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.

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.

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 ith 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.

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 ith 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 $E_{FC}(p)/E_{DP}(p) = 1$ for any probability distribution p [8]. It has long been known that $E_{MF}(p)/E_{DP}(p) \leq 2$ [3, 7]. Rivest [8] showed that $E_T(p) \leq E_{MF}(p)$, 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 $C_{MF}(s) \leq 2C_{DF}(s)$ 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 $X_{MF}(s) = X_T(s) = X_{FC}(s) = 0$ and that $F_A(s)$ for any algorithm A is at most $C_A(s) - m$.

(After an access or insertion of the ith 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 Φ. If we do an operation that takes time t and changes the configuration to one with potential Φ', 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 ith 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 Φ is the initial potential and Φ' the final potential. Thus we can estimate the total running time by choosing a potential function and bounding Φ, Φ', 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 $2i - 1$.

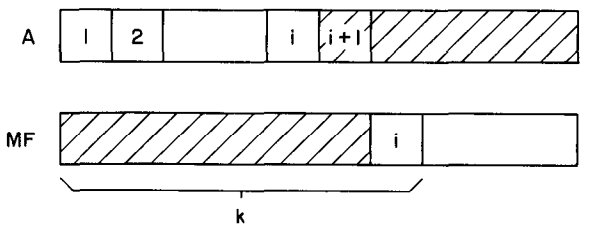

FIGURE 1. Arrangement of *A*'s and MF's lists in the proofs of Theorems 1 and 4. The number of items common to both shaded regions is x_i.

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*

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

PROOF.
The proof follows the same outline as that of Theorem 1. The potential function we use is d times the number of inversions in MF(d)'s list with respect to A's list. We shall show that the amortized time for MF(d) to access an item i (in position i in A's list) is at most $d(2i - 1)$, the amortized time for MF(d) to insert an item into a list of size i is at most $d(2i + 1) - 1$, and the amortized time for MF(d) to delete item i is at most i. Furthermore, the amortized time charged to MF(d) when A does an exchange is $-d$ if the exchange is free and at most d if the exchange is paid. These bounds are used as in the proof of Theorem 1 to give the result.

Consider an access to item i. Let k be the position of i in MF(d)'s list, and let p be the number of items past which MF(d) moves item i. Let x be the number of these items that occur after i in A's list. (See Figure 2.) The decrease in the number of inversions caused by this move is x, while the increase is $p - x$. Thus the potential increases by $d(p - 2x)$, and the amortized time of the insertion is $k + d(p - 2x)$. We want to show that this is less than $d(2i - 1)$. We know $k/d - 1 \leq p \leq x + i - 1$; the first inequality follows from the requirement on MF(d)'s update rule, and the second is true since each of the items past which i moves is either one of the $i - 1$ items preceeding i in A's list or one of the x items following i in A's list but preceding i in MF's list. Multiplying by d and using transitivity we get $k \leq d(x + i)$. Adding the second inequality multiplied by d gives $k + dp \leq d(2x + 2i - 1)$, which implies $k + d(p - 2x) \leq d(2i - 1)$, as desired.

A similar argument applies to insertion. In the case of deletion, the amortized time is at most $k - dx$, where x is defined as in Figure 1. This is at most $k - x$, which in turn is no greater than i, as shown in the proof of Theorem 1. Since i and d are at least 1, $i \leq 2i - 1 \leq d(2i - 1)$. Finally, a free exchange done by Algorithm A decreases the potential by d; a paid exchange increases or decreases it by d. Note that the $-d$ terms in the amortized times sum to give the $-dm$ term in the theorem. □

No analogous result holds for transpose or for frequency count. For transpose, a variant of the example mentioned previously is a counterexample; namely, insert n items and then repeatedly access the last two, alternating between them. A counterexample for frequency count is to insert an item and access it $k + n$ times, insert a second item and access it $k + n - 1$ times, and so on, finally inserting the nth item and accessing it $k + 1$ times. Here k is an arbitrary nonnegative integer. On this sequence, frequency count never rearranges the list and has cost $\Omega(kn^2) = \Omega(mn)$, whereas move-to-front has cost $m + O(n^2)$. The same example with the insertions omitted shows that using a static list ordered by decreasing access frequency is not within a constant factor of optimal.

