A Survey of Self-Organizing Data Structures[⋆]

Susanne Albers[⋆⋆] and Jeffery Westbrook[⋆⋆⋆]

1 Introduction

This paper surveys results in the design and analysis of self-organizing data structures for the search problem. The general search problem in pointer data structures can be phrased as follows. The elements of a set are stored in a collection of nodes. Each node also contains $O(1)$ pointers to other nodes and additional state data which can be used for navigation and self-organization. The elements have associated key values, which may or may not be totally ordered (almost always they are). Various operations may be performed on the set, including the standard dictionary operations of searching for an element, inserting a new element, and deleting an element. Additional operations such as set splitting or joining may be allowed. This survey considers two simple but very popular data structures: the unsorted linear list, and the binary search tree.

A self-organizing data structure has a rule or algorithm for changing pointers and state data after each operation. The self-organizing rule is designed to

[⋆] This survey will appear in a book on on-line algorithms edited by Amos Fiat and Gerhard Woeginger.

[⋆⋆] Max-Planck-Institut für Informatik, Im Stadtwald, 66123 Saarbrücken, Germany. E-mail: albers@mpi-sb.mpg.de.

[⋆⋆⋆] Department of Computer Science, Yale University, New Haven, CT 06520-2158, and AT&T Labs - Research, Murray Hill, NJ 07974. E-mail: jeffw@research.att.com.

respond to initially unknown properties of the input request sequence, and to get the data structure into a state that will take advantage of these properties and reduce the time per operation. As operations occur, a self-organizing data structure may change its state quite dramatically.

Self-organizing data structures can be compared to static or constrained data structures. The state of a static data structure is predetermined by some strong knowledge about the properties of the input. For example, if searches are generated according to some known probability distribution, then a linear list may sorted by decreasing probability of access. A constrained data structure must satisfy some structural invariant, such as a balance constraint in a binary search tree. As long as the structural invariant is satisfied, the data structure does not change.

Self-organizing data structures have several advantages over static and constrained data structures [65]. (a) The amortized asymptotic time of search and update operations is usually as good as the corresponding time of constrained structures. But when the sequence of operations has favorable properties, the performance can be much better. (b) Self-organizing rules need no knowledge of the properties of input sequence, but will adapt the data structure to best suit the input. (c) The self-organizing rule typically results in search and update algorithms that are simple and easy to implement. (d) Often the self-organizing rule can be implemented without any using any extra space in the nodes. (Such a rule is called "memoryless" since it saves no information to help make its decisions.)

On the other hand, self-organizing data structures have several disadvantages. (a) Although the total time of a sequence of operations is low, an individual operation can be quite expensive. (b) Reorganization of the structure has to be done even during search operations. Hence self-organizing data structures may have higher overheads than their static or constraint-based cousins.

Nevertheless, self-organizing data structures represent an attractive alternative to constraint structures, and reorganization rules have been studied extensively for both linear lists and binary trees. Both data structures have also received considerable attention within the study of on-line algorithms.

2 Unsorted linear lists

The problem of representing a dictionary as an unsorted linear list is also known as the *list update problem*. Consider a set S of items that has to be maintained under a sequence of *requests*, where each request is one of the following operations.

Access(x)**.** Locate item x in S.
Insert(x)**.** Insert item x into S.
Delete(x)**.** Delete item x from S.

Given that S shall be represented as an unsorted list, these operations can be implemented as follows. To access an item, a list update algorithm starts at the front of the list and searches linearly through the items until the desired item is found. To insert a new item, the algorithm first scans the entire list to verify that the item is not already present and then inserts the item at the end of the list. To delete an item, the algorithm scans the list to search for the item and then deletes it.

In serving requests a list update algorithm incurs cost. If a request is an access or a delete operation, then the incurred cost is i, where i is the position of the requested item in the list. If the request is an insertion, then the cost is $n+1$, where n is the number of items in the list before the insertion. While processing a request sequence, a list update algorithm may rearrange the list. Immediately after an access or insertion, the requested item may be moved at no extra cost to any position closer to the front of the list. These exchanges are called *free exchanges*. Using free exchanges, the algorithm can lower the cost on subsequent requests. At any time two adjacent items in the list may be exchanged at a cost of 1. These exchanges are called *paid exchanges*.

The cost model defined above is called the *standard model*. Manasse *et al.* [53] and Reingold *et al.* [60] introduced the P^d cost model. In the P^d model there are no free exchanges and each paid exchange costs d. In this survey, we will present results both for the standard and the P^d model. However, unless otherwise stated, we will always assume the standard cost model.

We are interested in list update algorithms that serve a request sequence so that the total cost incurred on the entire sequence is as small as possible. Of particular interest are *on-line* algorithms, i.e., algorithms that serve each request without knowledge of any future requests. In [65], Sleator and Tarjan suggested to compare the quality of an on-line algorithm to that of an *optimal off-line* algorithm. An optimal off-line algorithm knows the entire request sequence in advance and can serve it with minimum cost. Given a request sequence σ, let $C_A(\sigma)$ denote the cost incurred by an on-line algorithm A in serving σ, and let $C_{OPT}(\sigma)$ denote the cost incurred by an optimal off-line algorithm OPT. Then the on-line algorithm A is called c-competitive if there is a constant a such that for all size lists and all request sequences σ,

$$C_A(\sigma) \leq c \cdot C_{OPT}(\sigma) + a.$$

The factor c is also called the *competitive ratio*. Here we assume that A is a deterministic algorithm. The competitive ratio of a randomized on-line algorithm has to be defined in a more careful way, see Section 2.2. In Sections 2.1 and 2.2 we will present results on the competitiveness that can be achieved by deterministic and randomized on-line algorithms.

It is worthwhile to note that there is no algorithm known that computes in polynomial time the optimal way to process a given request sequence. Reingold and Westbrook [61] gave an optimal off-line algorithm that runs in time $O(2^n n! m)$, where n is the size of the list and m is the length of the request sequence.

Linear lists are one possibility to represent a dictionary. Certainly, there are other data structures such as balanced search trees or hash tables that, depending on the given application, can maintain a dictionary in a more efficient way. In general, linear lists are useful when the dictionary is small and consists of only a few dozen items [14]. Furthermore, list update algorithms have been used as subroutines in algorithms for computing point maxima and convex hulls [13, 31]. Recently, list update techniques have been very successfully applied in the development of data compression algorithms [17]. We discuss this application in detail in Section 4.

2.1 Deterministic on-line algorithms

There are three well-known deterministic on-line algorithms for the list update problem.

- **Move-To-Front**: Move the requested item to the front of the list.
- **Transpose**: Exchange the requested item with the immediately preceding item in the list.
- **Frequency-Count**: Maintain a frequency count for each item in the list. Whenever an item is requested, increase its count by 1. Maintain the list so that the items always occur in nonincreasing order of frequency count.

Other deterministic on-line algorithms that have been proposed in the literature are variants of the above algorithms, see [16, 32, 34, 42, 47, 62, 63, 65, 75]. Rivest [62], for instance, introduced a move-ahead-k heuristic that moves a requested item k positions ahead. Gonnet *et al.* [32] and Kan and Ross [41] considered a k-in-a-row rule, where an item is only moved after it is requested k times in a row. This strategy can be combined both with the Move-To-Front and Transpose algorithms.

The formulations of list update algorithms generally assume that a request sequence consists of accesses only. It is obvious how to extend the algorithms so that they can also handle insertions and deletions. On an insertion, the algorithm first appends the new item at the end of the list and then executes the same steps as if the item was requested for the first time. On a deletion, the algorithm first searches for the item and then just removes it.

In the following, we concentrate on the three algorithms Move-To-Front, Transpose and Frequency-Count. We note that Move-To-Front and Transpose are *memoryless* strategies, i.e., they do not need any extra memory to decide where a requested item should be moved. Thus, from a practical point of view, they are more attractive than Frequency-Count. Sleator and Tarjan [65] analyzed the competitive ratios of the three algorithms.

Theorem 1. *The Move-To-Front algorithm is 2-competitive.*

Proof. Consider a request sequence $\sigma = \sigma(1), \sigma(2), \ldots, \sigma(m)$ of length m. First suppose that σ consists of accesses only. We will compare simultaneous runs of Move-To-Front and OPT on σ and evaluate on-line and off-line cost using a potential function Φ. For an introduction to amortized analysis using potential functions, see Tarjan [72].

The potential function we use is the number of inversions in Move-To-Front's list with respect to OPT's list. An *inversion* is a pair x, y of items such that x occurs before y Move-To-Front's list and after y in OPT's list. We assume without loss of generality that Move-To-Front and OPT start with the same list so that the initial potential is 0.

For any t, $1 \leq t \leq m$, let $C_{MTF}(t)$ and $C_{OPT}(t)$ denote the actual cost incurred by Move-To-Front and OPT in serving $\sigma(t)$. Furthermore, let $\Phi(t)$ denote the potential after $\sigma(t)$ is served. The *amortized cost* incurred by Move-To-Front on $\sigma(t)$ is defined as $C_{MTF}(t) + \Phi(t) - \Phi(t-1)$. We will show that for any t,

$$C_{MTF}(t) + \Phi(t) - \Phi(t-1) \leq 2C_{OPT}(t) - 1. \tag{1}$$

Summing this expression for all t we obtain $\sum_{t=1}^{m} C_{MTF}(t) + \Phi(m) - \Phi(0) \leq \sum_{t=1}^{m} 2C_{OPT}(t) - m$, i.e., $C_{MTF}(\sigma) \leq 2C_{OPT}(\sigma) - m + \Phi(0) - \Phi(m)$. Since the initial potential is 0 and the final potential is non-negative, the theorem follows.

In the following we will show inequality (1) for an arbitrary t. Let x be the item requested by $\sigma(t)$. Let k denote the number of items that precede x in Move-To-Front's and OPT's list. Furthermore, let l denote the number of items that precede x in Move-To-Front's list but follow x in OPT's list. We have $C_{MTF}(t) = k + l + 1$ and $C_{OPT}(t) \geq k + 1$.

When Move-To-Front serves $\sigma(t)$ and moves x to the front of the list, l inversions are destroyed and at most k new inversions are created. Thus

$$\begin{aligned} C_{MTF}(t) + \Phi(t) - \Phi(t-1) &\leq C_{MTF}(t) + k - l \ = \ 2k + 1 \\ &\leq 2C_{OPT}(t) - 1. \end{aligned}$$

Any paid exchange made by OPT when serving $\sigma(t)$ can increase the potential by 1, but OPT also pays 1. We conclude that inequality (1) holds.

The above arguments can be extended easily to analyze an insertion or deletion. On an insertion, $C_{MTF}(t) = C_{OPT}(t) = n + 1$, where n is the number of items in the list before the insertion, and at most n new inversions are created. On a deletion, l inversions are removed and no new inversion is created. □

Bentley and McGeoch [14] proved a weaker version of Theorem 1. They showed that on any sequence of accesses, the cost incurred by Move-To-Front is at most twice the cost of the *optimum static off-line algorithm*. The optimum static off-line algorithm first arranges the items in order of decreasing request frequencies and does no further exchanges while serving the request sequence.

The proof of Theorem 1 shows that Move-To-Front is $(2 - \frac{1}{n})$-competitive, where n is the maximum number of items ever contained in the dictionary. Irani [38] gave a refined analysis of the Move-To-Front rule and proved that it is $(2 - \frac{2}{n+1})$-competitive.

Sleator and Tarjan [65] showed that, in terms of competitiveness, Move-To-Front is superior to Transpose and Frequency-Count.

Proposition 2. *The algorithms Transpose and Frequency-Count are not c-competitive for any constant c.*

Recently, Albers [3] presented another deterministic on-line algorithm for the list update problem. The algorithm belongs to the Timestamp(p) family of algorithms that were introduced in the context of randomized on-line algorithms and that are defined for any real number $p \in [0,1]$. For $p = 0$, the algorithm is deterministic and can be formulated as follows.

Algorithm Timestamp(0): Insert the requested item, say x, in front of the first item in the list that precedes x and that has been requested at most once since the last request to x. If there is no such item or if x has not been requested so far, then leave the position of x unchanged

Theorem 3. *The Timestamp(0) algorithm is 2-competitive.*

Note that Timestamp(0) is not memoryless. We need information on past requests in order to determine where a requested item should be moved. In fact, in the most straightforward implementation of the algorithm we need a second pass through the list to find the position where the accessed item must be inserted. Timestamp(0) is interesting because it has a better overall performance than Move-To-Front. The algorithm achieves a competitive ratio of 2, as does Move-To-Front. However, as we shall see in Section 2.3, Timestamp(0) is considerably better than Move-To-Front on request sequences that are generated by probability distributions.

El-Yaniv [29] recently presented a new family of deterministic on-line algorithms for the list update problem. This family also contains the algorithms Move-To-Front and Timestamp(0).

