6.897: Advanced Data Structures	Spring 2003
Lecture 5 — February 26, 2003	
Prof. Erik Demaine	*Scribe: Christopher Collier*

1 Overview

Lecture 4 introduced fusion trees [FW93]. This lecture finishes the description of fusion trees, then proceeds to discuss a new topic: self adjusting data structures.

2 Finishing the Description of Fusion Trees

2.1 Motivation

The comparison model is often used to provide a strict, universal lower bound of $\Omega(n \lg n)$ operations to sort a set of n items, and $\Omega(\lg n)$ to search for a given key in a sorted list of n items.

Fusion trees use a different model of computation and a greatly modified B-tree structure to provide (as we have seen them so far) a workable $O(\frac{n \lg n}{\lg \lg n})$ and $O(\frac{\lg n}{\lg \lg n})$ solution to sorting and searching, respectively. In fact, in this lecture, we will improve these bounds to $O(n\sqrt{\lg n})$ and $O(\sqrt{\lg n})$, respectively, via randomization.

More generally, the goal of fusion trees is to support the operations insert(x), delete(x), successor(x) and predecessor(x) in $o(\lg n)$ time on a transdichotomous RAM. The transdichotomous RAM model of computation was intrdouced in Lecture 2 (Feb. 12, 2003). The data-structure problem defined by these four operations is called the *fixed-universe successor problem* which was introduced in Lecture 1 (Feb. 10th, 2003). We have already studied two data structures that solve this problem, the van Emde Boas structure and y-fast trees, both achieving time bounds of $\Theta(\lg \lg u)$.

2.2 The Problem - Fixed-Universe Successor

The fixed-universe successor problem is the following: maintain a set S of elements of size n, where the elements are members of a bounded universe U of size u, subject to the following operations:

insert(x)	$(x \in U \setminus S)$
delete(x)	$(x \in S)$
successor(x)	$(x \in U)$: find the smallest element $\in s$ that is $> x$
predecessor(x)	$(x \in U)$: find the largest element $\in s$ that is $< x$

2.3 Overview of Fusion Trees

A fusion tree is a B-tree with three major modifications:

1. The branching factor, B, is $(\lg n)^{1/5}$. Note that this is an unbounded increasing function of n, unlike normal B-trees.

2. Because the branching factor of the tree is an increasing function of n and will most likely be larger than 2, some sort of scheme must be devised to choose among directions for the search path at each node in constant time. This is done by constructing a compressed representation of each key within each node, such that all of the key representations fit into one word. Word paralellism can then be used to make many comparisons at once. This compressed representation is called the *sketch* of x, given a key x.

3. The leaves of the tree store representative elements of standard B-tree subtrees of size approximately B^4, rather than directly storing the data itself. This is called *indirection.* A similar use of this technique was introduced in the discussion of y-fast trees (see Lecture 4). The motivation for this structural characteristic of fusion trees is to keep the amortized cost of updates to the tree consistent with that of search operations.

In Lecture 4, the first two of these three defining characteristics of fusion trees were described. This was enough information to build a static fusion tree. The fusion-tree component of this lecture reviews this structure for the static fusion tree, describes why the third characteristic needs to be added to make the fusion tree dynamic, and then shows how this third characteristic is added.

2.4 The Static Fusion Tree

Remember, a fusion tree is still a B-tree, so lookups are accomplished by walking down the root-to-leaf path. The difficulty comes in choosing the correct path at each node in constant time, as is required to keep time complexity to the desired $O(\frac{\lg n}{\lg \lg n})$ for an entire trip down the tree. (Recall that the height of a fusion tree is $O(\log_B n) = O(\frac{\lg n}{\lg \lg n})$.)

As previously mentioned, this branch decision is accomplished by constructing a compressed key representation, $sketch(x)$, for each key such that all $\leq B-1$ keys in a node fit in a single word, and then using word parallelism to do several comparisons at once. To see that such a compressed key representation can be constructed, consider the following:

1. The binary representation of any key (keys are integers) can be thought of as a path down a binary tree, with the most significant bit at the root of the tree.

