6.897: Advanced Data Structures Spring 2005

Lecture 24 — May 5, 2005

Prof. Erik Demaine *Scribe: Jeff Cohen*

1 Overview

In the last lecture, we looked at the cache-oblivious model, in which the machine has two levels of memory, one much larger and slower than the other, but where the memory system is controlled automatically, so that our algorithm doesn't see the size of the blocks or the cache. We also looked at cache-oblivious B-trees.

In this lecture, we look inside a black box from the last lecture: ordered file maintenance. We then examine ordered queries in lists, a topic related to ordered file maintenance. Finally, we learn how to implement cache-oblivious priority queues.

2 Ordered File Maintenance

The problem is posed as follows: maintain N elements in an array of size $O(N)$ in a specified order, with at most $O(1)$ gaps between two consecutive elements, subject to inserting and deleting in a specified place in the order. We show how to implement updates by moving a portion of the array of size $O(\lg^2 N)$, amortized. Since gaps are bounded, scanning k consecutive elements takes $O(1+\frac{k}{B})$ memory transfers, even cache-obliviously; inserting and deleting take $O(\frac{\lg^2 N}{B})$. The solution we present is due to Itai, Konheim, and Rodeh [1]. Worst-case bounds have also been achieved, originally by [2], and then simplified in [3].

Solution Overview. We divide the array into buckets of size $\Theta(\lg N)$. The rough idea is that each time we update, we first consider the updated bucket; if it has too many elements or too few, we look at its neighbors, and so on. We grow the explored interval until it has the right density of elements and evenly redistribute the elements in that interval.

To specify how this exploration proceeds and what "right density" means, we consider an implicit complete binary with the buckets in the leaves. Each non-leaf node represents an interval that is the union of its children's intervals, so that the root represents the whole array.

To insert an element x after y, we look in the leaf containing y and add x there by rearranging the $O(\lg N)$ elements there. If the leaf is too full (i.e. outside some threshold that we will define), we walk up to the nearest ancestor that is within threshold and redistribute the elements on that interval with uniform density. To delete an element, we do something similar. We remove it from its leaf and walk up the tree to the first interval that is within threshold, and we rearrange that interval to set all buckets within threshold.

Theshold definition. Define the *density* of an interval as the number of elements it contains divided by the number of slots in the interval where data could go. Let h be the height of the tree (which is roughly $\lg N$). At depth d, the density thresholds are:

$$\text{density} \geq \frac{1}{2} - \frac{1}{4}\frac{d}{h}$$

and

$$\text{density} \leq \frac{3}{4} + \frac{1}{4}\frac{d}{h}.$$

Note that the lower limit is always in $[\frac{1}{4}, \frac{1}{2}]$ and the upper limit is always in $[\frac{3}{4}, 1]$. The restrictions get stronger as you go up the tree (because d gets lower, so the lower limit gets close to $\frac{1}{2}$ and the upper limit gets close to $\frac{3}{4}$).

Analysis. After we re-balance an interval, it is within its threshold. Its children are within its threshold, as well. If the parent has depth d, the children have depth $d+1$, but are within the (stronger) thresholds for a node of depth d. Consequently, children are far within their thresholds, by a density close to $\frac{1/4}{\lg n} = \Theta(\frac{1}{\lg N})$. Before this node, which has size $s = 2^d \lg N$, is rebalanced again, one of its children must go outside its threshold, which requires $\Omega(\frac{s}{\lg N})$ insertions or deletions within its subtree. Then, if we charge $O(\lg N)$ per each of these insertions, we can pay for rebalancing this node again. Note, however, that each update is charged $O(\lg N)$ for future rebalancing by every one of its ancestors. Thus, the amortized cost of an update is $O(\lg^2 N)$.

3 Related Problems

List Labeling. In this problem, out goal is to maintain a linked list with an explicit label for each node such that the labels are monotonic throughout the list, subject to inserting and deleting at any given location. Note that maintaining N ordered items in an array of size S can also solve list labeling on N elements, with a tag space size of S (each move is a relabel operation).

There is a tradeoff between tag space and the number of relabelings per update:

- for tag space between $(1+\epsilon)n$ and $O(n \lg n)$, one can use the algorithm from above to achieve update time $O(\lg^2 n)$.
- for tag space between $n^{1+\epsilon}$ and $n^{O(1)}$, we need only spend $O(\lg n)$ time. This is based on the algorithm from above, with modified thresholds.
- for tag space $2^{O(n)}$, one can implement updates in $O(1)$ time. This is simple: insert each element at the middle point between its neighbors, and rebalance after n insertions.

There is also a lower bound of $\Omega(\lg n)$ for any polynomial tag space, due to [4]. This is only tight for the range of tag spaces corresponding to the middle bullet.

Order Queries. The Order Query problem consists of maintaining a linked list that supports insertion and deletion at a desired location and an order query, which, when given (x, y), determines if x precedes y in the list. We will achieve a constant-time updates and queries, by using list labeling with a polynomial tag space and indirection. This solution is due to [5].

On the top level, we will use a maximum label of size n^2, and have $\frac{n}{\lg n}$ elements. We will use the $O(\lg n)$ solution at this level. At the second level, we will have a number of data structures of size $O(\lg n)$, with a maximum label of n, and use the trivial $O(1)$-time solution with exponential tag space. The (amortized) time to update is $O(1)$. To find a label, we concatenate the label from the top and bottom parts. The tag space is $O(n^3)$, so labels have $O(\lg n)$ bits and can be compared in constant time.

