6.897: Advanced Data Structures Spring 2003

Lecture 14 — Monday April 14, 2003

Prof. Erik Demaine *Scribe: Kunal Agrawal, Vladimir Kiriansky*

1 Overview

Today we will first see a file-maintenance algorithm that allows random inserts and deletes in $O(\lg^2 n)$ time in a sequential file taking linear space.

Then we will move onto algorithms that care about memory hierarchy. In modern computers, memory is not flat. It has multiple levels of progressively slower and larger memory. Thus it is not equally costly to access all items in memory. For example, if you store a matrix in row-order, then scanning in row-order is much faster than scanning in column order.

Our goal to get models that capture this behavior and analyze why some algorithms do so well. We will see two memory hierarchy models that will be used later to analyze various algorithms and data structures.

The first algorithm we see is an ordered-file-maintainence algorithm which can be implemented in a file system for random insertions and deletions in a file. This basic algorithm will be later used as a black box for more complicated algorithms.

2 Ordered-File Maintenance [IKR81, BDFC00]

2.1 Problem Statement

Given n elements in some order, store them in that order in an array of size $O(n)$. The following operations must be supported:

- Insert an element between 2 given elements
- Delete any given element.

2.2 Naïve Solution

Store all elements in an array with no gaps, as shown on Figure 1. For insert and delete, we have to move all elements after the position where the insert or delete took place.

Cost: $O(n)$ in the worst case for both inserts and deletes.

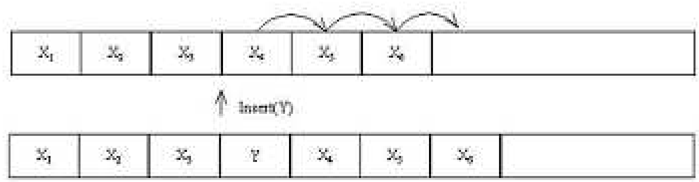

Figure 1: Insertion into an array

2.3 Better Solution

In the following sections we will discuss a better solution for this problem with cost $O(\lg^2 n)$ amortized for both inserts and deletes. It is possible to get $O(\lg^2 n)$ worst case [Wil92, BCD$^+$02], but the analysis is more complex and is not covered in this lecture. It is conjectured that these bounds are the best possible with linear space; see [DSZ94] for a matching lower bound but in a weak model.

2.4 Rough Idea

The idea is to keep gaps in the array so that the elements can usually be inserted without shifting the entire array. These gaps should be roughly evenly distributed and should remain evenly distributed in spite of insertions and deletions. So the rough idea of the algorithm is to ensure that, on insertion, the area around the element that was inserted doesn't get too crowded. If it does, then we find an interval around it that has enough blank spaces in total, and then redistribute the blank spaces evenly in the interval. The interval is grown exponentially, so that there have to be many inserts before it gets full again and thus we can charge to these inserts.

2.5 Tree View

For analysis purposes, we look at the conceptual tree view of the array. Divide the array into chunks of $\Theta(\lg n)$ slots. These chunks are the leaves of the binary tree.

The height of the tree is $h = \lg n - \lg\lg n$. Conceptually, each node of the tree represents an interval which is the combination of its children's intervals. The size of the interval of the leaves is $\Omega(\lg n)$ and as we walk up the tree the size of the interval grows exponentially. Thus when any interval is too full, we try to find an ancestor which is not too full (on average) and redistribute the spaces. If we reach the root of the tree and the interval is still full, we double the size of the array and redistribute the gaps.

2.6 Density

The *density* of a node represents how full or how empty the interval represented by that node is:

Definition 1 (Density). $\textit{Density(node)} = \frac{\textit{number of elements present in interval}}{\textit{number of slots in interval}}$

