6.851: Advanced Data Structures Spring 2007

Lecture 1 — February 12, 2007

Prof. Erik Demaine *Scribe: Ray C. He*

1 Overview

In this lecture we discussed the topics to be covered later in class. We also talked about linked lists as a data structure and how access times for elements can be optimized using a self-adjusting linked list, by using algorithms such as Move-To-Front and Order-By-Next-Request.

2 Linked Lists and the World of Self-Adjusting Data Structures

Self-adjusting data structures will change even as we access data. Ideally, we want the second access to our data structure to take $O(1)$ time, even if it longer to access the data the first time around.

The data structure we want to examine is the linked list, where each element contains a pointer to the next. The query sent to the list would be: $access(x)$, which would cost i, assuming x can be found in the ith position in the list.

2.1 Linear Search Model

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.

2.2 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 probabilities $p_1, p_2, \ldots, p_n$, 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 probabilities 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.

2.3 The General Model

The general (non-stochastic) 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 x_1, x_2,$\cdots$, x_m, 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 $p_i = f_i/m$.

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 $\sum_{i=1}^{n} if_i$, for an amortized cost per element of $\sum_{i=1}^{n} ip_i$.

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. To start, we have a an initially arbitrarily ordered list or an initially empty list (that is, $find(x)$ will append x to the list).

2.3.1 one-finger model

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

- $find(x)$ starts at the front of the list.
- To search and find element i at position i, the cost is i (moving forwards and backwards by 1 position costs 1).
- 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.

2.3.2 constant-finger model

All the fingers in the constant finger model start at the front fo the list, and each finger has the same behavior as the finger in the one-finger model. Operations such as standard pointer copying are allowed.

2.4 Natural $access(x)$ algorithms

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

1. *Frequency Count (FC)* — Count the requests for each element and store the request counts with each element. Then order the list of elements by those counts.
2. *Transpose* — Swap an element with the one in front of it upon request.
3. *Move-to-front (MTF)* — Move an element to the front upon each request for it.

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".

2.5 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. 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]

2.5.1 Transpose

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.

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

2.5.3 Dynamic Optimality

Until now, we have been comparing to "static OPT", which is defined to be the optimal list order subject to the constraint that the list order cannot change in the middle of the request sequence. (In the stochastic model, this constraint didn't matter, because we assumed that the search requests were independent: since we weren't able to predict what was coming next, no algorithm could preemptively optimize the list order to take advantage of information about future requests.) In some sense, comparing the MTF and FC algorithms to static optimality is "cheating", because we are using dynamic information to approximate a particular static ordering.

Henceforth, we drop the independence assumption; we will talk about request sequences. If an algorithm has the entire future request sequence available to it, then the algorithm will be able to outpace a statically optimum algorithm. In our model, we allow the optimum algorithm to be *omniscient*. (The terminology suggests that such a model would be unreasonable, but we talk about these algorithms all the time; usually we call them *offline* algorithms.) We expect that OPT would re-order its list dynamically to use its information about the future request sequence. If we're lucky, we'll still be able to design non-omniscient (online) algorithms that are nearly as good as the optimal omniscient (offline) algorithm.

2.5.4 Terms and background

First, we need to define our terms; we shall evaluate our algorithms using the following definition:

Definition 1. *(Competitiveness) Algorithm A is* c-competitive *if there is a number b such that, for all request sequences σ,*

$$\text{cost}_A(\sigma) \leq c \cdot \text{cost}_{OPT}(\sigma) + b,$$

where OPT is an omniscient optimum algorithm.

That is, for all inputs σ, our online algorithm A runs at most c times slower than the best possible algorithm. Note that algorithm A is handicapped: it only learns the request sequence as the searches happen, while OPT is an omniscient algorithm; it has access to the entire request sequence ahead of time.

