6.851: Advanced Data Structures Spring 2012

Lecture 8 — 13 March, 2012

Prof. Erik Demaine *Scribes: Robert Rusch (2017), Mahi Shafiullah (2017), Thomas Georgiou (2012), Pedram Razavi (2012), Tom Morgan (2010)*

1 From Last Lectures...

In the previous lecture, we discussed the External Memory and Cache Oblivious memory models. We additionally discussed some substantial results from data structures under each model; the most complex of these was the Cache Oblivious B-Tree, which can give us $O(\log_B N)$ runtime for insert, delete, and search. In that construction, we used the Ordered File Maintenance (OFM) data structure as a black box to maintain an ordered array with $O(\log^2 N)$ updates. Now we will fill in the details of the OFM. We will also come back to the List Labeling problem which was introduced in Lecture 1 as a black box for the full persistence data structures, specifically for linearizing the version tree.

2 Ordered File Maintenance [1] [2]

The OFM problem is to store N elements in an array of size $O(N)$, in a specified order. Additionally, the gaps between elements must be $O(1)$ elements wide, so that scanning k elements costs $O(\lceil \frac{k}{B} \rceil)$ memory transfers. The data structure must support deletion and insertion (between two existing elements). These updates are accomplished by re-arranging a contiguous block of $O(\log^2 N)$ elements using $O(1)$ interleaved scans. Thus the cost in memory transfers is $O(\frac{\log^2 N}{B})$; note that these bounds are amortized.

The OFM structure obtains its performance by guaranteeing that no part of the array becomes too densely or too sparsely populated. When a density threshold is violated, rebalancing (uniformly redistribute elements) occurs. To motivate the discussion, imagine that the array (size $O(N)$) is split into pieces of size $\Theta(\log N)$ each.

Now **imagine** a complete binary tree (depth: $O(\log N) - \Theta(\log \log N) = O(\log N)$) over these subsets. Each tree-node tracks the number of elements and the number of total array slots in its range. Density of an interval is then defined as the ratio of the number of elements stored in that interval to the number of total array slots in that interval.

2.1 Updates

To update element X,

- Update a leaf chunk of size $\Theta(\log N)$ containing X.

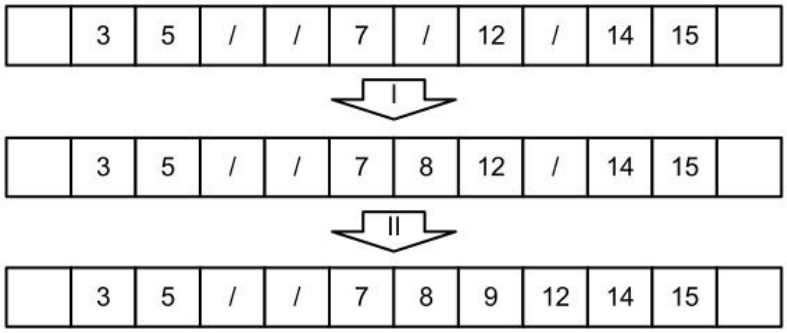

Figure 1: Example of two inserts in OFM. "/ " represents an empty gap, each of these gaps are $O(1)$ elements wide. I. insert element 8 after 7 II. insert element 9 after 8. 12 needs to get shifted first in this case.

- Walk up the tree to the first node **within the density threshold**.
- Uniformly redistribute the elements in this node's interval.

The density of a node, represented by ρ, is a measure to see how much of an interval is occupied, therefore we can define it as the **number of elements/array slots in the interval**. A density of 1 means that all the slots are full (too dense) and a density of 0 means that all slots are empty (too sparse). We want to avoid both of these extreme cases and maintain a density threshold. The density threshold is depth-dependent. The depth, represented by d, is defined such that the tree root has depth 0, and tree leaves have depth $h = \Theta(\log N)$. We require:

$$\rho \geq \frac{1}{2} - \frac{1}{4}\frac{d}{h} \in [\frac{1}{4}, \frac{1}{2}] : \text{ not too sparse}$$
$$\rho \leq \frac{3}{4} + \frac{1}{4}\frac{d}{h} \in [\frac{3}{4}, 1] : \text{ not too dense}$$

Notice that the density constraints are highest at the shallowest node. Intuitively, saving work (i.e., having tight constraints) at the deepest nodes gains relatively little performance because the working sets are comparatively small.