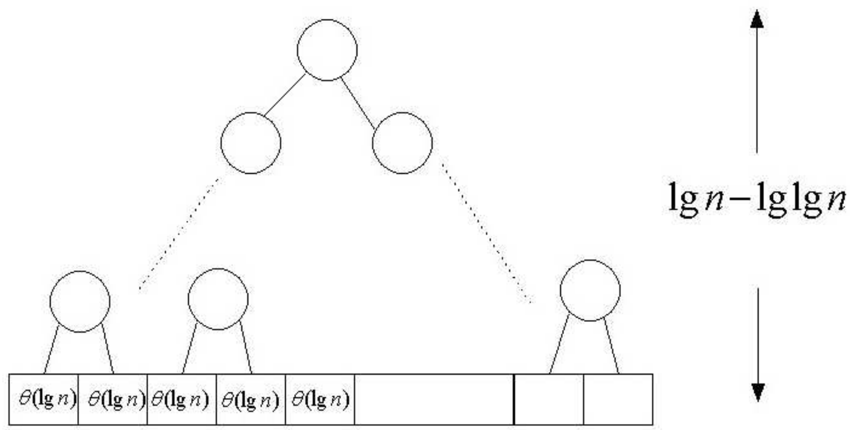

Figure 2: Conceptual Tree Representation

Density is always a constant between 0 (completely empty) and 1 (completely full).

2.6.1 Density Constraints

The density constraints define the lower and upper bounds on the density.

Node thresholds at depth d:

- lower density threshold and constraint

$$density \geq \frac{1}{2} - \frac{1}{4}\frac{d}{h} = ldt(d)$$

$$ldt \in \left[\frac{1}{4}, \frac{1}{2}\right]$$

- upper density threshold and constraint

$$density \leq \frac{3}{4} + \frac{1}{4}\frac{d}{h} = udt(d)$$

$$udt \in \left[\frac{3}{4}, 1\right]$$

- density threshold interval

$$thresholds(node) = \Big[ldt(depth(node)),\ udt(depth(node))\Big]$$

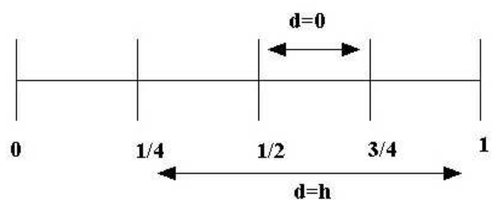

Figure 3: Density Constraints

Notice that the constraints are tighter as we go higher up the tree. In particular, at depth 0 (the root), $1/2 \leq \textit{density} \leq 3/4$ and, at depth h (the leaves), $1/4 \leq \textit{density} \leq 1$.

These constraints are not always satisfied for all nodes. They are violated occasionally. But if we ever see the violation, we fix it. If the violation is never seen, then it doesn't effect the algorithm.

The density constraints can be tuned to take less space. With these constraints, the array size is about twice the number of elements. This constant factor can be made arbitrarily small $(1+\epsilon)$ by changing the constraints.

2.7 Algorithm

To insert a new element x between y and z given pointers to y and z:

Insert(x, y, z):
    insert x into the same chunk as y (or z) using naïve insertion
    node $\leftarrow$ leaf node representing chunk
    if density(node) $\notin$ thresholds(node):
        while density(node) $\notin$ thresholds(node):
            node $\leftarrow$ parent(node)
        rebalance node by evenly redistributing all elements in the interval

Delete(x) is exactly the same except for the first step, which is to delete the element x from the chunk containing it using naïve deletion.

2.8 Description

It is guaranteed that there is at least one gap. So inserting into the leaf always succeeds in $\Theta(\lg n)$ time by the naïve shifting algorithm. Now, if the leaf is within threshold, we are done. If the leaf is outside threshold, we do not have enough gaps within this interval, so we need to find a larger interval with more gaps. When we reach a level at which there are enough gaps, i.e., at which the interval is within threshold, we redistribute the gaps evenly in that entire interval. Because the parents have tighter constraints than their children, all children will also now have enough gaps (be within threshold). The cost of redistribution can be amortized by charging to the inserts and deletes that brought a child node outside threshold. If we reach the root without finding any node within threshold, we rebuild the whole tree.

2.9 Analysis

The important observation is that thresholds get weaker as we go deeper in the tree. The consequence of this observation is simple but powerful.

Whenever we rebalance a node, it is because one if its children was outside of the thresholds, while the node itself is within thresholds. After rebalancing children, densities will also be within limits. Indeed children will have the same density as parent (disregarding rounding), but the childrens' constraints are weaker. Therefore children will be farther within density thresholds.

We can calculate by how much they will be within thresholds as follows. At adjacent levels, d differs by 1. Given that parent density is within its threshold, the density of the child is within its threshold by at least a factor of $1/(4h) = \Theta(1/\lg n)$.

Of course, this observation doesn't mean much for $n = 1$ element because of rounding. We should rather start with a large enough base case to get the absolute number of elements (the ratio times the size) at least 1. Therefore we need to start with an interval of size $\Theta(\lg n)$. This is the only reason we need the chunks at the leaves to be of this size.