Karp and Raghavan [42] developed a lower bound on the competitiveness that can be achieved by deterministic on-line algorithms. This lower bound implies that Move-To-Front and Timestamp(0) have an optimal competitive ratio.

Theorem 4. *Let A be a deterministic on-line algorithm for the list update algorithm. If A is c-competitive, then $c \geq 2$.*

Proof. Consider a list of n items. We construct a request sequence that consist of accesses only. Each request is made to the item that is stored at the last position in A's list. On a request sequence σ of length m generated in this way, A incurs a cost of $C_A(\sigma) = mn$. Let OPT' be the optimum static off-line algorithm. OPT' first sorts the items in the list in order of nonincreasing request frequencies and then serves σ without making any further exchanges. When rearranging the list, OPT' incurs a cost of at most $n(n-1)/2$. Then the requests in σ can be served at a cost of at most $m(n+1)/2$. Thus $C_{OPT}(\sigma) \leq m(n+1)/2 + n(n-1)/2$. For long request sequences, the additive term of $n(n-1)/2$ can be neglected and we obtain

$$C_A(\sigma) \geq \tfrac{2n}{n+1} \cdot C_{OPT}(\sigma).$$

The theorem follows because the competitive ratio must hold for all list lengths. □

The proof shows that the lower bound is actually $2 - \frac{2}{n+1}$, where n is the number of items in the list. Thus, the upper bound given by Irani on the competitive ratio of the Move-To-Front rule is tight.

Next we consider list update algorithms for other cost models. Reingold *et al.* [60] gave a lower bound on the competitiveness achieved by deterministic on-line algorithms.

Theorem 5. *Let A be a deterministic on-line algorithm for the list update algorithm in the P^d model. If A is c-competitive, then $c \geq 3$.*

Below we will give a family of deterministic algorithms for the P^d model. The best algorithm in this family achieves a competitive ratio that is approximately 4.56-competitive. We defer presenting this result until the discussion of randomized algorithms for the P^d model, see Section 2.2.2.

Sleator and Tarjan considered another generalized cost model. Let f be a nondecreasing function from the positive integers to the nonnegative reals. Suppose that an access to the i-th item in the list costs $f(i)$ and that an insertion costs $f(n+1)$, where n is the number of items in the list before the insertion. Let the cost of a paid exchange if items i and $i+1$ be $\Delta f(i) = f(i+1) - f(i)$. The function f is *convex* if $\Delta f(i) \geq \Delta f(i+1)$ for all i. Sleator and Tarjan [65] analyzed the Move-To-Front algorithm for convex cost functions. As usual, n denotes the maximum number of items contained in the dictionary.

Theorem 6. *If f is convex, then*

$$C_{MTF}(\sigma) \leq 2 \cdot C_{OPT}(\sigma) + \textstyle\sum_{i=1}^{n-1}(f(n) - f(i))$$

for all request sequences σ that consist only of accesses and insertions.

The term $\sum_{i=1}^{n-1}(f(n) - f(i))$ accounts for the fact that the initial lists given to Move-To-Front and OPT may be different. If the lists are the same, the term can be omitted in the inequality. Theorem 6 can be extended to request sequences that include deletions if the total cost for deletions does not exceed the total cost incurred for insertions. Here we assume that a deletion of the i-th item in the list costs $f(i)$.

2.2 Randomized on-line algorithms

The competitiveness of a randomized on-line algorithm is defined with respect to an adversary. Ben-David *et al.* [12] introduced three kinds of adversaries. They differ in the way a request sequence is generated and how the adversary is charged for serving the sequence.

- **Oblivious Adversary**: The oblivious adversary has to generate a complete request sequence in advance, before any requests are served by the on-line algorithm. The adversary is charged the cost of the optimum off-line algorithm for that sequence.

- **Adaptive On-line Adversary**: This adversary may observe the on-line algorithm and generate the next request based on the algorithm's (randomized) answers to all previous requests. The adversary must serve each request on-line, i.e., without knowing the random choices made by the on-line algorithm on the present or any future request.
- **Adaptive Off-line Adversary**: This adversary also generates a request sequence adaptively. However, it is charged the optimum off-line cost for that sequence.

A randomized on-line algorithm A is called c-competitive against any oblivious adversary if there is a constant a such that for all size lists and all request sequences σ generated by an oblivious adversary, $E[C_A(\sigma)] \leq c \cdot C_{OPT}(\sigma) + a$. The expectation is taken over the random choices made by A.

Given a randomized on-line algorithm A and an adaptive on-line (adaptive off-line) adversary ADV, let $E[C_A]$ and $E[C_{ADV}]$ denote the expected costs incurred by A and ADV in serving a request sequence generated by ADV. A randomized on-line algorithm A is called c-competitive against any adaptive on-line (adaptive off-line) adversary if there is a constant a such that for all size lists and all adaptive on-line (adaptive off-line) adversaries ADV, $E[C_A] \leq c \cdot E[C_{ADV}] + a$, where the expectation is taken over the random choices made by A.

Ben-David *et al.* [12] investigated the relative strength of the adversaries with respect to an arbitrary on-line problem. They showed that if there is a randomized on-line algorithm that is c-competitive against any adaptive off-line adversary, then there is also a c-competitive deterministic on-line algorithm. This immediately implies that no randomized on-line algorithm for the list update problem can be better then 2-competitive against any adaptive off-line adversary. Reingold *et al.* [60] proved a similar result for adaptive on-line adversaries.

Theorem 7. *If a randomized on-line algorithm for the list update problem is c-competitive against any adaptive on-line adversary, then $c \geq 2$.*

The optimal competitive ratio that can be achieved by randomized on-line algorithms against oblivious adversaries has not been determined yet. In the following we present upper and lower bounds known on this ratio.

2.2.1 Randomized on-line algorithms against oblivious adversaries

The first randomized on-line algorithm for the list update problem was presented by Irani [38, 39] and is called Split algorithm. Each item x in the list maintains a pointer $x.split$ that points to some other item in the list. The pointer of each item either points to the item itself or to an item that precedes it in the list.

Algorithm Split: The algorithm works as follows.
Initialization:

For all items x in the list, set $x.split \leftarrow x$.

If item x is requested:

```
    For all items $y$ with $y.split = x$, set $y.split \leftarrow$ item behind $x$ in the list.
    With probability 1/2:
        Move $x$ to the front of the list.
    With probability 1/2:
        Insert $x$ before item $x.split$.
        If $y$ preceded $x$ and $x.split = y.split$, then set $y.split \leftarrow x$.
    Set $x.split$ to the first item in the list.
```

Theorem 8. *The Split algorithm is* $(15/8)$*-competitive against any oblivious adversary.*

We note that $(15/8) = 1.875$. Irani [38, 39] showed that the Split algorithm is not better than 1.75-competitive in the $i-1$ cost model. In the $i-1$ cost model, an access to the i-th item in the list costs $i-1$ rather than i.

A simple and easily implementable list update rule was proposed by Reingold *et al.* [60].

Algorithm Bit: Each item in the list maintains a bit that is complemented whenever the item is accessed. If an access causes a bit to change to 1, then the requested item is moved to the front of the list. Otherwise the list remains unchanged. The bits of the items are initialized independently and uniformly at random.

Theorem 9. *The Bit algorithm is 1.75-competitive against any oblivious adversary.*

Reingold *et al.* analyzed Bit using an elegant modification of the potential function given in the proof of Theorem 1. Again, an inversion is a pair of items x, y such that x occurs before y in Bit's list and after y in OPT's list. An inversion has *type 1* if y's bit is 0 and *type 2* if y's bit is 1. Now, the potential is defined as the number of type 1 inversions plus twice the number of type 2 inversions.

The upper bound for Bit is tight in the $i-1$ cost model [60]. It was also shown that in the i cost model, i.e. in the standard model, Bit is not better than 1.625-competitive [4].

Reingold *et al.* [60] gave a generalization of the Bit algorithm. Let l be a positive integer and L be a non-empty subset of $\{0.1. \ldots, l-1\}$. The algorithm Counter(l, L) works as follows. Each item in the list maintains a mod l counter. Whenever an item x is accessed, the counter of x is decremented by 1 and, if the new value is in L, the item x is moved to the front of the list. The counters of the items are initialized independently and uniformly at random to some value in $\{0.1. \ldots, l-1\}$. Note that Bit is Counter(2,$\{1\}$). Reingold *et al.* chose parameters l and L so that the resulting Counter(l, L) algorithm is better than 1.75-competitive. It is worthwhile to note that the algorithms Bit and Counter(l, L) make random choices only during an initialization phase and run completely deterministically thereafter.

The Counter algorithms can be modified [60]. Consider a Counter$(l, \{0\})$ algorithm that is changed as follows. Whenever the counter of an item reaches

0, the counter is reset to j with probability p_j, $1 \leq j \leq l-1$. Reingold *et al.* [60] gave a value for l and a resetting distribution so that the algorithm achieves a competitive ratio of $\sqrt{3} \approx 1.73$.

Another family of randomized on-line algorithms was given by Albers [3]. The following algorithm works for any real number $p \in [0.1]$.

Algorithm Timestamp(p): Each request to an item, say x, is served as follows. With probability p execute Step (a).

(a) Move x to the front of the list.

With probability $1-p$ execute Step (b).

(b) Insert x in front of the first item in the list that precedes x and

(i) that was not requested since the last request to x

or

(ii) that was requested exactly once since the last request to x and the corresponding request was served using Step (b) of the algorithm.

If there is no such item or if x is requested for the first time, then leave the position of x unchanged.

Theorem 10. *For any real number $p \in [0,1]$, the algorithm Timestamp(p) is c-competitive against any oblivious adversary, where $c = \max\{2-p, 1+p(2-p)\}$.*

Setting $p = (3-\sqrt{5})/2$, we obtain a Φ-competitive algorithm, where $\Phi = (1+\sqrt{5})/2 \approx 1.62$ is the Golden Ratio. The family of Timestamp algorithms also includes two deterministic algorithms. For $p = 1$, we obtain the Move-To-Front rule. On the other hand, setting $p = 0$, we obtain the Timestamp(0) algorithm that was already described in Section 2.1.

In order to implement Timestamp(p) we have to maintain, for each item in the list, the times of the two last requests to that item. If these two times are stored with item, then after each access the algorithm needs a second pass through the list to find the position where the requested item should be inserted. Note that such a second pass is also needed by the Split algorithm. In the case of the Split algorithm, this second pass is necessary because pointers have to be updated.

Interestingly, it is possible to combine the algorithms Bit and Timestamp(0), see Albers *et al.* [6]. This combined algorithm achieves the best competitive ratio that is currently known for the list update problem.

Algorithm Combination: With probability 4/5 the algorithm serves a request sequence using Bit, and with probability 1/5 it serves a request sequence using Timestamp(0).

Theorem 11. *The algorithm Combination is 1.6-competitive against any oblivious adversary.*

Proof. The analysis consists of two parts. In the first part we show that given any request sequence σ, the cost incurred by Combination and OPT can be divided into costs that are caused by each unordered pair $\{x, y\}$ of items x and y. Then, in the second part, we compare on-line and off-line cost for each pair $\{x, y\}$. This method of analyzing cost by considering pairs of items was first introduced by Bentley and McGeoch [14] and later used in [3, 38]. In the following we always assume that serving a request to the i-th item in the list incurs a cost of $i-1$ rather than i. Clearly, if Combination is 1.6-competitive in this $i-1$ cost model, it is also 1.6-competitive in the i-cost model.

Let $\sigma = \sigma(1), \sigma(2), \ldots, \sigma(m)$ be an arbitrary request sequence of length m. For the reduction to pairs we need some notation. Let S be the set of items in the list. Consider any list update algorithm A that processes σ. For any $t \in [1, m]$ and any item $x \in S$, let $C_A(t, x)$ be the cost incurred by item x when A serves $\sigma(t)$. More precisely, $C_A(t, x) = 1$ if item x precedes the item requested by $\sigma(t)$ in A's list at time t; otherwise $C_A(t, x) = 0$. If A does not use paid exchanges, then the total cost $C_A(\sigma)$ incurred by A on σ can be written as

$$\begin{aligned} C_A(\sigma) &= \sum_{t\in[1,m]} \sum_{x\in S} C_A(t,x) = \sum_{x\in S} \sum_{t\in[1,m]} C_A(t,x) \\ &= \sum_{x\in S} \sum_{y\in S} \sum_{\substack{t\in[1,m]\\ \sigma(t)=y}} C_A(t,x) \\ &= \sum_{\substack{\{x,y\}\\ x\neq y}} \Big(\sum_{\substack{t\in[1,m]\\ \sigma(t)=x}} C_A(t,y) + \sum_{\substack{t\in[1,m]\\ \sigma(t)=y}} C_A(t,x)\Big). \end{aligned}$$

For any unordered pair $\{x, y\}$ of items $x \neq y$, let σ_{xy} be the request sequence that is obtained from σ if we delete all requests that are neither to x nor to y. Let $C_{BIT}(\sigma_{xy})$ and $C_{TS}(\sigma_{xy})$ denote the costs that Bit and Timestamp(0) incur in serving σ_{xy} on a two item list that consist of only x and y. Obviously, if Bit serves σ on the long list, then the relative position of x and y changes in the same way as if Bit serves σ_{xy} on the two item list. The same property holds for Timestamp(0). This follows from Lemma 12, which can easily be shown by induction on the number of requests processed so far.

