Cache-Oblivious Algorithms and Data Structures

Erik D. Demaine

MIT Laboratory for Computer Science, 200 Technology Square, Cambridge, MA 02139, USA, edemaine@mit.edu

Abstract. A recent direction in the design of cache-efficient and disk-efficient algorithms and data structures is the notion of *cache obliviousness*, introduced by Frigo, Leiserson, Prokop, and Ramachandran in 1999. Cache-oblivious algorithms perform well on a multilevel memory hierarchy without knowing any parameters of the hierarchy, only knowing the existence of a hierarchy. Equivalently, a single cache-oblivious algorithm is efficient on all memory hierarchies simultaneously. While such results might seem impossible, a recent body of work has developed cache-oblivious algorithms and data structures that perform as well or nearly as well as standard external-memory structures which require knowledge of the cache/memory size and block transfer size. Here we describe several of these results with the intent of elucidating the techniques behind their design. Perhaps the most exciting of these results are the data structures, which form general building blocks immediately leading to several algorithmic results.

Table of Contents

1 Overview

We assume that the reader is already familiar with the motivation for designing cache-efficient and disk-efficient algorithms: multilevel memory hierarchies are becoming more prominent, and their costs a dominant factor of running time, so for speed it is crucial to minimize these costs. We begin in Section 2 with a description of the cache-oblivious model, after a brief review of the standard external-memory model. Then in Section 3 we consider some simple to moderately complex cache-oblivious algorithms: reversal, matrix operations, and sorting. The bulk of this work culminates with the paper that first defined the notion of cache-oblivious algorithms [FLPR99]. Next in Sections 4 and 5 we examine static and dynamic cache-oblivious data structures that have been developed, mostly in the past few years (2000–2003). Finally, Section 6 summarizes where we are and what directions might be interesting to pursue in the (near) future.

2 Models

2.1 External-Memory Model

To contrast with the cache-oblivious model, we first review the standard model of a two-level memory hierarchy with block transfers. This model is known variously as the *external-memory model*, the *I/O model*, the *disk access model*, or the *cache-aware model* (to contrast with cache obliviousness). The standard reference for this model is Aggarwal and Vitter's 1988 paper [AV88] which also analyzes the memory-transfer cost of sorting in this model. Special versions of the model were considered earlier, e.g., by Floyd in 1972 [Flo72] in his analysis of the memory-transfer cost of matrix transposition.

The model[1] defines a computer as having two levels (see Figure 1):

1. the *cache* which is near the CPU, cheap to access, but limited in space; and
2. the *disk* which is distant from the CPU, expensive to access, but nearly limitless in space.

We use the terms "cache" and "disk" for the two levels to make the relative costs clear (cache is faster than disk); other papers ambiguously use "memory" to refer to either the slow level (when comparing to cache) or the fast level (when comparing to disk). We use the term "memory" to refer generically to the entire memory system, in particular, the total order in which data is stored.

The central aspect of the external-memory model is that transfers between cache and disk involve *blocks* of data. Specifically, the disk is partitioned into blocks of B elements each, and accessing one element on disk copies its entire block to cache. The cache can store up to M/B blocks, for a total size of M

[1] We ignore the parallel-disks aspect of the model described by Aggarwal and Vitter [AV88].

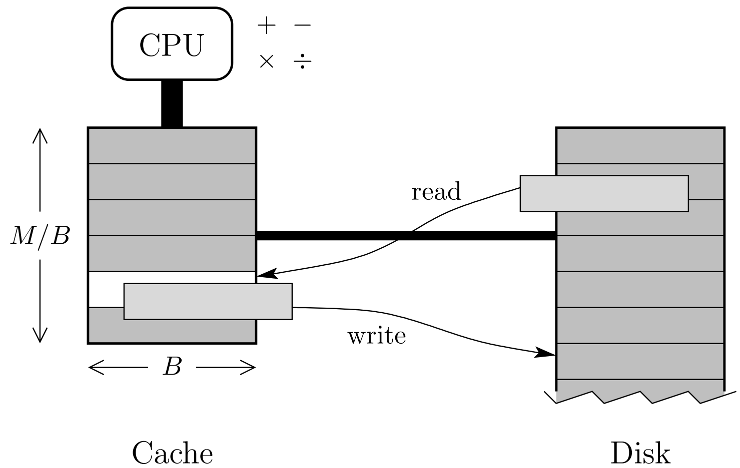

Figure 1. A two-level memory hierarchy shown with one block replacing another.

elements, $M \geq B$.[2] Before fetching a block from disk when the cache is already full, the algorithm must decide which block to evict from cache.

Many algorithms have been developed in this model; see Vitter's survey [Vit01]. One of its attractive features, in contrast to a variety of other models, is that the algorithm needs to worry about only two levels of memory, and only two parameters. Naturally, though, the existing algorithms depend crucially on B and M. Another point to make is that the algorithm (at least in principle) explicitly issues read and write requests to the disk, and (again in principle) explicitly manages the cache. These properties will disappear with the cache-oblivious model.

2.2 Cache-Oblivious Model

The *cache-oblivious model* was introduced by Frigo, Leiserson, Prokop, and Ramachandran in 1999 [FLPR99,Pro99]. Its principle idea is simple: design external-memory algorithms without knowing B and M. But this simple idea has several surprisingly powerful consequences.

One consequence is that, if a cache-oblivious algorithm performs well between two levels of the memory hierarchy (nominally called cache and disk), then it must automatically work well between any two adjacent levels of the memory hierarchy. This consequence follows almost immediately, though it relies on every two adjacent levels being modeled as an external memory, each presumably with different values for the parameters B and M, in such a way that blocks in memory levels nearer the CPU store subsets of memory levels farther from

[2] To avoid worrying about small additive constants in M, we also assume that the CPU has a constant number of *registers* that can store loop counters, partial sums, etc.

the CPU—the *inclusion property* [HP96, p. 723]. A further consequence, if the number of memory transfers is optimal up to a constant factor between any two adjacent memory levels, then any weighted combination of these counts (with weights corresponding to the relative speeds of the memory levels) is also within a constant factor of optimal. In this way, we can design and analyze algorithms in a two-level memory model, and obtain results for an arbitrary many-level memory hierarchy—provided we can make the algorithms cache-oblivious.

Another, more practical consequence is self-tuning. Typical cache-efficient algorithms require tuning to several cache parameters which are not always available from the manufacturer and often difficult to extract automatically. Parameter tuning makes code portability difficult. Perhaps the first and most obvious motivation for cache-oblivious algorithms is the lack of such tuning: a single algorithm should work well on all machines without modification. Of course, the code is still subject to some tuning, e.g., where to trim the base case of a recursion, but such optimizations should not be direct effects of the cache.

In contrast to the external-memory model, algorithms in the cache-oblivious model cannot explicitly manage the cache (issue block-read and block-write requests). This loss of freedom is necessary because the block and cache sizes are unknown. In addition, this automatic-management model more closely matches physical caches other than the level between main memory and disk: which block to replace is normally decided by the cache hardware according to a fixed *page-replacement strategy*, not by a general program.

But how are we to design algorithms that minimize the number of block transfers if we do not know the page-replacement strategy? An adversarial page-replacement strategy could always evict the next block that will be accessed, effectively reducing M to 1 in any algorithm. To avoid this problem, the cache-oblivious model assumes an *ideal cache* which advocates a utopian viewpoint: page replacement is optimal, and the cache is fully associative. These assumptions are important to understand, and will seem unrealistic, but theoretically they are justified as described in the next section.

The first assumption—*optimal page replacement*—specifies that the page-replacement strategy knows the future and always evicts the page that will be accessed farthest in the future. This omniscient ideal is the exact opposite of the adversarial page-replacement strategy described above. In contrast, of course, real-world caches do not know the future, and employ more realistic page-replacement strategies such as evicting the least-recently-used block (LRU) or evicting the oldest block ($FIFO$).

The second assumption—*full associativity*—may not seem like an assumption for those more familiar with external memory than with real-world caches: it says that any block can be stored anywhere in cache. In contrast, most caches above the level between main memory and disk have *limited associativity* meaning that each block belongs to a *cluster* between 1 and M, usually the block address modulo M, and at most some small constant c of blocks from a common cluster can be stored in cache at once. Typical real-world caches are either directed mapped ($c = 1$) or 2-way associative ($c = 2$). Some caches have more associativity—4-way or 8-way—but the constant c is certainly limited. Within each cluster, caches can apply a page-replacement strategy such as FIFO or LRU or OPT; but once the cluster fills, another block must be evicted. At first glance, such a policy would seem to limit M to c in the worst case.

2.3 Justification of Model

Frigo et al. [FLPR99,Pro99] justify the ideal-cache model described in the previous section by a collection of reductions that modify an ideal-cache algorithm to operate on a more realistic cache model. The running time of the algorithm degrades somewhat, but in most cases by only a constant factor. Here we outline the major steps, without going into the details of the proofs.

2.3.1 Replacement Strategy

The first reduction removes the practically unrealizable optimal (omniscient) replacement strategy that uses information about future requests.