2. Note that the only bits that contain useful information differentiating the keys are those that correspond to tree nodes that have representative bits in both of their subtrees. These are called *branching nodes.*

3. Because there are B keys in each node, there are at most $B-1$ branching nodes.

Ideally then, each key could be represented in $B-1$ bits, by extracting the bits corresponding to levels containing branching nodes from the binary-tree representation of all keys. This notion is referred to as the *perfect sketch.* A problem arises, however, in that there is no known good way to compress the branching node bits into consecutive blocks of bits. This problem is solved through the use of a procedure that relocates the bits into close, but not necessarily consecutive, positions, and retains their relative order. This relocation procedure consists of a series of multiplications

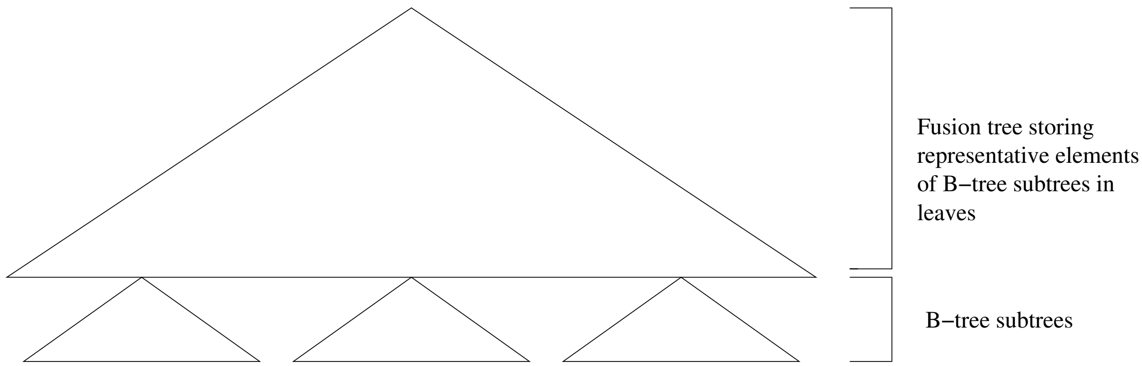

Figure 1: Fusion trees use indirection and store representative elements of B-tree subtrees in the leaves.

and bitwise AND operations, and was described in detail in Lecture 4. The most important characteristic of this nonperfect sketch representation is that it requires up to B^4 bits per element.

2.5 Making Fusion Trees Dynamic

In a regular B-tree, the upper bound on time complexity for insert/delete is constant amortized time. The problem with achieving this bound in fusion trees is that all of the sketches must be recomputed upon each insert or delete. This would give $\Theta(B^4)$ time for every insert/delete, because for each node the *sketch* operation takes time linear in the number of bits of the sketch representation. This isn't terrible, but it is possible to do updates in $O(\lg \lg n)$ time.

This improvement is obtained using indirection. Use the leaves of a fusion tree to store representative elements of subtrees, which are normal B-trees with $\Theta(B^4)$ elements each. Changes to the main fusion tree cost $O(B^4)$ amortized time, but each subtree costs $O(1)$ amortized time to update, because these subtrees are regular B-trees. When a subtree gets too big, it splits into two and a new node in the main tree is created to hold the new tree made from one half of the subtree.

This scheme increases the search time, because the regular B-tree subtrees require $O(\lg n')$ time for their portion of the search, but because each B-tree holds only $\Theta(B^4)$ elements, and $B = (\lg n)^{1/5}$, the addition of the B-trees incurs only an additive increase of $O(\lg \lg n)$ slowdown. On the other hand, the update time is reduced by a factor of B^4.

This indirection is not very balanced. The term *balanced* means that the time complexities for differently structured parts of a data structure are the same. In this analysis, however, the time complexities for the main fusion tree and the B-tree subtrees are different, which suggests a weak overall bound.

3 Making Fusion Trees Faster

We can make a fusion tree run in $O(\sqrt{\lg n})$ time if we don't mind randomization.

First we observe that the van Emde Boas structure performs well when u is not much larger then n. Remember that the query complexity for the van Emde Boas structure is $O(\lg \lg u)$. To obtain

$\lg\lg u \leq \sqrt{\lg n}$, we need $(\lg\lg u)^2 \leq \lg n$, or equivalently $n \geq 2^{(\lg\lg u)^2}$.

