})$ (the *linear* or *scanning* bound). Aggarwal and Vitter[6] showed that $\Theta(\frac{N}{B}\log_{M/B}\frac{N}{B})$ memory transfers are necessary and sufficient to sort $N$ elements. In this paper, we use $\mathrm{sort}(N)$ to denote $\frac{N}{B}\log_{M/B}\frac{N}{B}$ (the *sorting* bound). The number of memory transfers needed to search for an element among a set of $N$ elements is $\Omega(\log_B N)$ (the *searching* bound) and this bound is matched by the B-tree, which also supports updates in $O(\log_B N)$ memory transfers [18, 37, 41, 40]. An important consequence of these bounds is that, unlike in the RAM model, one cannot sort optimally with a search tree—inserting $N$ elements in a B-tree takes $O(N\log_B N)$ memory transfers, which is a factor of $(B \log_B N)/(\log_{M/B} \frac{N}{B})$ from optimal. Finally, permuting $N$ elements according to a given permutation takes $\Theta(\min\{N, \mathrm{sort}(N)\})$ memory transfers and for all practical values of $N, M$ and $B$ this is $\Theta(\mathrm{sort}(N))$ [6]. This represents another fundamental difference between the RAM and I/O model, since $N$ elements can be permuted in $O(N)$ time in the RAM model.

The I/O model can also be used to model the two-level hierarchy between cache and main memory; refer e.g. to [55, 43, 45, 16, 51, 53]. Some of the main shortcomings of the I/O model on this level is the lack of explicit application control of placement of data in the cache and the low associativity of many caches. However, as demonstrated by Sen et al. [55], I/O model results can often be used to obtain results in more realistic two-level cache main memory models. Algorithms that are efficient on the two-level cache main memory levels have also been considered by e.g. LaMarca and Ladner [43, 44].

### 1.1.2 Cache-oblivious model

One of the main disadvantages of two-level memory models is that they force the algorithm designer to focus on a particular level of the hierarchy. Nevertheless, the I/O model has been widely used because it is convenient to consider only two levels of the hierarchy. Very recently, a new model that combines the simplicity of the I/O mode with the realism of more complicated hierarchical models was introduced by Frigo et al. [39]. The idea in the *cache-oblivious model* is to design and analyze algorithms in the I/O model but without using the parameters $M$ and $B$ in the algorithm description. It is assumed that when an algorithm accesses an element that is not stored in main memory, the relevant block is automatically fetched into memory with a *memory transfer*. If the main memory is full, the *ideal* block in main memory is elected for replacement based on the future characteristics of the algorithm, that is, an *optimal* offline paging strategy is assumed. While this model may seem unrealistic, Frigo et al. [39] showed that it can be simulated by essentially any memory system with only a small constant-factor overhead. For example, the least-recently-used (LRU) block-replacement strategy approximates the omniscient strategy within a constant factor, given a cache larger by a constant factor [39, 56]. The main advantage of the cache-oblivious model is that it allows us to reason about a simple two-level memory model, but prove results about an unknown, multilevel memory hierarchy; because an analysis of an algorithm in the two-level model holds for any block and main memory size, it holds for *any* level of the memory hierarchy. As a consequence, if the algorithm is optimal in the two-level model, it is optimal on *all* levels of a multilevel memory hierarchy.

Frigo et al. [39] developed optimal cache-oblivious algorithms for matrix multiplication, matrix transposition, Fast Fourier Transform, and sorting. Subsequently, several authors developed dynamic cache-oblivious B-trees with a search and update cost of $O(\log_B N)$ matching the standard (cache-aware) B-tree [22, 28, 23, 50, 21]. Recently, several further results have been obtained, e.g. [24, 58, 25, 26, 27, 19, 2, 14, 17, 20, 29, 34]. See also the survey in [13]. Some of these results assume that $M \geq B^2$ (the *tall-cache* assumption), and we will also make the assumption in this paper. Brodal and Fagerberg [27] showed that an assumption of this type (actually $M = \Omega(B^{1+\epsilon})$ for some $\epsilon > 0$) is necessary to obtain the I/O-model sorting bound in the cache-oblivious model.

### 1.1.3 Priority queues

A priority queue maintains a set of elements each with a priority (or key) under *insert* and *delete-min* operations, where a delete-min operation finds and deletes the element with the minimum key in the queue. Sometimes *delete* of an arbitrary element is also supported, often, as in this paper, assuming that the key of the element to be deleted is known. The heap is a standard implementation of a priority queue and a balanced search tree can, of course, also easily be used to implement a priority queue. In the I/O model, a priority queue based on a B-tree would support all operations in $O(\log_B N)$ memory transfers. The standard heap can also be easily modified (to have fanout $B$) so that all operations are supported in the same bound (see e.g. [45]). The existence of a cache-oblivious B-tree immediately implies the existence of an $O(\log_B N)$ cache-oblivious priority queue.

As discussed in Section 1.1.1, the use of an $O(\log_B N)$ search tree (or priority queue) to sort $N$ elements results in an I/O model algorithm that is a factor of $(B \log_B N)/(\log_{M/B}(N/B))$ from optimal. To sort optimally we need a data structure supporting the relevant operations in $O(\frac{1}{B} \log_{M/B} \frac{N}{B})$ memory transfers. Note that for reasonable values of $N$, $M$, and $B$, this bound is less than 1 and we can, therefore, only obtain it in an amortized sense. To obtain such a bound, Arge developed the *buffer tree technique* [11] and showed how it can be used on a B-tree to obtain a priority queue supporting all operations in $O(\frac{1}{B} \log_{M/B} \frac{N}{B})$ amortized memory transfers [11]. This structure seems hard to make cache-oblivious since it involves periodically finding the $\Theta(M)$ smallest key elements in the structure and storing them in internal memory. Efficient I/O model priority queues have also been obtained by using the buffer tree technique on heap structures [38, 42]: The heap structure by Fadel et al. [38] seems hard to make cache-oblivious because it requires collecting $\Theta(M)$ insertions and performing them all on the structure at the same time. The structure by Kumar and Schwabe [42] avoids this by using the buffer technique on a tournament tree data structure. However, they only obtained $O(\frac{1}{B} \log_2 N)$ bounds, mainly because their structure was designed to also support an *update* operation. Finally, Brodal and Katajainen [30] developed a priority queue structure based on a $M/B$-way merging scheme. Very recently, and after the appearance of the conference version of this paper, Brodal and Fagerberg [26] managed to develop a similar merging based cache-oblivious priority queue based on ideas they developed in [25]. Brodal et al. [29], as well as Chowdhuey and Ramachandran [34], have also developed cache-oblivious priority queues that support updates in the same bound as the I/O-efficient structure of Kumar and Schwabe [42].

### 1.1.4 I/O model graph algorithms

The super-linear lower bound on permutation in the I/O model has important consequences for the I/O-complexity of graph algorithms, because the solution of almost any graph problem involves somehow permuting the $V$ vertices or $E$ edges of the graph. Thus $\Omega(\min\{V, \text{sort}(V)\})$ is in general a lower bound on the number of memory transfers needed to solve most graph problems. Refer to [9, 32, 47]. As mentioned in Section 1.1.1, this bound is $\Omega(\text{sort}(V))$ in all practical cases. Still, even though a large number of I/O model graph algorithms have been developed (see [60, 61] and references therein), not many algorithms match this bound. Below we review the results most relevant to our work.

Like for PRAM graph algorithms [52], list ranking—the problem of ranking the elements in a linked list stored as an unordered sequence in memory—is the most fundamental I/O model graph problem. Using PRAM techniques, Chiang et al. [32] developed the first efficient I/O model list ranking algorithm. Using an I/O-efficient priority queue, Arge [11] showed how to solve the problem in $O(\text{sort}(V))$ memory transfers. The list ranking algorithm and PRAM techniques can be used in the development of $O(\text{sort}(V))$ algorithms for many problems on trees, such as computing an Euler Tour, Breadth-First-Search (BFS), Depth-First-Search (DFS), and centroid decomposition [32].

The best known DFS and BFS algorithms for general directed graphs use $O(V + \frac{EV}{BM})$ [32] or $O((V + E/B)\log_2 V + \text{sort}(E))$ [31] memory transfers. For undirected graphs, improved $O(V + \text{sort}(E))$ and $O(\sqrt{\frac{V \cdot E}{B}} + \text{sort}(E))$ BFS algorithms have been developed [47, 46]. The best known algorithms for computing the connected components and the minimum spanning forest of a general undirected graph both use $O(\text{sort}(E) \cdot \log_2 \log_2(\frac{VB}{E}))$ or $O(V + \text{sort}(E))$ memory transfers [47, 15].

### 1.2 Our results

The main result of this paper is an optimal cache-oblivious priority queue. Our structure supports insert, delete, and delete-min operations in $O(\frac{1}{B}\log_{M/B}\frac{N}{B})$ amortized memory transfers and $O(\log N)$ amortized computation time; it is described in Section 2. The structure is based on a combination of several new ideas with ideas used in previous recursively defined cache-oblivious algorithms and data structures [39, 22], the buffer technique of Arge [11, 42], and the $M/B$-way merging scheme utilized by Brodal and Katajainen [30]. When the conference version of this paper appeared, our structure was the only cache-oblivious priority queue to obtain the same bounds as in the cache-aware case.