Lemma 12. *At any time during the processing of σ, x precedes y in Timestamp(0)'s list if and only if one of the following statements holds: (a) the last requests made to x and y are of the form xx, xyx or xxy; (b) x preceded y initially and y was requested at most once so far.*

Thus, for algorithm $A \in \{\text{Bit, Timestamp(0)}\}$ we have

$$C_A(\sigma_{xy}) = \sum_{\substack{t\in[1,m]\\ \sigma(t)=x}} C_A(t,y) + \sum_{\substack{t\in[1,m]\\ \sigma(t)=y}} C_A(t,x)$$

$$C_A(\sigma) = \sum_{\substack{\{x,y\}\\ x\neq y}} C_A(\sigma_{xy}). \tag{2}$$

Note that Bit and Timestamp(0) do not incur paid exchanges. For the optimal off-line cost we have

$$C_{OPT}(\sigma_{xy}) \le \sum_{\substack{t\in[1,m]\\ \sigma(t)=x}} C_{OPT}(t,y) + \sum_{\substack{t\in[1,m]\\ \sigma(t)=y}} C_{OPT}(t,x) \; + \; p(x,y)$$

and

$$C_{OPT}(\sigma) \geq \sum_{\substack{\{x,y\}\\ x\neq y}} C_{OPT}(\sigma_{xy}), \tag{3}$$

where $p(x,y)$ denotes the number of paid exchanges incurred by OPT in moving x in front of y or y in front of x. Here, only inequality signs hold because if OPT serves σ_{xy} on the two item list, then it can always arrange x and y optimally in the list, which might not be possible if OPT serves σ on the entire list. Note that the expected cost $E[C_{CB}(\sigma_{xy})]$ incurred by Combination on σ_{xy} is

$$E[C_{CB}(\sigma_{xy})] = \frac{4}{5}E[C_{BIT}(\sigma_{xy})] + \frac{1}{5}E[C_{TS}(\sigma_{xy})]. \tag{4}$$

In the following we will show that for any pair $\{x,y\}$ of items $E[C_{CB}(\sigma_{xy})] \leq 1.6C_{OPT}(\sigma_{xy})$. Summing this inequality for all pairs $\{x,y\}$, we obtain, by equations (2),(3) and (4), that Combination is 1.6-competitive.

Consider a fixed pair $\{x,y\}$ with $x \neq y$. We partition the request sequence σ_{xy} into phase. The first phase starts with the first request in σ_{xy} and ends when, for the first time, there are two requests to the same item and the next request is different. The second phase starts with that next request and ends in the same way as the first phase. The third and all remaining phases are constructed in the same way as the second phase. The phases we obtain are of the following types: x^k for some $k \geq 2$; $(xy)^k x^l$ for some $k \geq 1, l \geq 2$; $(xy)^k y^l$ for some $k \geq 1$, $l \geq 1$. Symmetrically, we have y^k, $(yx)^k y^l$ and $(yx)^k x^l$.

Since a phase ends with (at least) two requests to the same item, the item requested last in the phase precedes the other item in the two item list maintained by Bit and Timestamp(0). Thus the item requested first in a phase is always second in the list. Without loss of generality we can assume the same holds for OPT, because when OPT serves two consecutive requests to the same item, it cannot cost more to move that item to the front of the two item list after the first request. The expected cost incurred by Bit, Timestamp(0) (denoted by TS(0)) and OPT are given in the table below. The symmetric phases with x and y interchanged are omitted. We assume without generality that at the beginning of σ_{xy}, y precedes x in the list.

Phase	Bit	$TS(0)$	OPT
x^k	$\frac{3}{2}$	2	1
$(xy)^k x^l$	$\frac{3}{2}k+1$	$2k$	$k+1$
$(xy)^k y^l$	$\frac{3}{2}k+\frac{1}{4}$	$2k-1$	k

The entries for OPT are obvious. When Timestamp(0) serves a phase $(xy)^k x^l$, then the first two request xy incur a cost of 1 and 0, respectively, because x is left behind y on the first request to x. On all subsequent requests in the phase, the requested item is always moved to the front of the list. Therefore, the total cost on the phase is $1 + 0 + 2(k-1) + 1 = 2k$. Similarly, Timestamp(0) serves $(xy)^k y^l$ with cost $2k-1$.

For the analysis of Bit's cost we need two lemmata.

Lemma 13. *For any item x and any $t \in [1, m]$, after the t-th request in σ, the value of x's bit is equally likely to be 0 or 1, and the value is independent of the bits of the other items.*

Lemma 14. *Suppose that Bit has served three consecutive requests yxy in σ_{xy}, or two consecutive requests xy where initially y preceded x. Then y is in front of x with probability $\frac{3}{4}$. The analogous statement holds when the roles of x and y are interchanged.*

Clearly, the expected cost spent by Bit on a phase x^k is $1 + \frac{1}{2} + 0(k-2) = \frac{3}{2}$. Consider a phase $(xy)^k x^l$. The first two requests xy incur a expected cost of 1 and $\frac{1}{2}$, respectively. By Lemma 14, each remaining request in the string $(xy)^k$ and the first request in x^l have an expected cost of $\frac{3}{4}$. Also by Lemma 14, the second request in x^l costs $1 - \frac{3}{4} = \frac{1}{4}$. All other requests in x^l are free. Therefore, Bit pays an expected cost of $1 + \frac{1}{2} + \frac{3}{2}(k-1) + \frac{3}{4} + \frac{1}{4} = \frac{3}{2}k + 1$ on the phase. Similarly, we can evaluate a phase $(xy)^k y^l$.

The Combination algorithm serves a request sequence with probability $\frac{4}{5}$ using Bit and with probability $\frac{1}{5}$ using Timestamp(0). Thus, by the above table, Combination has an expected cost of 1.6 on a phase x^k, a cost of $1.6k + 0.8$ on a phase $(xy)^k x^l$, and a cost $1.6k$ on a phase $(xy)^k y^l$. In each case this is at most 1.6 times the cost of OPT.

In the above proof we assume that a request sequence consist of accesses only. However, the analysis is easily extended to the case that insertions and deletions occur, too. For any item x, consider the time intervals during which x is contained in the list. For each of these intervals, we analyze the cost caused by any pair $\{x, y\}$, where y is an item that is (temporarily) present during the interval. □

Teia [70] presented a lower bound for randomized list update algorithms.

Theorem 15. *Let A be a randomized on-line algorithm for the list update problem. If A is c-competitive against any oblivious adversary, then $c \geq 1.5$.*

An interesting open problem is to give tight bounds on the competitive ratio that can be achieved by randomized on-line algorithms against oblivious adversaries.

2.2.2 Results in the P^d cost model

As mentioned in Theorem 5, no deterministic on-line algorithm for the list update problem in the P^d model can be better than 3-competitive. By a result of Ben-David *et al.* [12], this implies that no randomized on-line algorithm for the list update problem in the P^d model can be better than 3-competitive against any adaptive off-line adversary. Reingold *et al.* [60] showed that the same bound holds against adaptive on-line adversaries.

Theorem 16. *Let A be a randomized on-line algorithm for the list update problem in the P^d model. If A is c-competitive against any adaptive on-line adversary, then $c \geq 3$.*

Reingold *et al.* [60] analyzed the Counter($l, \{l-1\}$) algorithms, l being a positive integer, for list update in the P^d model. As described in Section 2.2.1, these algorithms work as follows. Each item maintains a mod l counter that is decremented whenever the item is requested. When the value of the counter changes to $l-1$, then the accessed item is moved to the front of the list. In the P^d cost model, this movement is done using paid exchanges. The counters are initialized independently and uniformly at random to some value in $\{0, 1, \ldots, l-1\}$.

Theorem 17. *In the P^d model, the algorithm Counter($l, \{l-1\}$) is c-competitive against any oblivious adversary, where $c = \max\{1 + \frac{l+1}{2d}, 1 + \frac{1}{l}(2d + \frac{l+1}{2})\}$.*

The best value for l depends on d. As d goes to infinity, the best competitive ratio achieved by a Counter($l, \{l-1\}$) algorithm decreases and goes to $(5+\sqrt{17})/4 \approx 2.28$.

We now present an analysis of the deterministic version of the Counter($l, \{l-1\}$) algorithm. The deterministic version is the same as the randomized version, except that all counters are initialized to zero, rather than being randomly initialized.

Theorem 18. *In the P^d model, the deterministic algorithm Counter($l, \{l-1\}$) is c-competitive, where $c = \max\{3 + \frac{2l}{d}, 2 + \frac{2d}{l}\}$.*

Proof. The analysis is similar in form to that of Combination. Consider a pair of items $\{x, y\}$. Let $c(x)$ and $c(y)$ denote the values of the counters at items x and y, respectively. We define a potential function Φ. Assume w.l.o.g. that OPT's list is ordered (x, y). Then

$$\Phi = \begin{cases} (1 + 2d/l)c(y) & \text{if Counter's list is ordered } (x, y) \\ k + d - c(x) + (1 + 2d/l)c(y) & \text{if Counter's list is ordered } (y, x) \end{cases}$$

The remainder of the proof follows by case analysis. For each event in each configuration, we compare the amortized cost incurred by Counter to the actual cost incurred by OPT. (See the proof of competitiveness of MTF.) □

As in the randomized case, the optimum value of l for the deterministic Counter algorithm depends on d. As d goes to infinity, the best competitive ratio decreases and goes to $(5 + \sqrt{17})/2 \approx 4.56$, exactly twice the best randomized value.

2.3 Average case analyses of list update algorithms

In this section we study a restricted class of request sequences: request sequences that are generated by a probability distribution. Consider a list of n items $x_1, x_2, \ldots, x_n$, and let $\mathbf{p} = (p_1, p_2, \ldots, p_n)$ be a vector of positive probabilities p_i with $\sum_{i=1}^{n} p_i = 1$. We study request sequences that consist of accesses only, where each request it made to item x_i with probability p_i, $1 \leq i \leq n$. It is convenient to assume that $p_1 \geq p_2 \geq \cdots \geq p_n$.

There are many results known on the performance of list update algorithms when a request sequence is generated by a probability distribution, i.e. by a discrete memoryless source. In fact, the algorithms Move-To-Front, Transpose and Frequency-Count given in Section 2.1 as well as their variants were proposed as heuristics for these particular request sequences.

We are now interested in the asymptotic expected cost incurred by a list update algorithm. For any algorithm A, let $E_A(\mathbf{p})$ denote the asymptotic expected cost incurred by A in serving a single request in a request sequence generated by the distribution $\mathbf{p} = (p_1, \ldots, p_n)$. In this situation, the performance of an on-line algorithm has generally been compared to that of the *optimal static ordering*, which we call STAT. The optimal static ordering first arranges the items x_i in nonincreasing order by probabilities and then serves a request sequence without changing the relative position of items. Clearly, $E_{STAT}(\mathbf{p}) = \sum_{i=1}^{n} ip_i$ for any distribution $\mathbf{p} = (p_1, \ldots, p_n)$.

As in Section 2.1, we first study the algorithms Move-To-Front(MTF), Transpose(T) and Frequency-Count(FC). By the strong law of large numbers we have $E_{FC}(\mathbf{p}) = E_{STAT}(\mathbf{p})$ for any probability distribution $\mathbf{p}$ [62]. However, as mentioned in Section 2.1, Frequency-Count may need a large amount of extra memory to serve a request sequence. It was shown by several authors [16, 18, 34, 45, 55, 62] that

$$E_{MTF}(\mathbf{p}) = 1 + 2 \sum_{1 \le i < j \le n} \frac{p_i p_j}{(p_i + p_j)}$$

for any $\mathbf{p} = (p_1, \ldots, p_n)$. A simple, closed-form expression for the asymptotic expected cost of the Transpose rule has not been found. The expression for $E_{MTF}(\mathbf{p})$ was used to show that $E_{MTF}(\mathbf{p}) \le 2E_{STAT}(\mathbf{p})$ for any distribution $\mathbf{p}$. However, Chung *et al.* [21] showed that Move-To-Front performs better.

Theorem 19. *For any probability distribution* $\mathbf{p}$, $E_{MTF}(\mathbf{p}) \le \frac{\pi}{2} E_{STAT}(\mathbf{p})$.

This bound is tight as was shown by Gonnet *et al.* [32].