Lemma 1 ([FLPR99, Lemma 12]). *If an algorithm makes T memory transfers on a cache of size $M/2$ with optimal replacement, then it makes at most $2T$ memory transfers on a cache of size M with LRU or FIFO replacement (and the same block size B).*

In other words, LRU and FIFO replacement do just as well as optimal replacement up to a constant factor of memory transfers and up a constant factor wastage of the cache. This competitiveness property of LRU and FIFO goes back to a 1985 paper of Sleator and Tarjan [ST85a]. In the algorithmic setting, as long as the number of memory transfers depends polynomially on the cache size M, then halving M will only affect the running time by a constant factor. More generally, what is needed is the *regularity condition* that $T(B, M) = O(T(B, M/2))$. Thus we have this reduction:

Corollary 1 ([FLPR99, Corollary 13]). *If the number of memory transfers made by an algorithm on a cache with optimal replacement, $T(B, M)$, satisfies the regularity condition, then the algorithm makes $\Theta(T(B, M))$ memory transfers on a cache with LRU or FIFO replacement.*

2.3.2 Associativity and Automatic Replacement

The reductions to convert full associativity into 1-way associativity (no associativity) and to convert automatic replacement into manual memory management are combined inseparably into one:

Lemma 2 ([FLPR99, Lemma 16]). *For some constant $\alpha > 0$, an LRU cache of size αM and block size B can be simulated in M space such that an access to a block takes $O(1)$ expected time.*

The basic idea is to use 2-universal hash functions to implement the associativity with only $O(1)$ conflicts. Of course, this reduction requires knowledge of B and M.

2.4 Tall-Cache Assumption

It is common to assume that a cache is taller than it is wide, that is, the number of blocks, M/B, is larger than the size of each block, B. Often this constraint is written $M = \Omega(B^2)$. Usually a weaker condition also suffices: $M = \Omega(B^{1+\gamma})$ for any constant $\gamma > 0$. We will refer to this constraint as the *tall-cache assumption*.

This property is particularly important in some of the more sophisticated cache-oblivious algorithms and data structures, where it ensures that the cache provides a polynomially large "buffer" for guessing the block size slightly wrong. It is also commonly assumed in external-memory algorithms.

3 Algorithms

Next we look at how to cause few memory transfers in the cache-oblivious model. We start with some simple but nonetheless useful algorithms based on "scanning" in Section 3.1. Then we look at several examples of the divide-and-conquer approach in Section 3.2. Section 3.3 considers the classic sorting problem, where the standard divide-and-conquer approach (e.g., mergesort) is not optimal, and more sophisticated techniques are needed. Finally, Section 3.4 gives references to cache-oblivious algorithms in computational geometry and graph algorithms.

3.1 Scanning

3.1.1 Traversal and Aggregates

To make the existence of efficient cache-oblivious algorithms more intuitive, let us start with a trivial example. Suppose we need to traverse all of the elements in a set, e.g., to compute an aggregate (sum, maximum, etc.). On a flat memory hierarchy (uniform-cost RAM), such a procedure requires $\Theta(N)$ time for N elements.[3] In the external-memory model, if we store the elements in $\lceil N/B \rceil$ blocks of size B, then the number of blocks transfers is $\lceil N/B \rceil$.

To achieve a similar bound in the cache-oblivious model, we can lay out the elements of the set in a contiguous segment of memory, in any order, and implement the N-element traversal by scanning the elements one-by-one in the order they are stored. This layout and traversal algorithm do not require knowledge of B (or M). The analysis may seem trivial (see Figure 2), but to see how such arguments go, let us describe it in detail:

Theorem 1. *Scanning N elements stored in a contiguous segment of memory costs at most $\lceil N/B \rceil + 1$ memory transfers.*

[3] The problem size N, memory size M, and block size B are normally written in capital letters in the external-memory context, originally so that the lower-case letters can be used to denote the number of blocks in the problem and in cache, which are smaller: $n = N/B$, $m = M/B$. The lower-case notation seems to have fallen out of favor (at least, I find it easily confusing), but for consistency (and no loss of clarity) the upper-case letters have stuck.

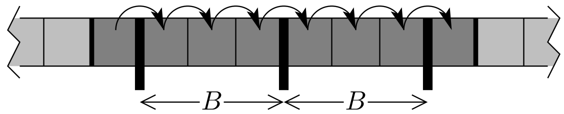

Figure 2. Scanning an array of N elements arbitrarily aligned with blocks may cost one more memory transfer than $\lceil N/B \rceil$.

Proof. The main issue here is alignment: where the block boundaries are relative to the beginning of the contiguous segment of memory. In the worst case, the first block has just one element, and the last block has just one element. In between, though, every block is fully occupied, so there are at most $\lfloor N/B \rfloor$ such blocks, for a total of at most $\lfloor N/B \rfloor + 2$ blocks. If B does not evenly divide N, this bound is the desired one. If B divides N and there are two nonfull blocks, then there are only $N/B - 1$ full blocks, and again we have the desired bound. □

The main structural element to point out here is that, although the algorithm does not use B or M, the analysis naturally does. The analysis is implicitly over all values of B and M (in this case, just B is relevant). In addition, we considered all possible alignments of the block boundaries with respect to the memory layout. However, this issue is normally minor, because at worst it divides one "ideal" block (according to the algorithm's alignment) into two physical blocks (according to the generic, possibly adversarial, alignment of the memory system). Above, we were concerned with the precise constant factors, so we were careful to consider the possible alignments.

Because of precisely the alignment issue, the cache-oblivious bound is an additive 1 away from the external-memory bound. Such error is ideal. Normally our goal is to match bounds within multiplicative constant factors. This goal is reasonable in particular because the model-justification reductions in Section 2.3 already lost multiplicative constant factors.

3.1.2 Array Reversal

While the aggregate application may seem trivial, albeit useful, a very similar idea leads to a particularly elegant algorithm for *array reversal*: reversing the elements of an array without extra storage. Bentley's array-reversal algorithm [Ben00, p. 14] makes two parallel scans, one from each end of the array, and at each step swaps the two elements under consideration. See Figure 3. Provided $M \geq 2B$, this cache-oblivious algorithm uses the same number of memory reads as a single scan.

3.2 Divide and Conquer

After scanning, the first major technique for designing cache-oblivious algorithms is *divide and conquer*. A classic technique in general algorithm design, this approach is particularly suitable in the cache-oblivious context. Often it leads to

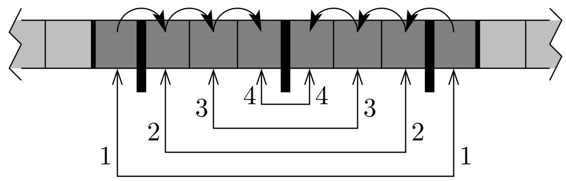

Figure 3. Bentley's reversal of an array.

algorithms whose memory-transfer count is optimal within a constant factor, although not always.

The basic idea is that divide-and-conquer repeatedly refines the problem size. Eventually, the problem will fit in cache (size at most M), and later, the problem will fit in a single block (size at most B). While the divide-and-conquer recursion completes all the way down to constant size, the analysis can consider the moments at which the problem fits in cache and fits in a block, and prove that the number of memory transfers is small in these cases. For a divide-and-conquer recursion dominated by the leaf costs, i.e. in which the number of leaves in the recursion tree is polynomially larger than the divide/merge cost, such an algorithm will usually then use within a constant factor of the optimal number of memory transfers. The constant factor arises here because we do not consider problems of exactly size M or B, but rather when a refined problem has size at most M or B; if we reduce the problem size by a constant factor c in each step, this will happen with problems of size at least M/c or B/c. On the other hand, if the divide and merge can be done using few memory transfers, then the divide-and-conquer approach will be efficient even when the cost is not dominated by the leaves.

3.2.1 Median and Selection

Let us start with a simple specific example of a divide-and-conquer algorithm: finding the median of an array in the comparison model. As usual, we will solve the more general problem of *selecting* the element of a given rank. The algorithm is a mix of scanning and a divide-and-conquer.

We begin with the classic deterministic $O(N)$-time flat-memory algorithm [BFP$^+$73]:

1. Conceptually partition the array into $\lceil N/5 \rceil$ quintuplets of five adjacent elements each.
2. Compute the median of each quintuplet using $O(1)$ comparisons.
3. Recursively compute the median of these medians (which is not necessarily the median of the original array).
4. Partition the elements of the array into two groups, according to whether they are at most or strictly greater than this median.
5. Count the number of elements in each group, and recurse into the group that contains the element of the desired rank.

To make this algorithm cache-oblivious, we specify how each step works in terms of memory layout and scanning. Step 1 is just conceptual; no work needs to be done. Step 2 can be done by two parallel scans, one reading the array 5 elements at a time, and the other writing a new array of computed medians. Assuming that the cache holds at least two blocks, this parallel scan uses $\Theta(1 + N/B)$ memory transfers. Step 3 is just a recursive call of size $\lceil N/5 \rceil$. Step 4 can be done with three parallel scans, one reading the array, and two others writing the partitioned arrays. Again, the parallel scans use $\Theta(1 + N/B)$ memory transfers provided $M \geq 3B$. Step 5 is another recursive call of size at most $\frac{7}{10}N$ (as in the classic algorithm).

