6.851: Advanced Data Structures Spring 2010

Lecture 20 — 22 April, 2010

Prof. Erik Demaine *Scribe: Eric Liu*

1 Memory Hierarchies and Models of Them

So far in class, we have worked with models of computation like the word RAM or cell probe models. These models account for communication with memory one word at a time: if we need to read 10 words, it costs 10 units.

On modern computers, this is virtually never the case. Modern computers have a memory hierarchy to attempt to speed up memory operations. The typical levels in the memory hiearchy are:

Memory Level	Size	Response Time
CPU registers	$\approx$ 100B	$\approx$ 0.5ns
L1 Cache	$\approx$ 64KB	$\approx$ 1ns
L2 Cache	$\approx$ 1MB	$\approx$ 10ns
Main Memory	$\approx$ 2GB	$\approx$ 150ns
Hard Disk	$\approx$ 1TB	$\approx$ 10ms

It is clear that the fastest memory levels are substantially smaller than the slowest ones. Generally, each level has a direct connection to the level immediately below it. (In addition, the faster, smaller levels are substantially more expensive to produce, so do not expect 1GB of register space any time soon.) Additionally, many of the levels communicate in blocks. For example, asking RAM to read one integer will typically also transmit a "block" of nearby data. So processing the block members costs no additional memory transfers. This issue is exacerbated when communicating with the disk: the 10ms is dominated by the time needed to find the data (move the read head over the disk). Modern disks are circular, spinning at 7200rpm, so once the head is in position, reading all of the data on that "ring" is practically free.

This speaks to a need for algorithms that are designed to deal with "blocks" of data. Algorithms that properly take advantage of the memory hiearchy will be much faster in practice; and memory models which correctly describe the hiearchy will be more useful for analysis. We will see some fundamental models with some associated results today.

2 External Memory Model

The external memory model was introduced by Aggarwal and Vitter in 1988 [1]; it is also called the "I/O Model" or the "Disk Access Model" (DAM). The external memory model simplifies the memory hierachy to just two levels. The CPU is connected to a fast cache of size M; this cache in turn is connected to a slower disk of effectively infinite size. Both cache and disk are divided into blocks of size B. Reading and writing one block from cache to disk costs 1 unit. Operations on blocks in RAM are free.

Clearly any algorithm from say the word RAM model with running time $T(N)$ requires no worse

than $T(N)$ memory transfers in the external memory model (at most one memory transfer per operation). The lower bound, which is usually harder to obtain, is $\frac{T(N)}{B}$, where we take perfect advantage of cache locality; i.e., each block is only read/written a constant number of times.

Note that, the external memory model is a good first approximation to the slowest connection in the memory hiearchy. For a large database, "cache" could be system RAM and "disk" could be the hard disk. For a small simulation, "cache" might be L2 and "disk" could be system RAM.

2.1 Scanning

Clearly, scanning N items costs $O(\lceil \frac{N}{B} \rceil)$ memory transfers.

2.2 Searching

Searching is accomplished with a B-Tree using a branching factor that is ΘB. Insert, delete, and predecessor/successor searches are then handled with $O(\log B + 1N)$ memory transfers. This will require $O(\log N)$ time in the comparision[1] model.

The $O(\log_{B+1} N)$ bound is in fact optimal; we can see this from an information theoretic argument. To locate a query among N items, we need $\log(N+1)$ bits of information. Each read from cache (one block) tells us where the query fits among B items, yielding $\log(B+1)$ bits of information. Thus we need at least $\frac{\log(N+1)}{\log(B+1)}$ or $\Omega(\log_{B+1} N)$ memory transfers to reveal all $\log(N+1)$ bits.

2.3 Sorting

In the word RAM model, a B-Tree can sort in optimal time: insert all elements and successively delete the minimal element. The same technique yields $O(\frac{N}{B} \log_{B+1} N)$ (amortized) memory transfers in the external memory model, which is **not optimal**.

An optimal algorithm is a $\frac{M}{B}$-way version of mergesort. It obtains performance by solving subproblems that fit in cache, leading to a total of $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ memory transfers. This bound is actually optimal in the comparison model [1].

2.4 Permutation

The permutation problem is: given N elements in some order and a new ordering, rearrange the elements to appear in the new order. Naively, this takes $O(N)$ operations: just swap each element into its new home. It may be faster to make the key equal to the permutation ordering and then apply the above optimal sort. This gives us a bound of $O(\min\{N, \frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B}\})$ (amortized). Note that this result holds in the "indivisible model," where words cannot be cut up and re-packed into other words.

[1] This is not the standard comparison model; here we mean that the only permissible operation on elements is to compare them pairwise.