Theorem 20. *For any* $\epsilon > 0$, *there exists a probability distribution* $\mathbf{p}_\epsilon$ *with* $E_{MTF}(\mathbf{p}_\epsilon) \ge (\frac{\pi}{2} - \epsilon) E_{STAT}(\mathbf{p}_\epsilon)$.

The distributions used in the proof of Theorem 20 are of the form

$$p_i = 1/(i^2 H_n^2) \qquad i = 1, \ldots, n$$

where $H_n^2 = \sum_{i=1}^{n} 1/i^2$. These distributions obey *Lotka's Law.* There are probability distributions $\mathbf{p}_0$ for which the ratio of $E_{MTF}(\mathbf{p}_0)/E_{STAT}(\mathbf{p}_0)$ can be smaller than $\pi/2 \approx 1.58$. Let $p_i = 1/(iH_n)$, $1 \le i \le n$, with $H_n = \sum_{i=1}^{n} 1/i$. This distribution is called *Zipf's Law.* Knuth [45] showed that for this distribution $\mathbf{p}_0$, $E_{MTF}(\mathbf{p}_0) \le (2 \ln 2) E_{STAT}(\mathbf{p}_0)$. We note that $2 \ln 2 \approx 1.386$.

Rivest [62] proved that Transpose performs better than Move-To-Front on distributions.

Theorem 21. *For any distribution* $\mathbf{p} = (p_1, \ldots, p_n)$, $E_T(\mathbf{p}) \leq E_{MTF}(\mathbf{p})$. *The inequality is strict unless* $n = 2$ *or* $p_i = 1/n$ *for* $i = 1, \ldots, n$.

Rivest conjectured that Transpose is optimal among all *permutation rules*. A permutation rule, when accessing an item at position j, applies a permutation π_j to the first j positions in the list. However, Anderson *et al.* [8] found a counterexample to this conjecture. Bitner [16] showed that while $E_T(\mathbf{p}) \leq E_{MTF}(\mathbf{p})$, the Move-To-Front rule converges faster to its asymptotic expected cost than Transpose.

The algorithms Move-To-Front, Transpose and Frequency-Count were also analyzed experimentally [14, 62, 74]. Rivest [62] generated request sequences that obeyed Zipf's law. On these sequences, Transpose indeed performed better than Move-To-Front. In contrast, Bentley and McGeoch [14] considered request sequences that came from word counting problems in text and Pascal files. In their tests, Transpose always performed worse than Move-To-Front and Frequency Count, with Move-To-Front usually being better then Frequency-Count. In general, STAT achieved a smaller average search time than the three on-line algorithms.

Finally, we consider the Timestamp(0) algorithm that was also presented in Section 2.1. It was shown in [5] that Timestamp(0) has a better performance then Move-To-Front if request sequences are generated by probability distributions. Let $E_{TS}(\mathbf{p})$ denote the asymptotic expected cost incurred by Timestamp(0).

Theorem 22. *For any probability distribution* $\mathbf{p}$, $E_{TS}(\mathbf{p}) \leq 1.34 E_{STAT}(\mathbf{p})$.

Theorem 23. *For any probability distribution* $\mathbf{p}$, $E_{TS}(\mathbf{p}) \leq 1.5 E_{OPT}(\mathbf{p})$.

Note that $E_{OPT}(\mathbf{p})$ is the asymptotic expected cost incurred by the optimal off-line algorithm OPT, which may dynamically rearrange the list while serving a request sequence. Thus, this algorithm is much stronger than STAT. The algorithm Timestamp(0) is the only algorithm whose asymptotic expected cost has been compared to $E_{OPT}(\mathbf{p})$.

The bound given in Theorem 23 holds with high probability. More precisely, for every distribution $\mathbf{p} = (p_1, \ldots, p_n)$, and $\epsilon > 0$, there exist constants c_1, c_2 and m_0 dependent on $\mathbf{p}, n$ and ϵ such that for any request sequence σ of length $m \geq m_0$ generated by $\mathbf{p}$,

$$Prob\{C_{TS}(\sigma) > (1.5 + \epsilon) C_{OPT}(\sigma)\} \leq c_1 e^{-c_2 m}.$$

2.4 Remarks

List update techniques were first studied in 1965 by McCabe [55] who considered the problem of maintaining a sequential file. McCabe also formulated the algorithms Move-To-Front and Transpose. From 1965 to 1985 the list update problem was studied under the assumption that a request sequence is generated by a probability distribution. Thus, most of the results presented is Section 2.3 were developed earlier than the results in Sections 2.1 and 2.2. A first survey about list update algorithms when a request sequence is generated by a distribution was written by Hester and Hirschberg [36]. The paper [65] by Sleator and Tarjan is a fundamental paper in the entire on-line algorithms literature. It made the competitive analysis of on-line algorithms very popular. Randomized on-line algorithms for the list update problem have been studied since the early nineties. The list update problem is a classical on-line problem that continues to be interesting both from a theoretical and practical point of view.

3 Binary search trees

Binary search trees are used to maintain a set S of elements where each element has an associated key drawn from a totally ordered universe. For convenience we assume each element is given a unique key, and that the n elements have keys $1, \ldots, n$. We will generally not distinguish between elements and their keys.

A binary search tree is a rooted tree in which each node has zero, one, or two children. If a node has no left or right child, we say it has a *null* left or right child, respectively. Each node stores an element, and the elements are assigned to the nodes in *symmetric order*: the element stored in a node is greater than all elements in descendents of its left child, and less than all elements in descendents of its right child. An inorder traversal of the tree yields the elements in sorted order. Besides the elements, the nodes may contain additional information used to maintain states, such as a color bit or a counter.

The following operations are commonly performed on binary search trees.

Successful-Access(x)**.** Locate an element $x \in S$.
Unsuccessful-Access(x)**.** Determine that an element x is **not** in S.
Insert(x)**.** Add a new element x to S. The tree is modified by adding a new node containing the element as a leaf, so that symmetric order is maintained.
Delete(x)**.** Remove element x from S. The resultant tree has one fewer nodes. There are several different deletion algorithms for binary search trees.
Split(x)**.** Split S into two sets: $S_1 = \{y \mid y \in S, y \leq x\}$ and $S_2 = \{y \mid y \in S, y > x\}$. S_1 and S_2 must be contained in two search trees.
Meld(S_1, S_2)**.** The inverse of a split (for all $x \in S_1, y \in S_2$, $x < y$).

Adel'son-Vel'skii and Landis [2] introduced a primitive operation for restructuring a binary search tree, the **edge rotation**. Figure 1 shows examples of left and right rotation of edge $\langle x, y\rangle$. Rotation preserves symmetric order.

In the *standard search tree model* [66][77] a search tree algorithm has the following behavior:

> The algorithm carries out each (successful) access by traversing the path from the root to the node containing the accessed item, at a cost of one plus the depth of the node containing the item, Between accesses the algorithm performs an arbitrary number of rotations anywhere in the tree, at a cost of one per rotation.[1]

[1] This definition is a slightly modified form of the one given in [66].

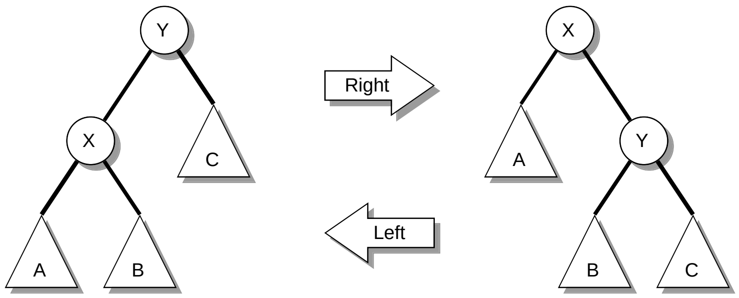


Fig. 1. Right and left rotations of $\langle x, y\rangle$.

The path from the root to a node is called the *access path*. If the path consists of only left edges, it is called a *left path*. A *right path* is defined analogously.

We expand the model to include unsuccessful searches, insertions and deletions as follows. Let T_0 be a search tree on n elements. T_0 is extended to search tree T by replacing any null child in the original tree by a leaf. Each leaf has an associated key range. If leaf l replaces the null left child of node x, then the range of l is the set of key values that are less than x but greater than the predecessor of x in the tree. If x is the least key in the original tree, then the range of l is all keys less than x. Similarly, if l replaces a null right child of x, then its range is all keys greater than x but less than the successor of x, if one exists. This is a well-known extension, and it is easy to see that the leaf ranges are disjoint. Successful searches are carried out as before. Between any operation, any number of rotations can be performed at cost 1 per rotation.

> The algorithm carries out an unsuccessful access to key i by traversing the path from the root to the leaf whose range contains i. The cost is 1 plus the length of the path.

> The algorithm carries out an insertion of a new element x (not already in the tree by assumption) by performing an unsuccessful search for x. Let l be the leaf reached by the search. Leaf l is replaced by a new node containing x. The new node has two new leaves containing the two halves of the the range originally in l. The cost is 1 plus the length of the path.

The model for deletion is rather more complicated, as deletion is itself a more complicated operation and can be done in a number of ways.

> The algorithm deletes an element x (already in the tree by assumption) in several phases. In the first phase, a successful search for x is performed, at the usual cost. In the second phase, any number of rotations are performed at cost 1 per rotation. In the third phase, let S_x be the subtree rooted at x after phase two. Let p be the predecessor of x in S_x, or x itself if x is the least element in S_x. Similarly, let s the successor

of x in S_x, or x itself if x has no successor. The algorithm chooses one of p or s, say p w.l.o.g., and traverses the path from x to p, at cost 1 plus the path length. In phase four, the algorithm changes pointers in x, p, and their parents, to construct two search trees, one consisting only of x, and the other containing all the remaining elements. The singleton tree is discarded. The cost of phase four is 1.

The successful and unsuccessful access can be implemented in a comparison model with a well-known recursive procedure (see [26, Chapter 13]): the search key is compared to the element in the root of the tree. If they are equal, the root element is returned. Otherwise the left subtree or right subtree of the root is recursively searched, according to whether the key is less than or greater than the root element, respectively. If the desired subtree is null, the procedure returns an unsuccessful search indicator. Examples of various deletion routines algorithms can be found in [26] or [66].

For what follows, we will restrict our attention to request sequences that consist only of successful accesses, unless otherwise stated. For an algorithm, A, in the standard search model, let $A(\sigma, T_0)$ denote the sum of the costs of accesses and rotations performed by A in response to access sequence σ starting from tree T_0, where T_0 is a search tree on n elements and σ accesses only the elements in T_0. Let $\mathrm{OPT}(\sigma, T_0)$ denote the minimum cost of servicing σ, starting from search tree T_0 containing elements $1, \ldots, n$, including both access costs and rotation costs.

Definition 24. An algorithm A is $f(n)$-competitive if for all σ, T_0

$$A(\sigma, T_0) \leq f(n) \cdot \mathrm{OPT}(\sigma, T_0) + O(n) \tag{5}$$

Let S denote the *static* search algorithm, *i.e.*, the algorithm which performs no rotations on the initial tree T_0.

Definition 25. An algorithm A is $f(n)$-static-competitive if for all σ, T_0

$$A(\sigma, T_0) \leq f(n) \cdot \min_T S(\sigma, T) + O(n) \tag{6}$$

where T is a search tree on the same elements as T_0.

Note that $S(\sigma, T)$ is given by

$$\sum_{i=1}^{n} f_\sigma(i) d_T(i)$$

where $f_\sigma(i)$ is the number of times element i is accessed in σ, and $d_T(i)$ denotes the depth of element i in search tree T.

A final definition deals with probabilistic request sequences, in which each request is chosen at random from among the possible requests according to some distribution $\mathcal{D}$. That is, on each request element i is requested with probability p_i, $i = 1 \cdots n$. For a fixed tree T, the expected cost of a request is

$$1 + \sum_{i=1}^{n} p_i d_T(i)$$

We denote this $S(\mathcal{D}, T)$, indicating the cost of distribution $\mathcal{D}$ on static tree T. Finally, let $\bar{S}(\mathcal{D}) = \min_T S(\mathcal{D}, T)$ denote the expected cost per request of the optimal search tree for distribution $\mathcal{D}$. For a search tree algorithm A, let $\bar{A}(\mathcal{D}, T_0)$ denote the *asymptotic expected cost* of servicing a request, given that the request is generated by $\mathcal{D}$ and the algorithm starts with T_0.