2.5.5 Dynamic Optimality Results

Given a cost model and a definition of c-competitive, we can discuss the goodness of the algorithms we've seen so far. [ST85] showed the following:

- The Move-to-Front algorithm is 2-competitive.
- Frequency-Count is not $O(1)$-competitive.
- Transpose is not $O(1)$-competitive.

We will present the competitiveness proof for Move-To-Front shortly. Intuitively, Transpose is not $O(1)$-competitive because the request sequence `C D A B A B A B ...` will never bring either `A` or `B` to the front, giving an arbitrarily bad cost. Similarly, Frequency-Count can be foiled by a sequence of n `A`s, followed by $n-1$ `B`s, followed by $n-2$ `C`s, etc.

Can we find an algorithm that is better than 2-competitive? The following unpublished result by Karp and Raghavan answers this question in the negative:

Theorem 2. *No deterministic algorithm is c-competitive, where $c < 2$.*

2.6 Order By Next Request (OBNR)

We finish our discussion of Munro's Order-By-Next-Request (OBNR) strategy [ONR] by showing that the amortized cost of the algorithm achieves the entropy bound. OBNR is an offline (omniscient) algorithm, in that we know for each element the time that it will be accessed. So, upon request to an element at position i, we do the following:

1. Continue scan to position $\lceil\lceil i\rceil\rceil$, where $\lceil\lceil i\rceil\rceil = 2^{\lceil \lg i\rceil}$ is the hyperceiling of i.
2. Sort these elements according to when they will next be requested.

The total cost for a permutation of the elements was $\Theta(n \lg n)$. By contrast, Move-To-Front (or any online strategy) can cost $\Theta(n^2)$ (with the startup model, or with a nasty permutation of the list matching its initial order). Per element, the costs are $\Theta(\lg n)$ versus $\Theta(n)$, which is an exponential discrepancy.

While OBNR may seem "unfair" because it requires "knowing the future" (i.e., knowing when each element will be accessed), it is important to realize that this assumption is sometimes realistic. For example, when generating a minimum spanning tree, it is possible to predict the order of the nodes that are going to be visited—so in this sense we "know" the future.

Comparing ONR and MTF. Consider a list with n distinct keys; the access sequence consists of a repetition of a particular permutation of the keys. For definiteness, we fix the request sequence consisting of repetitions of $(1, 2, \ldots, 16)$. Table 1 illustrates the execution of ONR on this request sequence. Note that every ith access costs 2^i. The total cost of n accesses for one instance of the permutation is $n \cdot (1/2 \cdot 2 + 1/4 \cdot 4 + 1/8 \cdot 8 + \cdots)$ which is $O(n \lg n)$, giving an amortized cost per access of $O(\lg n)$.

Table 1: Execution of the Order-by-Next-Request algorithm. At each point the list is a sequence of blocks, with elements in each block sorted in next-request order.

Search for...	Cost	List after searching (next request circled)															
initialize	16	1	②	3	4	5	6	7	8	9	10	11	12	13	14	15	16
2	2	2	1	③	4	5	6	7	8	9	10	11	12	13	14	15	16
3	4	3	④	1	2	5	6	7	8	9	10	11	12	13	14	15	16
4	2	4	3	1	2	⑤	6	7	8	9	10	11	12	13	14	15	16
5	8	5	⑥	7	8	1	2	3	4	9	10	11	12	13	14	15	16
6	2	6	5	⑦	8	1	2	3	4	9	10	11	12	13	14	15	16
7	4	7	⑧	5	6	1	2	3	4	9	10	11	12	13	14	15	16
8	2	8	7	5	6	1	2	3	4	⑨	10	11	12	13	14	15	16
9	16	9	⑩	11	12	13	14	15	16	1	2	3	4	5	6	7	8

By contrast, MTF (or any online algorithm) might cost $\Theta(n)$ per request, since we can choose a request sequence that always requests the element that is currently last in the list.