Before the parent needs to be rebalanced again, one of its children's density must be outside of its threshold. That will happen only after $\Omega(capacity/\lg n)$ inserts or deletes in the subtree. Capacity here is the capacity of the child, or that of the parent for that matter, because they are equal up to a constant factor ($capacity(parent) = \Theta(capacity(child))$).

2.9.1 Amortized Bound

At first, the amortized bound seems to be only $O(\lg n)$, but we shouldn't forget that we need to mark all ancestor subtrees whenever we insert a node.

The amortized cost of inserting or deleting an element into one subtree is $O(\lg n)$. The depth of the tree is $h = \Theta(\lg n)$. Whenever we insert an element, each of the $O(h)$ ancestor subtrees charges us. The total amortized number of moves is thus $O(\lg^2 n)$.

3 Related Problems

We will now look at two related problems in which we can apply this data structure for ordered-file maintenance.

3.1 Order Queries in a List

The Order-Maintenance Problem is to maintain a list representing a total order, subject to the following operations:

Tag space	Time per update
$\Theta(n)$ or $\Theta(n \lg n)$	$O(\lg^2 n)$
$\Theta(n^2)$ or $\Theta(n^{1+\epsilon})$	$O(\lg n)$
$\Theta(2^n)$	$O(1)$

Table 1: Tag space vs update time tradeoffs

Updates: exactly the same update operations as in the ordered file maintenance problem.

- Insert (X, Y, Z): Insert a new element X between two given elements
- Delete (X): Remove an element X from the list

Query: Order (X, Y): Given pointers to two nodes, determine whether X precedes Y in the list.

We will show a solution with using $O(1)$ amortized time per operation. In particular, other than amortization, updates take constant time just like in a linked-list representation.

3.2 List-Labeling Problem

A harder problem is to maintain a linked list with regard to same updates as above, and in addition store a *tag* with each node. The following operations must be supported:

Updates:

- Insert (X, Y, Z): Insert a new element X between two given elements
- Delete (X): Remove an element X from the list.

Query: Order queries should be answered correctly by direct comparison of tags.

3.3 Tag Space vs. Update Time

We should maintain monotonically ordered *tags* for each node in the list, such that the order of the tags matches the order of the nodes. (If we had real numbers as tags they would have served great for this purpose.) Further, if our tags fit in a constant number of words, order queries can be answered in $O(1)$ time by comparing tags.

Table 1 summarizes several results on this problem.

We can use ordered-file maintenance to keep an ordered array of $O(n)$ elements where we use the indices of the elements in that array as their labels. Whenever we re-arrange items we also need to update their labels. Thus we get updates in $O(\lg^2 n)$ amortized time, and the tag space is linear, so queries are $O(1)$.

By modification to thresholds, we can similarly obtain an $O(\lg n)$ amortized solution given $\Theta(n^{1+\epsilon})$ tag space. (Note that, even with $O(n \lg n)$ tag space, the update time is still $O(\lg^2 n)$; we need the polynomial tag space to get $O(\lg n)$.)

Lastly, for the trivial solution with exponential tag space but constant update time, we just add an extra bit for each node to subdivide the tag space.

3.4 Goal

We want to solve the order-maintenance problem (not list labeling) with $O(1)$ amortized update and query time using only the $O(\lg^2 n)$ data structure for list labeling.

3.5 Indirection

Indirection can help us solve order queries using "implicit" tags. We start with the almost-working list-labeling black box and add indirection.

$\boxed{\text{list labeling on } \frac{n}{\lg^2 n} \text{ blocks}}$

$\boxed{\Theta(\lg^2 n)}\ \boxed{\Theta(\lg^2 n)}\ \cdots\ \boxed{\Theta(\lg^2 n)}$

Figure 4: Indirection using a list labeling black box

The top-level nodes give labels for comparison of bottom-level blocks. We use an ordered-file-maintenance structure as a black box for the top-level nodes. The indirection delays updates to the top structure by at least $\Theta(\lg^2 n)$. The bottom-level blocks must have size $\Theta(\lg^2 n)$. The question now is how to represent the blocks?

Question: Can you use the exponential representation on the blocks?

