6.851: Advanced Data Structures Spring 2012

Lecture 7 — 8 March, 2012

Prof. Erik Demaine

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 only the level directly below it in the hierarchy. In addition, the faster, smaller levels are substantially more expensive to produce, so do not expect 1GB of register space any time soon. Many of the levels communicate in blocks. For example, asking Main Memory to read one integer will typically also transmit a "block" of nearby data. So processing the other block members requires 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 much faster.

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 much slower disk of effectively infinite size. Both cache and disk are divided into blocks of size B, so there are $\frac{M}{B}$ blocks in the cache. Transfering one block from cache to disk (or vice versa) costs 1 unit. Memory operations on blocks resident in the cache are free. Thus, the natural goal is to minimize the number of transfers between cache and disk.

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

Scanning N items trivially costs $O(\lceil \frac{N}{B} \rceil)$ memory transfers. The ceiling is important, because if $N = o(B)$ then we end up reading more items than necessary.

2.2 Searching

Searching is accomplished with a B-Tree using a branching factor that is $\Theta(B)$. In practice, we would want it to be exactly $B+1$ so that a single node fits in one memory block and we always have a branching factor ≥ 2. Insert, delete, and predecessor/successor searches are then handled with $O(\log_{B+1} N)$ memory transfers. This will require $O(\log N)$ time in the comparision[1] model; so there is an improvement by a factor of $O(\log B)$.

The $O(\log_{B+1} N)$ bound is in fact **optimal** for searches; we can see this from an information theoretic argument. We want to figure out where our items fits amongst all the N items. There are $N+1$ positions where our item can fit and we need $\Theta(\log N + 1)$ bits of information to specify one of those positions. Each read from cache (one block) tells us where the item fits among B items, yielding $O(\log(B+1))$ bits of information. Thus we need at least $\Theta(\frac{\log N}{\log(B+1)})$ or $\Omega(\log_{B+1} N)$ memory transfers to reveal all $\Theta(\log(N+1))$ bits.

For insert/delete, however, this bound is **not optimal**.

2.3 Sorting

In the word RAM model, a B-Tree can sort in optimal time: just insert all elements and then do an inorder traversal. However, the same technique yields $O(N \log_{B+1} N)$ (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].

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

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 position. It may be faster to assign each item a key equal to its permutation ordering and then apply the aforementioned optimal sort algorithm. This gives us a bound of $O(\min\{N, \frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\})$ (amortized). Note that this result only holds in the "indivisible model," where words cannot be cut up and re-packed into other words.

2.5 Buffer Trees

Buffer trees are essentially a dynamic version of sorting. Buffer trees achieve $O(\frac{1}{B}\log_{\frac{M}{B}}\frac{N}{B})$ (amortized) memory transfers per operation. Bound has to be amortized because it is typically $o(1)$. They also achieve the optimal sorting bound if all elements are inserted then the minimum deleted sequentially. The operations are batched updates (insert, delete-min) and delayed queries. If we do a query, we don't get the answer right then; it's delayed. There's a new operation called flush, which returns answers to all unanswered queries so far in $O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$ time. So if we do $O(N)$ operations and then do flush, it doesn't take any extra time. Find-min queries are free; the min item is maintained all the time. They can be used as efficient external memory priority queues.

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. In particular, our algorithms will look like normal RAM algorithms, but we will analyze them differently.

For modeling assumptions, we will assume that the caching is done automatically. We will also assume that the caching mechanism is optimal (in the offline sense). In practice, this can be achieved with either LRU (Least Recently Used) or FIFO (First In First Out) since they are $O(1)$-competitive with the offline optimal algorithm if they have a cache with twice the size. Since none of our algorithms change by replacing the cache size M with $2M$, this does not change our bounds.

Good algorithms for this model give us good algorithms for all values of B and M. They are especially useful for multi-level caches and for caches with changing values of B and M.

3.1 Scanning

The bound is identical to external memory: $O(\lceil\frac{N}{B}\rceil)$ memory transfers. We can use the same algorithm as before, since we only depend on B in the analysis.

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 will be done in the next lecture.

3.4 Permuting

The min{} is no longer possible[10, 7], since that depends on knowing M and B. Both component bounds (sorting or linear) from the external memory model are still valid, we just don't which gives the minimum.

3.5 Priority Queues

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]. This assumes a tall-cache which states: $M = \Omega(B^{1+\epsilon})$.