Thus we obtain the following recurrence on the number of memory transfers, $T(N)$:

$$T(N) = T(N/5) + T(7N/10) + O(1 + N/B).$$

It turns out that the base case of this recurrence is important to analyze. To see why, let us start with the simple assumption that $T(O(1)) = O(1)$. Then there are N^c leaves in the recursion tree, where $c \approx 0.8397803$,[4] and each leaf incurs a constant number of memory transfers. So $T(N)$ is at least $\Omega(N^c)$, which is larger than $O(1 + N/B)$ when N is larger than B but smaller than BN^c.

Fortunately, we have a stronger base case: $T(O(B)) = O(1)$, because once the problem fits into $O(1)$ blocks, all five steps incur only a constant number of memory transfers. Then there are only $(N/B)^c$ leaves in the recursion tree, which cost only $O((N/B)^c) = o(N/B)$ memory transfers. Thus the cost per level decreases geometrically from the root, so the total cost is the cost of the root: $O(1 + N/B)$.

This analysis proves the following result:

Theorem 2. *The worst-case linear-time median algorithm, implemented with appropriate scans, uses $O(1 + N/B)$ memory transfers, provided $M \geq 3B$.*

To review, the key part of the analysis was to identify the relevant base case, so that the "overhead term" (the +1 in the divide/merge term) did not dominate the cost for small problem sizes relative to the cache. In this case, the only relevant threshold was B, essentially because the target running time depended only on B and not M. Other than the new base case, the analysis was the same as the classic analysis.

3.2.2 Binary Search (A Failed Attempt)

Another simple example of divide and conquer is binary search, with recurrence

$$T(N) = T(N/2) + O(1).$$

[4] More precisely, c is the solution to $(\frac{1}{5})^c + (\frac{7}{10})^c = 1$ which arises from plugging $L(N) = N^c$ into the recurrence for the number $L(N)$ of leaves: $L(N) = L(N/5) + L(7N/10)$, $L(1) = 1$.

In this case, each problem has only one subproblem, the left half or the right half. In other words, each node in the recursion tree has only a single branch, a sort of degenerate case of divide and conquer.

More importantly, in this situation, the cost of the leaves balance with the cost of the root, meaning that the cost of every level of the recursion tree is the same, introducing an extra $\lg N$ factor. We might hope that the $\lg N$ factor could be reduced in a blocked setting by using a stronger base case. But the stronger base case, $T(O(B)) = O(1)$, does not help much, because it reduces the number of levels in the recursion tree by only an additive $\Theta(\lg B)$. So the solution to the recurrence becomes $T(N) = \lg N - |\Theta(\lg B)|$, proving the following theorem:

Theorem 3. *Binary search on a sorted array incurs $\Theta(\lg N - \lg B)$ memory transfers.*

In contrast, in external memory, searching in a sorted list can be done in as few as $\Theta(\lg N/\lg B) = \Theta(\log_B N)$ memory transfers using B-trees. This bound is also optimal in the comparison model. The same bound is possible in the cache-oblivious setting, also using divide-and-conquer, but this time using a layout other than the sorted order. We will return to this structure in the section on data structures, specifically Section 4.1.

3.2.3 Matrix Multiplication

Matrix multiplication is a particularly good application of divide and conquer in the cache-oblivious setting. The algorithm described below, designed for square matrices, originates in [BFJ+96] where it was described in a different framework. Then the algorithm was extended to rectangular matrices and described in the cache-oblivious framework in the original paper on cache obliviousness [FLPR99]. Here we focus on square matrices, and for the moment on matrices whose dimensions are powers of two.

For contrast, let us start by analyzing the straightforward algorithm for matrix multiplication. Suppose we want to compute $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$, where each matrix is $N \times N$. For each element c_{ij} of $\mathbf{C}$, the algorithm scans in parallel row i of $\mathbf{A}$ and column j of $\mathbf{B}$. Ideally, $\mathbf{A}$ is stored in row-major order and $\mathbf{B}$ is stored in column-major order. Then each element of $\mathbf{C}$ requires at most $O(1 + N/B)$ memory transfers, assuming that the cache can hold at least three blocks. The cost could only be smaller if M is large enough to store a previously visited row or column. If $M \geq N$, the relevant row of $\mathbf{A}$ will be remembered for an entire row of $\mathbf{C}$. But for a column of $\mathbf{B}$ to be remembered, M would have to be at least N^2, in which case the entire problem fits in cache. Thus the cost of scanning $\mathbf{B}$ dominates, and we have the following theorem:

Theorem 4. *Assuming that $\mathbf{A}$ is stored in row-major order and $\mathbf{B}$ is stored in column-major order, the standard matrix-multiplication algorithm uses $O(N^2 + N^3/B)$ memory transfers if $3B \leq M < N^2$ and $O(1 + N^2/B)$ memory transfers if $M \geq 3N^2$.*

The point of this result is that, even with an ideal storage order of $\mathbf{A}$ and $\mathbf{B}$, the algorithm still requires $\Theta(N^3/B)$ memory transfers unless the entire problem fits in cache. In fact, it is possible to do better, and achieve a running time of $\Theta(N^2/B + N^3/B\sqrt{M})$. In the external-memory context, this bound was first achieved by Hong and Kung in 1981 [HK81], who also proved a matching lower bound for any matrix-multiplication algorithm that executes these additions and multiplications (as opposed to Strassen-like algorithms). The cache-oblivious solution uses the same idea as the external-memory solution: block matrices.

We can write a matrix multiplication $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$ as a divide-and-conquer recursion using block-matrix notation:

$$\begin{aligned}\begin{pmatrix} \mathbf{C}_{11} & \mathbf{C}_{12} \\ \mathbf{C}_{21} & \mathbf{C}_{22} \end{pmatrix} &= \begin{pmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{pmatrix} \cdot \begin{pmatrix} \mathbf{B}_{11} & \mathbf{B}_{12} \\ \mathbf{B}_{21} & \mathbf{B}_{22} \end{pmatrix} \\ &= \begin{pmatrix} \mathbf{A}_{11}\mathbf{B}_{11} + \mathbf{A}_{12}\mathbf{B}_{21} & \mathbf{A}_{11}\mathbf{B}_{12} + \mathbf{A}_{12}\mathbf{B}_{22} \\ \mathbf{A}_{21}\mathbf{B}_{11} + \mathbf{A}_{22}\mathbf{B}_{21} & \mathbf{A}_{21}\mathbf{B}_{12} + \mathbf{A}_{22}\mathbf{B}_{22} \end{pmatrix}.\end{aligned}$$

In this way, we reduce an $N \times N$ multiplication problem down to eight $(N/2) \times (N/2)$ multiplication subproblems. In addition, there are four $(N/2) \times (N/2)$ addition subproblems; each of these can be solved by a single scan in $O(1+N^2/B)$ memory transfers. Thus we obtain the following recurrence:

$$T(N) = 8T(N/2) + O(1 + N^2/B).$$

To make small matrix blocks fit into blocks or main memory, the matrix is not stored in row-major or column-major order, but rather in a recursive layout. Each matrix $\mathbf{A}$ is laid out so that each block $\mathbf{A}_{11}, \mathbf{A}_{12}, \mathbf{A}_{21}, \mathbf{A}_{22}$ occupies a consecutive segment of memory, and these four segments are stored together in an arbitrary order.

The base case becomes trickier for this problem, because now both B and M are relevant. Certainly, $T(O(\sqrt{B})) = O(1)$, because an $O(\sqrt{B}) \times O(\sqrt{B})$ submatrix fits in a constant number of blocks. But this base case turns out to be irrelevant. More interesting is that $T(c\sqrt{M}) = O(M/B)$, where the constant c is chosen so that three $c\sqrt{M} \times c\sqrt{M}$ submatrices fit in cache, and hence each block is read or written at most once.

With this stronger base case, the number of leaves in the recursion tree is $\Theta((N/\sqrt{M})^3)$, and each leaf costs $O(M/B)$, so the total leaf cost is $O(N^3/B\sqrt{M})$. The divide/merge cost at the root of the recursion tree is $O(N^2/B)$. These two costs balance when $N = \Theta(\sqrt{M})$, when the depth of the tree is $O(1)$. Thus the total running time is the maximum of these two terms, or equivalently up to a constant factor, the summation of the two terms.

So far we have assumed that the square matrices have dimensions that are powers of two, so that we can repeatedly divide in half. To avoid this problem, we can extend the matrix to the next power of two. So if we have an $N \times N$ matrix $\mathbf{A}$, we extend it to an $\lceil\!\lceil N \rceil\!\rceil \times \lceil\!\lceil N \rceil\!\rceil$ matrix, where $\lceil\!\lceil N \rceil\!\rceil = 2^{\lceil \lg N \rceil}$ is the *hyperceiling* operator [BDFC00]. The matrix size N^2 is increased by less than a factor of 4, so the running time increases by only a constant factor. Thus we obtain the following theorem:

Theorem 5. *For square matrices, the recursive block-matrix cache-oblivious matrix-multiplication algorithm uses $O(N^2/B + N^3/B\sqrt{M})$ memory transfers, assuming that $M \geq 3B$.*

As mentioned above, Frigo et al. [FLPR99,Pro99] showed how to generalize this algorithm to rectangular matrices. In this context, the algorithm only splits one dimension at a time, the largest of the three dimensions N_1, N_2, N_3 where $\mathbf{A}$ is $N_1 \times N_2$ and $\mathbf{B}$ is $N_2 \times N_3$. The problem then reduces to two matrix multiplications and up to one addition. The recursion then becomes more complicated to allow the case in which one matrix fits in cache, but another matrix is much larger. In these cases, several similar recurrences arise, and the number of memory transfers can go up by an additive $O(N_1 + N_2 + N_3)$.