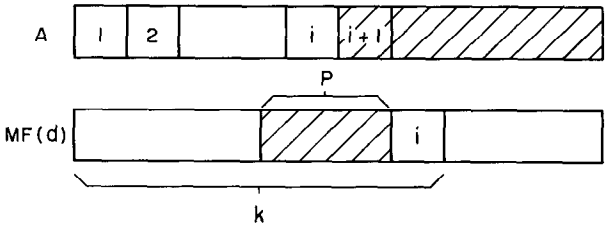

FIGURE 2. Arrangement of A's and MF(d)'s lists in the proof of Theorem 2. The number of items common to both shaded regions is x.

Theorem 1 is a very strong result, which implies the average-case optimality result for move-to-front mentioned at the beginning of the section. Theorem 1 states that on *any* sequence of operations, move-to-front is to within a constant factor as efficient as *any* algorithm, including algorithms that base their behavior on advance knowledge of the entire sequence of operations. If the operations must be performed on-line, such off-line algorithms are unusable. What this means is that knowledge of future operations cannot significantly help to reduce the cost of current operations.

More quantitatively, Theorem 1 provides us with a way to measure the inherent complexity of a sequence. Suppose we begin with the empty set and perform a sequence of insertions and accesses. We define the complexity of an access or insert operation on item i to be 1 plus the number of items accessed or inserted since the last operation on i. The complexity of the sequence is the sum of the complexities of its individual operations. With this definition the complexity of a sequence is exactly the cost of move-to-front on the sequence and is within a factor of 2 of the cost of an optimum algorithm.

3. SELF-ORGANIZING LISTS WITH GENERALIZED ACCESS COST

In this section we explore the limits of Theorem 1 by generalizing the cost measure. Let f be any nondecreasing function from the positive integers to the nonnegative reals. We define the cost of accessing the ith item to be $f(i)$, the cost of inserting an item to be $f(i + 1)$ if the list contains i items, and the cost of a paid exchange of the ith and $(i + 1)$st items to be $\Delta f(i) = f(i + 1) - f(i)$. (We define free and paid exchanges just as in Section 2.) If A is an algorithm for a sequence of operations s, we denote by $C_A(s)$ the total cost of A not counting paid exchanges and by $X_A(s)$ the cost of the paid exchanges, if any. To simplify matters we shall assume that there are no deletions.

We begin by studying whether $\Delta f(i)$ is a reasonable amount to charge for exchanging the ith and $(i + 1)$st items. If we charge significantly less than $\Delta f(i)$ for an exchange, then it may be cheaper to access an item by moving it to the front of the list, accessing it, and moving it back, than by accessing it without moving it. On

the other hand, if we charge at least $\Delta f(i)$ for an exchange, then as the following theorem shows, paid exchanges do not reduce the cost of a sequence.

THEOREM 3.
Let A be an algorithm for a sequence s of insertions and accesses. Then there is another algorithm for s that is no more expensive than A and does no paid exchanges.

PROOF.
Note that this theorem does not require the initial set to be empty. Also note that we can replace each insertion by an access without changing the cost. We do this by adding all the items to be inserted to the rear of the initial list, in order of insertion, and replacing each **insert**(i) operation by **access**(i). Thus we can assume without loss of generality that s contains only accesses.

