*Programming Techniques and Data Structures*

*Ellis Horowitz Editor*

# Amortized Efficiency of List Update and Paging Rules

DANIEL D. SLEATOR and ROBERT E. TARJAN

*ABSTRACT: In this article we study the amortized efficiency of the "move-to-front" and similar rules for dynamically maintaining a linear list. Under the assumption that accessing the ith element from the front of the list takes $\Theta(i)$ time, we show that move-to-front is within a constant factor of optimum among a wide class of list maintenance rules. Other natural heuristics, such as the transpose and frequency count rules, do not share this property. We generalize our results to show that move-to-front is within a constant factor of optimum as long as the access cost is a convex function. We also study paging, a setting in which the access cost is not convex. The paging rule corresponding to move-to-front is the "least recently used" (LRU) replacement rule. We analyze the amortized complexity of LRU, showing that its efficiency differs from that of the off-line paging rule (Belady's MIN algorithm) by a factor that depends on the size of fast memory. No on-line paging algorithm has better amortized performance.*

## 1. INTRODUCTION

In this article we study the amortized complexity of two well-known algorithms used in system software. These are the "move-to-front" rule for maintaining an unsorted linear list used to store a set and the "least recently used" replacement rule for reducing page faults in a two-level paged memory. Although much previous work has been done on these algorithms, most of it is average-case analysis. By studying the amortized complexity of these algorithms, we are able to gain additional insight into their behavior.

By amortization we mean averaging the running time of an algorithm over a worst-case sequence of executions. This complexity measure is meaningful if successive executions of the algorithm have correlated behavior, as occurs often in manipulation of data structures. Amortized complexity analysis combines aspects of worst-case and average-case analysis, and for many problems provides a measure of algorithmic efficiency that is more robust than average-case analysis and more realistic than worst-case analysis.

The article contains five sections. In Section 2 we analyze the amortized efficiency of the move-to-front list update rule, under the assumption that accessing the $i$th element from the front of the list takes $\Theta(i)$ time. We show that this algorithm has an amortized running time within a factor of 2 of that of the optimum off-line algorithm. This means that no algorithm, on-line or not, can beat move-to-front by more than a constant factor, on any sequence of operations. Other common heuristics, such as the transpose and frequency count rules, do not share this approximate optimality.

In Section 3 we study the efficiency of move-to-front under a more general measure of access cost. We show that move-to-front is approximately optimum as long as the access cost is convex. In Section 4 we study paging, a setting with a nonconvex access cost. The paging rule equivalent to move-to-front is the "least recently used" (LRU) rule. Although LRU is not within a constant factor of optimum, we are able to show that its amoritzed cost differs from the optimum by a factor that depends on the size of fast memory, and that no on-line algorithm has better amortized performance. Section 5 contains concluding remarks.

---

A preliminary version of some of the results was presented at the Sixteenth Annual ACM Symposium on Theory of Computing, held April 30–May 2, 1984 in Washington, D.C.

## 2. SELF-ORGANIZING LISTS

The problem we shall study in this article is often called the *dictionary problem*: Maintain a set of items under an intermixed sequence of the following three kinds of operations:

| | |
|---|---|
| **access**($i$): | Locate item $i$ in the set. |
| **insert**($i$): | Insert item $i$ in the set. |
| **delete**($i$): | Delete item $i$ from the set. |

In discussing this problem, we shall use $n$ to denote the maximum number of items ever in the set at one time and $m$ to denote the total number of operations. We shall generally assume that the initial set is empty.

A simple way to solve the dictionary problem is to represent the set by an unsorted list. To access an item, we scan the list from the front until locating the item. To insert an item, we scan the entire list to verify that the item is not already present and then insert it at the rear of the list. To delete an item, we scan the list from the front to find the item and then delete it. In addition to performing access, insert, and delete operations, we may occasionally want to rearrange the list by exchanging pairs of consecutive items. This can speed up later operations.

We shall only consider algorithms that solve the dictionary problem in the manner described above. We define the cost of the various operations as follows. Accessing or deleting the $i$th item in the list costs $i$. Inserting a new item costs $i + 1$, where $i$ is the size of the list before the insertion. Immediately after an access or insertion of an item $i$, we allow $i$ to be moved at no cost to any position closer to the front of the list; we call the exchanges used for this purpose *free*. Any other exchange, called a *paid* exchange, costs 1.

Our goal is to find a simple rule for updating the list (by performing exchanges) that will make the total cost of a sequence of operations as small as possible. Three rules have been extensively studied, under the rubric of *self-organizing linear lists*:

*Move-to-front (MF).* After accessing or inserting an item, move it to the front of the list, without changing the relative order of the other items.