Keep in mind that the BST is never physically constructed. It is only a conceptual tool useful for understanding and analyzing the OFM structure. To perform the tree search operations, we can instead examine the binary representation of the left/right edges of a "node" to determine whether this node is a left/right child. Then the scan can proceed left/right accordingly; this is the "interleaved scan."

2.2 Analysis

Consider a node D at depth d and its left child and right child at depth $d+1$; say node D is within its threshold, so we need to rebalance its interval. Observe that post-balancing, all descendants of node D will have the same density as node D. Since density constraints become stricter at shallower depths, this means that D's descendants will be (relatively) well-within their thresholds.

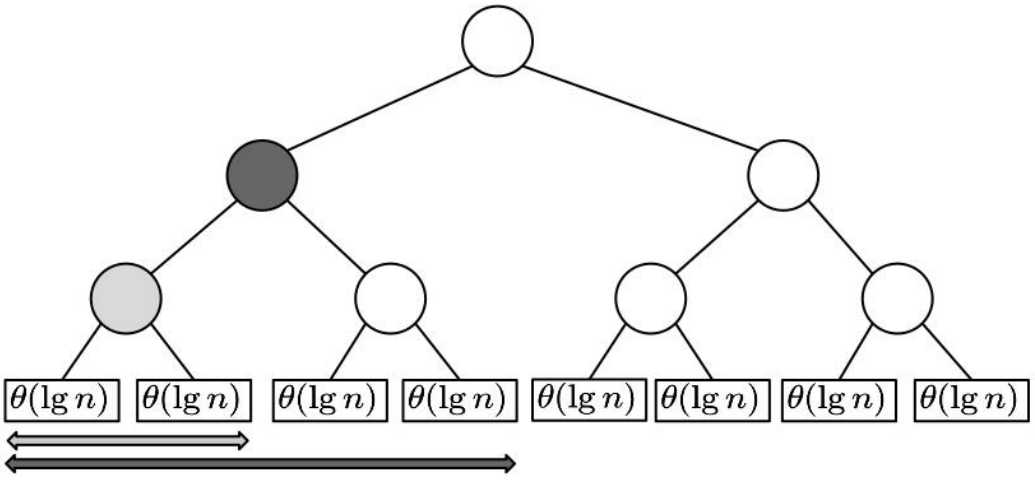

Figure 2: Conceptual binary tree showing how to build the intervals. If we insert a new element in a leaf and there is not enough room for it and interval is too dense (showed in light gray), we are going to walk up the tree and look at a bigger interval (showed in dark gray) until we find the right interval. In the end we at most get to the root which means redistribution of the the entire array

Specifically, the two children of D require around $\frac{m}{4h}$ or $\Omega(\frac{m}{\log N})$ (where m is the size of the child's interval) for their densities to violate their thresholds. So we can charge the cost of rebalancing the parent interval, which has size $\leq 2m$, to those $\Omega(\frac{m}{\log N})$ updates. Each update is charged $O(\log N)$ times each because there are only $\frac{m}{4 \log N}$ updates (until the child violates a threshold) but m charges in the parent interval. Now we deal with the fact that we have only considered depth d and $d+1$. Each leaf is below $O(\log N)$ ancestors, so each update may be charged at each tree level. This makes for a total of $O(\log^2 N)$ (amortized) memory moves.

This analysis is an amortized result. It can be made worst-case[3], but we did not discuss it in class. It also conjectured that $\Omega \log^2 N$ is an appropriate lower bound for this problem.