To eliminate the paid exchanges, we move them one by one after the accesses. Once this is done, we can eliminate them without increasing the cost. Consider an exchange that occurs just before an access. Identify the items by their positions just before the exchange. Let i and $i + 1$ be the items exchanged and let j be the item accessed. There are three cases. If $j \notin \{i, i + 1\}$, we can move the exchange after the access without changing the cost. If $j = i$, we save $\Delta f(i)$ on the access by performing the exchange afterwards. If $j = i + 1$, we spend $\Delta f(i)$ extra on the access by postponing the exchange, but we can then perform the exchange for free, saving $\Delta f(i)$. Thus, in no case do we increase the cost, and we either convert the paid exchange to a free exchange or move it after an access. The theorem follows by induction. □

Theorem 3 holds for any nondecreasing cost function, whereas our generalization of Theorem 1, which follows, requires convexity of the cost function. We say f is *convex* if $\Delta f(i) \geq \Delta f(i + 1)$ for all i.

THEOREM 4.
If f is convex and A is any algorithm for a sequence s of insertions and accesses starting with the empty set,

$$C_{MF}(s) \leq 2C_A(s) + X_A(s) - mf(1).$$

PROOF.
The proof is just like the proof of Theorem 1. We run A and MF in parallel on s and use a potential function, defined as follows. At any time during the running of the algorithms, we identify the items by their positions in A's list. For each item i, let x_i be the number of items $j > i$ preceding i in MF's list. (See Figure 1.) The value of the potential is $\sum_i (f(i + x_i) - f(i))$, where the sum is over all the items in the lists.

We shall show that the amortized time for MF to access item i is at most $2f(i) - f(1)$, the amortized time to insert an item in a list of size i is at most $2f(i + 1) - f(1)$, and the amortized time charged to MF when A exchanges items i and $i + 1$ is at most $\Delta f(i)$ if the exchange is paid, at most zero if the exchange is free. Since the initial configuration has zero potential and the final configuration has nonnegative potential, the actual cost of the sequence of operations is bounded by the sum of the operations' amortized times. The theorem will therefore follow from a proof of the bounds in this paragraph.

Consider an access of an item i. Let k be the position of i in MF's list. The amortized time for the access is $f(k)$ (the actual cost) plus the increase in the potential caused by the access. The increase in potential can be divided into three parts: the increase due to items $j > i$, that due to item i, and that due to items $j < i$. Let x_j be as defined just before the exchange. For $j > i$ the value x_j does not change when i is moved to the front of MF's list, and there is no corresponding change in the potential. The potential corresponding to item i changes from $f(i + x_i) - f(i)$ to zero, an increase of $f(i) - f(i + x_i)$. For each item $j < i$, the value of x_j can increase by at most 1, so the increase in potential for all $j < i$ is at most

$$\sum_{j<i} (f(j + x_j + 1) - f(j + x_j)) = \sum_{j<i} \Delta f(j + x_j) \leq \sum_{j<i} \Delta f(j) = f(i) - f(1).$$

Combining our estimates we obtain an amortized time of at most $f(k) + f(i) - f(i + x_i) + f(i) - f(1) = 2f(i) - f(1) + f(k) - f(i + x_i)$. Each of the $k - 1$ items preceding i in MF's list either follows i in A's list (there are x_i such items) or precedes i in A's list (there are at most $i - 1$ such items). Thus $k \leq i + x_i$, and the amortized time for the access is at most $2f(i) - f(1)$.

As in the proof of Theorem 2, we can treat each insertion as an access if we regard all the unaccessed items as added to the rear of the list, in order of insertion. The potential corresponding to each unaccessed item is zero. Thus the argument for access applies as well to insertions.