In the second part of the paper, Section 3, we use our priority queue to develop several cache-oblivious graph algorithms. We first show how to solve the list ranking problem in $O(\text{sort}(V))$ memory transfers. Using this result we develop $O(\text{sort}(V))$ algorithms for fundamental problems on trees, such as the Euler Tour, BFS, and DFS problems. The complexity of all of these algorithms matches the complexity of the best known cache-aware algorithms. Next we consider DFS and BFS on general graphs. Using modified versions of the data structures used in the $O((V + E/B)\log_2 V + \text{sort}(E))$ DFS and BFS algorithms for directed graphs [31], we make these algorithms cache-oblivious. We also discuss how the $O(V + \text{sort}(E))$ BFS algorithm for undirected graphs [47] can be made cache-oblivious. Very recently, and after the appearance of the conference version of this paper, Brodal et al. [29] developed two other cache-oblivious algorithms for undirected BFS based on the ideas in [46]. Finally, we develop two cache-oblivious algorithms for computing a minimum spanning forest (MSF), and thus also for computing connected components, of an undirected graph using $O(\text{sort}(E) \cdot \log_2 \log_2 V)$ and $O(V + \text{sort}(E))$ memory transfers, respec-

| **Problem** | **Our cache-oblivious result** | **Best cache-aware result** | |
|---|---|---|---|
| Priority queue | $O(\frac{1}{B}\log_{M/B}\frac{N}{B})$ | $O(\frac{1}{B}\log_{M/B}\frac{N}{B})$ | [11] |
| List ranking | $O(\text{sort}(V))$ | $O(\text{sort}(V))$ | [32, 11] |
| Tree algorithms | $O(\text{sort}(V))$ | $O(\text{sort}(V))$ | [32] |
| Directed BFS and DFS | $O((V + E/B)\log_2 V + \text{sort}(E))$ | $O((V + E/B)\log_2 V + \text{sort}(E))$ | [31] |
| | | $O(V + \frac{EV}{BM})$ | [32] |
| Undirected BFS | $O(V + \text{sort}(E))$ | $O(V + \text{sort}(E))$ | [47] |
| | | $O(\sqrt{\frac{V \cdot E}{B}} + \text{sort}(E))$ | [46] |
| Minimum spanning forest | $O(\text{sort}(E) \cdot \log_2 \log_2 V)$ | $O(\text{sort}(E) \cdot \log_2 \log_2 \frac{VB}{E})$ | [15] |
| | $O(V + \text{sort}(E))$ | $O(V + \text{sort}(E))$ | [15] |

Table 1: Summary of our results (priority queue bounds are amortized).

tively. The two algorithms can be combined to compute the MSF in $O(\text{sort}(E) \cdot \log_2 \log_2 \frac{V}{V'} + V')$ memory transfers for any $V'$ independent of $B$ and $M$. Table 1 summarizes our results. Note that recently, cache-oblivious algorithm for undirected shortest path computation have also been developed [29, 34].

# 2 Priority Queue

In this section we describe our optimal cache-oblivious priority queue. In Section 2.1 we define the data structure and in Section 2.2 we describe the supported operations.

## 2.1 Structure

### 2.1.1 Levels

Our priority queue data structure containing $N$ elements consists of $\Theta(\log\log N_0)$ *levels* whose sizes vary from $N_0 = \Theta(N)$ to some small size $c$ beneath a constant threshold $c_t$. The size of a level corresponds (asymptotically) to the number of elements that can be stored within it. The $i$th level from above has size $N_0^{(2/3)^{i-1}}$ and for convenience we refer to the levels by their size. Thus the levels from largest to smallest are level $N_0$, level $N_0^{2/3}$, level $N_0^{4/9}$, ..., level $X^{9/4}$, level $X^{3/2}$, level $X$, level $X^{2/3}$, level $X^{4/9}$, ..., level $c^{9/4}$, level $c^{3/2}$, and level $c$. Intuitively, smaller levels store elements with smaller keys or elements that were recently inserted. In particular, the element with minimum key and the most recently inserted element in the structure are in the smallest (lowest) level $c$. Both insertions and deletions are initially performed on the smallest level and may propagate up through the levels.

### 2.1.2 Buffers

A level stores elements in a number of *buffers*, which are also used to transfer elements between levels. Refer to Figure 1. Level $X^{3/2}$ consists of one *up buffer* $u^{X^{3/2}}$ and at most $\lceil X^{1/2} \rceil + 1$ *down buffers* $d_1^{X^{3/2}}, \ldots, d_{\lceil X^{1/2} \rceil + 1}^{X^{3/2}}$. The up buffer can store up to $\lfloor X^{3/2} \rfloor$ elements and the first down buffer can store up to $2\lfloor X \rfloor - 1$ elements, while each of the other down buffers can store between $\lfloor X \rfloor$ and $2\lfloor X \rfloor - 1$ elements. We refer to the maximum possible number of elements that can be

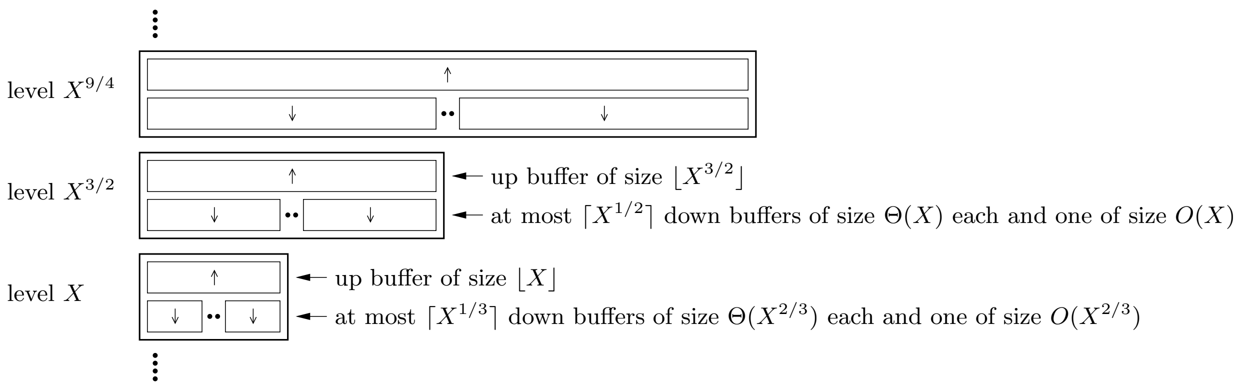


Figure 1: Levels $X$, $X^{3/2}$, and $X^{9/4}$ of the priority queue data structure.

stored in a buffer or level as its *size*; we refer to the number of elements currently in a buffer or level as the *occupancy*. Thus the size of level $X^{3/2}$ is $\Theta(X^{3/2})$. Note that the size of a down buffer at one level matches the size (up to a constant factor) of the up buffer one level down.

We maintain three invariants about the relationships between the elements in buffers of various levels:

**Invariant 1** *At level $X^{3/2}$, elements are sorted* **among** *the down buffers, that is, elements in $d_i^{X^{3/2}}$ have smaller keys than elements in $d_{i+1}^{X^{3/2}}$, but the elements within $d_i^{X^{3/2}}$ are unordered.*

The element with largest key in each down buffer $d_i^{X^{3/2}}$ is called a *pivot element*. Pivot elements mark the boundaries between the ranges of the keys of elements in down buffers.

**Invariant 2** *At level $X^{3/2}$, elements in the down buffers have smaller keys than the elements in the up buffer $u^{X^{3/2}}$.*

**Invariant 3** *The elements in the down buffers at level $X^{3/2}$ have smaller keys than the elements in the down buffers at the next higher level $X^{9/4}$.*

The three invariants ensure that the keys of the elements in the down buffers get larger as we go from smaller to larger levels of the structure. Furthermore, there is an order between the buffers on one level; keys of elements in the up buffer are larger than keys of elements in down buffers. Therefore, down buffers are drawn below up buffers in Figure 1. However, the keys of the elements in an up buffer are unordered relative to the keys of the elements in down buffers one level up. Intuitively, up buffers store elements that are "on their way up", that is, they have yet to be resolved as belonging to a particular down buffer in the next (or higher) level. Analogously, down buffers store elements that are "on their way down"—these elements are partitioned into several clusters so that we can quickly find the cluster of elements with smallest keys of size roughly equal to the next level down. In particular, the element with overall minimum key is in the first down buffer at level $c$.

### 2.1.3 Layout

We store the priority queue in a linear array as follows. The levels are stored consecutively from smallest to largest with each level occupying a single region of memory. For level $X^{3/2}$ we reserve space for the up buffers of size $\lfloor X^{3/2} \rfloor$ and for $\lceil X^{1/2} \rceil + 1$ possible down buffers of size $2\lfloor X \rfloor$. The up buffer is stored first, followed by the down buffers stored in an arbitrary order but linked together to form an ordered linked list. Thus the total size of the array is $\sum_{i=0}^{\log_{3/2} \log_c N_0} O(N_0^{(2/3)^i}) = O(N_0)$.

## 2.2 Operations

To implement the priority queue operations we will use two general operations, *push* and *pull*. Push inserts $\lfloor X \rfloor$ elements (with larger keys than all elements in the down buffers of level $X$) into level $X^{3/2}$, and pull removes and returns the $\lfloor X \rfloor$ elements with smallest keys from level $X^{3/2}$ (and above). Generally, whenever an up buffer on level $X$ overflows we push the $\lfloor X \rfloor$ elements in the buffer into level $X^{3/2}$, and whenever the down buffers on level $X$ become too empty we pull $\lfloor X \rfloor$ elements from level $X^{3/2}$.

### 2.2.1 Push

We push $\lfloor X \rfloor$ elements (with larger keys than all elements in the down buffers of level $X$) into level $X^{3/2}$ as follows: We first sort the elements. Then we distribute them into the down buffers of level $X^{3/2}$ by scanning through the sorted list and simultaneously visiting the down buffers in (linked) order. More precisely, we append elements to the end of the current down buffer $d_i^{X^{3/2}}$, and advance to the next down buffer $d_{i+1}^{X^{3/2}}$ as soon as we encounter an element with larger key than the pivot of $d_i^{X^{3/2}}$. Elements with larger keys than the pivot of the last down buffer are inserted in the up buffer $u^{X^{3/2}}$. During the distribution of elements a down buffer may become overfull, that is, contain $2\lfloor X \rfloor$ elements. In this case, we split the buffer into two down buffers each containing $\lfloor X \rfloor$ elements. If the level has at most $\lceil X^{1/2} \rceil + 1$ down buffers after the split, we place the new buffer in any free down-buffer spot for the level and update the linked list accordingly. Otherwise, we first remove the last down buffer by moving its at most $2\lfloor X \rfloor - 1$ elements into the up buffer; then we place the new buffer in the free down-buffer spot and update the linked list. If the up buffer runs full during the process, that is, contains more than $\lfloor X^{3/2} \rfloor$ elements, we recursively *push* $\lfloor X^{3/2} \rfloor$ elements into level $X^{9/4}$ (the next level up), leaving at most $\lfloor X^{3/2} \rfloor$ elements in the up buffer.