Answer: No, it will result in super linear space requirement, since $2^{\lg^2 n} > n^{O(1)}$. (Another useful inequality to know is $\lg^{O(1)} n < 2^{\sqrt{\lg n}} < n^{\Theta(1)}$.)

A better solution is to use B-trees on the $\Theta(\lg^2 n)$ block elements. We label each edge in the B-tree according to its index from the parent node. The bottom-level tag of an element in the leaves is the concatenation of the edge labels along the root-to-leaf path. The overall tag of an element is the concatenation of its top-level tag and bottom-level tag. We need $O(\lg n)$ tag bits for the top level and $O(\lg \lg^2 n) = O(\lg \lg n)$ bits for the bottom. The tags will thus fit in $O(1)$ words for comparison.

With B-trees, we can afford to relabel all descendants of a node whenever it is touched. For example, after an insert in the 2-4 tree in the following figure, the labels of the right-hand descendants have to have part of their label updated from 0 to 1.

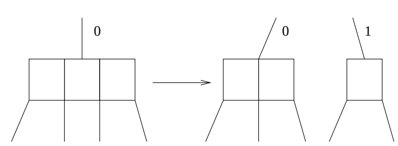

Figure 5: Descendant relabeling on insert in a 2-4-tree

In this way, we can achieve amortized $O(1)$ update cost. It is also possible to achieve worst-case $O(1)$ update cost [DS87, BCD+02].

Question: What about using indirection again on the $\lg^2 n$ blocks?

Answer: Yes, we can also use the $\Theta(n^2)$ tag-space representation at the second level, which will give us $O((\lg \lg n)^2)$ update cost, hence we need the bottom-level blocks to have at least $(\lg \lg n)^2$ elements. Even if the bottom-level blocks store $\Theta(\lg n)$ elements, we can afford the $O(2^n)$ tag-space representation and we achieved again an amortized $O(1)$ cost.

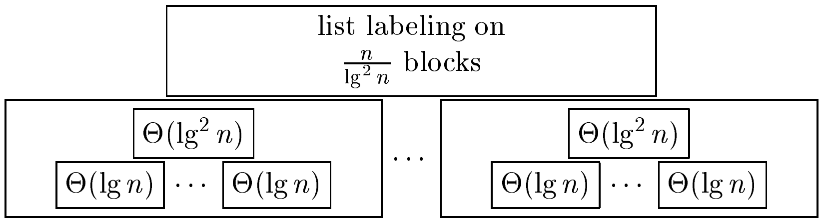

Figure 6: Double indirection using a list-labeling black box

Alternative indirection: Another approach would be to use as a black box the $O(n^2)$ tag-space list-labeling structure with update of $O(\lg n)$. Thus we only need to take care of bottom-level blocks of size $\Theta(\lg n)$. The $O(2^n)$ representation on the bottom level gives us linear space.

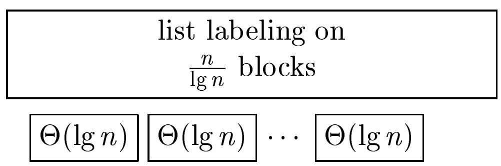

Figure 7: Indirection using a $\Theta(n^2)$ list labeling black box

4 Memory-Hierarchy Models

Most modern computers do not have a flat memory. They have a memory hierarchy consisting of multiple levels of memory with each level getting progressively slower and larger.

4.1 External-Memory Model [AV88]

In this model, we only consider a two-level memory hierarchy, on the assumption that the last two levels dominate the total cost. The memory closer to the CPU is called the *cache* and the level further away is the *disk*. The cache has size M divided into blocks of size B. Thus there are M/B blocks. The disk is conceptually infinite memory, again with block size B. The CPU always accesses the cache. If the requested item is not in the cache, then the block containing the item is brought in from the disk into the cache. If the cache is full, some block is thrown out to make space for the new block. The primary cost model is that we only charge for the memory transfers between the disk and the cache.