Consider an exchange done by A. The amortized time charged to MF when A exchanges items i and $i + 1$ is just the increase in potential, since the actual cost to MF is zero. This increase is

$$f(i + x_i') - f(i + x_i) + f(i + 1 + x_{i+1}') - f(i + 1 + x_{i+1}),$$

where the primes denote values after the exchange. Note that the exchange causes item i to become item $i + 1$ and vice-versa. If the original item i precedes the original item $i + 1$ in MF's list, then $x_i' = x_{i+1} + 1$ and $x_{i+1}' = x_i$, which means the amortized time for the exchange is $\Delta f(i + x_i) \leq \Delta f(i)$. If the original item i follows the original item $i + 1$ in MF's list, as is the case when the exchange is free, then $x_i' = x_{i+1}$ and $x_{i+1}' = x_i - 1$, which means the amortized time for the exchange is $-\Delta f(i + x_{i+1}) \leq 0$. □

As is true for Theorem 1, Theorem 4 can be generalized to allow the initial set to be nonempty and the initial lists to be different. The effect is to add to the bound on $C_{MF}(s)$ a term equal to the initial potential, which depends on the inversions and is at most $\sum_{i=1}^{n-1} (f(n) - f(i))$. We can also obtain a result similar to

Theorem 4 if we charge for an insertion not $f(i+1)$, where i is the length of the list before the insertion, but $f(i)$, where i is the position of the inserted item after the insertion.

Extending the results of this section to include deletions is problematic. If we charge $f(i)$ for a deletion in a list of size i, Theorem 4 holds if we allow deletions, and the total deletion cost is bounded by the total insertion cost. We leave the discovery of an alternative way to handle deletions to the ambitious reader.

4. PAGING

The model of Section 3 applies to at least one situation in which the access cost is not convex, namely paging. Consider a two-level memory divided into *pages* of fixed uniform size. Let n be the number of pages of fast memory. Each operation is an *access* that specifies a page of information. If the page is in fast memory, the access costs nothing. If the page is in slow memory, we must swap it for a page in fast memory, at a cost of one *page fault*. The goal is to minimize the number of page faults for a given sequence of accesses.

Such a two-level memory corresponds to a self-organizing list with the following access cost: $f(i) = 0$ if $i \le n$, $f(i) = 1$ if $i > n$. Since $\Delta f(i) = 0$ unless $i = n$, the items in positions 1 through n may be arbitrarily reordered for free, as may the items in positions greater than n; thus the access cost at any time depends only on the set of items in positions 1 through n. The only difference between the paging problem and the corresponding list update problem is that a page in slow memory must be moved to fast memory when accessed, whereas the corresponding list reordering is optional. To make our results easy to compare to previous work on paging, we shall, in this section, use standard paging terminology and require that each accessed page be moved to fast memory.

As with list updating, most previous work on paging [2, 5, 6, 9] is average-case analysis. Among paging rules that have been studied are the following:

Least recently used (LRU). When swapping is necessary, replace the page whose most recent access was earliest.

First-in, first-out (FIFO). Replace the page that has been in fast memory longest.

Last-in, first-out (LIFO). Replace the page most recently moved to fast memory.

Least frequently used (LFU). Replace the page that has been accessed the least.

Longest forward distance (MIN). Replace the page whose next access is latest.

All these rules use *demand paging*: They never move a page out of fast memory unless room is needed for a newly accessed page. It is well known that premature paging cannot reduce the number of page faults. Theorem 3 can be regarded as a generalization of this observation. Least recently used paging is equivalent to move-to-front; least frequently used paging corresponds to frequency count. All the paging rules except longest forward distance are on-line algorithms; that is, they require no knowledge of future accesses. Longest forward distance exactly minimizes the number of page faults [2], which is why it is known as the MIN algorithm.

We shall compare various on-line algorithms to the MIN algorithm. In making such a comparison, it is revealing to let the two algorithms have different fast memory sizes. If A is any algorithm and s is any sequence of m accesses, we denote by n_A the number of pages in A's fast memory and by $F_A(s)$ the number of page faults made by A on s. When comparing A and MIN, we shall assume that $n_A \ge n_{\text{MIN}}$. Our first result shows how poorly any on-line algorithm performs compared to MIN.