Frigo et al. [FLPR99,Pro99] show two other facts about matrix multiplication whose details we omit here. First, the algorithms have the same memory-transfer cost when the matrices are stored in row-major or column-major order, provided we have the tall-cache assumption described in Section 2.4. Second, Strassen's algorithm for matrix multiplication leads to analogous improvement in memory-transfer cost. Strassen's algorithm has running time $O(N^{\lg 7}) \approx O(N^{2.8073549})$, and its natural application to the recursive framework described above results in a memory-transfer bound of $O(N^2/B + N^{\lg 7}/B\sqrt{M})$.

Other matrix problems can be solved via block recursion. These problems include LU factorization without pivoting [BFJ+96], LU factorization with pivoting [Tol97], and matrix transpose and fast Fourier transform [FLPR99,Pro99].

3.3 Sorting

The sorting problem is one of the most-studied problems in computer science. In external-memory algorithms, it plays a particularly important role, because sorting is often a lower bound and even an upper bound, for other problems. The original paper of Aggarwal and Vitter [AV88] proved that the number of memory transfers to sort in the comparison model is $\Theta(\frac{N}{B}\lceil\log_{M/B}\frac{N}{B}\rceil)$.

3.3.1 Mergesort

The external-memory algorithm that achieves this bound [AV88] is an (M/B)-way mergesort. During the merge, each memory block maintains the first B elements of each list, and when a block empties, the next block from that list is loaded. So a merge effectively corresponds to scanning through the entire data, for a cost of $\Theta(N/B)$ memory transfers.

The total number of memory transfers for this sorting algorithm is given by the recurrence

$$T(N) = \tfrac{M}{B}T(N/\tfrac{M}{B}) + \Theta(N/B),$$

with a base case of $T(O(B)) = O(1)$. The recursion tree has $\Theta(N/B)$ leaves, for a leaf cost of $\Theta(N/B)$. The root node has divide-and-merge cost $\Theta(N/B)$ as well, as do all levels in between. The number of levels in the recursion tree is $\log_{M/B} N$, so the total cost is $\Theta(\frac{N}{B}\log_{M/B}\frac{N}{B})$.

In the cache-oblivious context, the most obvious algorithm is to use a standard 2-way mergesort, but then the recurrence becomes

$$T(N) = 2T(N/2) + \Theta(N/B),$$

which has solution $T(N) = \Theta(\frac{N}{B} \log_2 \frac{N}{B})$. Our goal is to increase the base in the logarithm from 2 to M/B, without knowing M or B.

3.3.2 Funnelsort

Frigo et al. [FLPR99,Pro99] gave two optimal cache-oblivious algorithms for sorting: a new *funnelsort* and an adaptation of the existing *distribution sort*. We will describe a simplification to the first algorithm, called *lazy funnelsort*, which was introduced by Brodal and Fagerberg [BF02a]. Funnelsort, in turn, is a sort of lazy mergesort.

This algorithm will be our first application of the tall-cache assumption (see Section 2.4). For simplicity, we assume that $M = \Omega(B^2)$. The same results can be obtained when $M = \Omega(B^{1+\gamma})$ by increasing the constant 3; refer to [BF02a] for details. Interestingly, optimal cache-oblivious sorting is not achievable without the tall-cache assumption [BF03].

The heart of the funnelsort algorithm is a static data structure which we call a *funnel*. We delay the description of funnels to Section 4.2 when we have built up some necessary tools in the context of static data structures. For now, we treat a K-funnel as a black box that merges K sorted lists of total size K^3 using $O(\frac{K^3}{B} \log_{M/B} \frac{K^3}{B} + K)$ memory transfers. The space occupied by a K-funnel is $\Theta(K^2)$.

Once we have such a fast merging procedure, we can sort using a K-way mergesort. How should we choose K? The larger the K, the faster the algorithm, because we cannot predict the optimal (M/B) multiplicity of the merge. This property suggests choosing $K = N$, in which case the entire sorting algorithm is in the merge.[5] In fact, however, a K-funnel is fast only if it is fed at least K^3 elements. Also, a K-funnel occupies $\Theta(K^2)$ space, and we want a linear-space algorithm. Thus, we choose $K = N^{1/3}$.

Now the sorting algorithm proceeds as follows:

1. Split the array into $K = N^{1/3}$ contiguous segments each of size $N/K = N^{2/3}$.
2. Recursively sort each segment.
3. Apply the K-funnel to merge the sorted segments.

Memory transfers are made just in Steps 2 and 3, leading to the recurrence:

$$T(N) = N^{1/3}T(N^{2/3}) + O(\tfrac{N}{B} \log_{M/B} \tfrac{N}{B} + N^{1/3}).$$

The base case is $T(O(B^2)) = O(B)$ because the tall-cache assumption says that $M \geq B^2$. Above the base case, $N = \Omega(B^2)$, so $B = O(\sqrt{N})$, and the $\frac{N}{B} \log(\cdots)$ cost dominates the $N^{1/3}$ cost.

[5] What a cheat!

The recursion tree has N/B^2 leaves, each costing $O(B \log_{M/B} B + B^{1/3}) = O(B)$ memory transfers, for a total leaf cost of $O(N/B)$. The root divide-and-merge cost is $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$, which dominates the recurrence. Thus, modulo the details of the funnel, we have proved the following theorem:

Theorem 6. *Assuming $M = \Omega(B^2)$, funnelsort sorts N comparable elements in $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ memory transfers.*

It can also be shown that the number of comparisons is $O(N \lg N)$; see [BF02a] for details.

3.4 Computational Geometry and Graph Algorithms

Some of the latest cache-oblivious algorithms solve "application-level" problems in computational geometry and graph algorithms, usually with memory-transfer costs matching the best known external-memory algorithms. We refer the interested reader to [ABD+02,BF02a,KR03] for details.

4 Static Data Structures

While data structures are normally thought of as dynamic, there is a rich theory of data structures that only support queries, or otherwise do not change in form. Here we consider two such data structures in the cache-oblivious setting:

Section 4.1: search trees, which statically correspond to binary search (improving on the attempt in Section 3.2.2);
Section 4.2: funnels, a multiway-merge structure needed for the funnelsort algorithm in Section 3.3.2;

4.1 Static Search Tree (Binary Search)

The static search tree [BDFC00,Pro99] is a fundamental tool in many data structures, indeed most of those described from here down. It supports *searching* for an element among N comparable elements, meaning that it returns the matching element if there is one, and returns the two adjacent elements (next smallest and next largest) otherwise. The search cost is $O(\log_B N)$ memory transfers and $O(\lg N)$ comparisons. First let us see why these bounds are optimal up to constant factors:

Theorem 7. *Starting from an initially empty cache, at least $\log_B N + O(1)$ memory transfers and $\lg N + O(1)$ comparisons are required to search for a desired element, in the average case.*

Proof. These are the natural information-theoretic lower bounds. A general (average) query element encodes $\lg(2N+1) + O(1) = \lg N + O(1)$ bits of information, because it can be any of the N elements or in any of the $N + 1$ positions between the elements. (The additive $O(1)$ comes from Kolmogorov complexity;

see [LV97].) Each comparison reveals at most 1 bit of information, proving the the $\lg N + O(1)$ lower bound on the number of comparisons. Each block read reveals where the query element fits among those B elements, which is at most $\lg(2B+1) = \lg B + O(1)$ bits of information. (Here we are measuring conditional Kolmogorov information; we suppose that we already know everything about the N elements, and hence about the B elements.) Thus, the number of block reads is at least $(\lg N + O(1))/(\lg B + O(1)) = \log_B N + O(1)$. $\square$

The idea behind obtaining the matching upper bound is simple. Construct a complete binary tree with N nodes storing the N elements in search-tree order. Now store the tree sequentially in memory according to a recursive layout called the *van Emde Boas layout*[6]; see Figure 4. Conceptually split the tree at the middle level of edges, resulting in one *top recursive subtree* and roughly $\sqrt{N}$ *bottom recursive subtrees*, each of size roughly $\sqrt{N}$. Recursively lay out the top recursive subtree, followed by each of the bottom recursive subtrees. In fact, the order of the recursive subtrees is not important; what is important is that each recursive subtree is laid out in a single segment of memory, and that these segments are stored together without gaps.

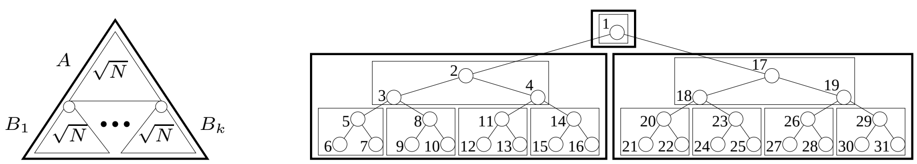

Figure 4. The van Emde Boas layout (left) in general and (right) of a tree of height 5.