The invariants are all maintained during a push of $\lfloor X \rfloor$ elements into level $X^{3/2}$. Because we sort the elements to distribute them among the down buffers, it is clear we maintain Invariant 1. Only elements with keys larger than the pivot of the last down buffer are placed in the up buffer, so Invariant 2 is also maintained. Finally, Invariant 3 is maintained since (by definition) the $\lfloor X \rfloor$ elements all have keys larger than the elements in the down buffers of level $X$, and since by Invariant 2 all recursively pushed elements have keys larger than all elements in the down buffers (as required to perform the recursive push).

Ignoring the cost of recursive push operations, a push of $\lfloor X \rfloor$ elements into level $X^{3/2}$ can be performed cache-obliviously in $O(\frac{X}{B} \log_{M/B} \frac{X}{B})$ memory transfers and $O(X \log_2 X)$ time using $O(M)$ of main memory. First note that since $M = \Omega(B^2)$ (the tall-cache assumption), all levels of size less than $B^2$ (of total size $O(B^2)$) fit in memory. If all these levels are kept in main memory at all times, all push costs associated with them would be eliminated. The optimal paging strategy is able to do so. Thus we only need to consider push costs when $X^{3/2} > B^2$, that is, when $X > B^{4/3}$ (note that in that case $\frac{X}{B} > 1$). When performing the push operation, the initial sort of the $\lfloor X \rfloor$ elements can then be performed cache-obliviously using $O(\frac{X}{B} \log_{M/B} \frac{X}{B})$ memory transfers and $O(X \log_2 X)$ time [39]. The scan of the elements in the distribution step then takes $O(X/B)$ memory transfers and $O(X)$ time. However, even though we do not insert elements in every down buffer, we still might perform a memory transfer for each of the $\lceil X^{1/2} \rceil + 1$ possible buffers; a block of each buffer may have to be loaded and written back without transferring a full block of elements into the buffer. If $X \geq B^2$ we trivially have that $\lceil X^{1/2} \rceil + 1 = O(\frac{X}{B})$. If, on the other hand, $B^{4/3} < X < B^2$ the $\lceil X^{1/2} \rceil + 1$ term can dominate the memory transfer bound and we have to analyze the cost more carefully. In this case we are working on a level $X^{3/2}$ where $B^2 \leq X^{3/2} < B^3$. There is only one such level and because $X^{1/2} \leq B$ and $M = \Omega(B^2)$ (the tall-cache assumption), a block for each of its down buffers can fit into main memory. Consequently, if a fraction of the main memory is used to keep a partially filled block of each buffer of level $X^{3/2}$ ($B^2 \leq X^{3/2} \leq B^3$) in memory at all times, and full blocks are written to disk, the $X^{1/2} + 1$ cost is eliminated at this level. The optimal paging strategy is able to do so. Thus the total cost of distributing the $\lfloor X \rfloor$ elements is $O(X/B)$ memory transfers and $O(X + X^{1/2}) = O(X)$ time.

The split of an overfull down buffer during the distribution, that is, split of a buffer of occupancy $2\lfloor X \rfloor$, can be performed in $O(X/B)$ memory transfers and $O(X)$ time by first finding the median of the elements in the buffer in $O(X/B)$ transfers and $O(X)$ time [39], and then partitioning the elements into the two new buffers of occupancy $\lfloor X \rfloor$ in a simple scan. Since we maintain that any new down buffer has occupancy $\lfloor X \rfloor$, and thus that $\lfloor X \rfloor$ elements have to be inserted it it before it splits, the amortized splitting cost per element is $O(1/B)$ transfers and $O(1)$ time. Thus in total, the amortized number of memory transfers and time used on splitting buffers while distributing the $\lfloor X \rfloor$ elements are $O(X/B)$ and $O(X)$, respectively.

**Lemma 1** *Using $O(M)$ main memory, a push of $\lfloor X \rfloor$ elements into level $X^{3/2}$ can be performed in $O(\frac{X}{B} \log_{M/B} \frac{X}{B})$ memory transfers and $O(X \log_2 X)$ amortized time, not counting the cost of any recursive push operations, while maintaining Invariants 1–3.*

### 2.2.2 Pull

To pull the $\lfloor X \rfloor$ elements with smallest keys from level $X^{3/2}$ (and above), we consider three different cases.

If the occupancy of the first down buffer of level $X^{3/2}$ is $Y \geq \lfloor X \rfloor$, we sort the $Y < 2\lfloor X \rfloor$ elements in the down buffer, remove the $\lfloor X \rfloor$ elements with smallest keys, and leave the remaining $Y - \lfloor X \rfloor$ elements in the buffer. We return the $\lfloor X \rfloor$ removed elements, since by Invariants 1–3 they are the elements with smallest keys in level $X^{3/2}$ (and above). It is easy to see that Invariants 1–3 are maintained is this case.

If the occupancy of the first down buffer of level $X^{3/2}$ is $Y < \lfloor X \rfloor$, but level $X^{3/2}$ has at least one other down buffer, we first remove the $Y$ elements in the first buffer. Next we sort the between $\lfloor X \rfloor$ and $2\lfloor X \rfloor - 1$ elements in the new first down buffer, remove the $\lfloor X \rfloor - Y$ elements with smallest keys, and leave the remaining elements in the buffer. We return the $\lfloor X \rfloor$ removed elements, since by Invariants 1–3 they are the elements with smallest keys in level $X^{3/2}$. As in the first case, it is easy to see that Invariants 1–3 are maintained.

Finally, if the occupancy of the first down buffer of level $X^{3/2}$ is $Y < \lfloor X \rfloor$ and level $X^{3/2}$ has no other down buffers, we remove the $Y$ elements and then we recursively *pull* the $\lfloor X^{3/2} \rfloor$ elements with smallest keys from level $X^{9/4}$ (and above). Because these $\lfloor X^{3/2} \rfloor$ elements do not necessarily have smaller keys than the $U$ elements in the up buffer $u^{X^{3/2}}$, we then sort all the $\lfloor X^{3/2} \rfloor + U \leq 2\lfloor X^{3/2} \rfloor$ elements, insert the $U$ elements with largest keys in the up buffer, and remove the $\lfloor X \rfloor - Y$ elements with smallest keys. Finally, we distribute the remaining $\lfloor X^{3/2} \rfloor + Y - \lfloor X \rfloor < \lfloor X^{3/2} \rfloor \leq X \cdot X^{1/2} \leq (\lfloor X \rfloor + 1) \cdot X^{1/2} = \lfloor X \rfloor \cdot X^{1/2} + X^{1/2} \leq \lfloor X \rfloor \cdot \lceil X^{1/2} \rceil + \lfloor X \rfloor$ elements into one down buffer with occupancy between 1 and $\lfloor X \rfloor$ and at most $\lceil X^{1/2} \rceil$ down buffers of occupancy $\lfloor X \rfloor$ each. As in the first two cases, we return the $\lfloor X \rfloor$ removed elements, since Invariant 1 and sorting the recursively pulled elements and the $U$ elements in the up buffer ensures that they are the elements with the smallest keys in level $X^{3/2}$ and above. As in the first two cases, it is easy to see that Invariants 1–3 are also maintained in this case.

To analyze a pull operation we assume, as in the push case, that all levels of size $O(B^2)$ are kept in main memory (so that we only need to consider levels $X^{3/2}$ with $X^{3/2} > B^2$, that is, the case where $X > B^{4/3}$). The first two cases are both performed by sorting and scanning $O(X)$ elements using $O(\frac{X}{B} \log_{M/B} \frac{X}{B}) + O(\frac{X}{B})$ memory transfers and $O(X \log_2 X) + O(X)$ time. In the third case we also sort and scan (distribute) $O(X^{3/2})$ elements. However, the cost of doing so is dominated by the cost of the recursive pull operation itself. Thus, ignoring these costs (charging them to the recursive pull), we have the following:

**Lemma 2** *Using $O(M)$ main memory, a pull of $\lfloor X \rfloor$ elements from level $X^{3/2}$ can be performed in $O(\frac{X}{B} \log_{M/B} \frac{X}{B})$ memory transfers and $O(X \log_2 X)$ amortized time, not counting the cost of any recursive pull operations, while maintaining Invariants 1–3.*

### 2.2.3 Insert and Delete-Min

To support insert and delete-min operations using push and pull on our structure we (or rather, the optimal paging strategy) maintain an insertion and a deletion buffer of at most $\lfloor c^{2/3} \rfloor$ elements each in main memory. The deletion buffer contains elements that have smaller keys than all other elements in the structure, while intuitively the insertion buffer contains the most recently inserted elements. We perform an insertion simply by comparing the key of the element to be inserted with the maximal key of an element in the deletion buffer: if the key of the new element is largest, we simply insert it into the insertion buffer; otherwise we instead insert it into the deletion buffer and move the element with largest key from the deletion buffer to the insertion buffer. In both cases, the occupancy of the insertion buffer is increased by one, and if it runs full we empty it by pushing its $\lfloor c^{2/3} \rfloor$ elements into level $c$. Similarly, we perform a delete-min by deleting and returning the element with smallest key in the deletion buffer; if the deletion buffer becomes empty we pull $\lfloor c^{2/3} \rfloor$ elements from level $c$ and fill up the deletion buffer with the $\lfloor c^{2/3} \rfloor$ smallest of these elements and the elements in the insertion buffer (without changing the occupancy of the insertion buffer). The correctness of the insert and delete-min operations follow directly from Invariants 1–3 and the definition of the push and pull operations.

Except for the possible push and pull operations on level $c$, which may require recursive pushes or pulls on higher levels, the insert and delete-min operations are performed in constant time without incurring any memory transfers. Below we will use a credit argument to prove that including all push and pull operations the amortized cost of an insert or delete-min is $O(\frac{1}{B} \log_{M/B} \frac{N_0}{B})$ memory transfers; then we will argue that the operations take $O(\log_2 N_0)$ amortized computation time.