Notice that this seems to contradict the lower bound for list labeling. The reason it does not is that we aren't maintaining explicit labels here, so our problem is easier. Relabelings in the top level actually change the labels for $O(\lg n)$ elements at once.

4 Cache-oblivious priority queues

We will describe cache-oblivious priority queues which achieve $O(\frac{1}{B} \lg_{M/B} \frac{N}{B})$ amortized memory transfers per operation [6]. Thus, N calls to the priority queue are as efficient as sorting.

The main idea is to use $\lg \lg N$ levels of sizes $N, N^{2/3}, N^{4/9}, \ldots$, reminescent of exponential trees. Each level has an up buffer and several down buffers, to store elements moving up and down, respectively. At level $X^{3/2}$, the up buffer will have size $X^{3/2}$, and there will be at most $X^{1/2}$ down buffers, each of size $\Theta(X)$, except the first, which may be smaller.

We maintain a number of invariants on our data structure:

- the keys in the down buffers at each level are less than those in the up buffer.
- keys in the down buffers at level $X^{3/2}$ are less than those in the down buffers at level $X^{9/4}$.
- the down buffers at each level are sorted: the keys in the first are less than those in the second, and so on.

Pushing. We first define a PUSH procedure, which will be used to implement insertions. It pushes X elements from below into level $X^{3/2}$. These elements all have keys above all elements in the down buffers at level X. The procedure works as follows. First, sort the elements. Then, the elements are distributed among the down buffers (scanning elements and visiting buffers in parallel, both in sorted order). When a down buffer overflows, split it in half and link the halves together. When the number of down buffers gets too large, move the last down buffer up to the up buffer. When the up buffer overflows, push it up to the next level, $X^{9/4}$.

Now, to insert an element x, append it to the smallest up buffer. Then, swap it with the largest element in the smallest down buffer if needed. When an up buffer overflows, call PUSH.

Pulling. The PULL procedure will be used to implement DELETE-MIN. It pulls the X smallest elements into level X from level $X^{3/2}$ (and above if necessary). To execute it, first we check if

there are enough elements in the down buffers. If so, we sort the first two down buffers and extract the elements we need. Otherwise, we recursively pull $X^{3/2}$ elements from level $X^{9/4}$, sort them together with the up buffer, put the largest q of them into the up buffer, where q is the number that were in the up buffer before we sorted, take the elements we need to get the X smallest, and put the rest in the down buffers.

To delete the minimum element, if the smallest down buffer underflows, pull elements down into it, and then return the smallest element of the smallest down buffer.

Analysis. We first show that PUSH and PULL at level $X^{3/2}$ take $O(\frac{X}{B}\log_{M/B}\frac{X}{B})$ memory transfers. Our ideal memory manager keeps all levels of size at most M in the cache, so operations on those levels are free. When this is not the case, $X^{3/2} > M$, so, by the tall cache assumption from last lecture, $X^{3/2} > B^2$, or $X > B^{4/3}$.

First consider PUSH. Sorting on such a level costs $O(\frac{X}{B}\log_{M/B}\frac{X}{B})$. The distribution step costs $O(\frac{X}{B} + \sqrt{X})$, where the first term is for scanning and the second for a startup cost of 1 for each down buffer. If $X > B^2$, then the distribution cost is $O(\frac{X}{B})$, so it's hidden by sorting. There is only one problematic level with $B^{4/3} < X \leq B^2$. But for this level, we can keep one page per down buffer in cache, making the startup cost disappear. This is possible because there are $\sqrt{X} \leq B$ down buffers, and the tall cache assumption guaranteees we have $\frac{M}{B} \geq B$ pages in the cache. Thus, in all cases, the push time is dominated by sorting.

In the case of PULL at a level $X^{3/2} > B^2$, sorting again costs $O(\frac{X}{B}\log_{M/B}\frac{X}{B})$ memory transfers. There is another potential sort on $O(X^{3/2})$ elements when we pull recursively, but we can charge the cost of our sort to that recursive sort in that case. Thus the time is once again dominated by sorting.

Thus, pushing and pulling at level $X^{3/2}$ costs $O(\frac{X}{B}\log_{M/B}\frac{X}{B})$ memory transfers. Such a push or pull moves X elements up or down, so the cost per element is $O(\frac{1}{B}\log_{M/B}\frac{X}{B})$. One can show that an element is only charged once per level per direction (essentially, an element goes up, then down; see [6] for details). Thus, the total cost per element is $O(\frac{1}{B}\sum_X \log_{M/B}\frac{X}{B})$. Because X increases doubly exponentially, the logarithm increases exponentially, so the sum is dominated by the last term. Then, the total amortized cost per element is $O(\frac{1}{B}\log_{M/B}\frac{N}{B})$.

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), p. 417-431, 1981.

[2] D. 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), p. 150-204, Apr. 1992.

[3] 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), p. 152-164, 2002.

[4] P. Dietz, J. Seiferas, J. Zhang, *A Tight Lower Bound for Online Monotonic List Labeling*, SIAM Journal on Discrete Mathematics, 18(3) , p. 626 - 637, 2005.

[5] P. Dietz, and D. Sleator, *Two algorithms for maintaining order in a list*, Annual ACM Symposium on Theory of Computing (STOC), p. 365-372, 1987.

[6] L. Arge, M. A. Bender, E. D. Demaine, B. Holland-Minkley, and J. I. Munro, *Cache-oblivious priority queue and graph algorithm applications*, Proceedings of the 34th Annual ACM Symposium on Theory of Computing, p. 268-276. ACM Press, 2002