To handle trees whose height h is not a power of two (as in Figure 4, right), the split rounds so that the bottom recursive subtrees have heights that are powers of two, specifically $\lceil\lceil h/2 \rceil\rceil$. This process leaves the top recursive subtree with a height of $h - \lceil\lceil h/2 \rceil\rceil$, which may not be a power of two. In this case, we apply the same rounding procedure to split it.

The van Emde Boas layout is a kind of divide-and-conquer, except that just the layout is divide-and-conquer, whereas the search algorithm is the usual tree-search algorithm: look at the root, and go left or right appropriately. One way to support the search navigation is to store left and right pointers at each node. Other implicit (pointerless) methods have been developed [BFJ02,LFN02,Oha01], although the best practical approach may yet to be discovered.

To analyze the search cost, consider the *level of detail* defined by recursively splitting the tree until every recursive subtree has size at most B. In other words, we stop the recursive splitting whenever we arrive at a recursive subtree with at most B nodes; this stopping time may differ among different subtrees because

[6] For those familiar with the van Emde Boas $O(\lg \lg u)$ priority queues, this layout matches the dominant idea of splitting at the middle level of a complete binary tree.

of rounding. Now each recursive subtree at this level of detail is stored in an interval of memory of size at most B, so it occupies at most two blocks. Each recursive subtree except the topmost has the same height [BDFC00, Lemma 1]. Because we are cutting trees at the middle level in each step, this height may be as small as $(\lg B)/2$, for a subtree of size $\Theta(\sqrt{B})$, but no smaller.

The search visits nodes along a root-to-leaf path of length $\lg N$, visiting a sequence of recursive subtrees along the way. All but the first recursive subtree has height at least $(\lg B)/2$, so the number of visited recursive subtrees is at most $1 + 2(\lg N)/(\lg B) = 2 \log_B N$. Each recursive subtree may incur up to two memory transfers, for a total cost of at most $2 + 4 \log_B N$ memory transfers. Thus we have proved the following theorem:

Theorem 8. *Searching in a complete binary tree with the van Emde Boas layout incurs at most* $2 + 4 \log_B N$ *memory transfers and* $\lg N$ *comparisons.*

The static search tree can also be generalized to complete trees with node degrees varying between 2 and some constant $\Delta \geq 2$; see [BDFC00]. This generalization was a key component in the first cache-oblivious dynamic search tree [BDFC00], which used a variation on a B-tree with constant branching factor.

4.2 Funnels

With static search trees in hand, we are ready to fill in the details of the funnelsort algorithm described in Section 3.3.2. Our goal is to develop a *K-funnel* which merges K sorted lists of total size K^3 using $O(\frac{K^3}{B} \log_{M/B} \frac{K^3}{B} + K)$ memory transfers and $\Theta(K^2)$ space. We assume here that $M \geq B^2$; again, this assumption can be weakened to $M = \Omega(B^{1+\gamma})$ for any $\gamma > 0$, as described in [BF02a].

A K-funnel is a complete binary tree with K leaves, stored according to the van Emde Boas layout. Thus, each of the recursive subtrees of a K-funnel is a $\sqrt{K}$-funnel. In addition to the nodes, edges in a K-funnel store *buffers*; see Figure 5. The edges at the middle level of a K-funnel, partitioning the funnel into two recursive $\sqrt{K}$-subfunnels, have size $K^{3/2}$ each, for a total buffer size of K^2 at that level. Buffers within the subfunnels are recursively smaller. We store these buffers of size $K^{3/2}$ in the recursive layout alongside the recursive $\sqrt{K}$-subfunnels within the K-funnel. The buffers can be stored in an arbitrary order along with the recursive subtrees.

First we analyze the space occupied by a K-funnel, which is important for knowing when a funnel fits in memory:

Lemma 3. *A K-funnel occupies* $\Theta(K^2)$ *storage, and at least* K^2 *storage.*

Proof. The size of a K-funnel, $S(K)$, satisfies the following recurrence:

$$S(K) = (1 + \sqrt{K})\ S(\sqrt{K}) + K^2.$$

This recurrence has $O(K)$ leaves each costing $O(1)$, for a total leaf cost of $O(K)$. The root divide-and-merge cost is K^2, which dominates. □

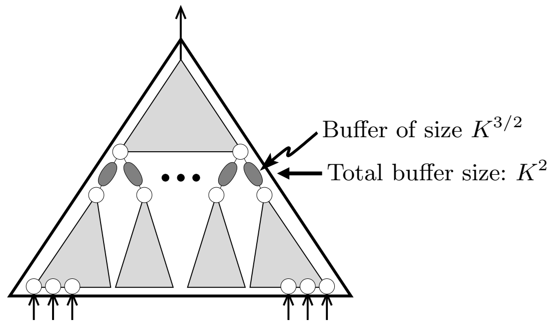

Figure 5. A K-funnel. Lightly shaded regions are $\sqrt{K}$-funnels.

For consistency in describing the algorithms, we view a K-funnel as having an additional buffer of size K^3 along the edge connecting the root of the tree to its imaginary parent. To maintain the lemma above that the storage is $O(K^2)$, this buffer is not actually stored; rather, it can be viewed as the output mechanism.

The algorithm to *fill* this buffer above the root node, thereby merging the entire input, is a simple recursion. We merge the elements in the buffers along the left and right children edges of the node, as long as those two buffers remain nonempty. (Initially, all buffers are empty.) Whenever either of the buffers becomes empty, we recursively fill it. At the bottom of the tree, a *leaf buffer* (a buffer immediately below a leaf) corresponds to one of the input lists.

The analysis considers the coarsest level of detail at which J-funnels occupy less than $M/4$ storage, so that J^2-funnels occupy at least $M/4$ storage. In symbols, $cJ^2 \le M/4$ where $c \ge 1$ according to Lemma 3. By the tall-cache assumption, the cache can fit one J-funnel and one block for each of the J leaf buffers of that J-funnel, because $cJ^2 \le M/4$ implies

$$JB \le \sqrt{M/4c}\,\sqrt{M} = M/\sqrt{4c} \le M/2.$$

Now consider the cost of extracting J^3 elements from the root of a J-funnel. Because we choose J by repeatedly taking the square root of K, and by Lemma 3, J-funnels are not too small, occupying $\Omega(\sqrt{M}) = \Omega(B)$ storage, so $J = \Omega(M^{1/4}) = \Omega(\sqrt{B})$. Loading the entire J-funnel and one block of each of the leaf buffers costs $O(J^2/B + J)$ memory transfers, which is $O(J^3/B)$ because $J = \Omega(\sqrt{B})$. The J-funnel can then merge its inputs cheaply, incurring extra block reads only to read additional blocks from the leaf buffers, until a leaf buffer empties.

When a leaf buffer empties, we recursively extract at least J^3 elements from the $\ge J$-funnel below this J-funnel, and most likely evict all of the J-funnel from cache. But these J^3 elements "pay" for the $O(J^3/B)$ cost of reading the elements back in. That is, the number of memory transfers is at most $1/B$ per element per buffer of size at least J^3 that the element enters. Because $J = \Omega(M^{1/4})$, each element enters $O(1 + (\lg K)/\frac{1}{4}\lg M) = O(1 + \log_M K)$ such buffers. In addition, we might have to pay at least one memory transfer per input list, even if they have fewer than B elements. Thus, the total number of memory transfers is

$$O(\tfrac{K^3}{B}(1+\log_M K)+K) = O(\tfrac{K^3}{B}\log_M K + K).$$

This bound is not quite the standard sorting bound plus $O(K)$, but it turns out to be equivalent in this context with a bit of manipulation. Because $M = \Omega(B^2)$, we have $\lg M \geq 2\lg B + O(1)$, so

$$\log_{M/B} K = \frac{\lg K}{\lg(M/B)} = \frac{\lg K}{\lg M - \lg B} = \frac{\lg K}{\Theta(\lg M)} = \Theta(\log_M K).$$

Thus, the logarithm base of M is equivalent to the standard logarithm base of M/B. Also, if $K = \Omega(B^2)$, then $\log_M \frac{K}{B} = \log_M K - \log_M B = \Omega(\log_M K)$, while if $K = o(B^2)$, then $\frac{K}{B}\log_M K = o(B\log_M B) = o(K)$, so the $+O(K)$ term dominates. Hence, the $+O(K)$ term dominates any savings that would result from replacing the $\log_M K$ with $\log_M \frac{K}{B}$. Finally, the missing cube on K in the $\log_M K$ term contributes only a constant factor, which is absorbed by the O.

In conclusion, we obtain the following theorem, filling in the hole in the proof of Theorem 6:

Theorem 9. *If the input lists have K^3 elements total, then a K-funnel fills the output buffer in $O(\frac{K^3}{B}\log_{M/B}\frac{K^3}{B} + K)$ memory transfers.*

5 Dynamic Data Structures

Dynamic data structures are perhaps the most exciting and most interesting kind of cache-oblivious structure. In this context, it is harder to plan ahead, because the sequence of operations performed on the structure is not known in advance. Consequently, the desired access pattern to the elements is unpredictable and more difficult to cluster into blocks.