We define a *level-$X$ push coin* and a *level-$X^{3/2}$ pull coin* to be worth $\Theta(\frac{1}{B} \log_{M/B} \frac{X}{B})$ memory transfers each, that is, $\lfloor X \rfloor$ level-$X$ push coins can pay for a push of $\lfloor X \rfloor$ elements into level $X^{3/2}$, and $\lfloor X \rfloor$ level-$X^{3/2}$ coins can pay for a pull of $\lfloor X \rfloor$ elements from level $X^{3/2}$. We maintain the following *coin invariants*:

**Invariant 4** *On level $X^{3/2}$, each element in the first half of a down buffer has a pull coin for level $X^{3/2}$ and each level below.*

**Invariant 5** *On level $X^{3/2}$, each element in the second half of a down buffer and in the up buffer has a push coin for level $X^{3/2}$ and each level above, as well as pull coins for all levels.*

**Invariant 6** *Each element in the insertion buffer has a push and a pull coin for each level.*

Intuitively, the first two coin invariants mean that an element in the first half of a down buffer can pay for being pulled down through all lower levels, while elements in the second half of a down buffer and in an up buffer can pay for being pushed up through all higher levels and pulled down through all levels.

To fulfill Invariant 6, we simply have to give each inserted element a push and a pull coin for each level, since an insert operation increases the occupancy of the insertion buffer by one and a delete-min does not change the occupancy. We will show that we can then pay for all recursive push and pull operations with released coins while maintaining Invariants 4 and 5. Thus a delete-min is free amortized and an insertion costs $O(\sum_{i=0}^{\infty} \frac{1}{B} \log_{M/B}(N_0^{(2/3)^i}/B)) = O(\frac{1}{B} \log_{M/B} \frac{N_0}{B})$ amortized memory transfers.

First consider an insertion that results in a sequence of push operations. We will show that we can maintain the coin invariants while paying for each push operation with released coins, if we require that when pushing $\lfloor X \rfloor$ elements into level $X^{3/2}$, each of the pushed elements has a push coin for level $X$ and each level above, as well as a pull coin for all levels (the *push requirement*). Note that Invariants 5 and 6 ensure that the push requirement is fulfilled, since the pushed elements come from the insertion buffer or the up buffer of level $X$.

When performing a push on level $X^{3/2}$ we first distribute the $\lfloor X \rfloor$ elements into the down and the up buffers. In the worst case (when all elements are placed in the second half of a down buffer or in the up buffer) each element needs a push coin for level $X^{3/2}$ and each level above and pull coins for all levels to fulfill Invariants 4 and 5. Since they have a push coin for level $X$ and each level above and pull coins for all levels, this leaves us with $\lfloor X \rfloor$ level-$X$ push coins, which we can use to pay the $O(\frac{X}{B} \log_{M/B} \frac{X}{B})$ push cost (Lemma 1). If a down buffer of occupancy $2\lfloor X \rfloor$ splits into two buffers of occupancy $\lfloor X \rfloor$ during the distribution process, $\lfloor X \rfloor$ push coins for level $X^{3/2}$ and each level above and $\lfloor X \rfloor$ pull coins for level $X^{9/4}$ and each level above are released, since the $\lfloor X \rfloor$ elements in the second half of the full buffer must fulfill Invariant 5 before the split but only Invariant 4 after the split. On the other hand, if splitting a down buffer results in the movement of the elements in the last down buffer to the up buffer, the at most $\lfloor X \rfloor$ elements in the second half of the down buffer needs $\lfloor X \rfloor$ push coins for level $X^{3/2}$ and each level above and $\lfloor X \rfloor$ pull coins for level $X^{9/4}$ and each level above, since they must fulfill Invariant 5 after the move but only Invariant 4 before the move. Since we never move elements from a down buffer to the up buffer without having split a down buffer, we can reestablish Invariant 4 and 5 with the released coins. Finally, as mentioned above, if the up buffer runs full and we perform a recursive push of $\lfloor X^{3/2} \rfloor$ elements into level $X^{9/4}$, each of the pushed elements has a push coin for level $X^{3/2}$ and each level above, as well as pull coins for all levels, as required (Invariant 5).

Next consider a delete-min that results in a sequence of pull operations. We will show that we can maintain the coin invariants while paying for each pull operation with released coins, if we require that when pulling $\lfloor X \rfloor$ elements from level $X^{3/2}$ down to level $X$, each of the pulled elements has a pull coin for level $X$ and each level below (the *pull requirement*).

When performing a pull on level $X^{3/2}$ with at least $\lfloor X \rfloor$ elements in the down buffers (the first two cases), we effectively remove the $\lfloor X \rfloor$ smallest elements from the first two down buffers. It is straightforward to see that the remaining elements still fulfill Invariants 4 and 5. From Invariant 4 and 5 we know that each of the removed (pulled) elements (at least) has a pull coin for level $X^{3/2}$ and each level below. Thus, since they only need a pull coin for level $X$ and each level below to fulfill the pull requirement, this leaves us with $\lfloor X \rfloor$ level-$X^{3/2}$ pull coins, which we can use to pay the $O(\frac{X}{B} \log_{M/B} \frac{X}{B})$ pull cost (Lemma 2). If, on the other hand, level $X^{3/2}$ contains $Y < \lfloor X \rfloor$ elements in the down buffers (the third case), we perform a recursive pull of $\lfloor X^{3/2} \rfloor$ elements from level $X^{9/4}$, and effectively remove the $\lfloor X \rfloor$ elements with smallest keys among the $\lfloor X \rfloor + Y$ elements. Before the recursive pull, level $X^{3/2}$ has one down buffer with $Y$ elements and an up buffer with $U$ elements; after the recursive pull and removal of the $\lfloor X \rfloor$ elements, it has one down buffer with less than $\lfloor X \rfloor$ elements, at most $\lceil X^{1/2} \rceil$ down buffers with $\lfloor X \rfloor$ elements, and an up buffer with $U$ elements. The coins on the $U$ elements in the up buffer before the recursive pull can be used to fulfill Invariant 5 for the $U$ elements in the up buffer after the recursive pull. By the pull requirement, each of the $\lfloor X^{3/2} \rfloor$ pulled elements has a pull coin for level $X^{3/2}$ and each level below. The same is true for each of the $Y$ original elements (Invariant 4). Thus we have enough coins for all the elements in the down buffers after the pull, since each down buffer contains at most $\lfloor X \rfloor$ elements and each element therefore only needs to fulfill Invariant 4. Similarly, since each of the $\lfloor X \rfloor$ removed (pulled) elements only needs a pull coin for level $X$ and each level below to fulfill the pull requirement, the pull cost can (as in the first two cases) be payed by the remaining $\lfloor X \rfloor$ level-$X^{3/2}$ pull coins.

The above argument shows that all push and pulls can be paid if each inserted element is given a push and a pull coin for each level of the structure, that is, that a delete-min is free amortized and an insertion costs $O(\frac{1}{B}\log_{M/B}\frac{N_0}{B})$ amortized memory transfers. By a completely analogous argument, it is easy to see that the operations are performed in $O(\sum_{i=0}^{\infty}\log_2(N_0^{(2/3)^i})) = O(\log_2 N_0)$ amortized time.

Finally, to maintain that $N_0 = \Theta(N)$ we completely rebuild the structure bottom-up after every $N_0/4$ operations (often referred to as *global rebuilding* [48]). We choose the size $N_0$ of the largest level to be $2N$ and compute the largest value $c < c_t$ such that $c^{3/2^i} = 2N$ for some integer $i$. Then we construct levels $c, c^{3/2}, \ldots, N_0^{2/3}$ such that all up buffers are empty and such that each level-$X^{3/2}$ has exactly $\lceil X^{1/2} \rceil$ down buffers of size $\lfloor X \rfloor$. The remaining elements are placed in level $N_0$ such that it has at most $\lceil N_0^{1/3} \rceil$ down buffers of size exactly $\lfloor N_0^{2/3} \rfloor$ and one of size less than $\lfloor N_0^{2/3} \rfloor$. We can easily perform the rebuilding in a sorting and a scanning step using a total of $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ memory transfers. At the same time, we can place pull coins on the elements in order to fulfill Invariant 4 (no elements need to fulfill Invariant 5), for a total cost of $O(\sum_{i=0}^{\infty}(N_0^{(2/3)^i}/B)\log_{M/B}(N_0^{(2/3)^i}/B)) = O(\frac{N_0}{B}\log_{M/B}\frac{N_0}{B})$ memory transfers. Thus the rebuilding adds only $O(\frac{1}{B}\log_{M/B}\frac{N_0}{B})$ amortized transfers to the cost of an insert or delete-min operation. In the same way we can argue that it adds only $O(\log_2 N_0)$ amortized time per operation.

Since the up buffer of level $N_0$ is of size $\lfloor 2N \rfloor = 2N$ after the rebuilding, we will not need further levels during the next $N_0/4 = N/2$ operations (insertions). At the same time, the size $N_0$ of the largest level remains $\Theta(N)$ during the next $N/2$ operations (deletions). Thus the size of our structure remains linear in $N$ and it supports insert and delete-min operations in $O(\frac{1}{B}\log_{M/B}\frac{N}{B})$ amortized memory transfers and $O(\log_2 N)$ amortized computation time.

**Lemma 3** *Using $O(M)$ main memory, a set of $N$ elements can be maintained in a linear-space cache-oblivious priority queue data structure supporting each insert and delete-min operation in $O(\frac{1}{B}\log_{M/B}\frac{N}{B})$ amortized memory transfers and $O(\log_2 N)$ amortized computation time.*

### 2.2.4 Delete

Using ideas from [11, 42] we can easily support a delete operation in $O(\frac{1}{B}\log_{M/B}\frac{N}{B})$ memory transfers and $O(\log_2 N)$ amortized time, provided that we are given the key of the element to be deleted. To perform a deletion, we simply insert a special element in the priority queue, with key equal to the element to be deleted. At some point during the sorts performed during a push, pull, or rebuild, this special element and the element to be deleted will be compared, as they have the same key. When this happens, we remove both elements from the structure. Note that the structure cannot contain more than a constant fraction of special elements or elements to be deleted, as all such elements will be compared and removed when we rebuild the structure.

To analyze a delete operation, we first note that it behaves exactly like an insertion, except that the special delete element and the element to be deleted are both removed when they "meet".