3 List Labeling

List Labeling is an easier problem than OFM. It involves maintaining explicit (e.g., integer) labels in each node of a linked list. These labels should increase monotonically over the list. The data structure needs to support insertion between two labels and deletion of a label. Depending on the size of the label space, we are aware of a spread of time-bounds:

Label Space	Best Update Query Time	Comments
$(1+\epsilon)N...N \lg N$	$O(\lg^2 N)$	Use OFM (linear space).
$N^{1+\epsilon}...N^{O(1)}$	$\Theta(\lg N)$	OFM method with thresholds at $\approx \frac{1}{\alpha^d}$, $1 < \alpha \leq 2$.
2^N	$\Theta(1)$	Simple: have N bits for N items, so insert by bisecting intervals (may rebuild once per N ops).

3.1 List Ordered Maintenance

Here we need to maintain an ordered linked list subject to insertion and deletion. Additionally, it should answer "ordering queries": Does node x come before node y? $O(1)$ updates and queries are possible using indirection [4].

Consider a "top" summary structure covering a set of "bottom" structures. The bottom structures are each size $\Theta(\log N)$. To perform the labeling, we can use the simple exponential label space list labeling algorithm. The required label space is $2^n = 2^{\log N} = N$, which is affordable. The summary structure is size $O(\frac{N}{\log N})$; each element points to one of the bottom members. For this list, use the $\Theta(\log N)$ list labeling algorithm. The cost is $O(1)$ amortized since it costs $\Theta(\log N)$ per update, but updates only happen once every $\Theta(\log N)$ updates to the bottom members.

Then we can form an implicit label for every bottom member element in the form of (top label, bottom label). This requires $O(\log N)$ bits to store, so we can compare two labels in $O(1)$ time. The key point here is that changing one top label updates numerous bottom labels simultaneously. This is the reason we are able to obtain better performance than the optimal result quoted in the above table. Lastly, worst-case bounds are also obtainable [4].

4 Cache-Oblivious Priority Queues [5]

Our objective with the cache-oblivious priority queue is to support all operations with $O(\frac{1}{B} \log_{M/B} \frac{N}{B})$ amortized memory transfers per operation in the cache-oblivious model. The data structure is divided up into $\Theta(\log \log n)$ levels of sizes $N, N^{2/3}, N^{4/9}, ..., O(1)$. Each level consists of a single up buffer, and many down buffers which store elements moving up and down in the data structure as linked lists. For a level of size $X^{3/2}$, the single up buffer is of size $X^{3/2}$ and there are at most $X^{1/2}$ down buffers. Each down buffers has $\Theta(X)$ elements, except for the first one which may be smaller.

We maintain the following invariants on the order of the elements:

1. In a given level, all of the elements in the down buffers must be smaller than all of the elements in the up buffer.
2. In a given level, the down buffers must be in increasing order - all elements in the ith down buffer must be smaller than all of the elements in the $(i+1)$th down buffer.
3. The down buffers are ordered globally - all elements in a down buffer in a smaller level must be smaller than all elements in any down buffer in a larger level.

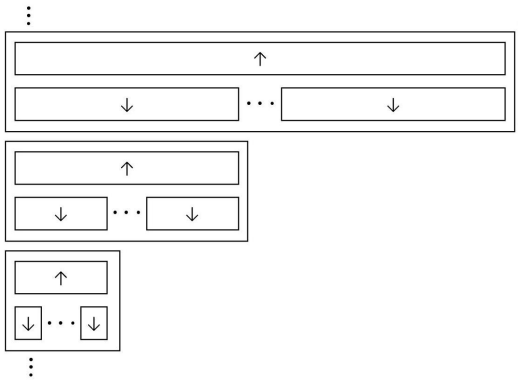

Figure 3: A cache-oblivious priority queue

4.1 Find-min

The minimum element in the priority queue should always be in the first down buffer at the smallest level, so we simply find it there.

4.2 Delete-min

The minimum element in the priority queue should always be in the first down buffer at the smallest level, so we simply go there and remove it. However, this may cause the down buffers to under flow, in which case we perform a pull operation.

4.3 Insertion

Insertion a new element is done in three steps:

1. Append the element to the up buffer at the smallest level.
2. If the new element is smaller than an element in a down buffer, swap it into the down buffers as necessary.
3. If the up buffer overflows, perform a push operation on it.

