6.897: Advanced Data Structures Spring 2005

Lecture 23 — May 3, 2005

Prof. Erik Demaine *Scribe: Dan Ports*

1 Overview

In this lecture, we examined data structures for two new models of computation. Previously we worked in the RAM or cell probe models, in which the cost of an algorithm depends on the total number of accesses to memory locations. However, this doesn't reflect the *memory hierarchy* of real computers: there is more than one layer of memory, with different access characteristics.

In a modern computer, the CPU operates on values in its registers, and can fetch values from main memory, usually via several layers of caches. Large files are stored on disk, and data is transferred from disk to main memory. The memory closest to the CPU is fastest but smallest, and the memory farthest from the CPU is largest but has the highest latency. Moreover, the slower layers of memory have greater parallelism: it takes about as long to fetch a consecutive kilobyte from disk as it does to fetch a single byte, since most of the cost is latency for the disk to find the right location. So we might as well always transfer a full *block* of data at a time; the slower layers of memory have larger block sizes. This prompts the need for algorithms and data structures that exploit locality of reference, arranging the layout of memory with data that is frequently accessed together placed in the same blocks, in order to minimize the number of blocks that need to be accessed.

This lecture covered two models of computation that take this memory hierarchy into account: the *external-memory model* and its refinement, the *cache-oblivious model*.

2 External-Memory Model

The external-memory model was introduced by Aggarwal and Vitter in 1988 [1], and is sometimes also known as the *I/O model* or *disk access model (DAM)*. It captures a two-layered memory hierarchy. The model is based on a computer with a CPU connected directly to a fast cache of size M, which is connected to a much larger and slower disk. Both the cache and disk are divided into blocks of size B; the cache thus holds $\frac{M}{B}$ blocks, while the disk can hold many more. The CPU can only operate directly on the data stored in cache. Algorithms can make *memory transfer* operations, which read a block from disk to cache, or write a block from cache to disk. The cost of an algorithm is the number of memory transfers required; operations on cached data are considered free.

Clearly any algorithm that has running time $T(N)$ in the RAM model can be trivially converted into an external-memory algorithm that requires no more than $T(N)$ memory transfers, by ignoring locality. We want to do better than this. Ideally, we would like to achieve $\frac{T(N)}{B}$, but this optimum is often hard to achieve.

2.1 Searching

The ideal search tree structure for the external-memory model is a B-tree with branching factor $\Theta(B)$, such that each node fits in a block. This allows us to perform INSERT, DELETE, and SEARCH operations using $O\left(\log_{B+1} N\right)$ memory transfers and $O(\log N)$ time in the comparison model.

This is an optimal bound for the external-memory comparison-model searching, as an information theoretic argument shows. Suppose that we wish to discover where some element x is located in an array of n elements. Expressing the answer (e.g. as an index into the array) requires at least $\lg(N+1)$ bits. Each time a block is transferred, it reads at most $\Theta(B)$ elements, learning where x fits into these B elements. This provides no more than $O(\lg(B+1))$ bits of information. Since we need to determine $\lg(N+1)$ bits of information, at least $\frac{\lg(N+1)}{O(\lg(B+1))} = \Omega\left(\lg_{B+1} N\right)$ memory transfers are required.

2.2 Sorting

The natural companion to the searching problem is sorting. In the RAM model, we can sort N elements by inserting them into a B-tree and performing N DELETE-MIN operations. This gives us $O(N \lg N)$ runtime, which we know to be optimal. We can do the same in the external-memory model, and perform sorting in $O\left(n \lg_{B+1} N\right)$ memory transfers. But this is *not* optimal.

We can do better with $\frac{M}{B}$*-way mergesort*, which divides a sorting problem into $\frac{M}{B}$ subproblems, recursively sorts them, and merges them. This gives us a cost of $O\left(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B}\right)$ memory transfers. In fact, this is optimal: it can be shown by a similar information-theoretic argument to the one above that sorting N elements in the comparison model requires $\Omega\left(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B}\right)$ memory transfers [1].

2.3 Permutation

The permutation problem is, given N elements and a permutation, to rearrange the elements according to that permutation. We can do this in $O(N)$ memory transfers by moving each element to its new position, ignoring locality. We can also solve the problem with $O\left(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B}\right)$ memory transfers by sorting the elements, sorting the permutation, and then applying the permutation in reverse. This gives us a bound of $O\left(\min\left(N,\ \frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B}\right)\right)$.

In the *indivisible model*, where elements cannot be divided between blocks, there is a matching lower bound. It remains an open problem whether one can do better in a general model.

2.4 Buffer Trees

We noted in Section 2.2 that B-trees are optimal for searching, but cannot be used as the basis for an optimal sorting algorithm. The buffer tree [2] is a data structure that provides INSERT, DELETE, DELETE-MIN, and BATCHED-SEARCH operations using $O\left(\frac{1}{B} \log_{\frac{M}{B}} \frac{N}{B}\right)$ amortized memory transfers per operation. (Notice that this bound is usually $o(1)$, so it has to be amortized!) BATCHED-

SEARCH is a delayed query: it performs a search for a particular value in the tree as it appears at a particular time, but does not give the results immediately; they become available later, after other operations have been performed.