2.6.1 OBNR cost analysis

Theorem 3. *The amortized cost of OBNR to access an element is* $\leq 1+4\lceil \lg r \rceil$, *where* r = *number of distinct elements accessed since this element was last accessed (including the element itself).*

Corollary 4. *The total cost of OBNR is* $\leq m + 4\sum_{i=1}^{n} \lceil \lg \frac{m}{f_i} \rceil$, *where* f_i = *number of occurences of* i. *Let* $p_i = \frac{f_i}{m}$, *the probability of accessing element* i. *Then the following statements are true:*

1. *The amortized cost for element* i *is* $O(\lg \frac{1}{p_i})$.
2. *The "expected" cost for a random element is* $O(\sum_{i=1}^{n} p_i \lg \frac{1}{p_i})$.
3. *The total cost is* $O(m \sum_{i=1}^{n} p_i \lg \frac{1}{p_i})$.

The bound in statement 2 above represents the *entropy* of the probability distribution characterized by the p_i. This is promising because it shows that the expected cost of OBNR for a random element approaches the information-theoretic lower bound.

Now let's prove the above theorem.

Proof: We begin with two failed attempts (at calculating the cost of OBNR) before demonstrating the correct approach.

Proposition 1: *Charge the elements in the last block (i.e. the largest block).*

This is a bad idea because most of these elements might never be requested.

Proposition 2: *Charge the elements in the front (i.e. the small blocks, for some notion of small).*

Unfortunately, this is still a bad idea because there aren't enough elements in these blocks; most of the elements (a constant fraction) are actually in the two largest blocks.

Proposition 3: *Soak the middle class: charge a cost of* $\lceil \lceil i \rceil \rceil$ *to the penultimate block* b *of size* $\frac{\lceil \lceil i \rceil \rceil + 1}{4}$ *(unless the request is for the first element, in which case we charge the first element).*

Claim 5. *An element* i *is charged at most once in any block, between two requests for* i*.*

Suppose i is in block b. There are two cases to consider:

Case 1: i gets charged in block b and either stays in b or moves forward. This is shown by the solid arrow in Figure 1. Elements from the new position of i to the end of block $b+1$ serve as a buffer because we know they will remain in the same relative order until i is accessed (i.e. they will only be accessed after i is accessed first). In this case, i can't be charged unless it moved to block $b+1$ and the resulting buffer did not extend to fill block $b+2$ (and then an element in block $b+2$ was requested).

Case 2: i moves to block $b+1$. This is shown by the dashed arrow in Figure 1. In this case, i may get charged by a request in block $b+2$, but again this will only happen once between two requests for i (see Claim 3).

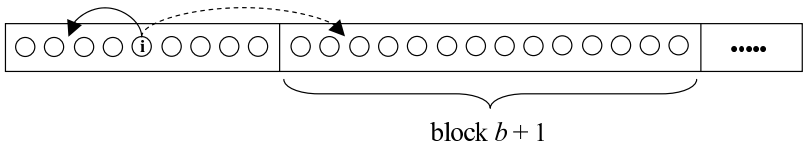

Figure 1: Possible movement of element i after a request for i. Case 1 is illustrated by the solid arrow; Case 2 is illustrated by the dashed arrow.

Now we ask the following question: how many blocks can i possibly be in? We know that i is charged at most once per block. Let r be the number of distinct elements accessed between two accesses to i. Then the farthest right i can possibly move is position r. This leads us to the following two conclusions:

1. i can be in at most $\lceil \lg r \rceil$ blocks, so there will be $\leq \lceil \lg r \rceil$ charges attributed to i.
2. The amortized cost to access element i will therefore be $\leq 4\lceil \lg r \rceil + 1$, where the additional 1 represents the base charge of accesses to the front of the list.

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.

[ONR] J.I. Munro. On the Copmletitiveness of Linear Search. *Proceedings of the 8th Annual European Symposium on Algorithms*. 338–345, 2000.

[ST85] Daniel Dominic Sleator and Robert Endre Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM* 28(2):202–208, Feb. 1985.