2.5 Buffer Trees

Buffer trees are the external memory priority queue. They also achieve the sorting bound if all elements are inserted then the minimum deleted sequentially. Buffer trees achieve $O(\frac{1}{B}\log_{\frac{M}{B}}\frac{N}{B})$ (amortized) memory transfers per operation. The operations are batched updates and delayed queries; delete-min queries are free.

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 block size, B, or the total memory size, M. This means that the blocking is implicit: when the algorithm asks for a piece of data, it gets a whole block of memory. But it does not know how much data arrived; i.e., the algorithms must function for any B and M combintaion.

For modeling assumptions, automatic block transfers are triggered by word access with an offline optimal block replacement scheme[2]. More practical schemes like LRU (Least Recently Used) or FIFO (First In First Out) are 2-competitive with the offline optimal algorithm if the offline routine's cache size is only half as large. But putting a constant factor on M does not change our bounds.

Also, we will be using the "Tall Cache Assumption." That is, even though we do not know M, we assume that $M \geq cB$ for some constant c. That is, the cache is able to hold "enough" blocks.

3.1 Scanning

The bound is identical to external memory: $O(\lceil\frac{N}{B}\rceil)$ memory transfers.

3.2 Search Trees

A cache-oblivious variant of the B-tree [4, 5, 9] provides the `INSERT`, `DELETE`, and `SEARCH` operations with $O\log_{B+1} N$ (amortized) memory transfers, as in the external-memory model. The latter half of this lecture concentrates on cache-oblivious B-Trees.

3.3 Sorting

As in the external-memory model, sorting N elements can be performed cache-obliviously using $O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$ memory transfers [10, 7]. This algorithm uses a tall cache assumption: $M = \Omega(B^{1+\epsilon})$, which is necessary [10, 7].

[2]Offline optimal here means that given the user's sequence of memory accesses, the fewest possible number of blocks are moved.

3.4 Permuting

The min{} is no longer possible[10, 7], but both component bounds from the external memory model are still valid.

3.5 Priority Queues

Again using a tall cache assumption, a priority queue can be implemented that executes the `INSERT`, `DELETE`, and `DELETE-MIN` operations in $O\left(\frac{1}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$ (amortized) memory transfers [3, 6]. We will explore the details of this data structure in the next lecture.

4 Cache Oblivious B-Trees

Now we will discuss and analyze the data structure leading to the previously stated cache-oblivious search tree result. We will use a data structure that shares many features with the standard B-tree. It will require modification since we do not know B, unlike in the external memory model. To start, we will build a static structure supporting searches in $O(\log_{B+1} N)$ time.

4.1 Static Search Trees

First, we will construct a *complete* binary search tree over all N elements. To achieve the $\log_{B+1}$ complexity on memory transfers, the tree will be represented on disk in the *van Emde Boas* layout[11]. The vEB layout is defined recursively. The tree will be split in half by height; the upper subtree has height $\frac{1}{2}\log N$ and it holds $O(\sqrt{N})$ elements. The top subtree in turn links to $O(sqrtN)$ subtrees each with $O(\sqrt{N})$ elements. Each of the $\sqrt{N}+1$ subtrees is in turn divided up according to the vEB layout. On disk, the upper subtree is stored first, with the bottom subtrees laid out sequentially after it. This layout can be generalized to trees where the height is not a power of 2 with a $O(1)$ branching factor (≥ 2)[4].

Note that when the recursion reaches a subtree that is small enough (size less than B), we can stop. Smaller working sets would not gain or lose anything since they would not require any additional memory transfers.

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

Proof. Consider the level of detail that "straddles" B. That is, continue cutting the tree in half until the height of *each subtree* first becomes $\leq \log B$. At this point, the height must also be greater than $\frac{1}{2}\log B$. Note that the size of the subtrees is at least $\sqrt{B}$ and at most B, giving between B and B^2 elements at this "straddling" level.

In the (search) walk from root to leaf, we will access no more than $\frac{\log N}{\frac{1}{2}\log B}$ subtrees[3]. Each subtree

[3]The tree height is $O(\log N)$; the subtree heights are $\Omega(\log B)$.

at this level then requires at most 2 memory transfers to access[4]. Thus the entire search requires $O(4 \log_B N)$ memory transfers.

□

4.2 Dynamic Search Trees

Note that the following description is modeled after the work of [5], which is a simplification of [4].

4.2.1 Ordered File Maintenance

First, we will need an additional supporting data structure that solves the Ordered File Maintenance (OFM) problem. For now, treat it as a black box; the details will be given in the next lecture.

The OFM problem involves storing N elements in an array of size $O(N)$ with a specified ordering. Note that this implies gaps of size $O(1)$ are permissible. The OFM data structure then supports `INSERT` (between two consecutive items) and `DELETE`. It accomplishes each operation by moving elements in an interval of size $O(\log^2 N)$ (amortized) via $O(1)$ interleaved scans. One might imagine this data structure backing the file system on a computer.