We incur the standard insertion cost when inserting a special delete element in the structures. The removal of the two elements on a level $X$ eventually contributes to a pull operation being performed on level $X^{3/2}$; the cost incurred by the two deletions is upper bounded by the cost of two delete-min operations. Thus in total we have obtained the following:

**Theorem 1** *Using $O(M)$ main memory, a set of $N$ elements can be maintained in a linear-space cache-oblivious priority queue data structure supporting each insert, delete-min, and delete operation in $O(\frac{1}{B}\log_{M/B}\frac{N}{B})$ amortized memory transfers and $O(\log_2 N)$ amortized computation time.*

# 3 Graph Algorithms

In this section we discuss how our cache-oblivious priority-queue can be used to develop several cache-oblivious graph algorithms. We first consider the simple list ranking problem and algorithms on trees, and then we go on and consider BFS, DFS and minimum spanning forest algorithms for general graphs.

## 3.1 List ranking

In the list ranking problem we are given a linked list of $V$ nodes stored as an unordered sequence. More precisely, we have an array with $V$ nodes, each containing the position of the next node in the list (an edge). The goal is to determine the *rank* of each node $v$, that is, the number of edges from $v$ to the end of the list. In a more general version of the problem, each edge has a weight and the goal is to find for each node $v$ the sum of the weights of edges from $v$ to the end of the list. Refer to Figure 2.

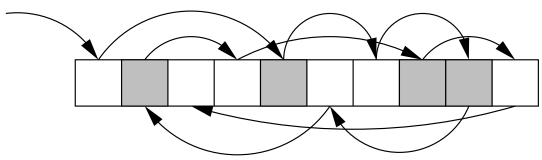

Figure 2: List ranking problem. An independent set of size 4 is marked. There are two forward lists (on top) and two backwards lists (on bottom).

Based on ideas from efficient PRAM algorithms [8, 35], Chiang et al. [32] designed an $O(\text{sort}(V))$ I/O model list ranking algorithm. The main idea in the algorithm is as follows. An *independent set* of $\Theta(V)$ nodes (nodes without edges to each other) is found, nodes in the independent set are "bridged out" (edges incident to nodes in this set are contracted), the remaining list is recursively ranked, and finally the contracted nodes are reintegrated into the list (their ranks are computed). The main innovation in the algorithm by Chiang et al. [32] was an $O(\text{sort}(V))$ memory transfer algorithm for computing an independent set of size $V/c$ for some constant $c > 0$. The rest of the non-recursive steps of the algorithm can easily be performed in $O(\text{sort}(V))$ memory transfers using a few scans and sorts of the nodes of the list as follows: To bridge out the nodes in the independent set, we first identify nodes with a successor in the independent set. We do so by creating a copy of the list of nodes, sorting it by successor position, and simultaneously scanning the two lists. During this process, we can also mark each predecessor of an independent set node $v$ with the position of the successor $w$ of $v$, as well as with the weight of the edge $(v, w)$. Next, in a simple scan, we create a new list where the two edges incident to each independent set node $v$ have been replaced with an edge from the predecessor of $v$ to the successor of $v$. The new edge has weight equal to the sum of that of the two old edges. Finally, we create the list to be ranked recursively by removing the independent set nodes and "compressing" the remaining nodes, that is, storing them in an array of size $V(1 - 1/c)$. We do so by scanning through the list, while creating a list of the nodes not in the independent set, as well as a list that indicates the old and new position of each node (that is, the position of each node in the old and new array). Then we update the successor fields of the first list by sorting it by successor position, and simultaneously scanning it and the list of new positions. After having ranked the compressed list recursively, we reintegrate the removed nodes, while computing their ranks. The rank of an independent set node $v$ is simply the rank of its successor $w$ minus the weight of edge $(v, w)$. The reintegration of the independent set nodes can be performed in a few scans and sorts similar to the way we bridged out the independent set. Overall, not counting the independent set computation, the number of memory transfers used to rank a $V$ node list is $T(V) = O(\text{sort}(V)) + T(V/c) = O(\text{sort}(V))$.

Since we only use scans and sorts in the above algorithm, all that remains in order to obtain a cache-oblivious list ranking algorithm is to develop a cache-oblivious independent set algorithm. Under different assumptions about the memory and block size, Chiang et al. [32] developed several independent set algorithms based on 3-coloring; in a 3-coloring every node is colored with one of three colors such that adjacent nodes have different colors. The independent set (of size at least $V/3$) then consists of the set of nodes with the most popular color. Arge [11] and Kumar and Schwabe [42] later removed the main memory and block assumptions.

One way of computing a 3-coloring is as follows [11, 32]: We call an edge $(v, w)$ a *forward* edge if $v$ appears before $w$ in the (unordered) sequence of nodes—otherwise it is called a *backward* edge. First we imagine splitting the list into two sets consisting of forward running segments (forward lists) and backward running segments (backward lists). Each node is included in at least one of these sets, and nodes at the head or tail of a segment (nodes at which there is a reversal of the direction) will be in both sets. Refer to Figure 2. Next we color the nodes in the forward lists red or blue by coloring the head nodes red and the other nodes alternately blue and red. Similarly, the nodes in the backward lists are colored green and blue, with the head nodes being colored green. In total, every node is colored with one color, except for the heads/tails, which have two colors. It is easy to see that we obtain a 3-coloring if we color each head/tail node the color it was colored as the head of a list [32].

In the above 3-coloring algorithm we can cache-obliviously color the forward lists as follows (the backwards lists can be colored similarly). In a single sort and scan we identify the head nodes and for each such node $v$ we insert a red element in a cache-oblivious priority queue with key equal to the position of $v$ in the unordered list. We then repeatedly extract the minimal key element $e$ from the queue. If $e$ corresponds to a node $v$, we access $v$ in the list, color it the same color as $e$, and insert an element corresponding to its successor in the queue. We color the inserted element in the opposite color of $e$. After processing all elements in the queue we have colored all forward lists. The initial sort and scan is performed cache-obliviously in $O(\text{sort}(V))$ memory transfers, and since we use a cache-oblivious priority queue we can also perform the $O(V)$ priority queue operations in $O(\text{sort}(V))$ memory transfers. Apart from this, we also perform what appears to be random accesses to the $O(V)$ nodes in the list. However, since we only process the forward list nodes in position order, the accesses overall end up corresponding to a scan of the list. Thus they only require $O(V/B)$ transfers. Therefore we can compute a 3-coloring cache-obliviously in $O(\text{sort}(V))$

memory transfers, and thus overall we obtain the following.

**Theorem 2** *The list ranking problem on a $V$ node list can be solved cache-obliviously in $O(\text{sort}(V))$ memory transfers.*

### 3.2 Algorithms on trees

Many efficient PRAM algorithms on undirected trees use Euler tour techniques [57, 59]. An Euler tour of a graph is a cycle that traverses each edge exactly once. Not every graph has an Euler tour but a tree where each undirected edge has been replaced with two directed edges does (Refer to Figure 3). When, in the following, we refer to an Euler tour of an undirected tree, we mean a tour in the graph obtained by replacing each edge in the tree with two directed edges.

To cache-obliviously compute an Euler tour of an undirected tree, that is, to compute an ordered list of the edges along the tour, we use ideas from similar PRAM algorithms. Consider imposing a (any) cyclic order on the nodes adjacent to each node $v$ in the tree. In [52] it is shown that an Euler tour is obtained if we traverse the tree such that a visit to $v$ from $u$ (through the incoming edge $(u, v)$) is followed by a visit to the node $w$ following $u$ in the cyclic order (through the outgoing edge $(v, w)$). Thus we can compute the successor edge of each edge $e$, that is, the edge following $e$ in the Euler tour, as follows: We first construct a list of incoming edges to each node $v$ sorted according to the cyclic order. If two edges $(u, v)$ and $(w, v)$ are stored next to each other in this list, the successor edge for the (incoming) edge $(u, v)$ is simply the (outgoing) edge $(v, w)$. Therefore we can compute all successor edges in a scan of the list. Given a list of all edges augmented with their successor edge, we can then compute the Euler tour simply by ranking the list and the sorting the edges by their rank. Thus overall we compute an Euler tour of an undirected tree using a few sorts and scans and a list ranking step, that is, using $O(\text{sort}(V))$ memory transfers.

Using our cache-oblivious Euler tour algorithm, we can easily compute a depth-first search (DFS) numbering of the nodes of a tree starting at a source node $s$ [52]. First note that if we start a walk of the Euler tour in the source node $s$ it is actually a DFS tour of the tree. To compute the numbering, we therefore first classify each edge as being either a *forward* or a *backward* edge; an edge is a forward edge if it is traversed before its reverse in the tour. After numbering the edges along the tour and sorting the list of edges such that reverse edges appear consecutively, we can classify each edge in a simple scan of the list. Then we assign each forward edge weight 1 and each backward edge weight 0. The DFS number of a node $v$ is then simply the sum of the weights on the edges from $s$ to $v$. Thus we can obtain the DFS numbering by solving the general version of

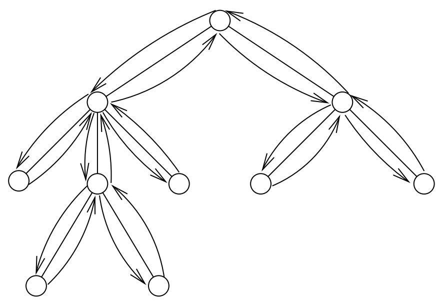

Figure 3: An undirected tree and an Euler tour of the corresponding directed graph.

list ranking. Since we only use Euler tours computation, list ranking, sorting, and scanning, we compute the DFS numbering cache-obliviously in a total of $O(\text{sort}(V))$ memory transfers.

Using an Euler tour, list ranking, and sorting, we can also compute a breadth-first search (BFS) numbering of the nodes of a tree cache-obliviously in $O(\text{sort}(V))$ memory transfers in a similar way [52]. Using standard PRAM ideas, and the tools developed here, we can also e.g. compute the centroid decomposition of a tree in $O(\text{sort}(V))$ memory transfers [57, 59, 32]. The centroid of a tree is the node that, if removed, minimizes the size of the largest of the remaining subtrees. The centroid decomposition of a tree is a recursive partition of a tree into subtrees around the centroid.

**Theorem 3** *The Euler tour, BFS, DFS, and centroid decomposition problems on a tree with $V$ nodes can be solved cache-obliviously in $O(\text{sort}(V))$ memory transfers.*

## 3.3 DFS and BFS

We now consider the DFS and BFS numbering problems for general graphs. We first describe a cache-oblivious DFS algorithm for directed graphs and then we modify it to compute a BFS numbering. Finally we develop an improved BFS algorithm for undirected graphs.