THEOREM 5.
Let A be any on-line algorithm. Then there are arbitrarily long sequences s such that

$$F_A(s) \ge (n_A/(n_A - n_{MIN} + 1))F_{MIN}(s).$$

PROOF.
We shall define a sequence of n_A accesses on which A makes n_A faults and MIN makes only $n_A - n_{\text{MIN}} + 1$ faults. The first $n_A - n_{\text{MIN}} + 1$ accesses are to pages in neither A's nor MIN's fast memory. Let S be the set of $n_A + 1$ pages either in MIN's memory initially or among the $n_A - n_{\text{MIN}} + 1$ newly accessed pages. Each of the next $n_{\text{MIN}} - 1$ accesses is to a page in S not currently in A's fast memory. On the combined sequence of n_A accesses, A faults every time. Because MIN retains all pages needed for the last $n_{\text{MIN}} - 1$ accesses, it faults only $n_A - n_{\text{MIN}} + 1$ times. This construction can be repeated as many times as desired, giving the theorem. □

REMARK.
If $n_A < n_{\text{MIN}}$, there are arbitrarily long sequences on which A always faults and MIN never faults. □

Our second result shows that the bound in Theorem 5 is tight to within an additive term for LRU.

THEOREM 6.
For any sequence s,

$$F_{LRU}(s) \le (n_{LRU}/(n_{LRU} - n_{MIN} + 1))F_{MIN}(s) + n_{MIN}.$$

PROOF.
After the first access, the fast memories of LRU and MIN always have at least one page in common; namely the page just accessed. Consider a subsequence t of s not including the first access and during which LRU faults $f \le n_{\text{LRU}}$ times. Let p be the page accessed just before t. If LRU faults on the same page twice during t, then t must contain accesses to at least $n_{\text{LRU}} + 1$ different pages. This is also true if LRU faults on p during t. If

neither of these cases occurs, then LRU faults on at least f different pages, none of them p, during t. In any case, since $f \leq n_{LRU}$, MIN must fault at least $f - n_{MIN} + 1$ times during t.

Partition s into $s_0, s_1, \ldots, s_k$ such that s_0 contains the first access and at most n_{LRU} faults by LRU, and s_i for $i = 1, \ldots, k$ contains exactly n_{LRU} faults by LRU. On each of $s_1, \ldots, s_k$, the ratio of LRU faults to MIN faults is at most $n_{LRU}/(n_{LRU} - n_{MIN} + 1)$. During s_0, if LRU faults f_0 times, MIN faults at least $f_0 - n_{MIN}$ times. This gives the theorem. □

The additive term of n_{MIN} in the bound of Theorem 6 merely reflects the fact that LRU and MIN may initially have completely different pages in fast memory, which can result in LRU faulting n_{MIN} times before MIN faults at all. If we allow LRU and MIN to have arbitrarily different initial fast memory contents, then we can increase the lower bound in Theorem 4 by n_{MIN}. On the other hand, if we assume that LRU's fast memory initially contains all the pages in MIN's fast memory and no others more recently used, then we can decrease the upper bound in Theorem 6 by n_{MIN}. In either case we get exactly matching upper and lower bounds.

Essentially the same argument shows that Theorem 6 holds for FIFO. We merely refine the definition of $s_0, s_1, \ldots, s_k$ so that during s_i for $i = 1, \ldots, k$, LRU faults exactly n_{LRU} times and also on the access just before s_i. We can find such a partition by scanning s from back to front. The rest of the proof is the same.

It is easy to construct examples that show that a result like Theorem 6 holds neither for LIFO nor for LFU. A counterexample for LIFO is a sequence of $n_{LIFO} - 1$ accesses to different pages followed by repeated alternating accesses to two new pages. On such a sequence s, $F_{LIFO}(s) = m$, but $F_{MIN}(s) = n_{LIFO} + 1$ if $n_{MIN} \geq 2$. A counterexample for LFU is a sequence of $k + 1$ accesses to each of $n_{LFU} - 1$ pages, followed by $2k$ alternating accesses to two new pages, k to each, with this pattern repeated indefinitely (so that all but the initial $n_{LFU} - 1$ pages have k accesses). On such a sequence, $F_{LFU}(s) = m - k(n_{LFU} - 1)$, but $F_{MIN}(s) \leq m/k$ if $n_{MIN} \geq 2$.