4.2.2 Back to Search Trees: Linking vEB layout and OFM

First, initialize and construct an OFM over the N keys. Recall that the OFM is backed by an array, so we can build a complete BST on top of the OFM array. The BST has one leaf for each entry in the OFM array, *including the blank spaces*. Only the leaves contain original keys; the non-leaf nodes only contain copies. Non-leaf nodes take their value from the maximum of their two children (empty is considered to be smaller than any other value). The BST is static since the will only move data around within its array.

With this structure, `SEARCH` still requires $O(\log_{B+1} N)$ memory transfers. Searching now involves checking each node's left child in to decide whether to branch left or right; this at most doubles the number of comparisons over a standard BST. `INSERT(X)` is more complex. The operations are as follows: 1) search for X to find its predecessor and successor; 2) use the OFM to insert X in position (which changes $O(\log^2 N)$ elements); and 3) fix the BST by updating leaves changed by the OFM. Note that updates must be performed *post-order* since both child keys are required to compute the parent. `DELETE` is essentially the same: search in the BST, delete from OFM, update BST.

To prove that `INSERT` and `DELETE` fall within the desired memory transfer bound, we will first prove a weaker result which we will fix afterward.

Claim 2. *If the OFM changes k leaves, then updating the BST requires $O(\frac{k}{B} + \log_{B+1} N)$ memory transfers.*

[4] Although the subtrees each have size at most B, they may not align to cache boundaries; e.g., half in cache-line i and half in line $i + 1$.

Proof. As before, consider the level of recursion "straddling" B. Consider the bottom two levels: 1) chunks of size $\leq B$; and 2) chunks of size $> B$. Type 2 chunks contain $O(B^2 + 1)$ type 1 chunks. In particular, we want to consider the group of type 2 chunks spanning the k leaves that the OFM changed. As long as $M \geq 4B^5$, the cost of updating one type 2 chunk (equal to the cost of scanning it) is $O(\frac{|type2|}{B})$ memory transfers. Summing across all the type 2 chunks, these bottom-most updates cost $O(\frac{k}{B})$ memory transfers.

Say that each type 2 chunk has size J. After the previous series of updates at the bottom 2 levels, each type 2 chunks is effectively reduced to 1 node. Now consider the larger subtrees composed of those type 2 chunks; again these larger subtrees also have size $J > B$ since we are at the level of detail "straddling" B. So there are $O(\frac{k}{J}) = O(\frac{k}{B})$ BST nodes to traverse before the LCA of the k changed elements is reached. Above the LCA, there are $O(\log_{B+1} N)$ elements remaining on the path to the root, each of which may incur a memory transfer. So the total then is $O(\frac{k}{B} + \log_{B+1} N)$ memory transfers.

□

At this point, we can perform updates with $O(\frac{\log^2 N}{B} + \log_{B+1} N)$ memory transfers. This is too costly if $B = o(\log N \log\log N)$. We can rid ourselves of the $\frac{\log^2 N}{B}$ component using a technique we saw first in the lecture about y-fast trees: *indirection*.

4.2.3 Wrapping Up: Adding Indirection

Recall that indirection involves grouping the elements into $\Theta(\frac{N}{\log N}$ groups of size $\Theta(\log N)$ elements each. Now we will create the OFM structure over the *minimum* of each group. As a result, the vEB-style BST over the OFM array will act over $\Theta(\frac{N}{\log N}$ leaves instad of $\Theta(N)$ leaves.

The vEB storage allows us to search the top structure in $O(\log_{B+1} N)$; we will also have to scan one lower group at cost $O(\frac{\log N}{B})$ for a total search cost of $O(\log_{B+1} N)$ memory transfers. Now `INSERT` and `DELETE` will require us to reform an entire group at a time, but this costs $O(\frac{\log N}{B}) = O(\log_B N)$ memory transfers, which is sufficiently cheap. As with y-fast trees, we will also want to manage the size of the groups: they should be between 25% and 100% full. Groups that are too small or too full can be merged then split or merged (respectively) as necessary by destroying and/or forming new groups. We will need $\Omega(\log N)$ updates to cause a merge or split. Thus the merge and split costs can be charged to the updates, so their amortized cost is $O(1)$. The minimum element only needs to be updated when a merge or split occurs. So expensive updates to the vEB structure only occur every $O(\log N)$ updates at cost $O(\frac{\log_{B+1} N + \frac{\log^2 N}{B}}{\log N}) = O(\frac{\log N}{B}) = O(\log_{B+1} N)$. Thus all operations (`SEARCH`, `INSERT`, and `DELETE`) cost $O(\log_{B+1} N)$.

References

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

[5] At any given time, we will be updating one of the bottom type 1 chunks within a single type 2 chunk. This requires interacting with that bottom chunk *and* the top chunk in this level of the vEB layout.

[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.