6.851: Advanced Data Structures Spring 2007

Lecture 20 — Apr 30, 2007

Prof. Erik Demaine *Scribe: Robert Bryant*

1 Overview

Last lecture we discussed the cache-oblivious model for algorithm and data structure design, and surveyed current results within this model. During this survey, we developed the idea of a cache-oblivious dynamic B-tree. In order to construct this data structure, we utilized a black box component data structure that could maintain an array of N elements in order in $O(N)$ space and achieve insert/delete operations in $O(\lg^2 N)$ ($O(\frac{\lg^2 N}{B})$ memory transfers in the cache-oblivious model). This problem is called Ordered File Maintenance in the literature, and is the topic of the first half of this lecture. In the second half of this lecture, we complete our survey of cache-oblivious algorithms by investigating cache-oblivious priority queues.

2 Ordered File Maintenance [1] [2]

Formally the Ordered File Maintenance problem is to maintain an array of N elements in $O(N)$ space by having only $O(1)$ size empty array-slot "gaps" between elements. This leads to scanning any k items in $O(\lceil k/B \rceil)$ memory transfers and performing updates (insert/delete) in $\frac{\lg^2 N}{B}$ memory transfers by rearranging only $O(\lg^2 N)$ amortized consecutive elements. Because both scanning and updates involve only consecutive elements, and therefore the $\frac{1}{B}$ factor in the number of memory transfers is obvious, we will continue in our investigation considering only the traditional data structure without any caching.

The rough idea of the OFM data structure is as follows: on any update of an element e, choose progressively larger intervals of the array that contains e until the element density of the interval is "within threshold", then evenly redistribute all elements in the interval.

We choose intervals from our array using a symbolic complete binary search tree. We divide our array into segments of $\Theta(\lg N)$ elements each, and use these as the leaves of our symbolic BST. When an update occurs, we walk up our tree until the segment of the array covered by the current subtree is within our density threshold.

We define our desity threshold on a node at depth d in a tree of height h as follows:

density $\geq \frac{1}{2} - \frac{1}{4}\frac{d}{h}, \in [\frac{1}{4}, \frac{1}{2}]$

density $\leq \frac{3}{4} + \frac{1}{4}\frac{d}{h}, \in [\frac{3}{4}, 1]$

2.1 Analysis

The cost of rebalancing a node v is capacity(v), where the capacity(v) is the maximum number of elements that could be in v's subtree. The important part of the OFM anaylsis is in how often we have to rebalance v. Towards this end, suppose we rebalance a node v at depth d. Then, because we evenly redistribute the entire segment under v, density(all descendants of v) = density(v). However, our density threshold per node is dependent on the depth of that node. Therefore the children of v are far within the threshold, by at least $\frac{1}{4h} = \Theta(\lg N)$. This means that in the worst case, one of v's children will have to have a density change of $O(\lg N)$ elements for v to be rearranged again. Thus we have:

$\Delta\text{density} = O(\lg N)$

$\frac{\Delta numelements}{\text{capacity of v}} = 1/O(\lg N)$

$\Delta numelements = \text{capacity of v}/O(\lg N)$

Therefore our amortized cost of rebalancing a node v is capacity(v)/(capacity of v$/O(\lg N)$) = $O(\lg N)$. This cost can be made worst case as in [3], but it is very ugly, so we did not cover it in class. Because every node v is in $O(\lg N)$ subtrees, the overall amortized cost of the OFM problem is $O(\lg^2 N)$.

2.2 Related Problems

Ordered File Maintenance was originally developed because of its similarity to two different non-cache oblivious algorithms: List Labelling and Order Queries. In List Labelling, we attempt to maintain an explicit tag for each of N nodes in a linked list such that tags increase monotonically over the list, subject to insert/delete here queries. Depending on the size of the tag space, this problem is achievable in varying bounds:

|Tag Space|	Best Update Query Time
$(1+\epsilon)N ... N\lg N$*	$O(\lg^2 N)$
$N^{1+\epsilon} ... N^{O(1)}$	$\Theta(\lg N)$
2^N	$O(1)$

*This is the OFM Problem.

The List Labelling Problem is useful in another problem called Order Queries. In the Order Queries problem, we attempt to maintain an ordered list of N elements subject to the query order(x, y) that returns if a node x is before a node y. We can achieve $O(1)$ queries and updates for this problem by using data structure from [4].

First, we utilize indirection, creating a top data structure of $\frac{N}{\lg N}$ elements, each pointing to a child data structure of $\lg N$ elements. We store the top elements in our second List Labelling data structure, using a tag space of size N^2, yielding queries/updates in $O(\lg N)$, amortized $O(1)$. We store each set of $\lg N$ child elements in our third List Labelling data structure, using a tag space of size 2^N, which yields a maximum label of size n and has trivial $O(1)$ updates/queries. We define a bottom element's label as the concatenation of its top and bottom labels. Because this overall tag space is $O(N^3)$, we can represent it in $O(\lg N)$ bits, allowing us to compare tags in constant time.

3 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 [5].

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 each level are less than those in the down buffers of the next-largest level.
- the down buffers at each level are sorted: the keys in the first are less than those in the second, and so on (but the keys are not sorted within buffers).

Insertion. The basic insertion algorithm is as follows:

1. Insert new node into smallest up buffer
2. Fix the ordering of smallest level
3. If up buffer overflows, **push** the up buffer to the next level

Pushing. We now define the **push** procedure, which will be used to implement insertions. It pushes elements from one up buffer to that level above that up buffer. 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 recursively.

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 is very similar to push, and has the same bounds.

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.

As mentioned previously, the bounds for **pull** are similar.

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). 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)*, 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.