Definition 26. An algorithm A is called $f(n)$-distribution-competitive if for all n, all distributions $\mathcal{D}$ on n elements and all initial trees T_0 on n elements,

$$\bar{A}(\mathcal{D}, T_0) \leq f(n) \cdot \bar{S}(\mathcal{D}) \tag{7}$$

Definitions 25 and 26 are closely related, since both compare the cost of an algorithm on sequence σ to the cost of a fixed search tree T on σ. The fixed tree T that achieves the minimum cost is the tree that minimizes the *weighted path length* $\sum_{i=1}^{n} w(i) d_T(i)$, where the weight of node i is the total number of accesses to i, in the case of static competitiveness, or the probability of an access to i, in the case of distribution optimality. Knuth [44] gives an $O(n^2)$ algorithm for computing a tree that minimizes the weighted path length. By information theory [1], for a request sequence σ, $m = |\sigma|$,

$$S(\sigma, T) = \Omega(m + \sum_{i=1}^{n} f_\sigma(i) \log(m/f_\sigma(i))).$$

In the remainder of this section, we will first discuss the off-line problem, in which the input is the entire request sequence σ and the output is a sequence of rotations to be performed after each request so that the the total cost of servicing σ is minimized. Little is known about this problem, but several characterizations of optimal sequences are known and these suggest some good properties for on-line algorithms.

Next we turn to on-line algorithms. An $O(\log n)$ competitive ratio can be achieved by any one of several balanced tree schemes. So far, no on-line algorithm is known that is $o(\log n)$-competitive. Various adaptive and self-organizing rules have been suggest over the past twenty years. Some of them have good properties against probability distributions, but most perform poorly against arbitrary sequences. Only one, the *splay* algorithm, has any chance of being $O(1)$-competitive. We review the known properties of splay trees, the various conjectures made about their performance, and progress on resolving those conjectures.

3.1 The off-line problem and properties of optimal algorithms

An off-line search tree algorithm takes as input a request sequence σ and initial tree T_0 and outputs a sequence of rotations, to be intermingled with servicing successive requests in σ, that achieves the minimum total cost $\mathrm{OPT}(\sigma, T_0)$.

It is open whether $\mathrm{OPT}(\sigma, T_0)$ can be computed in polynomial time. There are $\frac{1}{n+1}\binom{2n}{n}$ binary search trees on n nodes, so the dynamic programming algorithm for solving metrical service systems or metrical task systems requires exponential space just to represent all possible states.

A basic subproblem in the dynamic programming solution is to compute the rotation distance, $d(T_1, T_2)$, between two binary search trees T_1 and T_2 on n nodes. The rotation distance is the minimum number of rotations needed to transform T_1 into T_2. It is also open whether the rotation distance can be computed in polynomial time. Upper and lower bounds on the worst-case rotation distance are known, however.

Theorem 27. *The rotation distance between any two binary trees is at most 2n-6, and there is an infinite family of trees for which this bound is tight.*

An upper bound of $2n - 2$ was shown by Crane [27] and Culik and Wood [46]. This bound is easily seen: a tree T can be converted into a right path by repeatedly rotating an edge hanging off the right spine onto the right spine with a right rotation. This process must eventually terminate with all edges on the right spine. Since there are $n - 1$ edges, the total number of rotations is $n - 1$. To convert T_1 to T_2, convert T_1 to a right path then compute the rotations needed to convert T_2 to a right path, and apply them in reverse to the right spine into which T_1 has been converted. At most $2n - 2$ rotations are necessary. Sleator, Tarjan and Thurston [67] improved the upper bound to $2n-6$ using a relation between binary trees and triangulations of polyhedra. They also demonstrated the existence of an infinite family in which $2n - 6$ rotations were required. Makinen [52] subsequently showed that a weaker upper bound of $2n-5$ can be simply proved with elementary tree concepts. His proof is based on using either the left path or right path as an intermediate tree, depending on which is closer. Luccio and Pagli [51] showed that the tight bound $2n - 6$ could be achieved by adding one more possible intermediate tree form, a root whose left subtree is a right path and whose right subtree is a left path.

Wilber [77] studied the problem of placing a lower bound on the cost of the optimal solution to specific families of request sequences. He described two techniques for calculating lower bounds, and used them to show the existence of request sequences on which the optimal cost is $\Omega(n \log n)$. We give one of his examples below. Let i and k be non-negative integers, $i \in [0, 2^k - 1]$. The *k-bit reversal* of i, denoted $br_k(i)$, is the integer j given by writing the binary representation of i backwards. Thus $br_k(6) = 3$ ($110 \Rightarrow 011$). The bit reversal permutation on $n = 2^k$ elements is the sequence $B^k = br_k(0), br_k(1), \ldots, br_k(n-1)$.

Theorem 28. [77] *Let k be a nonnegative integer and let $n = 2^k$. Let T_0 be any search tree with nodes $0, 1, \ldots, n-1$. Then* $\mathrm{OPT}(B^k, T_0) \geq n \log n + 1$.

Lower bounds for various on-line algorithms can be found using two much simpler access sequences: the *sequential access sequence* $\sigma^S = 1, 2, \ldots, n$, and the reverse order sequence $\sigma^R = n, n-1, \ldots, 1$. It is easy to see that $\mathrm{OPT}(\sigma^S, T_0) = O(n)$. For example, T_0 can be rotated to a right spine in $n-1$ time, after which each successive accessed element can be rotated to the root with one rotation. This gives an amortized cost of $2 - 1/n$ per access. The optimal static tree is the completely balanced tree, which achieves a cost of $\Theta(\log n)$ per access. The sequential access sequence can be repeated k times, for some integer k, with the same amortized costs per access.

Although no polynomial time algorithm is known for computing $\mathrm{OPT}(\sigma, T_0)$, there are several characterizations of the the properties of optimal and near-optimal solutions. Wilber [77] and Lucas [50] note that there a solution within a factor of two of optimum in which each element is at the root when it is accessed. The near-optimal algorithm imitates the optimal algorithm except that it rotates the accessed item to the root just prior to the access, and then undoes all the rotations in reverse to restore the tree to the old state. Hence one may assume that the accessed item is always at the root, and all the cost incurred by the optimum strategy is due to rotations.

Lucas [50] proves that there is an optimum algorithm in which rotations occur prior to the access and the rotated edges form a connected subtree containing the access path. In addition, Lucas shows that if in the initial tree T_0 there is a node x none of whose descendents (including x) are ever accessed, then x need never be involved in a rotation. She also studies the "rotation graph" of a binary tree and proves several properties about the graph [48]. The rotation graph can be used to enumerate all binary search trees on n nodes in $O(1)$ time per tree.

Lucas makes several conjectures about the off-line and on-line search tree algorithms.

Conjecture 1 (Greedy Rotations) *There is a c-competitive off-line algorithm (and possibly on-line) such that each search is to the element at the root and all rotations decrease the depth of the next element to be searched for.*

Observe that the equivalent conjecture for the list update problem (that all exchanges decrease the depth of the next accessed item) is true for list update, since an off-line algorithm can be 2-competitive by moving the next accessed item to the front of the list just prior to the access. The cost is the same as is incurred by the move-to-front heuristic, since the exchanges save one unit of access cost. But for list update, there is no condition on relative position of elements (*i.e.* no requirement to maintain symmetric order), so the truth of the conjecture for search trees is non-obvious.

Lucas proposes a candidate polynomial-time off-line algorithm, and conjectures that it provides a solution that is within a constant factor of optimal, but does not prove the conjecture. The proposed "greedy" algorithm modifies the tree prior to each access. The accessed element is rotated to the root. The edges that are rotated off the path during this process form a collection of connected subtrees, each of which is a left path or a right path. These paths are then converted into connected subtrees that satisfy the heap property where the heap key of a node is the time it or a descendent off the path will be next accessed. This tends to move the element that will be accessed soonest up the tree.

3.2 On-line algorithms

An $O(\log n)$-competitive solution is achievable with a balanced tree; many balanced binary tree algorithms are known that handle unsuccessful searches, insertions, deletions, melds, and splits in $O(\log n)$ time per operation, either amortized or worst-case. Examples include AVL-trees, red/black trees, and weight-balanced trees; see the text [26] for more information. These data structures make no attempt to self-organize, however, and are concerned solely with keeping the maximum depth of any item at $O(\log n)$. Any heuristic is, of course, $O(n)$-competitive.

A number of candidate self-organizing algorithms have been proposed in the literature. These can generally be divided into memoryless and state-based algorithms. A memoryless algorithm maintains no state information besides the current tree. The proposed memoryless heuristics are:

1. Move to root by rotation [7].
2. Single-rotation [7].
3. Splaying [66].

The state-based algorithms are

1. Dynamic monotone trees [16].
2. WPL trees [20].
3. D-trees [57, 58].

3.3 State-based algorithms

Bitner [16] proposed and analyzed *dynamic monotone trees.* Dynamic monotone trees are a dynamic version of a data structure suggested by Knuth [45] for approximating optimal binary search trees given a distribution $\mathcal{D}$. The element with maximum probability of access is placed at the root of the tree. The left and right subtrees of the root are then constructed recursively. In dynamic monotone trees, each node contains an element and a counter. The counters are initialized to zero. When an element is accessed, its counter is incremented by one, and the element is rotated upwards while its counter is greater than the counter in its parent. Thus the tree stores the elements in symmetric order by key, and in max-heap order by counter. A similar idea is used in the "treap," a randomized search tree developed by Aragon and Seidel [9].

Unfortunately, monotone and dynamic monotone trees do poorly in the worst case. Mehlhorn [56] showed that the distribution-competitive ratio is $\Omega(n/\log n)$. This is easily seen by assigning probabilities to the elements $1, \ldots, n$ so that

$p_1 > p_2 > \ldots > p_n$ but so that all are very close to $1/n$. The monotone tree corresponding to these probabilities is a right path, and by the law of large numbers a dynamic monotone tree will asymptotically tend to this form. The monotone tree is also no better than $\Omega(n)$-competitive; repetitions of sequential access sequence give this lower bound. Bitner showed, however, that monotone trees do perform better on probability distributions with low entropy (the bad sequence above has nearly maximum entropy.) Thus as the distributions become more skewed, monotone trees do better.

Bitner also suggested several conditional modification rules. When an edge is rotated, one subtree decreases in depth while another increases in depth. With conditional rotation, an accessed node is rotated upwards if the total number of accesses to nodes in the subtree that moves up is greater than the total number of accesses to the subtree that moves down. No analysis of the conditional rotation rules is given, but experimental evidence is presented which suggests they perform reasonably well. No such rule is better than $\Omega(n)$-competitive, however, for the same reason single exchange is not competitive.

Oommen *et al.* [20] generalized the idea of conditional rotations by adding two additional counters. One stores the number of accesses to descendents of a node, and one stores the total weighted path length (defined above) of the subtree rooted at the node. After an access, the access path is processed bottom-up and successive edges are rotated as long as the weighted path length (WPL) of the whole tree diminishes. The weight of a node at time t for this purpose is taken to be the number of accesses to that node up to time t. Oommen *et al.* claim that their WPL algorithm asymptotically approaches the tree with minimum weighted path length, and hence is $O(1)$-distribution-competitive. By the law of large numbers, the tree that minimizes the weighted path length will asymptotically approach the optimal search tree for the distribution. The main contribution of Oommen *et al.* is to show that changes in the weighted path length of the tree can be computed efficiently. The WPL tree can be no better than $\Omega(\log n)$-competitive (once again using repeated sequential access) but it is unknown if this bound is achieved. It is also unknown whether WPL-trees are $O(1)$-static-competitive.

Mehlhorn [57, 58] introduced the D-tree. The basic idea behind a D-tree is that each time an element is accessed the binary search tree is extended by adding a dummy node as a leaf in the subtree rooted at accessed element. The extended tree is then maintained using a weight-balanced or height-balanced binary tree (see [58] for more information on such trees). Since nodes that are frequently accessed will have more dummy descendents, they will tend to be higher in the weight-balanced tree. Various technical details are required to implement D-trees in a space-efficient fashion. Mehlhorn shows that the D-tree is $O(1)$-distribution-competitive.

At the end of an access sequence the D-tree is near the optimal static binary tree for that sequence, but it is not known if it is $O(1)$-static-competitive, since a high cost may be incurred in getting the tree into the right shape.

3.4 Simple memoryless heuristics

The two memoryless heuristics *move-to-root* (MTR) and *simple exchange* (SE) were proposed by Allen and Munro as the logical counterparts in search trees of the move-to-front and transpose rules of sequential search.

In simple exchange, each time an element is accessed, the edge from the node to its parent is rotated, so the element moves up. With the MTR rule, the element is rotated to the root by repeatedly applying simple exchanges. Allen and Munro show that MTR is fares well when the requests are generated by a probability distribution, but does poorly against an adversary.

Theorem 29. [7] *The move-to-root heuristic is $O(1)$-distribution-competitive.*

Remark: the constant inside the $O(1)$ is $2\ln 2 + o(1)$.

Theorem 30. [7] *On the sequential access sequence $\sigma = (1, 2, \ldots, n)^k$ the MTR heuristic incurs $\Omega(n)$ cost per request when $k \geq 2$.*

Remark: Starting from any initial tree T_0, the sequence $1, 2, \ldots, n$ will cause MTR to generate the tree consisting of a single left path. Thereafter the cost of a request to i will have cost $n - i + 1$.