4.4 Push

The objective of pushing is to empty out the up buffer at level X by moving all of its elements into level $X^{3/2}$. To do this, we first sort all of these elements, and then distribute them among the down and up buffers in level $X^{3/2}$. This distribution is performed by scanning through the buffers in increasing order (first down buffers, then up buffer) and inserting our elements where they belong. If one of the down buffers overflows we simply split it in half, and if this causes the number of down buffers to grow too large, we move the last one into the up buffer. If up buffer overflows, we simply perform a push on it.

4.5 Pull

Our goal here is to refill the down buffers at level X by bringing down the smallest X elements from level $X^{3/2}$. Once we get these elements, we will sort them together with the up buffer. We will leave the largest elements in the up buffer (the same number of elements as were there before the pull) and then split the remaining elements between the down buffers.

We recall that each of the down buffers at level $X^{3/2}$ except for perhaps the first one should have $\Theta(X)$ elements, so we simply sort the first two down buffers in level $X^{3/2}$ and bring the smallest X elements from them to be put into level X's down buffers. However, if there are fewer than X elements here, then we recursively pull $X^{3/2}$ elements from level $X^{9/4}$.

4.6 Analysis

We will first argue that ignoring recursion, the number of memory transfers during a push or pull operation on level $X^{3/2}$ is $O(\frac{X}{B}\log_{M/B}\frac{X}{B})$. First we assume that any level of size less than M will be entirely in cache, so those are free. In the remaining cases, we use the tall cache assumption to say $X^{3/2} > M \geq B^2$, thus $X > B^{4/3}$.

First consider a push at level $X^{3/2}$. Sorting costs $O(\frac{X}{B}\log_{M/B}\frac{X}{B})$, and distribution costs $O(\frac{X}{B} + X^{1/2})$. $\frac{X}{B}$ is the cost for scanning, and $X^{1/2}$ is the cost for loading the first part of each down buffer. If $X \geq B^2$ then this distribution costs $O(X/B)$ which is dwarfed by sorting. There is only one level remaining in which $B^{4/3} \leq X \leq B^2$ and for this level we will simply assume that our ideal cache always holds on to one line buffer down buffer, so we don't have to pay the $X^{1/2}$ start up cost. We can do this because $X \leq B^2$ so by the tall cache assumption $X^{1/2} \leq B \leq M/B$. Therefore sorting is always the dominant cost. The analysis for pulling is essentially identical.

Thus, X elements involved in pushing or pulling cost $O(\frac{X}{B}\log_{M/B}\frac{X}{B})$. One can prove that each element can only be charged for a constant number of pushes and pulls per level. The intuition is basically that each element goes up and then down. Thus the cost per element is $O(\frac{1}{B}\sum_X \log_{M/B}\frac{X}{B})$. Since X grows doubly exponentially, the $\log_{M/B}\frac{X}{B}$ term grows exponentially. Therefore, the last term, $\log_{M/B}\frac{N}{B}$ dominates the sum and we are left with $O(\frac{1}{B}\log_{M/B}\frac{N}{B})$ per operation as desired.

References

[1] Alon Itai, Alan G. Konheim, and Michael Rodeh. A Sparse Table Implementation of Priority Queues. *International Colloquium on Automata, Languages, and Programming (ICALP)*, p417-432, 1981.

[2] M. A. Bender, R. Cole, E. Demaine, M. Farach-Colton, and J. Zito. Two Simplified Algorithms for Maintaining Order in a List. *Proceedings of the 10th European Symposium on Algorithms (ESA)*, p152-164, 2002.

[3] D.E. Willard. A density control algorithm for doing insertions and deletions in a sequentially ordered file in good worst-case time. In *Information and Computation*, 97(2), p150-204, April 1992.

[4] P. Dietz, and D. Sleator. Two algorithms for maintaining order in a list. In *Annual ACM Symposium on Theory of Computing (STOC)*, p365-372, 1987.

[5] 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.