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 $O(\sqrt{N})$ 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[2]. Each subtree at this level then requires at most 2 memory transfers to access[3]. 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 in the structure. The OFM data structure then supports `INSERT` (between two given consecutive items) and `DELETE` operations. It accomplishes each by moving elements in an interval of size $O(\log^2 N)$ (amortized) via $O(1)$ interleaved scans. It follows that the operations require $O\left(\frac{\log^2 N}{B}\right)$ transfers.

OPEN It is conjectured that $O(\log^2 N)$ (amortized) time is optimal.

4.2.2 Back to Search Trees: Linking vEB layout and OFM

Now, we want to also enable fast `SEARCH` operations.

To do so, we construct an OFM over the N keys, plus the necessary gaps. Then, we create a vEB-arranged tree over the OFM items, adding pointers between the leaves of the tree and the corresponding items in the OFM. The result is illustrated in Figure 1.

Notice that the keys are all at the leaves. Inner nodes contain the maximum of the subtrees rooted in their children (gaps count as $-\infty$).

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

[3] 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 $i+1$.

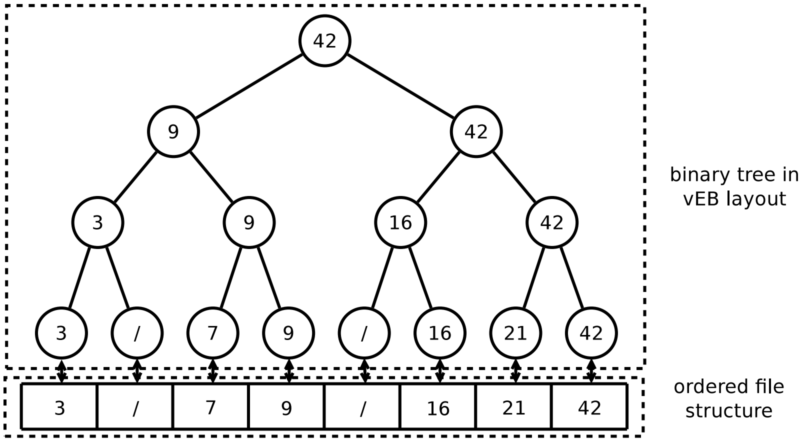

Figure 1: A tree with vEB layout to search in an OFM. The / symbol represents gaps.

With this structure, SEARCH is done by checking each node's left child to decide whether to branch left or right. It takes $O(\log_{B+1} N)$.

INSERT requires first searching for the successor of the item to be inserted. Then, we insert the item in the OFM, and finally we update the tree, from the leaves that were changed up to the root. Note that updates must be performed in *post-order*, so that each node can compute the new maximum of its children. The process is illustrated in Figure 2. DELETE behaves analogously: we search in the tree, delete from OFM, and update the tree.

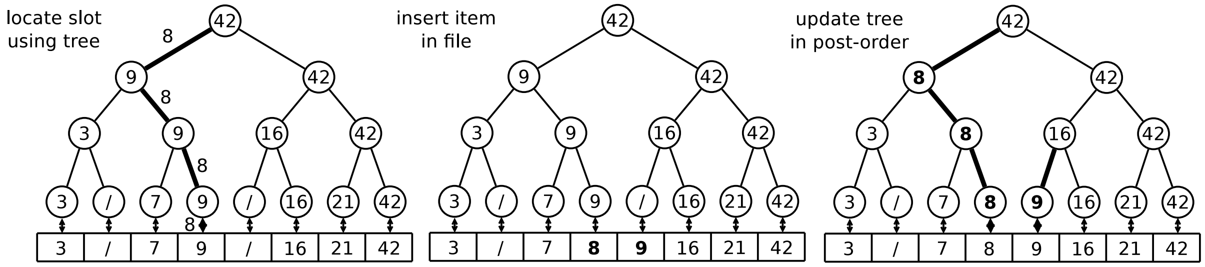

Figure 2: Inserting the key '8' in the data structure.

Now, we want to analyze the update operations.

Claim 2. *INSERT and DELETE take* $O\left(\log_{B+1} N + \frac{\log^2 N}{B}\right)$ *block reads.*

Proof. We notice immediately that finding the item to update takes $O(\log_{B+1} N)$ block reads, and updating the OFM $O\left(\frac{\log^2 N}{B}\right)$.

To see how long it takes to update the tree, we need to consider three separate tree layers independently. For the first layer, we define small vEB triangles to be subtrees in the tree which 1) have size smaller (or equal) to B and 2) are in one of the last two levels of the tree. Also, we are going to consider large vEB triangle, each of them having 1) size larger than B and 2) contain two levels of small vEB triangles. For the second layer, we consider the subtree whose leaves are the roots of the larger vEB blocks. For the third layer, we consider the path from the root of the subtree to the root of the whole tree. This division is illustrated in Figure 3.