### 3.3.1 Depth-First Search

In the RAM model, directed DFS can be solved in linear time using a stack $S$ containing vertices $v$ that have not yet been visited but have an edge $(w, v)$ incident to a visited vertex $w$, as well as an array $A$ containing an element for each vertex $v$, indicating if $v$ has been visited or not. The top vertex $v$ of $S$ is repeatedly popped and $A$ is accessed to determine if $v$ has already been visited. If $v$ has not yet been visited, it is marked as visited in $A$ and all vertices adjacent to $v$ are pushed onto $S$. It is easy to realize that if the stack $S$ is implemented using a doubling array then a *push* or *pop* requires $O(1/B)$ amortized cache-oblivious memory transfers, since the optimal paging strategy can always keep the last block of the array (accessed by both push and pop) in main memory. However, each access to $A$ may require a separate memory transfer resulting in $\Omega(E)$ memory transfers in total.

In the I/O model, Chiang et al. [32] modified the above algorithm to obtain an $O(V + \frac{E}{B}\frac{V}{M})$ algorithm. In their algorithm all visited vertices (marked vertices in array A) are stored in main memory. Every time the number of visited vertices grows larger than the main memory, all visited vertices and all their incident edges are removed from the graph. Since this algorithm relies crucially on knowledge of the main memory size, it seems hard to make it cache-oblivious. Buchsbaum et al. [31] described another $O((V + \frac{E}{B})\log_2 V + \text{sort}(E))$ I/O model algorithm. In the following we describe how to make it cache-oblivious. The algorithm uses a number of data structures: $V$ priority queues, a stack, and a so-called *buffered repository tree*. As discussed above, a stack can trivially be made cache-oblivious. Below we first describe how to make the buffered repository tree cache-oblivious. Next we describe what we call a *buffered priority tree* that we use in the algorithm rather than our cache-oblivious priority queue; we cannot simply use our priority queue since it requires $O(M)$ space. Finally, we describe the algorithm by Buchsbaum et al. [31] and how the use of the cache-oblivious structures leads to a cache-oblivious version of it.

**Buffered repository tree.** A buffered repository tree (BRT) maintains $O(E)$ elements with keys in the range $[1..V]$ under operations *insert* and *extract*. The insert operation inserts a new element, while the extract operation reports and deletes all elements with a certain key.

Our cache-oblivious version of the BRT consists of a static binary tree with the keys 1 through $V$ in sorted order in the leaves. A buffer is associated with each node and leaf of the tree. The buffer of a leaf $v$ contains elements with key $v$ and the buffers of the internal nodes are used to perform insertions in a batched manner. We perform an insertion simply by inserting the new element into the root buffer. To perform an extraction of elements with key $v$ we traverse the path from the root to the leaf containing $v$. At each node $\mu$ on this path we scan the associated buffer and report and delete elements with key $v$. During the scan we also distribute the remaining elements among the two buffers associated with the children of $\mu$; we distribute an element with key $w$ to the buffer of the child of $\mu$ on the path to $w$. We place elements inserted in a buffer during the same scan consecutively in memory (but not necessarily right after the other elements in the buffer). This way the buffer of a node $\mu$ can be viewed as consisting of a linked list of *buckets* of elements in consecutive memory locations, with the number of buckets being bounded by the number of times the buffer of $\mu$'s parent buffer has been emptied since the last time $\mu$'s buffer was emptied. To avoid performing a memory transfer on each insertion, we implement the root buffer slightly different, namely as a doubling array (like a stack). Since only the root buffer is implemented this way, the optimal paging strategy can keep the last block of the array in main memory and we obtain an $O(1/B)$ amortized root buffer insertion bound. Refer to Figure 4 for an illustration of a BRT.

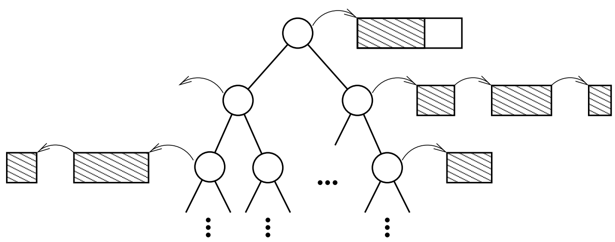

Figure 4: Buffered repository tree (BRT). Each non-root node has a buffer of elements stored as a linked list of buckets. The root node buffer is implemented using a doubling array.

**Lemma 4** *A cache-oblivious buffered repository tree uses $\Theta(B)$ main memory space and supports insert and extract operations in $O(\frac{1}{B}\log_2 V)$ and $O(\log_2 V)$ amortized memory transfers, respectively.*

*Proof*: As discussed, an insertion in the root buffer requires $O(1/B)$ amortized memory transfers. During an extract operation, we use $O(X/B+K)$ memory transfers to empty the buffer of a node $\mu$ containing $X$ elements in $K$ buckets (since we access each element and each bucket). We charge the $X/B$-term to the insert operations that inserted the $X$ elements in the BRT. Since each element is charged at most once on each level of the tree, an insert is charged $O(\frac{1}{B}\log_2 V)$ transfers. We charge the $K$-term to the extract operations that created the $K$ buckets. Since an extract operation creates two buckets on each level of the tree, it is charged a total of $O(\log_2 V)$ memory transfers. $\square$

**Buffered Priority Tree.** A buffered priority tree is constructed on $E_v$ elements with $E_v$ distinct keys, and maintains these elements under *buffered delete* operations. Given $E'$ elements, all of which are currently stored in the structure, a buffered delete operation first deletes the $E'$ elements and then deletes and reports the minimal key element among the remaining elements in the structure.

The buffered priority tree is implemented similarly to the buffered repository tree. It consists of a static binary tree with the $E_v$ keys in sorted order in the leaves, where a buffer is associated with each node. Initially, the $E_v$ elements are stored in the leaves containing their keys. The buffers are used to store elements intended to be deleted from the structure, and with each node $v$ we maintain a counter storing the number of (undeleted) elements stored in the tree rooted in $v$. To perform a buffered delete we first insert the $E'$ elements in the buffer of the root and update its counter. Next we traverse the path from the root to the leaf $l$ containing the minimal key elements (among the undeleted elements), while emptying the buffers associated with the encountered nodes: At the root, we scan the buffer and distribute the elements among the two buffers associated with the children (exactly as in the buffered repository tree). During the distribution, we also update the counters in the two children. If the counter of the left child $v_l$ is still greater than zero, i.e. the minimal element is in the tree rooted in $v_l$, we then recursively visit $v_l$. Otherwise we visit the right child $v_r$. (Note that when reaching leaf $l$ it has an empty buffer). After finding $l$, we report and delete the element stored in $l$. Finally we decrement the counters on the path from the root to $l$.

**Lemma 5** *Using no permanent main memory, a cache-oblivious buffered priority tree supports buffered deletes of $E'$ elements in $O((\frac{E'}{B}+1)\log_2 V)$ amortized memory transfers. It can be constructed in $O(\mathrm{sort}(E_v))$ memory transfers.*

*Proof*: The number of transfers needed to construct the tree is dominated by the $O(\mathrm{sort}(E_v))$ memory transfers needed to sort the $E_v$ elements and construct the leaves; after that, the tree can be constructed in $O(E_v/B)$ transfers level-by-level bottom-up. The amortized cost of a buffered delete is equal to the cost of inserting $E'$ elements into a BRT, plus the cost of extracting an element from a BRT, that is, $O((\frac{E'}{B}+1)\log_2 V)$ memory transfers. □

**DFS algorithm.** As mentioned, the directed DFS numbering algorithm by Buchsbaum et al. [31] utilizes a number of data structures: A stack $S$ containing vertices on the path from the root of the DFS tree to the current vertex, a priority queue $P(v)$ for each vertex $v$ containing edges $(v,w)$ connecting $v$ with a possibly unvisited vertex $w$, as well as one buffered repository tree $D$ containing edges $(v,w)$ incident to a vertex $w$ that has already been visited but where $(v,w)$ is still present in $P(v)$. The key of an edge $(v,w)$ is $v$ in $D$ and $w$ in $P(v)$. For the priority queues $P(v)$ we use our buffered priority tree.

Initially each $P(v)$ is constructed on the $E_v$ edges $(v,w)$ incident to $v$, the source vertex is placed on the stack $S$, and the BRT $D$ is empty. To compute a DFS numbering, the vertex $u$ on the top of the stack is repeatedly considered. All edges in $D$ of the form $(u,w)$ are extracted and deleted from $P(u)$ using a buffered delete operation. If $P(u)$ is now empty, so that no minimal element (edge $(u,v)$) is returned, all neighbors of $u$ have already been visited and $u$ is popped off $S$. Otherwise, if edge $(u,v)$) is returned, vertex $v$ is visited next: It is numbered and pushed on $S$, and all edges $(w,v)$ (with $w \neq u$) incident to it are inserted in $D$. In [31] it is shown that this algorithm correctly computes a DFS numbering.

To analyze the above algorithm, we first note that each vertex is considered on the top of $S$ a number of times equal to one greater than its number of children in the DFS tree. Thus the total number of times we consider a vertex on top of $S$ is $2V-1$. When considering a vertex $u$ we first perform a stack operation on $S$, an extract operation on $D$, and a buffered delete operation on $P(u)$. The stack operation requires $O(1/B)$ memory transfers and the extraction $O(\log_2 V)$ transfers, since the optimal paging strategy can keep the relevant $O(1)$ blocks of $S$ and $D$ in main memory at all times. Thus overall these costs add up to $O(V \log_2 V)$. The buffered delete operation requires $O((1+\frac{E'}{B})\log_2 V)$ memory transfers if $E'$ is the number of deleted elements (edges). Each edge is deleted once, so overall the buffered deletes costs add up to $O((V+\frac{E}{B})\log_2 V)$. Next we insert the, say $E''$, edges incident to $v$ in $D$. This requires $O(1+\frac{E''}{B}\log_2 V)$ memory transfers, or $O(V+\frac{E}{B}\log_2 V)$ transfers over the whole algorithm. (Note that the $E''$ edges are not immediately deleted directly from the relevant $P(w)$'s since that could cost a memory transfer per edge). In addition, the initial construction of the buffered priority trees requires $O(\text{sort}(E))$ memory transfers. Thus overall the algorithm uses $O((V+\frac{E}{B})\log_2 V + \text{sort}(E))$ memory transfers.