Corollary 31. *The competitive ratio of the MTR heuristic is $\Theta(n)$, and the static competitive ratio of MTR is $\Omega(n/\log n)$.*

The corollary follows from the observation of Section 3.1 that the sequential access sequence can be satisfied with O(1) amortized cost per request, and with $O(\log n)$ cost per request if a fixed tree is used.

Although MTR is not competitive, it does well against a probability distribution. The simple exchange heuristic, however, is not even good against a probability distribution.

Theorem 32. [7] *If $p_i = 1/n$, $1 \leq i \leq n$, then the asymptotic expected time per request of the simple exchange algorithm is $\sqrt{\pi n} + o(\sqrt{n})$.*

For this distribution the asymptotic expected cost of a perfectly balanced tree is $O(\log n)$. This implies:

Corollary 33. *The distribution competitive ratio of SE is $\Omega(\sqrt{n}/\log n)$.*

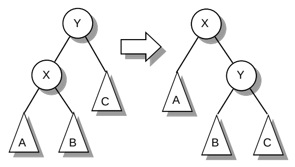
Corollary 34. *The competitive ratio of SE is $\Theta(n)$, and the static competitive ratio of SE is $\Omega(n/\log n)$.*

Corollary 34 can be proved with a sequential access sequence, except that each single request to i is replaced by enough consecutive requests to force i to the root. After each block of consecutive requests to a given element i, the resulting tree has the same form as the tree generated by MTR does after a single request to i.

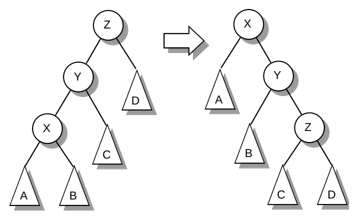
3.5 Splay trees

To date, the only plausible candidate for a search tree algorithm that might be $O(1)$-competitive is the splay tree, invented by Sleator and Tarjan [66]. A splay tree a binary search tree in which all operations are performed by means of a primitive called *splaying*. A splay at node x is a sequence of rotations on the path that moves x the root of the tree. The crucial difference between splaying and simple move-to-root is that while move-to-root rotates each edge on the path from x to the root in order from top to bottom, splaying rotates some higher edges before some lower edges. The order is chosen so that the nodes on the path decrease in depth by about one half. This halving of the depth does not happen with the move-to-root heuristic.

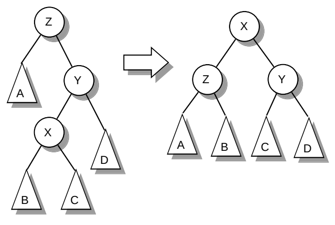
The splaying of node x to the root proceeds by repeatedly determining which of the three cases given in Fig. 2 applies, and performing the diagramed rotations. Sleator and Tarjan call these cases respectively the zig, zig-zig, and zig-zag cases. Note that in the zig and zig-zag cases, the rotations that occur are precisely those that would occur with the move-to-root heuristic. But the zig-zig is different. Simple move-to-root applied to a long left or right path leads to another long left or right path, while repeatedly executing zig-zig makes a much more balanced tree.



(a) Zig case.



(b) Zig-zig case.



(c) Zig-zag case.

Fig. 2. Three cases of splaying. Case (a) applies only when y is the root. Symmetric cases are omitted.

The fundamental lemma regarding splay trees is the *splay access lemma*. Let

$w_1, w_2, \ldots, w_n$ be a set of arbitrary real weights assigned to elements 1 through n respectively, and let $W = \sum_{i=1}^{n} w_i$.

Lemma 35. [66] *The cost of splaying item i to the root is $O(\log(W/w_i))$.*

This result is based on the following potential function (called a *centroid* potential function by Cole [23, 24]). Let s_i be the sum of the weights of the elements that are descendents of i in the search tree, including i itself. Let $r_i = \log s_i$. Then $\Phi = \sum_{i=1}^{n} r_i$. The amortized cost of a zig-zig or zig-zag operation is $3(r_z - r_x)$ while the cost of a zig operation is $3r_y$ (with reference to Fig. 2).

Theorem 36. [66] *The splay tree algorithm is $O(\log n)$-competitive.*

This follows from Lemma 35 with $w_i = 1$ for all i, in which case the amortized cost of an access is $O(\log n)$. Therefore splay trees are as good in an asymptotic sense as any balanced search tree. They also have other nice properties. Operations such as delete, meld, and split can be implemented with a splay operation plus a small number of each pointer changes at the root. Splay trees also adapt well to unknown distributions, as the following theorem shows.

Theorem 37. [66] *The splay tree is $O(1)$-static-competitive.*

This theorem follows by letting $w_i = f_\sigma(i)$, the frequency with which i is accessed in σ, and comparing with the information theoretic lower bound giving at the beginning of this section.

Sleator and Tarjan also made several conjectures about the competitiveness of splay trees. The most general is the "dynamic optimality" conjecture.

Conjecture 2 (Dynamic optimality) *The splay tree is $O(1)$-competitive.*

Sleator and Tarjan made two other conjectures, both of which are true if the dynamic optimality conjecture is true. (The proof of these implications are non-trivial, but have not been published. They have been reported by Sleator and Tarjan [66], Cole *et al.* [25] and Cole [23, 24].)

Conjecture 3 (Dynamic finger) *The total time to perform m successful accesses on an n-node splay tree is $O(m + n + \sum_{j=1}^{m-1} \log(|i_{j+1} - i_j| + 1)$, where for $1 \leq i \leq m$ the jth access is to item i_j (we denote items by their symmetric order position).*

Conjecture 4 (Traversal) *Let T_1 and T_2 be any two n-node binary search trees containing exactly the same items. Suppose we access the items in T_1 one after another using splaying, accessing them in order according to their preorder number in T_2. Then the total access time is $O(n)$.*

There are a number of variations of basic splaying, most of which attempt to reduce the number of rotations per operation. Sleator and Tarjan suggested *semisplaying*, in which only the topmost of the two zig-zig rotations is done, and *long splaying*, in which a splay only occurs if the path is sufficiently long. Semisplaying still achieves an $O(\log n)$ competitive ratio, as does long splaying with an appropriate definition of "long." Semisplaying may still be $O(1)$-competitive, but long splaying cannot be. Klostermeyer [43] also considered some variants of splaying but provides no analytic results.

3.6 Progress on splay tree conjectures

In this section we describe subsequent progress in resolving the original splay tree conjectures, and several related conjectures that have since appeared in the literature.

Tarjan [73] studied the performance of splay trees on two restricted classes of inputs. The first class consists of sequential access sequences, $\sigma = 1, 2, \ldots, n$. The dynamic optimality conjecture, if true, implies that the time for a splay tree to perform a sequential access sequence must be $O(n)$, since the optimal time for such a sequence is at most $2n$.

Theorem 38. [73] *Given an arbitrary n-node splay tree, the total time to splay once at each of the nodes, in symmetric order, is $O(n)$.*

Tarjan called this the *scanning theorem*. The proof of the theorem is based on an inductive argument about properties of the tree produced by successive accesses. Subsequently Sundar [68] gave a simplified proof based on a potential function argument.

In [73] Tarjan also studied request sequences consisting of double-ended queue operations: PUSH, POP, INJECT, EJECT. Regarding such sequences he made the following conjecture.

Conjecture 5 (Deque) *Consider the representation of a deque by a binary tree in which the* i*th node of the binary tree in symmetric order corresponds to the* i*th element of the deque. The splay algorithm is used to perform deque operations on the binary tree as follows:* POP *splays at the smallest node of the tree and removes it from the tree;* PUSH *makes the inserted item the new root, with null left child and the old root as right child;* EJECT *and* INJECT *are symmetric. The cost of performing any sequence of m deque operations on an arbitrary n-node binary tree using splaying is $O(m + n)$.*

Tarjan observed that the dynamic optimality conjecture, if true, implies the deque conjecture. He proved that the deque conjecture is true when the request sequence does not contain any EJECT operations. That is, new elements can be inserted at both ends of the queue, but only removed from one end. Such a deque is called *output-restricted.*

Theorem 39. [73] *Consider a sequence of m* PUSH, POP, *and* INJECT *operations performed as described in the deque conjecture on an arbitrary initial tree T_0 containing n nodes. The total time required is $O(m + n)$.*

The proof uses an inductive argument.

Lucas [49] showed the following with respect to Tarjan's deque conjecture.

Theorem 40. [49] *The total cost of a series of ejects and pops is $O(n\alpha(n,n))$ if the initial tree is a simple path of n nodes from minimum node to maximum node.*[2]

Sundar [68, 69] came within a factor of $\alpha(n)$ of proving the deque conjecture. He began by considering various classes of restructurings of paths by rotations. A right 2-turn on a binary tree is a sequence of two right rotations performed on the tree in which the bottom node of the first rotation is identical to the top node of the second rotation. A 2-turn is equivalent to a zig-zig step in the splay algorithm. (The number of single right rotations can be $\Omega(n^2)$. See, for example, the remark above following Thm. 30.) As reported in [73], Sleator conjectured that the total number of right 2-turns in any sequence of right 2-turns and right rotations performed on an arbitrary n-node binary tree is $O(n)$. Sundar observed that this conjecture, if true, would imply that the deque conjecture was true. Unfortunately, Sundar disproved the turn conjecture, showing examples in which $\Omega(n \log n)$ right 2-turns occur.[3]

Sundar then considered the following generalizations of 2-turns

1. *Right twists.* For $k > 1$, a right k-twist arbitrarily selects k different edges from a left subpath of the binary tree and rotates the edges one after another in top-to-bottom order. From an arbitrary initial tree, $O(n^{1+1/k})$ right twists can occur and $\Omega(n^{1+1/k}) - O(n)$ are possible.
2. *Right turns:* For any $k > 1$ a right k-turn is a right k-twist that converts a left subpath of k edges in the binary tree into a right subpath by rotating the edges of the subpath in top-to-bottom order. $O(n\alpha(k/2, n))$ right twists can occur if $k \neq 3$ and $O(n \log\log n)$ can occur if $k = 3$. On the other hand, there are trees in which $\Omega(n\alpha(k/2, n)) - O(n)$ k-twists are possible if $k \neq 3$ and $\Omega(n \log\log n)$ are possible when $k = 3$.
3. *Right cascade:* For $k > 1$, a right k-cascade is a right k-twist that rotates every other edge lying on a left subpath of $2k-1$ edges in the binary tree. The same bounds hold for right k-cascades as for right turns.

(Symmetric definitions and results hold for left twists, turns, and cascades.)

Using these results, Sundar proved the following theorem.

Theorem 41. [68, 69] *The cost of performing an intermixed sequence of m deque operations on an arbitrary n-node binary tree using splaying is $O((m+n)\alpha(m+n, m+n))$.*

---

[2] $\alpha(i,j)$ is the functional inverse of the Ackermann function, and is a very slowly growing function. See, for example, [71] for more details about the inverse Ackermann function.

[3] Sundar reports that S.R. Kosaraju independently disproved the turn conjecture, with a different technique.

Sundar added one new conjecture to the splay tree literature. A *right ascent* of a node x is a maximal series of consecutive right rotations of the edge connecting a node and its parent.

Conjecture 6 (Turn-ascent) *The maximum number of right 2-turns in any intermixed series of right 2-turns and r right ascents performed on an n-node binary search tree is $O(n + r)$.*

Sundar observes that if this conjecture is true, it implies the truth of the deque conjecture.

The greatest success in resolving the various splay conjectures is due to Cole [23, 24], who was able to prove the truth of the dynamic finger conjecture. His paper is quite complex, and builds both on the work of Sundar and of Cole *et al.* [25] on splay sorting.

Theorem 42. Dynamic finger theorem. [66] *The total time to perform m successful accesses on an n-node splay tree is $O(m+n+\sum_{j=1}^{m-1}\log(|i_{j+1}-i_j|+1))$, where for $1 \leq i \leq m$ the jth access is to item i_j (we denote items by their symmetric order position).*

The proof of this theorem is very intricate, and we will not attempt to summarize it here.

In recent work, Chaudhuri and Hoft [19] prove if the nodes of an arbitrary n-node binary search tree T are splayed in the preorder sequence of T then the total time spent is $O(n)$. This is a special case of the traversal conjecture. Cohen and Fredman [22] give some further evidence in favor of the truth of the splay tree conjecture. They analyze several classes of request sequences generated from a random distribution, and show the splay tree algorithm is $O(1)$-competitive on these sequences.

3.7 Remarks

While exciting progress has been made in resolving special cases of the dynamic optimality conjecture for splay trees, it is unclear how this work will impact the full conjecture. In competitive analysis one usually compares the performance of an on-line algorithm to the performance of an (unknown) optimal off-line algorithm by means of some form of potential function. None of the results on the splay tree conjectures use such a potential function. Rather than comparing the splay tree algorithms to an optimal off-line algorithm, the proofs directly analyze properties of the splay tree on the special classes of requests. Finding some potential function that compares on-line to off-line algorithms is perhaps the greatest open problem in the analysis of the competitive binary search trees.