*Transpose (T).* After accessing or inserting an item, exchange it with the immediately preceding item.

*Frequency count (FC).* Maintain a frequency count for each item, initially zero. Increase the count of an item by 1 whenever it is inserted or accessed; reduce its count to zero when it is deleted. Maintain the list so that the items are in nonincreasing order by frequency count.

Bentley and McGeoch's paper [3] on self-adjusting lists contains a summary of previous results. These deal mainly with the case of a fixed set of $n$ items on which only accesses are permitted and exchanges are not counted. For our purposes the most interesting results are the following. Suppose the accesses are independent random variables and that the probability of accessing item $i$ is $p_i$. For any Algorithm $A$, let $E_A(p)$ be the asymptotic expected cost of an access, where $p = (p_1, p_2, \ldots, p_n)$. In this situation, an optimum algorithm, which we call *decreasing probability* (DP), is to use a fixed list with the items arranged in nonincreasing order by probability. The strong law of large numbers implies that $E_{\mathrm{FC}}(p)/E_{\mathrm{DP}}(p) = 1$ for any probability distribution $p$ [8]. It has long been known that $E_{\mathrm{MF}}(p)/E_{\mathrm{DP}}(p) \leq 2$ [3, 7]. Rivest [8] showed that $E_{\mathrm{T}}(p) \leq E_{\mathrm{MF}}(p)$, with the inequality strict unless $n = 2$ or $p_i = 1/n$ for all $i$. He further conjectured that transpose minimizes the expected access time for any $p$, but Anderson, Nash, and Weber [1] found a counterexample.

In spite of this theoretical support for transpose, move-to-front performs much better in practice. One reason for this, discovered by Bitner [4], is that move-to-front converges much faster to its asymptotic behavior if the initial list is random. A more compelling reason was discovered by Bentley and McGeoch [3], who studied the amortized complexity of list update rules. Again let us consider the case of a fixed list of $n$ items on which only accesses are permitted, but let $s$ be any sequence of access operations. For any Algorithm $A$, let $C_A(s)$ be the total cost of all the accesses. Bentley and McGeoch compared move-to-front, transpose, and frequeny count to the optimum static algorithm, called *decreasing frequency* (DF), which uses a fixed list with the items arranged in nonincreasing order by access frequency. Among algorithms that do no rearranging of the list, decreasing frequency minimizes the total access cost. Bentley and McGeoch proved that $C_{\mathrm{MF}}(s) \leq 2C_{\mathrm{DF}}(s)$ if MF's initial list contains the items in order by first access. Frequency count but not transpose shares this property. A counterexample for transpose is an access sequence consisting of a single access to each of the $n$ items followed by repeated accesses to the last two items, alternating between them. On this sequence transpose costs $mn - O(n^2)$, whereas decreasing frequency costs $1.5m + O(n^2)$.

Bentley and McGeoch also tested the various update rules empirically on real data. Their tests show that transpose is inferior to frequency count but move-to-front is competitive with frequency count and sometimes better. This suggests that some real sequences have a high locality of reference, which move-to-front, but not frequency count, exploits. Our first theorem, which generalizes Bentley and McGeoch's theoretical results, helps to explain this phenomenon.

For any Algorithm $A$ and any sequence of operations $s$, let $C_A(s)$ be the total cost of all the operations, not counting paid exchanges, let $X_A(s)$ be the number of paid exchanges, and let $F_A(s)$ be the number of free exchanges. Note that $X_{\mathrm{MF}}(s) = X_{\mathrm{T}}(s) = X_{\mathrm{FC}}(s) = 0$ and that $F_A(s)$ for any algorithm $A$ is at most $C_A(s) - m$.

(After an access or insertion of the $i$th item there are at most $i - 1$ free exchanges.)

THEOREM 1.
*For any Algorithm A and any sequence of operations s starting with the empty set,*

$$C_{MF}(s) \leq 2C_A(s) + X_A(s) - F_A(s) - m.$$

PROOF.
In this proof (and in the proof of Theorem 3 in the next section) we shall use the concept of a *potential function.* Consider running Algorithms $A$ and MF in parallel on $s$. A potential function maps a configuration of $A$'s and MF's lists onto a real number $\Phi$. If we do an operation that takes time $t$ and changes the configuration to one with potential $\Phi'$, we define the *amortized time* of the operation to be $t + \Phi' - \Phi$. That is, the amortized time of an operation is its running time plus the increase it causes in the potential. If we perform a sequence of operations such that the $i$th operation takes actual time $t_i$ and has amortized time $a_i$, then we have the following relationship:

$$\sum_i t_i = \Phi - \Phi' + \sum_i a_i$$

where $\Phi$ is the initial potential and $\Phi'$ the final potential. Thus we can estimate the total running time by choosing a potential function and bounding $\Phi$, $\Phi'$, and $a_i$ for each $i$.