5.1 Ordered-File Maintenance

A particularly useful tool for building dynamic data structures supports maintaining a sequence of elements in order in memory, with constant-size gaps, subject to insertion and deletion of elements in the middle of the order. More precisely, an insert operation specifies two adjacent elements between which the new element belongs; and a delete operation specifies an existing element to remove. Solutions to this *ordered-file maintenance* problem were pioneered by Itai, Konheim, and Rodeh [IKR81] and Willard [Wil92], and then adapted to the cache-oblivious context in the *packed-memory structure* of [BDFC00].

First attempts. First let us consider two extremes of trivial (inefficient) solutions, which give some intuition for the difficulty of the problem. If gaps must be completely avoided, then every insertion or deletion requires shifting the remaining elements to the right by one unit. If N is the number of elements currently in the array, such an insertion or deletion requires $\Theta(N)$ time and $\Theta(\lceil N/B \rceil)$ memory transfers in the worst case. On the other hand, if the gaps can be exponentially large, then to insert an element between two other elements, we can store it midway between those two elements. If we imagine an initial set of just two elements separated by a gap of 2^N, then N insertions can be supported in this way without moving any elements. Deletions only help, so up to N insertions and deletions can be supported in $O(1)$ time and memory transfers each.

Packed-memory structure. The packed-memory structure uses the first approach (complete re-organization) for problem sizes covered by the second approach, subranges of $\Theta(\lg N)$ elements. These subranges are organized as the leaves of a (conceptual) complete balanced binary tree. Each node of this tree represents the concatenation of several subranges (corresponding to the leaves below the node).

At each node, we impose a *density constraint* on the number of elements in that range: the range should not be too full or too empty, where the precise meaning of "too much" depends on the height of the node. More precisely, the *density* of a node is the number of elements stored below that node divided by the total capacity below that node (the length of the range of that node). Let h denote the height of the tree, so that $h = \lg N - \lg \lg N + O(1)$. The density of a node at depth d, $0 \le d \le h$, should nominally be at least $\frac{1}{2} - \frac{1}{4}d/h$ ($\in [\frac{1}{4}, \frac{1}{2}]$) and at most $\frac{3}{4} + \frac{1}{4}d/h$ ($\in [\frac{3}{4}, 1]$). Thus, as we go up in the tree, we force more stringent constraints on the density, but only by a constant factor. However, we allow a node's density to temporarily exceed these thresholds before the violation is "noticed" by the structure and then fixed.

To insert an element, we first attempt to add the node to the relevant leaf subrange. If this leaf is not completely full, we can accommodate the new element by relabeling possibly all of the elements. If the leaf is full, we say that it is *outside threshold*, and we walk up the tree until we find an ancestor that is *within threshold* in the sense that it is above its lower-bound threshold and below its upper-bound threshold. Then we *rebalance* this ancestor by redistributing all of its elements uniformly throughout the constituent leafs. Consequently, every descendent of that ancestor will be within threshold (because thresholds become only weaker farther down in the tree), so in particular there will be room in the original leaf for the new element.

Deletions are similar. We first attempt to remove the node from the relevant leaf subrange. If the leaf then falls below its lower-bound threshold ($\frac{1}{4}$ full), we walk up the tree until we find an ancestor that is within threshold, and rebalance that node. Again every descendent, in particular the original leaf, will then fall within threshold.

To support N changing drastically, we can apply the standard global rebuilding trick [Ove83]: whenever N grows or shrinks by a constant factor, rebuild the entire structure.

Analysis. The key property for the amortized analysis is that, when a node is rebalanced, its descendents are not just within threshold, but far within threshold. Specifically, the density of each node is below the upper-bound threshold and above the lower-bound threshold each by at least the difference in density thresholds between two adjacent levels: $\frac{1}{4}/h = \Theta(1/\lg N)$.[7] Thus, if the node has capacity for K elements, at least $\Theta(K/\lg N)$ elements must be inserted or deleted below the node before it falls outside threshold again. The amortized cost of inserting or deleting an element below a particular ancestor is therefore $\Theta(\lg N)$. Each element falls below $h = \Theta(\lg N)$ nodes in the tree, for a total amortized cost of $\Theta(\lg^2 N)$ per insertion or deletion.

Each rebalance operation can be viewed as two interleaved scans: one leftwards in memory for when we walk up from a right child, and one rightwards in memory for when we walk up from a left child. Thus, provided $M \geq 2B$, the block usage is optimal, so the number of memory transfers is $\Theta(\lceil(\lg^2 N)/B\rceil)$.

This analysis establishes the following theorem:

Theorem 10. *The packed-memory structure maintains N elements consecutive in memory with gaps of size $O(1)$, subject to insertions and deletions in $O(\lg^2 N)$ amortized time and $O(\lceil(\lg^2 N)/B\rceil)$ amortized memory transfers.*

The packed-memory structure has been further refined to satisfy the property that every update (in addition to every traversal) consists of $O(1)$ physical scans sequentially through memory [BCDFC02]. This property is useful in practice when caches use prefetching to speed up sequential accesses over random blocked accesses. The basic idea is to always grow the rebalancing window to the right, again to powers of two, and viewing the array as cyclic. Thus, instead of two interleaved scans as the window grows left and right, we have only one scan. The analysis can no longer use the implicit binary tree structure, but still the $O(\lceil(\lg^2 N)/B\rceil)$ amortized update bound holds.

5.2 B-trees

The standard external-memory B-tree has branching factor B, and supports insertions, deletions, and searches in $O(\log_B N)$. As we know from Theorem 7, this bound is optimal for searches. (On the other hand, better bounds can be obtained for insertions and deletions, at least in the amortized sense, when the location of the nodes to be inserted and deleted are given.) The first cache-oblivious B-tree to achieve these bounds was designed by Bender, Demaine, and Farach-Colton [BDFC00]. Two related simplifications were obtained by Bender,

[7] Note that this density actually corresponds to a positive number of elements because all nodes have at least $\Theta(\log N)$ elements below them; this fact is why we have leaves of $\Theta(\log N)$ elements.

Duan, Iacono, and Wu [BDIW02] and simultaneously by Brodal, Fagerberg, and Jacob [BFJ02].

Here we describe the simplification of [BDIW02], because it combines in a fairly simple way two structures we have already described: the static search tree from Section 4.1 and the packed-memory structure from Section 5.1. Specifically, we build a static complete binary tree with $\Theta(N)$ leaves, stored according to the van Emde Boas layout, and a packed-memory structure representing the elements. The structure maintains a fixed one-to-one correspondence (bidirectional pointers) between the cells in the packed-memory structure and the leaves in the tree. Some of these cells/leaves are occupied by elements, while others are blank.

Each internal node of the tree stores the maximum (nonblank) key of its two children, recursively. Thus, we can binary search through the tree, starting at the root, and at each step examining the key stored in the right child of the current node to decide whether to travel left or right. Because we stay entirely within the static search tree, the same analysis as Theorem 8 applies, for a cost of $O(\log_B N)$ memory transfers.

An insertion or deletion into the structure first searches for the location of the specified element (if it was not specified), and then issues a corresponding insert or delete operation to the packed-memory structure, which causes several nodes to move. Let K denote the number of moves, which is $O(\lg^2 N)$ in the amortized sense. To maintain the one-to-one correspondence, the affected $\Theta(K)$ cells each update the key of its corresponding leaf. These key changes are propagated up the tree to all ancestors, eventually updating the maximum stored in the root. The propagation proceeds as a post-order traversal of the leaves and their ancestors, so that a node is updated only after its children have been updated.

The claim is that this key propagation costs only $O(K/B + \log_B N)$ memory transfers. The analysis is similar to Theorem 8: we consider the coarsest level of detail at which each recursive subtree fits in a block, and hence stores between $\sqrt{B}$ and B nodes. The postorder traversal will proceed down the leftmost path of the tree to be updated. When it reaches a bottom recursive subtree, it will update all the elements in there; then it will go up to the recursive subtree above, and then return down to the next bottom recursive subtree. The next-to-bottom subtree and the bottom subtrees below have a total size of at least B, and assuming $M \geq 2B$, these updates use blocks optimally. In this way, the total number of memory transfers to update the next-to-bottom and bottom levels is $O(\lceil K/B \rceil)$ memory transfers. Above these levels, there are fewer than $\lceil K/B \rceil$ elements whose values must be propagated up to $\lceil K/B \rceil + \lg N$ other nodes. We can afford an entire memory transfer for each of the $\lceil K/B \rceil$ nodes. The remaining $\leq \lg N$ nodes up to the root are traversed consecutively for a cost of $O(\log_B N)$ memory transfers by Theorem 8.