5. REMARKS.

We have studied the amortized complexity of the move-to-front rule for list updating, showing that on any sequence of operations it has a total cost within a constant factor of minimum, among all possible updating rules, including off-line ones. The constant factor is 2 if we do not count the updating cost incurred by move-to-front and 4 if we do. This result is much stronger than previous average-case results on list update heuristics. Neither transpose nor frequency count shares this approximate optimality. Thus, even if one is willing to incur the time and space overhead needed to maintain frequency counts, it may not be a good idea.

Our results lend theoretical support to Bentley and McGeoch's experiments showing that move-to-front is generally the best rule in practice. As Bentley and McGeoch note, this contradicts the asymptotic average-case results, which favor transpose over move-to-front. Our tentative conclusion is that amortized complexity provides not only a more robust but a more realistic measure for list update rules than asymptotic average-case complexity.

We have generalized our result on move-to-front to any situation in which the access cost is convex. We have also studied paging, a setting with a nonconvex access cost. Our results for paging can be interpreted either positively or negatively. On the one hand, any on-line paging algorithm makes a number of faults in the worst case that exceeds the minimum by a factor equal to the size of fast memory. On the other hand, even in the worst case both LRU and FIFO paging come within a constant factor of the number of page faults made by the optimum algorithm with a constant factor smaller fast memory. More precisely, for any constant factor $c > 1$, LRU and FIFO with fast memory size n make at most c times as many faults as the optimum algorithm with fast memory size $(1 - 1/c)n$. A similar result for LRU was proved by Franaszek and Wagner [6], but only on the average.

REFERENCES

1. Anderson, E.J., Nash, P., and Weber, R.R. A counterexample to a conjecture on optimal list ordering. *J. Appl. Prob.*, to appear.
2. Belady, L.A. A study of replacement algorithms for virtual storage computers. *IBM Syst. J. 5* (1966), 78–101.
3. Bentley, J.L., and McGeoch, C. Worst-case analysis of self-organizing sequential search heuristics. In *Proceedings of 20th Allerton Conference on Communication, Control, and Computing*, (Univ. Illinois, Urbana-Champaign, Oct. 6–8, 1982), 1983, 452–461.
4. Bitner, J.R. Heuristics that dynamically organize data structures. *SIAM J. Comput. 8* (1979), 82–110.
5. Coffman, E.G., and Denning, P.J. *Operating Systems Theory*, Prentice-Hall, Englewood Cliffs, NJ, 1973.
6. Franaszek, P.A., and Wagner, T.J. Some distribution-free aspects of paging performance. *J. ACM 21*, 1 (Jan. 1974), 31–39.
7. Knuth, D.E. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, Reading, MA, 1973.
8. Rivest, R. On self-organizing sequential search heuristics. *Commun. ACM 19*, 2 (Feb. 1976), 63–67.
9. Spirn, J.R. *Program Behavior: Models and Measurements*. Elsevier, New York, 1977.

CR Categories and Subject Descriptors: D.4.2 [**Operating Systems**]: Storage Management—*swapping*; E.1 [**Data**]: Data Structures—*lists*; *tables*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*sorting and searching*

General Terms: Algorithms, Theory

Additional Key Words and Phrases: self-organizing, move-to-front, least recently used, paging.

Received 3/83; revised 6/83; accepted 8/84

Authors' Present Addresses: Daniel D. Sleator, Computing Science Research Center, AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974. Robert E. Tarjan, Mathematics and Statistics Research Center, AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974.