To obtain the theorem, we use as the potential function the number of inversions in MF's list with respect to $A$'s list. For any two lists containing the same items, an *inversion* in one list with respect to the other is an unordered pair of items, $i$, $j$, such that $i$ occurs anywhere before $j$ in one list and anywhere after $j$ in the other. With this potential function we shall show that the amortized time for MF to access item $i$ is at most $2i - 1$, the amortized time for MF to insert an item into a list of size $i$ is at most $2(i + 1) - 1$, and the amortized time for MF to delete item $i$ is at most $i$, where we identify an item by its position in $A$'s list. Furthermore, the amortized time charged to MF when $A$ does an exchange is $-1$ if the exchange is free and at most 1 if the exchange is paid.

The initial configuration has zero potential since the initial lists are empty, and the final configuration has a nonnegative potential. Thus the actual cost to MF of a sequence of operations is bounded by the sum of the operations' amortized times. The sum of the amortized times is in turn bounded by the right-hand side of the inequality we wish to prove. (An access or an insertion has amortized time $2c_A - 1$, where $c_A$ is the cost of the operation to $A$; the amortized time of a deletion is $c_A \leq 2c_A - 1$. The $-1$'s, one per operation, sum to $-m$.)

It remains for us to bound the amortized times of the operations. Consider an access by $A$ to an item $i$. Let $k$ be the position of $i$ in MF's list and let $x_i$ be the number of items that precede $i$ in MF's list but follow $i$ in $A$'s list. (See Figure 1.) Then the number of items preceding $i$ in both lists is $k - 1 - x_i$. Moving $i$ to the front of MF's list creates $k - 1 - x_i$ inversions and destroys $x_i$ other inversions. The amortized time for the operation (the cost to MF plus the increase in the number of inversions) is therefore $k + (k - 1 - x_i) - x_i = 2(k - x_i) - 1$. But $k - x_i \leq i$ since of the $k - 1$ items preceding $i$ in MF's list only $i - 1$ items precede $i$ in $A$'s list. Thus the amortized time for the access is at most $2i - 1$.

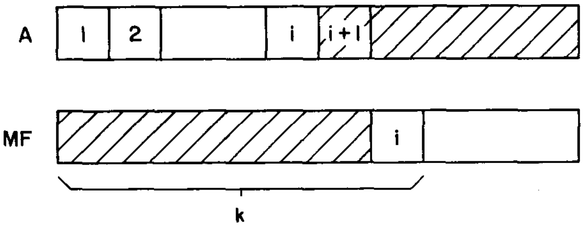

**FIGURE 1. Arrangement of *A*'s and MF's lists in the proofs of Theorems 1 and 4. The number of items common to both shaded regions is $x_i$.**

The argument for an access applies virtually unchanged to an insertion or a deletion. In the case of a deletion no new inversions are created, and the amortized time is $k - x_i \leq i$.

An exchange by $A$ has zero cost to MF, so the amortized time of an exchange is simply the increase in the number of inversions caused by the exchange. This increase is at most 1 for a paid exchange and is $-1$ for a free exchange. □

Theorem 1 generalizes to the situation in which the initial set is nonempty and MF and $A$ begin with different lists. In this case the bound is $C_{MF}(s) \leq 2C_A(s) + X_A(s) + I - F_A(s) - m$, where $I$ is the initial number of inversions, which is at most $\binom{n}{2}$. We can obtain a result similar to Theorem 1 if we charge for an insertion not the length of the list before the insertion but the position of the inserted item after the insertion.

We can use Theorem 1 to bound the cost of MF when the exchanges it makes, which we have regarded as free, are counted. Let the gross cost of Algorithm $A$ on sequence $s$ be $T_A(s) = C_A(s) + F_A(s) + X_A(s)$. Then $F_{MF}(s) \leq C_{MF}(s) - m$ and $X_{MF}(s) = 0$, which implies by Theorem 1 that $T_{MF}(s) \leq 4C_A(s) + 2X_A(s) - 2F_A(s) - 2m = 4T_A(s) - 2X_A(s) - 6F_A(s) - 2m$.

The proof of Theorem 1 applies to any update rule in which the accessed or inserted item is moved some fixed fraction of the way toward the front of the list, as the following theorem shows.

THEOREM 2.
*If* MF($d$) ($d \geq 1$) *is any rule that moves an accessed or inserted item at position k at least $k/d - 1$ units closer to the front of the list, then*

$$C_{MF(d)}(s) \leq d(2C_A(s) + X_A(s) - F_A(s) - m).$$