3 Cache-Oblivious Model

The cache-oblivious model is a variation of the external-memory model introduced by Frigo, Leiserson, Prokop, and Ramachandran in 1999 [10, 11]. In this model, the algorithm does not know the size M of its cache, or the block size B. This means that it cannot perform its own memory management, explicitly performing memory transfers. Instead, algorithms are RAM algorithms, and block transfers are performed automatically, triggered by element accesses, using the offline optimal block replacement strategy. Though the use of the offline optimal block replacement strategy sounds like it would pose a problem for practical applications, in fact there are a number of competitive block replacement strategies (FIFO, LRU, etc.) that are within a factor of 2 of optimal given a cache of twice the size.

Though the algorithm does not know the size M of its cache, we will routinely assume that $M \geq c \cdot B$ for any constant c, i.e. that the cache can hold at least c blocks. (In practice, however, the algorithms we consider will not require c to be very large at all.)

From a theoretical standpoint, the cache-oblivious model is appealing because it is very clean. A cache-oblivious algorithm is simply a RAM algorithm; it is only the analysis that differs. The cache-oblivious model also works well for multilevel memory hierarchies, unlike the external-memory model, which only captures a two-level hierarchy. Since a cache-oblivious algorithm provides the desired result for any cache size and block size, it will work with *every* cache and block size at each level of the hierarchy, thus giving the same bound overall.

3.1 Survey of Results

B-tree A cache-oblivious variant of the B-tree [4, 5, 9] provides the INSERT, DELETE, and SEARCH operations with $O\left(\log_{B+1} N\right)$ memory transfers, as in the external-memory model.

Sorting As in the external-memory model, sorting N elements can be performed cache-obliviously using $O\left(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B}\right)$ memory transfers [10, 7]. Note, however, that this requires the *tall-cache assumption*: that $M = \Omega\left(B^{1+\epsilon}\right)$. The external-memory sorting algorithm does not require this to be the case. In [8], it was shown that the tall-cache assumption is necessary.

Priority queue A priority queue can be implemented that executes the INSERT, DELETE, and DELETE-MIN operations in $O\left(\frac{1}{B} \log_{\frac{M}{B}} \frac{N}{B}\right)$ memory transfers [3, 6].

3.2 Static Search Trees

We will now see how to construct a static search tree that can perform searches using $O\left(\log_{B+1} N\right)$ memory transfers. We do this by constructing a complete binary search tree with the N elements stored in sorted order.

In order to achieve the desired memory transfer bound, we will store the tree on disk using a representation known as the *van Emde Boas layout* [11]. It uses the van Emde Boas idea of dividing the tree at the middle level of edges, giving a top subtree of $\sqrt{N}$ elements, and $\sqrt{N}$ subtrees of $\sqrt{N}$ elements each. We recursively lay out each of the $\sqrt{N} + 1$ subtrees, then concatenate them, ensuring that each subtree is stored consecutively.

Claim 1. *Performing a search on a search tree in the van Emde Boas layout requires* $O\left(\log_{B+1} N\right)$ *memory transfers.*

Proof. We can stop our analysis when we reach a subtree that has size less than B: the algorithm will continue, but since the subtree fits entirely within one block, it will fit in the cache, and no further memory transfers will be required. So consider the level of recursion that "straddles" B: the whole structure at that level has size greater than B, but each subtree has size at most B. Each subtree requires at most two memory transfers to access (it can fit in one block, but it might actually need to be stored in two blocks if it is "out of frame" with the block boundaries.)

Since we are considering the level that straddles B, each subtree must have height at least $\frac{1}{2} \lg B$: otherwise two levels of these subtrees would have height less than $\lg B$, and thus contain less than B elements, violating the definition of the level that straddles B. So we will need to access $\frac{\lg N}{\frac{1}{2} \lg B} = 2 \log_B N$ subtrees. Each subtree access can be done in 2 memory transfers, so we need no more than $4 \log_B N$ memory transfers to perform a search. □

Note that this technique can be generalized to trees whose height is not a power of 2, and non-binary trees of constant degree (except for degree 1, of course).

3.3 Ordered File Maintenance

Before we proceed to describe how to make these search trees dynamic, we will need a result to use as a black box.

The ordered file maintenance (OFM) problem is to store N elements in order in an array of size $O(N)$. This array can have gaps, but any two consecutive elements must be separated by at most $O(1)$ gaps. An ordered file maintenance data structure must support two operations: to INSERT an element between two other elements, preserving the order of the array, and to DELETE an element.

Our black box can accomplish these two operations by rearranging $O\left(\lg^2 N\right)$ consecutive elements, amortized. The next lecture will describe how to do this.