Splay trees have been generalized to multiway and k-ary search trees by Martel [54] and Sherk [64]. Some empirical results on self-adjusting trees and splay trees in particular have appeared. Moffat *et al.* [59] give evidence that sorting using splay trees is quite efficient. On the other hand, Bell and Gupta

[10] give evidence that on random data that is not particularly skewed, self-adjusting trees are generally slower than standard balanced binary trees. There still remains a great deal of work to be done on empirical evaluation of self-adjusting trees on data drawn from typical real-life applications.

4 Data compression: An application of self-organizing data structures

Linear lists and splay trees, as presented in Section 3.5, can be used to build locally adaptive data compression schemes. In the following we present both theoretical and experimental results.

4.1 Compression based on linear lists

The use of linear lists in data compression recently became of considerable importance. In [17], Burrows and Wheeler developed a data compression scheme using unsorted lists that achieves a better compression than Ziv-Lempel based algorithms. Before describing their algorithm, we first present a data compression scheme given by Bentley *et al.* [15] that is very simple and easy to implement.

In data compression we are given a string S that shall be *compressed*, i.e., that shall be represented using fewer bits. The string S consists of *symbols*, where each symbol is an element of the alphabet $\Sigma = \{x_1, \ldots, x_n\}$. The idea of data compression schemes using linear lists it to convert the string S of symbols into a string I of integers. An *encoder* maintains a linear list of symbols contained in Σ and reads the symbols in the string S. Whenever the symbol x_i has to be compressed, the encoder looks up the current position of x_i in the linear list, outputs this position and updates the list using a list update rule. If symbols to be compressed are moved closer to the front of the list, then frequently occurring symbols can be encoded with small integers.

A *decoder* that receives I and has to recover the original string S also maintains a linear list of symbols. For each integer j it reads from I, it looks up the symbol that is currently stored at position j. Then the decoder updates the list using the same list update rule as the encoder. Clearly, when the string I is actually stored or transmitted, each integer in the string should be coded again using a variable length prefix code.

In order to analyze the above data compression scheme one has to specify how an integer j in I shall be encoded. Elias [28] presented several coding schemes that encode an integer j with essentially $\log j$ bits. The simplest version of his schemes encodes j with $1 + 2\lfloor \log j \rfloor$ bits. The code for j consists of a prefix of $\lfloor \log j \rfloor$ 0's followed by the binary representation of j, which requires $1 + \lfloor \log j \rfloor$ bits. A second encoding scheme is obtained if the prefix of $\lfloor \log j \rfloor$ 0's followed by the first 1 in the binary representation of j is coded again using this simple scheme. Thus, the second code uses $1 + \lfloor \log j \rfloor + 2\lfloor \log(1 + \log j) \rfloor$ bits to encode j.

Bentley *et al.* [15] analyzed the above data compression algorithm if encoder and decoder use Move-To-Front as list update rule. They assume that an integer j is encoded with $f(j) = 1 + \lfloor \log j \rfloor + 2\lfloor \log(1 + \log j)\rfloor$ bits. For a string S, let $A_{MTF}(S)$ denote the average number of bits needed by the compression algorithm to encode one symbol in S. Let m denote the length of S and let m_i, $1 \le i \le n$, denote the number of occurrences of the symbol x_i in S.

Theorem 43. *For any input sequence S,*

$$A_{MTF}(S) \le 1 + H(S) + 2\log(1 + H(S)),$$

where $H(S) = \sum_{i=1}^{n} \frac{m_i}{m} \log(\frac{m}{m_i})$.

The expression $H(S) = \sum_{i=1}^{n} \frac{m_i}{m} \log(\frac{m}{m_i})$ is the "empirical entropy" of S. The empirical entropy is interesting because it corresponds to the average number of bits per symbol used by the optimal static Huffman encoding for a sequence. Thus, Theorem 43 implies that Move-To-Front based encoding is almost as good as static Huffman encoding.

Proof of Theorem 43. We assume without loss of generality that the encoder starts with an empty linear list and inserts new symbols as they occur in the string S. Let $f(j) = 1 + \lfloor \log j \rfloor + 2\lfloor \log(1 + \log j)\rfloor$. Consider a fixed symbol x_i, $1 \le i \le n$, and let $q_1, q_2, \ldots, q_{m_i}$ be the positions at which the symbol x_i occurs in the string S. The first occurrence of x_i in S can the encoded with $f(q_1)$ bits and the k-th occurrence of x_i can be encoded with $f(q_k - q_{k-1})$ bits. The m_i occurrences of x_i can be encoded with a total of

$$f(q_1) + \sum_{k=1}^{m_i} f(q_k - q_{k-1})$$

bits. Note that f is a concave function. We now apply Jensen's inequality, which states that for any concave function f and any set $\{w_1, \ldots, w_n\}$ of positive reals whose sum is 1, $\sum_{i=1}^{n} w_i f(y_i) \le f(\sum_{i=1}^{n} w_i y_i)$ [37]. Thus, the m_i occurrences of x_i can be encoded with at most

$$m_i f(\frac{1}{m_i}(q_1 + \sum_{k=2}^{m_i}(q_k - q_{k-1})) = m_i f(\frac{q_{m_i}}{m_i}) \le m_i f(\frac{m}{m_i})$$

bits. Summing the last expression for all symbols x_i and dividing by m, we obtain

$$A_{MTF}(S) = \sum_{i=1}^{n} \frac{m_i}{m} f(\frac{m}{m_i}).$$

The definition of f gives

$$\begin{aligned} A_{MTF}(S) &\le \sum_{i=1}^{n} \frac{m_i}{m} + \sum_{i=1}^{n} \frac{m_i}{m} \log(\frac{m}{m_i}) + \sum_{i=1}^{n} \frac{m_i}{m} 2\log(1 + \log(\frac{m}{m_i})) \\ &\le \sum_{i=1}^{n} \frac{m_i}{m} + \sum_{i=1}^{n} \frac{m_i}{m} \log(\frac{m}{m_i}) + 2\log(\sum_{i=1}^{n} \frac{m_i}{m} + \sum_{i=1}^{n} \frac{m_i}{m} \log(\frac{m}{m_i})) \\ &= 1 + H(S) + 2\log(1 + H(S)). \end{aligned}$$

The second inequality follows again from Jensen's inequality. □

Bentley *et al.* [15] also considered strings that are generated by probability distributions, i.e., by discrete memoryless sources $\mathbf{p} = (p_1, \ldots, p_n)$. The p_i's are positive probabilities that sum to 1. In a string S generated by $\mathbf{p} = (p_1, \ldots, p_n)$, each symbol is equal to x_i with probability p_i, $1 \le i \le n$. Let $B_{MTF}(\mathbf{p})$ denote the expected number of bits needed by Move-To-Front to encode one symbol in a string generated by $\mathbf{p} = (p_1 \ldots, p_n)$.

Theorem 44. *For any* $\mathbf{p} = (p_1, \ldots, p_n)$,

$$B_{MTF}(\mathbf{p}) \le 1 + H(\mathbf{p}) + 2\log(1 + H(\mathbf{p})),$$

where $H(\mathbf{p}) = \sum_{i=1}^{n} p_i \log(1/p_i)$ *is the entropy of the source.*

Shannon's source coding theorem (see e.g. Gallager [31]) implies that the number $B_{MTF}(\mathbf{p})$ of bits needed by Move-To-Front encoding is optimal, up to a constant factor.

Albers and Mitzenmacher [5] analyzed the data compression algorithm if encoder and decoder use Timestamp(0) as list update algorithm. They showed that a statement analogous to Theorem 43 holds. More precisely, for any string S, let $A_{MTF}(S)$ denote the average number of bits needed by Timestamp(0) to encode one symbol in S. Then, $A_{TS}(S) \le 1 + H(S) + 2\log(1 + H(S))$, where $H(S)$ is the empirical entropy of S. For strings generated by discrete memoryless sources, Timestamp(0) achieves a better compression than Move-To-Front.

Theorem 45. *For any* $\mathbf{p} = (p_1, p_2, \ldots, p_n)$,

$$B_{TS}(\mathbf{p}) \le 1 + \overline{H}(\mathbf{p}) + 2\log(1 + \overline{H}(\mathbf{p})),$$

where $\overline{H}(\mathbf{p}) = \sum_{i=1}^{n} p_i \log(1/p_i) + \log(1 - \sum_{i \le j} p_i p_j (p_i - p_j)^2/(p_i + p_j)^2)$.

Note that $0 \le \sum_{i \le j} p_i p_j (p_i - p_j)^2/(p_i + p_j)^2 < 1$.

The above data compression algorithm, based on Move-To-Front or Timestamp(0), was analyzed experimentally [5, 15]. In general, the algorithm can be implemented in two ways. In a *byte-level* scheme, each ASCII character in the input string is regarded as a symbol that is encoded individually. In contrast, in a *word-level* scheme each word, i.e. each longest sequence of alphanumeric and nonalphanumeric characters, represents a symbol. Albers and Mitzenmacher [5] compared Move-To-Front and Timestamp(0) based encoding on the Calgary Compression Corpus [78], which consists of files commonly used to evaluate data compression algorithms. In the byte-level implementations, Timestamp(0) achieves a better compression than Move-To-Front. The improvement is typically 6–8%. However, the byte-level schemes perform far worse than standard UNIX utilities such as pack or compress. In the word-level implementations, the compression achieved by Move-To-Front and Timestamp(0) is comparable to that of the UNIX utilities. However, in this situation, the improvement achieved by Timestamp(0) over Move-To-Front is only about 1%.

Bentley *et al.* [15] implemented a word-level scheme based on Move-To-Front that uses a linear list of limited size. Whenever the encoder reads a word from the input string that is not contained in the list, the word is written in non-coded form onto the output string. The word is inserted as new item at the front of the list and, if the current list length exceeds the allowed length, the last item of the list is deleted. Such a list acts like a cache. Bentley *et al.* tested the compression scheme with various list lengths on several text and Pascal files. If the list may contain up to 256 items, the compression achieved is comparable to that of word-based Huffman encoding and sometimes better.

Grinberg *et al.* [33] proposed a modification of Move-To-Front encoding, which they call *Move-To-Front encoding with secondary lists*. They implemented this new compression scheme but their simulations do not show an explicit comparison between Move-To-Front and Move-To-Front with secondary lists.

As mentioned in the beginning of this section, Burrows and Wheeler [17] developed a very effective data compression algorithm using self-organizing lists that achieves a better compression than Ziv-Lempel based schemes. The algorithm by Burrows and Wheeler first applies a reversible transformation to the string S. The purpose of this transformation is to group together instances of a symbol x_i occurring in S. The resulting string S' is then encoded using the Move-To-Front algorithm.

More precisely, the transformed string S' is computed as follows. Let m be the length of S. The algorithm first computes the m rotations (cyclic shifts) of S and sorts them lexicographically. Then it extracts the last character of these rotations. The k-th symbol of S' is the last symbol of the k-th sorted rotation. The algorithm also computes the index J of the original string S in the sorted list of rotations. Burrows and Wheeler gave an efficient algorithm to compute the original string S given only S' and J.

In the sorting step, rotations that start with the same symbol are grouped together. Note that in each rotation, the initial symbol is adjacent to the final symbol in the original string S. If in the string S, a symbol x_i is very often followed by x_j, then the occurrences of x_j are grouped together in S'. For this reason, S' generally has a very high locality of reference and can be encoded very effectively with Move-To-Front. The paper by Burrows and Wheeler gives a very detailed description of the algorithm and reports of experimental results. On the Calgary Compression Corpus, the algorithm outperforms the UNIX utilities compress and gzip and the improvement is 13% and 6%, respectively.

4.2 Compression based on splay trees

Splay trees have proven useful in the construction of dynamic Huffman codes, arithmetic codes and alphabetic codes [33, 40]. Furthermore they can be used as auxiliary data structure to speed up Ziv-Lempel based compression schemes [11].

Jones [40] studied dynamic Huffman codes based on splay trees. A Huffman code implicitly maintains a code tree. Associated with each leaf in the tree is a symbol of the given alphabet $\Sigma = \{x_1, \ldots, x_n\}$. The code for symbol x_i can be read by following the path from the root of the tree to the leaf containing x_i. Each left branch on the path corresponds to a 0, and each right branch corresponds to a 1. A dynamic Huffman code is obtained by splaying the code tree at certain nodes each time symbol x_i had to be encoded. Note that a Huffman code stores the information at the leaves of the tree, with the internal nodes being empty. Therefore, we may not execute regular splaying in which an accessed leaf would become an internal node, i.e. the root, of the tree. Jones presented a variant of splaying in which the set of leaves remains the same during the operation. He evaluated the algorithm experimentally and showed that the code achieves a very good compression on image data. On text and object files, the codes were not as good, in particular they performed worse than a dynamic Huffman code developed by Vitter [76].