### 3.3.2 Breadth-First Search

The DFS algorithm described above can be modified to perform a breadth-first search simply by replacing the stack $S$ with a queue. Queues, like stacks, can be implemented using a doubling array, and the optimal paging strategy can keep the two partial blocks in use by the *enqueue* and *dequeue* operations in memory, such that each queue operation requires $O(1/B)$ amortized memory transfers. Thus we also obtain an $O((V+\frac{E}{B})\log_2 V)$ directed BFS numbering algorithm.

Our directed DFS and BFS algorithms can of course also be used on undirected graphs. For undirected graphs, improved $O(V+\text{sort}(E))$ and $O(\sqrt{\frac{V \cdot E}{B}} + \text{sort}(E))$ I/O model algorithms have been developed [47, 46]. The idea in the algorithm by Munagala and Ranade [47], which can immediately be made cache-oblivious, is to visit the vertices in "layers" of vertices of equal distance from the source vertex $s$. The algorithm utilizes that in an undirected graph any vertex adjacent to a vertex in layer $i$ is either in layer $i-1$, layer $i$, or layer $i+1$. It maintains two sorted lists of vertices in the last two layers $i$ and $i-1$. To create a sorted list of vertices in layer $i+1$, a list of possible layer $i+1$ vertices is first produced by collecting all vertices with a neighbor in level $i$ (using a scan of the adjacency lists of layer $i$ vertices). Then this list is sorted, and in a scan of the list and the (sorted) lists of vertices in level $i$ and $i-1$ all previously visited vertices are removed. Apart from the sorting steps, overall this algorithm uses $O(V+E/B)$ memory transfers to access the edge lists for all vertices, as well as $O(E/B)$ transfers to scan the lists. Since each vertex is included in a sort once for each of its incident edges, the total cost of all sorting steps is $O(\text{sort}(E))$. Thus in total the algorithm uses $O(V+\text{sort}(E))$ memory transfers. Since it only uses scans and sorts, it is cache-oblivious without modification. Refer to [47] for full details. As mentioned in the introduction, Brodal et al. [29] have developed other undirected BFS algorithms based on the ideas in [46].

**Theorem 4** *The DFS or BFS numbering of a directed graph can be computed cache-obliviously in $O((V+\frac{E}{B})\log_2 V + \text{sort}(E))$ memory transfers. The BFS numbering of an undirected graph can be computed cache-obliviously in $O(V+\text{sort}(E))$ memory transfers.*

## 3.4 Minimum Spanning Forest

In this section we consider algorithms for computing the minimum spanning forest (MSF) of an undirected weighted graph. Without loss of generality, we assume that all edge weights are distinct. In the I/O model, a sequence of algorithms have been developed for the problem [32, 1, 42, 15], culminating in an algorithm using $O(\text{sort}(E) \cdot \log_2 \log_2(\frac{VB}{E}))$ memory transfers developed by Arge et al. [15]. This algorithm consists of two phases. In the first phase, an edge contraction algorithm inspired by PRAM algorithms [33, 36] is used to reduce the number of vertices to $O(E/B)$. In the second phase, a modified version of Prim's algorithm [49] is used to complete the MSF computation. Using our cache-oblivious priority queue we can relatively easily modify both of the phases to work cache-obliviously. However, since we cannot decide cache-obliviously when the first phase has reduced the number of vertices to $O(E/B)$, we are not able to combine the two phases as effectively as in the I/O model. Below we first describe how to make the algorithms used in the two phases cache-oblivious. Then we discuss their combination.

### 3.4.1 Phase 1

The basic edge contraction based MSF algorithm proceeds in stages [33, 32, 42]. In each stage the minimum weight edge incident to each vertex is selected and output as part of the MSF, and the vertices connected by the selected edges are contracted into super-vertices (that is, the connected components of the graph of selected edges are contracted). See e.g. [15] for a proof that the selected edges along with the edges in a MSF of the contracted graph constitute a MSF for the original graph.

In the following we sketch how we can perform a contraction stage on a graph $G$ cache-obliviously in $O(\text{sort}(E))$ memory transfers as in [15]. We can easily select the minimum weight edges in $O(\text{sort}(E))$ memory transfers using a few scans and sorts. To perform the contraction, we select a *leader vertex* in each connected component of the graph $G_s$ of selected edges, and replace every edge $(u, v)$ in $G$ with the edge $(leader(u), leader(v))$. To select the leaders, we use the fact that the connected components of $G_s$ are trees, except that one edge in each component (namely the minimal weight edge) appears twice [15]. In each component, we simply use one of the vertices incident to the edge appearing twice as leader. This way we can easily identify all the leaders in $O(\text{sort}(E))$ memory transfers using a few sorts and scans (by identifying all edges that appear twice). We can then use our cache-oblivious tree algorithms developed in Section 3.2 to distribute the identity of the leader to each vertex in each component in $O(\text{sort}(V))$ memory transfers: We add an edge between each leader in $G_s$ and a pseudo root vertex $s$ and perform a DFS numbering of the resulting tree starting in $s$; since all vertices in the same connected component (tree) will have consecutive DFS numbers, we can then mark each vertex with its leader using a few sorts and scans. Finally, after marking each vertex $v$ with $leader(v)$, we can easily replace each edge $(u, v)$ in $G$ with $(leader(u), leader(v))$ in $O(\text{sort}(E))$ memory transfers using a few sort and scan steps on the vertices and edges.

Since each contraction stage reduces the number of vertices by a factor of two, and since a stage is performed in $O(\text{sort}(E))$ memory transfers, we can reduce the number of vertices to $V' = V/2^i$ in $O(\text{sort}(E) \cdot \log_2(V/V'))$ memory transfers by performing $i$ stages after each other. Thus we obtain an $O(\text{sort}(E) \cdot \log_2 V)$ algorithm by continuing the contraction until we are left with a single vertex. In the I/O model, Arge et al. [15] showed how to improve this bound to $O(\text{sort}(E) \cdot \log_2 \log_2 V)$ by grouping the stages into "super-stages" and working only on a subset of the edges of $G$ in each super-stage. The extra steps involved in their improvement are all sorting or scanning of the edges and vertices, and therefore the improvement is immediately cache-oblivious.

**Lemma 6** *The minimum spanning forest of an undirected weighted graph can be computed cache-obliviously in $O(\text{sort}(E) \cdot \log_2 \log_2 V)$ memory transfers.*

### 3.4.2 Phase 2

Prim's algorithm [49] grows a minimum spanning tree (MST) of a connected graph iteratively from a source vertex using a priority queue $P$ on the vertices not already included in the MST. The key of a vertex $v$ in $P$ is equal to the weight of the minimal weight edge connecting $v$ to the current MST. In each step of the algorithm a delete-min is used to obtain the next vertex $u$ to add to the MST, and the keys of all neighbors of $u$ in $P$ are (possibly) updated. A standard implementation of this algorithm uses $\Omega(E)$ memory transfers, since a transfer is needed to obtain the current key of each neighbor vertex. In the I/O model, Arge et al. [15] showed how to modify the algorithm to use $O(V + \mathrm{sort}(E))$ memory transfers by storing edges rather than vertices in the priority queue. Below we describe this algorithm in order to show that it can be implemented cache-obliviously.

Like Prim's algorithm, the algorithm by Arge et al. [15] grows the MST iteratively. We maintain a priority queue $P$ containing (at least) all edges connecting vertices in the current MST with vertices not in the tree; $P$ can also contain edges between two vertices in the MST. Initially it contains all edges incident to the source vertex. In each step of the algorithm we extract the minimum weight edge $(u, v)$ from $P$. If $v$ is already in the MST we discard the edge; otherwise we include $v$ in the MST and insert all edges incident to $v$, except $(v, u)$, in the priority queue. We can efficiently determine if $v$ is already in the MST, since if $u$ and $v$ are both already in the MST then $(u, v)$ must be in the priority queue twice; thus if the next edge we extract from $P$ is $(v, u)$ then $v$ is already in the MST.

The correctness of the above algorithm follows immediately from the correctness of Prim's algorithm. During the algorithm we access the edges in the adjacency list of each vertex $v$ once (when $v$ is included in the MST) for a total of $O(V + E/B)$ memory transfers. We also perform $O(E)$ priority queue operations, for a total of $O(\mathrm{sort}(E))$ memory transfers. Thus the algorithm uses $O(V + \mathrm{sort}(E))$ memory transfers. All of the above can easily be modified to compute a MSF for an unconnected graph rather than a MST for a connected graph. Thus we have obtained the following.

**Lemma 7** *The minimum spanning forest of an undirected weighted graph can be computed cache-obliviously in $O(V + \mathrm{sort}(E))$ memory transfers.*

### 3.4.3 Combined algorithm

In the I/O model, an $O(\mathrm{sort}(E) \cdot \log_2 \log_2(\frac{VB}{E}))$ MSF algorithm can be obtained by running the phase 1 algorithm until the number of vertices has been reduced to $V' = E/B$ using $O((\mathrm{sort}(E) \cdot \log_2 \log_2(\frac{VB}{E}))$ memory transfers, and then finishing the MSF in $O(V' + \mathrm{sort}(E)) = O(\mathrm{sort}(E))$ memory transfers using the phase 2 algorithm. As mentioned, we cannot combine the two phases as effectively in the cache-oblivious model. In general however, we can combine the two algorithms to obtain an $O(\mathrm{sort}(E) \cdot \log_2 \log_2(V/V') + V')$ algorithm for any $V'$ independent of $B$ and $M$.

**Theorem 5** *The minimum spanning forest of an undirected weighted graph can be computed cache-obliviously in $O(\mathrm{sort}(E) \cdot \log_2 \log_2(V/V') + V')$ memory transfers for any $V'$ independent of $B$ and $M$.*

## 4 Conclusions

In this paper, we presented an optimal cache-oblivious priority queue and used it to develop efficient cache-oblivious algorithms for several graph problems. We believe the ideas utilized in the development of the priority queue and our graph algorithms will prove useful in the development of other cache-oblivious data structures.