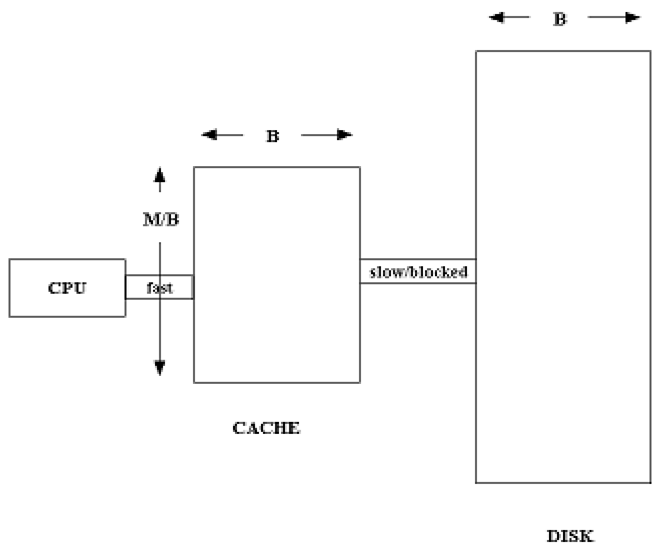

Figure 8: 2-level Memory Heirarchy

The transfers between disk and cache are always in terms of blocks. For these block transfers to be useful, we need locality of reference. For example, it is much cheaper to access the data sequentially and not randomly.

4.1.1 Examples

- Scanning N elements in order costs $\lceil N/B \rceil$ memory transfers.[1]
- Searching in a B-tree with branching factor equal to the block size B costs $O(\log_B N)$ memory transfers. This is because we can transfer one entire node in one memory transfer and thus the number of memory transfers is equal to the height of the tree.

  This bound is also information-theoretically optimal: you need $\lg N$ bits of information to find the index, and in each memory transfer we learn $\lg B$ bits (where the query element fits among the B read elements), so we need $\frac{\lg N}{\lg B} = \log_B N$ memory transfers to search. Thus the B-tree can be tuned to give optimal performance if M and B are known in the external-memory model.

4.2 Cache-Oblivious Model [FLPR99]

This model is similar to the external-memory model except that the algorithm is not aware of the values of M and B and is not allowed to find it by sampling or any other techniques. The algorithm should work exactly the same for all values of M and B.

[1]Note the historical notation of external-memory sizes with CAPS.

4.2.1 Advantages

- Self-tuning. The algorithm doesn't have to be tuned for various machines, with respect to B and M.
- Works well with multiple level of hierarchy. For example, consider the case of B-trees, if we have multiple levels of hierarchy, each with different M and B. To be optimal at all levels, we would have to nest B-trees inside every node of the B-trees recursively. On the other hand, if our algorithm does not depend on M and B, and is optimal for all M and B, then the analysis applies equally well to all levels of memory, so the algorithm is optimal on multilevel memory hierarchies.
- Clean and easy to use. Algorithms are phrased as if they are on the RAM; only the analysis is aware of a cache.
- It is often possible!

4.2.2 Examples

- Scanning N elements costs at most $\lceil N/B \rceil + 1$ memory transfers. There is an extra $+1$ to account for possible misalignment with the blocks, which we cannot predict.
- We'll see that it is also possible to have optimal performance in B-trees without knowing B!

References

[AV88] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *CACM*, 31(9):1116–1127, 1988.

[BCD$^+$02] Michael A. Bender, Richard Cole, Erik D. Demaine, Martin Farach-Colton, and Jack Zito. Two simplified algorithms for maintaining order in a list. In *Proceedings of the 10th Annual European Symposium on Algorithms (ESA 2002)*, volume 2461 of *Lecture Notes in Computer Science*, pages 152–164, Rome, Italy, September 17–21 2002.

[BDFC00] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Cache-oblivious b-trees. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science (FOCS 2000)*, pages 399–409, Redondo Beach, California, November 12–14 2000.

[DS87] P. F. Dietz and D. D. Sleator. Two algorithms for maintaining order in a list. In *19th ACM Symposium Theory of Computing*, pages 365–372, New York, May 1987.

[DSZ94] Paul F. Dietz, Joel I. Seiferas, and Ju Zhang. A tight lower bound for on-line monotonic list labeling. In *Proceedings of the 4th Scandinavian Workshop on Algorithm Theory*, volume 824 of *Lecture Notes in Computer Science*, pages 131–142, Aarhus, Denmark, July 1994.

[FLPR99] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science*, pages 285–298, New York, October 1999.

[IKR81] Alon Itai, Alan G. Konheim, and Michael Rodeh. A sparse table implementation of priority queues. In *Automata, Languages and Programming (ICALP)*, volume 115 of *Lecture Notes in Computer Science*, pages 417–431. Springer-Verlag, 1981.

[Wil92] Dan 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):150–204, 1992.