The total number of memory transfers for an update, $O(\log_B N + (\lg^2 N)/B)$ amortized, is not quite what we want. However, this bound is the best known if we also want to be able to quickly traverse elements in order. For the structure described so far, this is possible by simply traversing elements in the packed-memory structure. Thus we obtain the following theorem:

Theorem 11. *This cache-oblivious B-tree supports insertions and deletions in* $O(\log_B N + (\lg^2 N)/B)$ *amortized memory transfers, searches in* $O(\log_B N)$ *memory transfers, and traversing* K *consecutive elements in* $O(\lceil K/B \rceil)$ *memory transfers.*

Following [BDFC00], we can use a standard indirection trick to remove the $(\lg^2 N)/B$ term, although we lose the corresponding traversal bound. We apply the structure to a universe of size $\Theta(N/\lg N)$, where the "elements" denote clusters of $\lg N$ elements. These clusters can be inserted and deleted into by complete reorganization; when they fill or get a constant fraction empty, we split or merge them appropriately, causing an insertion or deletion in the main toplevel structure. Thus, updates to the toplevel structure are slowed down by a factor of $\Theta(\lg N)$, reducing the cost of all but the $O(\log_B N)$ search to $O((\lg N)/B)$ which becomes negligible.

Corollary 2. *The indirection structure supports searches in* $O(\log_B N)$ *memory transfers, and insertions and deletions in* $O((\lg N)/B)$ *amortized memory transfers plus the cost of finding the node to update.*

Bender, Cole, and Raman [BCR02] have strengthened this result in various directions. First, they obtain worst-case bounds of $O(\log_B N)$ memory transfers for both updates and queries. Second, they build a *partially persistent* data structure, meaning that it supports queries about past versions of the data structure. Third, they obtain fast finger queries (searches nearby previous searches).

5.3 Buffer Trees (Priority Queues)

Desired bound. When searches do not need to be answered immediately, operations can be performed faster than $\Theta(\log_B N)$ memory transfers. For example, if we tried to sort by inserting into a cache-oblivious B-tree and then extracting the elements, it would cost $O(N \log_B N)$, which is far greater than the sorting bound. In contrast, Arge's external-memory buffer trees [Arg95] support insertions and deletions in $O(\frac{1}{B} \log_{M/B} \frac{N}{B})$ amortized memory transfers, which leads to the desired sorting bound. Buffer trees have the property that queries may be answered later than they are asked, so that they can be batched together; often (e.g., sorting) this delay is not a problem. In fact, the special delete-min operation can be answered immediately using various modifications to the basic buffer tree; see [ABD$^+$02].

Results. Arge, Bender, Demaine, Holland-Minkley, and Munro [ABD$^+$02] showed how to build a cache-oblivious priority queue that supports insertions, deletions, and online delete-min's in $O(\frac{1}{B} \log_{M/B} \frac{N}{B})$ amortized memory transfers. The difference with buffer trees is that the cache-oblivious structure does not support delayed searches. Brodal and Fagerberg [BF02b] developed a simpler cache-oblivious priority queue using the funnels and funnelsort algorithm that we saw in Sections 4.2 and 3.3.2. Their data structure (at least as described) does not support deletion of elements other than the minimum. We briefly describe their data structure, called a *funnel heap*, now.

Funnel heap. The funnel heap is composed of a sequence of several funnels of doubly exponentially increasing size; see Figure 6. Above each of these funnels, there is a corresponding buffer and 2-funnel (binary merging element) that together enforce a reasonable ordering among output elements. Specifically, we guarantee that elements in the smaller buffers are smaller in value than elements in both larger buffers and in larger funnels. This invariant follows from the use of 2-funnels to merge funnel outputs with larger buffers.

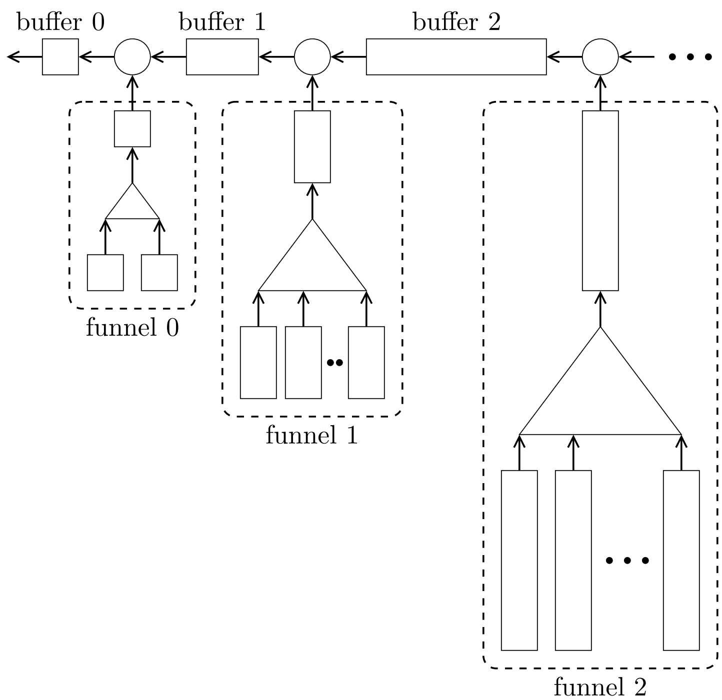

Figure 6. A funnel heap.

A delete-min operation is straightforward: it extracts the first element from the first buffer, recursively filling it if it was empty. The *filling* operation is just as in the funnel; see Section 4.2.

An insertion operation needs to be more aggressive in order to preserve the invariant. In general, we allow funnels to have *missing* input streams. Suppose that the ith funnel is the smallest funnel missing at least one input stream. Then we sort the entire contents up to the ith funnel, along with the inserted element and parts of the ith funnel, and distribute this sorted list among the buffers up to the ith funnel and a new input stream to the ith funnel. This "sweeping" process empties all funnels up to but not including the ith funnel, making it a long time before sweeping next reaches the ith funnel.

Sweeping proceeds in two main steps. First, we can immediately read off the sorted order of elements in the buffers up to and including the ith buffer (just before the ith funnel). We also include the elements within the ith funnel along the root-to-leaf path from the output stream to the missing input stream; the sorted order of these elements we can also be read off. We move these sorted elements to an auxiliary buffer, erasing the originals. Second, we extract the sorted order of the elements in funnels up to but not including the ith funnel by repeatedly calling delete-min and storing the elements into another auxiliary buffer. Finally, we merge the two sorted lists obtained from the two main steps, and distribute the first few elements among the buffers up to the ith funnel (so that they have the same size as before), then along the root-to-leaf path in the ith funnel, then storing the majority of the elements as a new input stream to the ith funnel.

The doubly exponential size increase is designed so that the size of an input stream of the ith funnel is just large enough to contain all the elements extracted from a sweep. More specifically, the ith buffer and an input to the ith funnel has size roughly $2^{(4/3)^i}$, while the number of inputs to the ith funnel is roughly the cubed root $2^{(4/3)^i/3}$. A fairly straightforward charging scheme shows that the amortized memory-transfer cost of a sweep is $O(\frac{1}{B}\log_{M/B} N^{(3/4)^j})$ for each level j. As with funnelsort, this analysis uses the tall-cache assumption; see [BF02b] for details. Even when summing from $j = 0$ to $j = \infty$, this amortized cost is $O(\frac{1}{B}\log_{M/B} N)$. Thus we obtain the following theorem:

Theorem 12. *The funnel heap supports insertion and delete-min operations in* $O(\frac{1}{B}\log_{M/B}\frac{N}{B})$ *amortized memory transfers.*

5.4 Linked Lists

A standard *linked list* supports three main operations: insertion and deletion of nodes, given the immediate predecessor or successor, and traversal to the next or previous node. These operations can be supported in $O(1)$ time on a normal pointer machine.

External-memory solution. In external memory, traversing K consecutive elements in the list can be supported in $O(\lceil K/B \rceil)$ amortized memory transfers, while maintaining $O(1)$ memory transfers per insertion or deletion. The idea is to maintain the list as a partition into $\Theta(N/B)$ pieces each with between $B/2$ and B consecutive elements. Each piece is stored consecutively in a single block, but the pieces are ordered arbitrarily in memory. To insert an element, we insert into the relevant piece, and if it does not fit, we split the piece in half, and place one half at the end of memory. To delete an element, we remove it from the relevant piece, and if that piece becomes too small, we merge it with a logically adjacent piece and resplit if necessary. The hole is filled by swapping with the last piece.

Summary of cache-oblivious solutions. In the cache-oblivious setting, we already know one solution, the packed-memory structure from Section 5.1, which supports updates in $O((\lg^2 N)/B)$ memory transfers. Because gaps are maintained to have size $O(1)$, traversing K consecutive elements in the packed-memory structure costs $O(\lceil K/B \rceil)$ memory transfers. This solution is essentially the best we know subject to the constraint of a worst-case traversal bound of $O(\lceil K/B \rceil)$.

If we are allowed a slightly larger worst-case traversal bound for a small range of K around B, namely, an additive B^ε term when $K \geq B^{1-\varepsilon}$, then there is a cache-oblivious data structure achieving $O((\lg \lg N)^{2+\varepsilon}/B)$ amortized memory transfers per update [BCDFC02]. This structure relies on the tall-cache assumption to allow the elements to get slightly out of order. The basic idea is to recursively split the N elements into $N^{1-\alpha}$ pieces of size N^α, where $0 < \alpha < 1$; the pieces store consecutive list elements in consecutive memory regions, but the pieces themselves may be stored in any order. We can afford to add an $O(1/(\lg \lg N)^{1+\varepsilon})$ fraction of extra space to each level of recursion, and the total space is still $O(N)$. This space turns out to be enough for local reorganization of pieces to obtain an $O((\lg \lg N)^{2+\varepsilon}/B)$ amortized update bound, paying $O((\lg \lg N)^{1+\varepsilon})$ at each of the $\Theta(\lg \lg N)$ levels of recursion.