On the other hand, if $n \leq 2^{(\lg\lg u)^2}$, then $2^{(1/5)\sqrt{\lg n}} \leq 2^{(1/5)\lg\lg u} = (\lg u)^{1/5}$. Therefore, if we use fusion trees with a branching factor of $(\lg u)^{1/5}$ instead of the usual $(\lg n)^{1/5}$, which will work just as well because a word has at least $\lg u$ bits, then we are also using a branching factor of at least $2^{(1/5)\sqrt{\lg n}}$. Thus, we obtain a search time in the top-level fusion tree of $\log_B n = \frac{\lg n}{\Theta(\sqrt{\lg n})} = \Theta(\sqrt{\lg n})$. The cost of searching in the bottom level is $\Theta(\lg B^4) = \Theta(\lg B) = \Theta(\sqrt{\lg n})$.

Therefore, in either case, we obtain a running time of $\Theta(\sqrt{lgn})$.

4 Self-Adjusting Data Structures

This is an entirely new topic. In this lecture, we will introduce a series of models that facilitate the analysis of the performance of self-adjusting data structres. We will also introduce some basic self-adjusting schemes for a linked-list data structure, and describe some of the main results.

We will use the comparison model throughout our discussion on self-adjusting data structures.

A regular linear search through a linked list of n elements has time complexity n in the worst case, and $n/2$ in the average case. But this average-case analysis assumes a randomly, uniformly distributed set of elements. What if the search probabilities are not uniformly distributed? The *stochastic model* is used to model the situation in which there is some probability distribution among elements.

4.1 The Stochastic Model

The stochastic model makes two assumptions:

- Searches request element i with probability p_i.
- The requests are independent events.

The optimal scheme for arranging a linked list then is to put the elements with the highest probability at the front of the list, i.e., to arrange the elements in order of decreasing probability. For our list of n elements with probablilties $p_1, p_2, \ldots, p_n$, if we relabel so that $p_1 \geq p_2 \geq p_3 \geq \cdots \geq p_n$, then the cost of this optimal scheme is $cost(OPT) = \sum_{i=1}^{n} ip_i$. (Note that OPT will be used here as an abbreviation for "optimal".)

What if the probablities in the distribution of elements are not known? One can change the list order on-the-fly to adapt to the input distribution, to make searches fast. This is the notion of self-adjusting data structures.

Following are three schemes for on-the-fly ordering:

1. *Frequency Count* — Count the requests for each element and store the request counts with each element. Then order the list of elements by those counts.
2. *Move-to-front* — Move an element to the front upon each request for it.

3. *Transpose* — Swap an element with the one in front of it upon request.

In the last two schemes, state is maintained entirely in the list itself. The frequency count scheme uses additional space for each element to keep its state. This extra space makes frequency count a little more "messy".

4.2 Performance of Dynamic Ordering Schemes

How do these dynamic ordering schemes perform in the stochastic model? The following is a summary of research that has been done in this area. MTF will be used as an abbreviation for "Move-to-front", and FC for "frequency count". Note that these analyses only hold if the request sequence is sufficiently long, which is necessary for the stochastic model to really "kick in".

- $cost(FC) \sim OPT$ [Bit79]
- $cost(Transpose) \geq cost(MTF)$ [Riv76]
  (This inequality is strict unless $n \leq 2$ or all p_i's are equal)
- $cost(MTF) < 2 \times OPT$ [Riv76]
- $cost(MTF) \leq \frac{\pi}{2} \times OPT$ [CHS88]
- For some distributions, the previous bound is tight [GMS81]

4.3 The General Model

The general (nonstochastic) model is used to model situations where the search requests are correlated. The general model can be described in terms of its definition of frequency and probability. In an arbitrary request sequence of length m on n elements, let f_i (the frequency of i) be the number of requests for i. Thus $\sum_{i=1}^{n} f_i = m$, the total number of requests. The "probability" (really a relative frequency) of i is $p_i = f_i/n$.