Grinberg *et al.* [33] studied alphabetic codes based on splay trees. Consider an alphabet $\Sigma = \{x_1, \ldots, x_n\}$ in which there is an alphabetic order among the symbols $x_1, \ldots, x_n$. In an alphabetic code, the code words for the symbols have to preserve this alphabetic order. As before, a code tree is maintained. In the algorithm proposed by Grinberg *et al.*, whenever a symbol x_i had to be coded, the code tree is splayed at the parent of the leaf holding x_i.

Grinberg *et al.* analyzed the compression achieved by this scheme. Let S be an arbitrary string of length m and let m_i be the number of occurrences of symbol x_i in S. Furthermore, let $m_{min} = \min\{m_i | 1 \leq i \leq n\}$. We denote by $A_{SP}(S)$ the average number of bits to encode one symbol in S.

Theorem 46. *For any input sequence S,*

$$A_{SP}(S) \leq 2 + 3H(S) + \frac{n}{m}\log(\frac{m}{m_{min}}),$$

where $H(S) = \sum_{i=1}^{n} \frac{m_i}{m} \log(\frac{m}{m_i})$.

Grinberg *et al.* also investigated alphabetic codes based on semisplaying, see Section 3.5. Let $A_{SSP}(S)$ denote the average number of bits needed by semisplaying to encode one symbol in S.

Theorem 47. *For any input sequence S,*

$$A_{SSP}(S) \leq 2 + 2H(S) + \frac{n}{m}\log(\frac{m}{m_{min}}),$$

where $H(S) = \sum_{i=1}^{n} \frac{m_i}{m} \log(\frac{m}{m_i})$.

Thus semisplaying achieves a slightly better performance than splaying.

References

1. N. Abramson. *Information Theory and Coding.* McGraw-Hill, New York, 1983.
2. G. M. Adel'son-Vel'skii and E. M. Landis. An algorithm for the organization of information. *Soviet Math. Dokl.*, 3:1259–1262, 1962.

3. S. Albers. Improved randomized on-line algorithms for the list update problem. In *Proc. of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 412–419, 1995.
4. S. Albers. Unpublished result.
5. S. Albers and M. Mitzenmacher. Average case analyses of list update algorithms, with applications to data compression. In *Proc. of the 23rd International Colloquium on Automata, Languages and Programming*, Springer Lecture Notes in Computer Science, Volume 1099, pages 514–525, 1996.
6. S. Albers, B. von Stengel and R. Werchner. A combined BIT and TIMESTAMP algorithm for the list update problem. *Information Processing Letters*, 56:135–139, 1995.
7. B. Allen and I. Munro. Self-organizing binary search trees. *Journal of the ACM*, 25(4):526–535, Oct. 1978.
8. E.J. Anderson, P. Nash and R.R. Weber. A counterexample to a conjecture on optimal list ordering. *Journal on Applied Probability*, 19:730–732, 1982.
9. C. R. Aragon and R. G. Seidel. Randomized search trees. In *Proc. 30th Symposium on Foundations of Computer Science*, pages 540–545, 1989.
10. J. Bell and G. Gupta. Evaluation of self-adjusting binary search tree techniques. *Software—Practice & Experience*, 23(4):369–382, Apr. 1993.
11. T. Bell and D. Kulp. Longest-match string searching for ziv-lempel compression. *Software– Practice and Experience*, 23(7):757–771, July 1993.
12. S. Ben-David, A. Borodin, R.M. Karp, G. Tardos and A. Wigderson. On the power of randomization in on-line algorithms. *Algorithmica*, 11:2-14,1994.
13. J.L. Bentley, K.L. Clarkson and D.B. Levine. Fast linear expected-time algorithms for computing maxima and convex hulls. In *Proc. of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 179–187, 1990.
14. J.L. Bentley and C.C. McGeoch. Amortized analyses of self-organizing sequential search heuristics. *Communication of the ACM*, 28:404–411, 1985.
15. J.L. Bentley, D.S. Sleator, R.E. Tarjan and V.K. Wei. A locally adaptive data compression scheme. *Communication of the ACM*, 29:320–330, 1986.
16. J.R. Bitner. Heuristics that dynamically organize data structures. *SIAM Journal on Computing*, 8:82–110, 1979.
17. M. Burrows and D.J. Wheeler. A block-sorting lossless data compression algorithm. DEC SRC Research Report 124, 1994.
18. P.J. Burville and J.F.C. Kingman. On a model for storage and search. *Journal on Applied Probability*, 10:697–701, 1973.
19. R. Chaudhuri and H. Hoft. Splaying a search tree in preorder takes linear time. *SIGACT News*, 24(2):88–93, Spring 1993.
20. R. P. Cheetham, B. J. Oommen, and D. T. H. Ng. Adaptive structuring of binary search trees using conditional rotations. *IEEE Transactions on Knowledge & Data Engineering*, 5(4):695–704, 1993.
21. F.R.K. Chung, D.J. Hajela and P.D. Seymour. Self-organizing sequential search and Hilbert's inequality. *Proc. 17th Annual Symposium on the Theory of Computing*, pages 217–223, 1985.
22. D. Cohen and M. L. Fredman. Weighted binary trees for concurrent searching. *Journal of Algorithms*, 20(1):87–112, Jan. 1996.
23. R. Cole. On the dynamic finger conjecture for splay trees. part 2: Finger searching. Technical Report 472, Courant Institute, NYU, 1989.
24. R. Cole. On the dynamic finger conjecture for splay trees. In *Proc. Symposium on Theory of Computing (STOC)*, pages 8–17, 1990.

25. R. Cole, B. Mishra, J. Schmidt, and A. Siegel. On the dynamic finger conjecture for splay trees. part 1: Splay sorting $\log n$-block sequences. Technical Report 471, Courant Institute, NYU, 1989.
26. T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. McGraw-Hill, New York, NY, 1990.
27. C. A. Crane. Linear lists and priority queues as balanced binary trees. Technical Report STAN-CS-72-259, Dept. of Computer Science, Stanford University, 1972.
28. P. Elias. Universal codeword sets and the representation of the integers. *IEEE Transactions on Information Theory*, 21:194–203, 1975.
29. R. El-Yaniv. There are infinitely many competitive-optimal online list accessing algorithms. Manuscript, May 1996.
30. R.G. Gallager. *Information Theory and Reliable Communication*. Wiley, New York, 1968.
31. M.J. Golin. PhD thesis, Department of Computer Science, Princeton University, 1990. Technical Report CS-TR-266-90.
32. G.H. Gonnet, J.I. Munro and H. Suwanda. Towards self-organizing linear search. In *Proc. 19th Annual IEEE Symposium on Foundations of Computer Science*, pages, 169–174, 1979.
33. D. Grinberg, S. Rajagopalan, R. Venkatesan and V.K. Wei. Splay trees for data compression. In *Proc. of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 522–530, 1995.
34. W.J. Hendricks. An extension of a theorem concerning an intersting Markov chain. *Journal on Applied Probability*, 10:886–890, 1973.
35. W.J. Hendricks. An account of self-organizing systems. *SIAM Journal on Computing*, 5:715–723, 1976.
36. J.H. Hester and D.S. Hirschberg. Self-organizing linear search. *ACM Computing Surveys*, 17:295–312, 1985.
37. G.H. Hardy, J.E. Littlewood and G. Polya. *Inequalities*. Cambridge University Press. Cambridge, England, 1967.
38. S. Irani. Two results on the list update problem. *Information Processing Letters*, 38:301–306, 1991.
39. S. Irani. Corrected version of the SPLIT algorithm. Manuscript, January 1996.
40. D. W. Jones. Application of splay trees to data compression. *Communications of the ACM*, 31(8):996–1007, Aug. 1988.
41. Y.C. Kan and S.M. Ross. Optimal list orders under partial memory constraints. *Journal on Applied Probability*, 17:1004–1015, 1980.
42. R. Karp and P. Raghavan. From a personal communication cited in [60].
43. W. F. Klostermeyer. Optimizing searching with self-adjusting trees. *Journal of Information & Optimization Sciences*, 13(1):85–95, Jan. 1992.
44. D. E. Knuth. Optimum binary search trees. *Acta Informatica*, pages 14–25, 1971.
45. D. E. Knuth. *The Art of Computer Programming, Sorting and Searching*, volume 3. Addison-Wesley, Reading, MA, 1973.
46. K. Kulik II and D. Wood. A note on some tree similarity measures. *Information Processing Letters*, 15:39–42, 1982.
47. K. Lam, M.K. Sui and C.T. Yu. A generalized counter scheme. *Theoretical Computer Science*, 16:271–278, 1981.
48. J. M. Lucas. The rotation graph of binary trees is hamiltonian. *Journal of Algorithms*, 8(4):503–535, Dec. 1987.
49. J. M. Lucas. Arbitrary splitting in splay trees. Technical Report DCS-TR-234, Rutgers University, 1988.

50. J. M. Lucas. Canonical forms for competitive binary search tree algorithms. Technical Report DCS-TR-250, Rutgers University, 1988.
51. F. Luccio and L. Pagli. On the upper bound on the rotation distance of binary trees. *Information Processing Letters*, 31(2):57–60, Apr. 1989.
52. E. Makinen. On the rotation distance of binary trees. *Information Processing Letters*, 26(5):271–272, Jan. 1988.
53. M.S. Manasse, L.A. McGeoch and D.D. Sleator. Competitive algorithms for on-line problems. In *Proc. 20th Annual ACM Symposium on Theory of Computing*, pages 322-33, 1988.
54. C. Martel. Self-adjusting multi-way search trees. *Information Processing Letters*, 38(3):135–141, May 1991.
55. J. McCabe. On serial files with relocatable records. *Operations Research*, 12:609–618, 1965.
56. K. Mehlhorn. Nearly optimal binary search trees. *Acta Informatica*, 5:287–295, 1975.
57. K. Mehlhorn. Dynamic binary search. *SIAM Journal on Computing*, 8(2):175–198, 1979.
58. K. Mehlhorn. *Data Structures and Algorithms*. Springer-Verlag, New York, 1984. (3 volumes).
59. G. Port and A. Moffat. A fast algorithm for melding splay trees. In *Proc. Workshop on Algorithms and Data Structures (WADS '89)*, pages 450–459, Berlin, West Germany, 1989. Springer-Verlag.
60. N. Reingold, J. Westbrook and D.D. Sleator. Randomized competitive algorithms for the list update problem. *Algorithmica*, 11:15–32, 1994.
61. N. Reingold, J. Westbrook. Optimum off-line algorithms for the list update problem. Technical Report YALEU/DCS/TR-805, Yale University, 1990.
62. R. Rivest. On self-organizing sequential search heuristics. *Communication of the ACM*, 19:63–67, 1976.
63. G. Schay Jr. and F.W. Dauer. A probabilistic model of a self-organizing file system. *SIAM Journal on Applied Mathematics*, 15:874–888, 1967.
64. M. Sherk. Self-adjusting k-ary search trees. *Journal of Algorithms*, 19(1):25–44, July 1995.
65. D.D. Sleator and R.E. Tarjan. Amortized efficiency of list update and paging rules. *Communication of the ACM*, 28:202–208, 1985.
66. D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32:652–686, 1985.
67. D. D. Sleator, R. E. Tarjan, and W. P. Thurston. Rotation distance, triangulations, and hyperbolic geometry. In *Proc. 18th Symposium on Theory of Computing (STOC)*, pages 122–135, 1986.
68. R. Sundar. Twists, turns, cascades, deque conjecture, and scanning theorem. In *Proc. 30th Symposium on Foundations of Computer Science (FOCS)*, pages 555–559, 1989.
69. R. Sundar. Twists, turns, cascades, deque conjecture, and scanning theorem. Technical Report 427, Courant Institute, New York University, Jan. 1989.
70. B. Teia. A lower bound for randomized list update algorithms. *Information Processing Letters*, 47:5–9, 1993.
71. R. E. Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, PA., 1983.
72. R.E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6:306–318, 1985.

73. R. E. Tarjan. Sequential access in splay trees takes linear time. *Combinatorica*, 5(4):367–378, 1985.

74. A. Tenenbaum. Simulations of dynamic sequential search algorithms. *Communication of the ACM*, 21:790–797, 1978.

75. A.M. Tenenbaum and R.M. Nemes. Two spectra of self-organizing sequential search. *SIAM Journal on Computing*, 11:557–566, 1982.

76. J.S. Vitter. Two papers on dynamic Huffman codes. Technical Report CS-85-13. Brown University Computer Science, Providence. R.I. Revised December 1986.

77. R. Wilber. Lower bounds for accessing binary search trees with rotations. *SIAM Journal on Computing*, 18(1):56–67, Feb. 1989.

78. I.H. Witten and T. Bell. The Calgary/Canterbury text compression corpus. Anonymous ftp from ftp.cpsc.ucalgary.ca : /pub/text.compression/corpus/ text.compression.corpus.tar.Z.