3.4 Dynamic Search Trees

Now we can turn to how to build a dynamic search tree. This description follows that of [5], a simplification of [4].

We store our elements in an ordered file structure, then build a static search tree "on top" of the ordered file structure: the leaves correspond to the array slots in the ordered file structure (including blank spaces where there are gaps in the ordered file array). The search tree invariant is that each internal node stores the maximum value of its children (ignoring the blank spaces due to empty slots).

This structure allows us to perform searches with $O\left(\lg_{B+1} N\right)$ memory transfers in the standard way. Performing an insertion is a bit more complicated. We begin by performing a SEARCH to find the predecessor or successor of our new element, and thus find out where to insert it into the ordered file maintenance structure. Performing the OFM insert changes $O\left(\lg^2 N\right)$ cells. Then, for all of these cells, we update the corresponding leaves of the search tree, and propagate the changes upward in a post-order traversal of the changed leaves and their ancestors.

Claim 2. *If k cells in the ordered file structure are changed, the cost of updating the corresponding leaves and ancestors in the search tree $O\left(\lg_{B+1} N + \frac{k}{B}\right)$ memory transfers.*

Proof. As before, consider the level of detail straddling B. Consider the bottom two levels of subtrees (each of size at most B). We are performing updates in a post-order traversal of the tree. For the bottom two levels, this is essentially a scanning over the subtrees, each of which fits in one block: we perform the updates in one subtree at the bottom level, then move up to the subtree at the next higher level to perform some updates before moving on to the next subtree at the bottom level. So as long as our cache is big enough to hold six blocks at once (for a block of the ordered file maintenance structure, the subtree at the bottom level, and the subtree at the second-from-bottom level — actually two each, since we do not know where the block boundaries are and may be "out of frame" as before), we need $O\left(\frac{k}{B}\right)$ memory transfers to update the bottom two levels.

Now we consider the updates above the bottom two levels. Note that the larger subtrees composed of the bottom two levels have size $J > B$ since we are considering the level of detail that straddles B. This means that after the bottom two levels, J leaves are reduced to one node, so there are $O\left(\frac{K}{J}\right) = O\left(\frac{K}{B}\right)$ elements to be traversed until the least common ancestor has been reached. We can afford one memory transfer per element. Then, after the least common ancestor has been reached, there are $O\left(\lg_{B+1} N\right)$ elements on the path to the root. So the total cost is $O\left(\lg_{B+1} N + \frac{K}{B}\right)$ memory transfers. □

We now have a tree that can perform INSERT operations in $O\left(\lg_{B+1} N + \frac{\lg^2 N}{B}\right)$ amortized memory transfers (and DELETE operations with the same cost, in a similar way), and searches in $O\left(\lg_{B+1} N\right)$ memory transfers.

To eliminate the $O\left(\frac{\lg^2 N}{B}\right)$ factor, we can use indirection. We cluster elements into groups of $O(\lg N)$ elements, and store the minimum of the group in the ordered file maintenance structure and search tree as before. Now performing a INSERT operation requires rewriting an entire group, but this costs $O\left(\frac{\lg N}{B}\right) = O\left(\lg_B N\right)$ memory transfers. If the size of a group grows too large after $O(\lg N)$ INSERTs, we may need to split the group, but we can amortize away the cost of this in the standard way. Now the cost of a update operation is $O\left(\lg_{B+1} N + \frac{\lg N}{B}\right) = O\left(\lg_{B+1} N\right)$ amortized memory transfers.

References

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.

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

[3] L. Arge, M. A. Bender, E. D. Demaine, B. Holland-Minkley, and J. I. Munro. Cache-oblivious priority queue and graph algorithm applications. In *Proc. STOC '02*, pages 268–276, May 2002.

[4] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. In *Proc. FOCS '00*, pages 399–409, Nov. 2000.

[5] M. A. Bender, Z. Duan, J. Iacono, and J. Wu. A locality-preserving cache-oblivious dynamic dictionary. In *Proc. SODA '02*, pages 29–38, 2002.

[6] G. S. Brodal and R. Fagerberg. Funnel heap — a cache oblivious priority queue. In *Proc. ISAAC '02*, pages 219–228, 2002.

[7] G. S. Brodal and R. Fagerberg. Cache oblivious distribution sweeping. In *Proc. ICALP '03*, page 426, 2003.

[8] G. S. Brodal and R. Fagerberg. On the limits of cache-obliviousness. In *Proc. STOC '03*, pages 307–315, 2003.

[9] G. S. Brodal, R. Fagerberg, and R. Jacob. Cache oblivious search trees via binary trees of small height. In *Proc. SODA '02*, pages 39–48, 2002.

[10] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. FOCS '99*, pages 285–298, 1999.

[11] H. Prokop. Cache-oblivious algorithms. Master's thesis, Massachusetts Institute of Technology, June 1999.