If we were omnicient and knew how many requests for each element were going to occur, we could store the elements by decreasing request frequency: $f_1 \geq f_2 \geq f_3 \geq \cdots \geq f_n$. Because the elements are pre-arranged and do not change place during the request sequence, this scheme is called the *static optimal.* Its cost is $\sum_{i=1}^{n} i f_i$, for an amortized cost per element of $\sum_{i=1}^{n} i p_i$.

We can do better than this static optimal however, if the order of the elements can be changed on the fly, leading to the notion of *dynamic optimality.* Two new model changes are now introduced that provide a framework for analysis of dynamic ordering schemes within the general model. These model changes are intentionally restrictive, to allow for clear-cut analysis of the ideas at hand.

4.4 Cost Model

The basic cost model, called the *Sleator-Tarjan cost model*, is as follows:

- To search and find element i at position i, the cost is i.

- To swap two adjacent elements, the cost is 1 (a "paid swap").
- Upon finding an element, moving it partially toward the front or completely to the front is free (a "free swap").

Thus, Move-to-Front and Tranpose make only free swaps.

4.5 Startup Model (venture capital)

For a finite request sequence, we need to be a little careful how the whole process gets started, because otherwise it depends on the initial order of the list. Thus we generally make the following startup assumptions to put all algorithms on a level playing field:

- The list starts empty.
- The first time an element is requested, it is placed at the end of the list.

4.6 Tranpose is Poor

Interestingly, $cost(Transpose)$ can be very bad under this scheme [BM85]. The following example shows why this is so. Suppose your list looks like $3, 4, 5, \ldots, n, 1, 2$ and the request sequence is $\underbrace{1, 2, 1, 2, \ldots}_{m}$. Then $cost(Transpose) = \sum_{i=1}^{n} i + (m-n)n \sim mn$ because elements 1 and 2 never make it out of the back of the list. On the other hand, $cost(static\ OPT)$ in this case is $\sum_{i=1}^{n} i + 1.5m \sim 1.5m$ because only 2 items are ever requested.

4.7 Static Optimality

It turns out that $cost(MTF\ or\ FC) \leq 2 \times cost(static\ OPT)$, so both these schemes will do much better than Transpose in this request sequence. In general, this property is called *static optimality.*

Why MTF performs this well can be shown by looking at how many unsuccessful comparisons are made when looking for an element. In other words, how many times are we looking for j but find i? If we then sum $cost_{ij}$ for all i, j, we get the total cost for searches for all elements in our set.

Suppose $f_i \leq f_j$. Static OPT orders i before j. So, in OPT, $cost_{ij} = f_i$, and the number of times i is found when looking for j is f_i, because i will always be encountered before j on each search for j.

The worst case for MTF is that we see (i) $f_i - f_j$ times, and we see (j, i) f_j times. Therefore, for MTF, $cost_{ij} \leq 2f_j$.

Therefore, $cost(MTF) \leq 2 \times cost(staticOPT)$.

4.8 Wrapup

Notice that, in the stochastic model, Transpose performed better than MTF, but in this general model, with the cost and startup modifications, MTF performs better than Transpose. This goes to show that the model really does matter in these type of performance analyses. We'll see this behavior many more times.

Next class we'll move on to dynamic optimality.

References

[BM85] J. L. Bentley and C. C. McGeoch. Amortized analyses of self-organizing sequential search heuristics. *Comm. ACM* 28:404–411, 1985.

[Bit79] J. R. Bitner. Heuristics that Dynamically Organize Data Structures. *SIAM J. Comput.* 8(1):82–110, Feb. 1979.

[CHS88] F. R. K. Chung, D. J. Hajela, P. D. Seymour: Self-organizing sequential search and Hilbert's inequalities. *J. Comp. Systems Sc.* 36(2):148–157, 1988.

[FW93] M. L. Fredman and D. E. Willard. Surpassing the Information Theoretic Bound with Fusion Trees. *J. Comp. System Sc.* 47(3):424–436, 1993.

[GMS81] G. H. Gonnet, J. I. Munro and H. Suwanda. Exegesis of Self-Orgainzing Linear Search. *SIAM J. Comput.* 10(3):613–637, Aug. 1981.

[Riv76] R. Rivest. On self-organizing sequential search heuristics. *Communications of the ACM* 19(2):63–67, Feb. 1976.