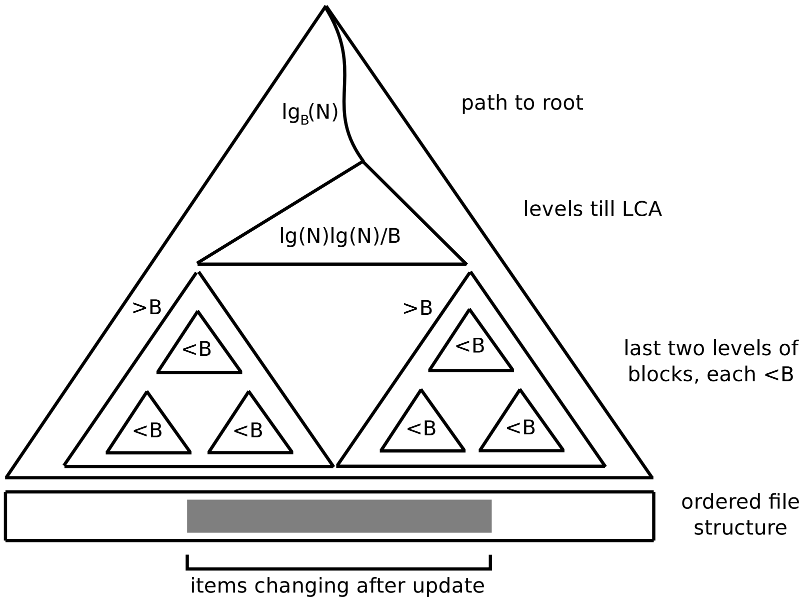

Figure 3: The structure division to analyze updates.

We start by noticing that a chunk of size less (or equal) to B can span at most two blocks in memory (when it is between the two). So, reading a small vEB triangle requires at most reading two blocks. Because we are updating the structure in post-order, when we are in the last two levels we only need to keep 4 blocks in cache, two for the child small vEB triangle and two for the parent small vEB triangle being evaluated. As long as we have at least 4 blocks of cache available, we can then update all the lowest two levels with an amortized constant amount of block reads per vEB triangles to update. Since there are $O\left(1+\frac{\log^2 N}{B}\right)$ such triangles in need for an update in the lowest two levels, updating them takes $O\left(\frac{\log^2 N}{B}\right)$ (amortized) block reads.

Proceeding upwards, we notice that the $O\left(\frac{\log^2 N}{B}\right)$ roots of large vEB triangles are the leaves of a $O\left(\frac{\log^2 N}{B}\right)$-sized tree τ. Notice that the root of τ is the LCA of all items that were modified at the leaves. The size of τ is equal (asymptotically) to the amount of reads we already did for the lowest two layers. Thus, we can afford reading an entire block to update each item of τ without adding to the runtime. So, any simple algorithm can be used to update τ.

Finally, to update the path from the root of τ to the root of the whole structure, we do $O\left(\log B + 1N\right)$ more block reads. Therefore, we can update the entire tree in $O\left(\log_{B+1} N + \frac{\log^2 N}{B}\right)$. □

At this point, we would still like to improve the bound even further. What we would like to achieve is $O\left(\log_{B+1} N\right)$ (what we can obtain with B-trees), and what we have is too costly if $B = o(\log N \log\log N)$. Fortunately, we can rid ourselves of the $\frac{\log^2 N}{B}$ component using the technique of *indirection*.

4.2.3 Wrapping Up: Adding Indirection

Indirection involves grouping the elements into $\Theta\left(\frac{N}{\log N}\right)$ groups of size $\Theta(\log N)$ elements each. Now we will create the structure we just described over the *minimum* of each $O(\log N)$ group. As a result, the vEB-style BST over the OFM array will act over $\Theta\left(\frac{N}{\log N}\right)$ 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 $O\left(\frac{\log N}{B}\right)$ 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 $O\left(\frac{\log N}{B}\right) = O(\log_B N)$ 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 $O\left(\frac{\log_{B+1} N + \frac{\log^2 N}{B}}{\log N}\right) = O\left(\frac{\log N}{B}\right) = O\left(\log_{B+1} N\right.$. Thus all `SEARCH`, `INSERT`, and `DELETE` operations 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.

[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.

6.851 Advanced Data Structures
Spring 2012