Many important problems still remain open in the area of cache-oblivious algorithms and data structures. In the area of graph algorithms, for example, it remains open to develop a cache-oblivious MSF algorithm with complexity matching the best known cache-aware algorithm.

## Acknowledgements

The authors wish to thank an anonymous reviewer for pointing out a mistake in an earlier draft of this paper, and all the reviewers for numerous suggestions for improving the presentation.

## References

[1] J. Abello, A. L. Buchsbaum, and J. R. Westbrook. A functional approach to external graph algorithms. *Algorithmica*, 32(3):437–458, 2002.

[2] P. K. Agarwal, L. Arge, A. Danner, and B. Holland-Minkley. Cache-oblivious data structures for orthogonal range searching. In *Proc. ACM Symposium on Computational Geometry*, pages 237–245, 2003.

[3] A. Aggarwal, B. Alpern, A. K. Chandra, and M. Snir. A model for hierarchical memory. In *Proc. ACM Symposium on Theory of Computation*, pages 305–314, 1987.

[4] A. Aggarwal and A. K. Chandra. Virtual memory algorithms. In *Proc. ACM Symposium on Theory of Computation*, pages 173–185, 1988.

[5] A. Aggarwal, A. K. Chandra, and M. Snir. Hierarchical memory with block transfer. In *Proc. IEEE Symposium on Foundations of Computer Science*, pages 204–216, 1987.

[6] A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[7] B. Alpern, L. Carter, E. Feig, and T. Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 12(2-3), 1994.

[8] R. J. Anderson and G. L. Miller. A simple randomized parallel algorithm for list-ranking. *Information Processing Letters*, 33:269–273, 1990.

[9] L. Arge. The I/O-complexity of ordered binary-decision diagram manipulation. In *Proc. International Symposium on Algorithms and Computation, LNCS 1004*, pages 82–91, 1995. A complete version appears as BRICS Technical Report RS-96-29, University of Aarhus.

[10] L. Arge. External memory data structures. In J. Abello, P. M. Pardalos, and M. G. C. Resende, editors, *Handbook of Massive Data Sets*, pages 313–358. Kluwer Academic Publishers, 2002.

[11] L. Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003.

[12] L. Arge, M. Bender, E. Demaine, B. Holland-Minkley, and J. I. Munro. Cache-oblivious priority queue and graph algorithm applications. In *Proc. ACM Symposium on Theory of Computation*, pages 268–276, 2002.

[13] L. Arge, G. S. Brodal, and R. Fagerberg. Cache-oblivious data structures. In D. Mehta and S. Sahni, editors, *Handbook on Data Structures and Applications*. CRC Press, 2005.

[14] L. Arge, G. S. Brodal, R. Fagerberg, and M. Laustsen. Cache-oblivious planar orthogonal range searching and counting. In *Proc. ACM Symposium on Computational Geometry*, 2005.

[15] L. Arge, G. S. Brodal, and L. Toma. On external memory MST, SSSP and multi-way planar graph separation. In *Proc. Scandinavian Workshop on Algorithms Theory, LNCS 1851*, pages 433–447, 2000.

[16] L. Arge, J. Chase, J. Vitter, and R. Wickremesinghe. Efficient sorting using registers and caches. *ACM Journal on Experimental Algorithmics*, 7(9), 2002.

[17] L. Arge, M. de Berg, and H. Haverkort. Cache-oblivious R-trees. In *Proc. ACM Symposium on Computational Geometry*, 2005.

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

[19] M. Bender, R. Cole, E. Demaine, and M. Farach-Colton. Scanning and traversing: Maintaining data for traversals in memory hierarchy. In *Proc. European Symposium on Algorithms*, pages 152–164, 2002.

[20] M. A. Bender, G. S. Brodal, R. Fagerberg, D. Ge, S. He, H. Hu, J. Iacono, and A. López-Ortiz. The cost of cache-oblivious searching. In *Proc. IEEE Symposium on Foundations of Computer Science*, pages 271–282, 2003.

[21] M. A. Bender, R. Cole, and R. Raman. Exponential structures for cache-oblivious algorithms. In *Proc. International Colloquium on Automata, Languages, and Programming*, pages 195–207, 2002.

[22] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. In *Proc. IEEE Symposium on Foundations of Computer Science*, pages 339–409, 2000.

[23] M. A. Bender, Z. Duan, J. Iacono, and J. Wu. A locality-preserving cache-oblivious dynamic dictionary. In *Proc. ACM-SIAM Symposium on Discrete Algorithms*, pages 29–38, 2002.

[24] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. An analysis of dag-consistent distributed shared-memory algorithms. In *Proc. ACM Symposium on Parallel Algorithms and Architectures*, pages 297–308, 1996.

[25] G. S. Brodal and R. Fagerberg. Cache oblivious distribution sweeping. In *Proc. International Colloquium on Automata, Languages, and Programming*, pages 426–438, 2002.

[26] G. S. Brodal and R. Fagerberg. Funnel heap - a cache oblivious priority queue. In *Proc. International Symposium on Algorithms and Computation, LNCS 2518*, pages 219–228, 2002.

[27] G. S. Brodal and R. Fagerberg. On the limits of cache-obliviousness. In *Proc. ACM Symposium on Theory of Computation*, 2003.

[28] G. S. Brodal, R. Fagerberg, and R. Jacob. Cache oblivious search trees via binary trees of small height. In *Proc. ACM-SIAM Symposium on Discrete Algorithms*, pages 39–48, 2002.

[29] G. S. Brodal, R. Fagerberg, U. Meyer, and N. Zeh. Cache-oblivious data structures and algorithms for undirected breadth-first search and shortest paths. In *Proc. Scandinavian Workshop on Algorithms Theory, LNCS 3111*, pages 480–492, 2004.

[30] G. S. Brodal and J. Katajainen. Worst-case efficient external-memory priority queues. In *Proc. Scandinavian Workshop on Algorithms Theory, LNCS 1432*, pages 107–118, 1998.

[31] A. L. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J. R. Westbrook. On external memory graph traversal. In *Proc. ACM-SIAM Symposium on Discrete Algorithms*, pages 859–860, 2000.

[32] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proc. ACM-SIAM Symposium on Discrete Algorithms*, pages 139–149, 1995.

[33] F. Chin, J. Lam, and I. Chen. Efficient parallel algorithms for some graph problems. *Communications of ACM*, 25:659–665, 1982.

[34] R. A. Chowdhuey and V. Ramachandran. Cache-oblivious shortest paths in graphs using buffer heap. In *Proc. ACM Symposium on Parallel Algorithms and Architectures*, pages 245–254, 2004.

[35] R. Cole and U. Vishkin. Deterministic coin tossing with applications to optimal list-ranking. *Information and Control*, 70(1):32–53, 1986.

[36] R. Cole and U. Vishkin. Approximate parallel scheduling. II. Applications to logarithmic-time optimal parallel graph algorithms. *Information and Computation*, 92(1):1–47, 1991.

[37] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[38] R. Fadel, K. V. Jakobsen, J. Katajainen, and J. Teuhola. Heaps and heapsort on secondary storage. *Theoretical Computer Science*, 220(2):345–362, 1999.

[39] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. IEEE Symposium on Foundations of Computer Science*, pages 285–298, 1999.

[40] S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982.

[41] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading MA, second edition, 1998.

[42] V. Kumar and E. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proc. IEEE Symp. on Parallel and Distributed Processing*, pages 169–177, 1996.

[43] R. Ladner, J. Fix, and A. LaMarca. Cache performance analysis of traversals and random accesses. *Journal of Algorithms*, 1999.

[44] A. LaMarca and R. Ladner. The influence of cache on the performace of sorting. In *Proc. ACM-SIAM Symposium on Discrete Algorithms*, pages 370–379, 1997.

[45] A. LaMarca and R. E. Ladner. The influence of caches on the performance of heaps. *Journal of Experimental Algorithmics*, 1(4), 1996.

[46] K. Mehlhorn and U. Meyer. External-memory breadth-first search with sublinear I/O. In *Proc. European Symposium on Algorithms, LNCS 2461*, pages 723–735, 2002.

[47] K. Munagala and A. Ranade. I/O-complexity of graph algorithms. In *Proc. ACM-SIAM Symposium on Discrete Algorithms*, pages 687–694, 1999.

[48] M. H. Overmars. *The Design of Dynamic Data Structures.* Springer-Verlag, LNCS 156, 1983.

[49] R. C. Prim. Shortest connection networks and some generalizations. *Bell Syst. Tech. J.*, 36:1389–1401, 1957.

[50] N. Rahman, R. Cole, and R. Raman. Optimised predecessor data structures for internal memory. In *Proc. Workshop on Algorithm Engineering, LNCS 2141*, pages 67–78, 2001.

[51] N. Rahman and R. Raman. Analysing cache effects in distribution sorting. In *Proc. Workshop on Algorithm Engineering*, 1999.

[52] J. H. Reif, editor. *Synthesis of Parallel Algorithms*, chapter 2, pages 61–114. Morgan Kaufmann, 1993.

[53] P. Sanders. Fast priority queues for cached memory. In *Proc. Workshop on Algorithm Engineering and Experimentation, LNCS 1619*, pages 312–327, 1999.

[54] J. E. Savage. Extending the Hong-Kung model to memory hierachies. In *Proc. Annual International Conference on Computing and Combinatorics, LNCS 959*, pages 270–281, 1995.

[55] S. Sen, S. Chatterjee, and N. Dumir. Towards a theory of cache-efficient algorithms. *Journal of the ACM*, 49(6):828–858, 2002.

[56] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28:202–208, 1985.

[57] R. E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM Journal on Computing*, 14(4):862–874, 1985.

[58] S. Toledo. Locality of reference in *LU* decomposition with partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 18(4):1065–1081, 1997.

[59] U. Vishkin. On efficient parallel strong orientation. *Information Processing Letters*, 20:235–240, 1985.

[60] J. S. Vitter. External memory algorithms and data structures: Dealing with MASSIVE data. *ACM Computing Surveys*, 33(2):209–271, 2001.

[61] N. Zeh. *I/O-Efficient Algorithms for Shortest Path Related Problems*. PhD thesis, School of Computer Science, Carleton University, 2002.