On the other hand, if the traversal bound can be amortized, we can build a *self-organizing* data structure that achieves the same bounds as the external-memory linked list, except that the bounds are all amortized. This structure is essentially described in [BCDFC02], but in the context of the $O((\lg \lg N)^{2+\varepsilon}/B)$ structure, where it is adapted to obtain an $O(\lceil K/B \rceil)$ amortized traversal bound in addition to the $O(\lceil K/B \rceil + [B^\varepsilon \text{ if } K > B^\varepsilon])$ worst-case traversal bound.

Self-organizing structure. Here we describe the self-organizing data structure, which is quite simple. Updates are trivial: an insertion adds an element at the end of memory, and links it into the list; a deletion unlinks the element from the list, and just erases the element. For now, we ignore the space wasted by leaving holes in the array; these holes will be dealt with later so that the used space is always $O(n)$.

The heart of the structure is in the self-organizing traversal operations. The traversal itself follows the pointers to traverse all K elements, while observing their access pattern. Suppose that the traversed elements are organized into r *runs* of consecutive elements. Then the cost of the traversal was $O(r + \lceil K/B \rceil)$ memory transfers. We charge $r-3$ of this cost to the updates that created all runs except the first two and the last one. Then the traversal operation fixes these runs by merging all runs except the first and last run into a single run, storing it at the end of memory, and erasing the original copies of these runs. This merge operation combines $r-2$ runs into 1 run, and ensures that the charged update operations will not be charged again. Thus, the amortized cost of an update remains $O(1)$ memory transfers, and the amortized cost of a traversal reduces to $O(\lceil K/B \rceil)$ memory transfers as desired.

When the space occupied by the structure grows by a constant factor, we traverse the entire list, reducing the structure to a single run, and then shift this run to start at the beginning of memory. If there were r runs in total, the cost of this traversal is $O(r + \lceil N/B \rceil)$ memory transfers. As before, we can charge $r - 1$ of this cost to the runs that were fixed by merging into a single run. The remaining $O(\lceil N/B \rceil)$ memory transfers can be charged to the memory transfers required to have grown the structure by a constant factor in the first place, as in the standard array-doubling trick but with a $1/B$ factor.

The data structure described in [BCDFC02] behaves slightly differently when the structure grows by a constant factor: it *recompatifies* by shifting all elements left, preserving their relative order, removing all gaps, and updating pointers as we go. Updating pointers can cost $\Theta(N)$ memory transfers in the worst case, which can be too much in the worst case. Fortunately, updating pointers is only expensive between elements in different runs, so the cost of recompactification is again $O(r + \lceil N/B \rceil)$ where r is the total number of runs, and the amortization goes through as in the previous paragraph.

6 Conclusion

The cache-oblivious model is an exciting innovation that has lead to a flurry of interesting research. My favorite aspect of the model is that it is theoretically clean, avoiding all parameterization in the algorithms. As a consequence, the model requires and has lead to several new approaches and tools for forcing data locality in such a broad model. Surprisingly, in most cases, cache-oblivious algorithms match the performance of the standard, more knowledgeable, external-memory algorithms.

In addition to exploring the boundary between external-memory and cache-oblivious algorithms, we might hope that cache-oblivious algorithms actually give new insight for entering new arenas or solving old problems in external memory. For example, what can be said along the lines of self-adjusting data structures such as splay trees [ST85b], van Emde Boas priority queues [vE77,vEKZ77], or fusion trees [FW93]?

Acknowledgments

Many thanks go to Michael Bender; through many joint discussions, our understanding of cache obliviousness and its associated issues has grown significantly. Also, the realization that the worst-case linear-time median algorithm is efficient in the cache-oblivious context arose in a discussion with Michael Bender, Stefan Langerman, and Ian Munro.

References

[ABD+02] Lars Arge, Michael A. Bender, Erik D. Demaine, Bryan Holland-Minkley, and J. Ian Munro. Cache-oblivious priority queue and graph algorithm applications. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing*, pages 268–276, Montréal, Canada, May 2002.

[Arg95] Lars Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *Proceedings of the 4th International Workshop on Algorithms and Data Structures*, pages 334–345, Kingston, Canada, August 1995.

[AV88] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, September 1988.

[BCDFC02] Michael A. Bender, Richard Cole, Erik D. Demaine, and Martin Farach-Colton. Scanning and traversing: Maintaining data for traversals in a memory hierarchy. In *Proceedings of the 10th Annual European Symposium on Algorithms*, volume 2461 of *Lecture Notes in Computer Science*, pages 139–151, Rome, Italy, September 2002.

[BCR02] Michael A. Bender, Richard Cole, and Rajeev Raman. Exponential structures for efficient cache-oblivious algorithms. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming*, volume 2380 of *Lecture Notes in Computer Science*, pages 195–207, Málaga, Spain, July 2002.

[BDFC00] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Cache-oblivious B-trees. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, pages 399–409, Redondo Beach, California, November 2000.

[BDIW02] Michael A. Bender, Ziyang Duan, John Iacono, and Jing Wu. A locality-preserving cache-oblivious dynamic dictionary. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 29–38, San Francisco, California, January 2002.

[Ben00] Jon Bentley. *Programming Pearls*. Addison-Wesley, Inc., 2nd edition, 2000.

[BF02a] Gerth Stølting Brodal and Rolf Fagerberg. Cache oblivious distribution sweeping. In *Proceedings of the 29th International Colloquium on Automata, Languages, and Programming*, volume 2380 of *Lecture Notes in Computer Science*, pages 426–438, Malaga, Spain, July 2002.

[BF02b] Gerth Stølting Brodal and Rolf Fagerberg. Funnel heap – a cache oblivious priority queue. In *Proceedings of the 13th Annual International Symposium on Algorithms and Computation*, volume 2518 of *Lecture Notes in Computer Science*, pages 219–228, Vancouver, Canada, November 2002.

[BF03] Gerth Stølting Brodal and Rolf Fagerberg. On the limits of cache-obliviousness. In *Proceedings of the 35th Annual ACM Symposium on Theory of Computing*, San Diego, California, June 2003. To appear.

[BFJ+96] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. An analysis of dag-consistent distributed shared-memory algorithms. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 297–308, Padua, Italy, June 1996.

[BFJ02] Gerth Stølting Brodal, Rolf Fagerberg, and Riko Jacob. Cache oblivious search trees via binary trees of small height. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 39–48, San Francisco, California, January 2002.

[BFP+73] Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert Endre Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, August 1973.

[Flo72] R. W. Floyd. Permuting information in idealized two-level storage. In R. Miller and J. Thatcher, editors, *Complexity of Computer Calculations*, pages 105–109. Plenum, 1972.

[FLPR99] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, pages 285–297, New York, October 1999.

[FW93] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, 1993.

[HK81] Jia-Wei Hong and H. T. Kung. I/O complexity: The red-blue pebble game. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computation*, pages 326–333, Milwaukee, Wisconsin, May 1981.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 2nd edition, 1996.

[IKR81] Alon Itai, Alan G. Konheim, and Michael Rodeh. A sparse table implementation of priority queues. In S. Even and O. Kariv, editors, *Proceedings of the 8th Colloquium on Automata, Languages, and Programming*, volume 115 of *Lecture Notes in Computer Science*, pages 417–431, Acre (Akko), Israel, July 1981.

[KR03] Piyush Kumar and Edgar Ramos. I/O-efficient construction of Voronoi diagrams. Manuscript, 2003.

[LFN02] Richard E. Ladner, Ray Fortna, and Bao-Hoang Nguyen. A comparison of cache aware and cache oblivious static search trees using program instrumentation. In *Experimental Algorithmics: From Algorithm Design to Robust and Efficient Software*, volume 2547 of *Lecture Notes in Computer Science*, pages 78–92, 2002.

[LV97] Ming Li and Paul Vitányi. *An Introduction to Kolmogorov Complexity and its Applications*. Springer-Verlag, New York, second edition, 1997.

[Oha01] Darin Ohashi. Cache oblivious data structures. Master's thesis, Department of Computer Science, University of Waterloo, Waterloo, Canada, 2001.

[Ove83] Mark H. Overmars. *The Design of Dynamic Data Structures*, volume 156 of *Lecture Notes in Computer Science*. Springer-Verlag, 1983.

[Pro99] Harald Prokop. Cache-oblivious algorithms. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, June 1999.

[ST85a] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[ST85b] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, July 1985.

[Tol97] Sivan Toledo. Locality of reference in LU decomposition with partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 18(4):1065–1081, October 1997.

[vE77] Peter van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6(3):80–82, 1977.

[vEKZ77] Peter van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10(2):99–127, 1977.

[Vit01] Jeffrey Scott Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, June 2001.

[Wil92] Dan E. Willard. A density control algorithm for doing insertions and deletions in a sequentially ordered file in good worst-case time. *Information and Computation*, 97(2):150–204, April 1992.