DIMACS Technical Report 2003-46
January 2004

Adaptive Sorting with AVL Trees

by

Amr Elmasry
Computer Science Department, Alexandria University,
Alexandria, Egypt.
elmasry@cs.rutgers.edu

DIMACS is a partnership of Rutgers University, Princeton University, AT&T Labs-Research, Bell Labs, NEC Research Institute and Telcordia Technologies (formerly Bellcore).

DIMACS was founded as an NSF Science and Technology Center, and also receives support from the New Jersey Commission on Science and Technology.

ABSTRACT

A new adaptive sorting algorithm is introduced. The new implementation relies on using the traditional AVL trees, and has the same performance limitations. More precisely, the number of comparisons performed by our algorithm, on an input sequence of length n that has I inversions, is at most $1.44n \lg \frac{I}{n} + O(n)$ [1]. Empirical results imply an average performance of about $1.02n \lg \frac{I}{n}$ comparisons for random sequences. Our algorithm runs in time $O(n \lg \frac{I}{n})$ and is practically efficient and easy to implement.

[1] $\lg x$ is the maximum of $\log_2 x$ and 1.

1 Introduction

An adaptive sorting algorithm is a sorting algorithm that benefits from the presortedness in the input sequence. In the literature there are plenty of adaptive sorting algorithms. One of the commonly recognized measures of presortedness is the number of inversions in the input sequence [11]. The number of inversions is the number of pairs of input items in the wrong order. For an input sequence X, the number of inversions of X, *Inv(X)*, is defined

$$Inv(X) = |\{(i,j) \mid 1 \leq i < j \leq n \text{ and } x_i > x_j\}|.$$

An adaptive sorting algorithm is optimal with respect to the number of inversions when it runs in $O(n \lg \frac{Inv(X)}{n})$ [9]. Unfortunately, most of the known theoretically optimal adaptive sorting algorithms are not practical and not easy to implement [9, 18, 3, 17, 15].

The number of comparisons is considered one of the main analytical measures to compare different sorting algorithms. The number of comparisons performed by an *Inv*-optimal sorting algorithm is $cn \lg \frac{Inv(X)}{n} + O(n)$ comparisons, for some constant $c \geq 1$. Among the adaptive sorting algorithms Splitsort [13] and Adaptive Heapsort [14] require $c = 2.5$. Finger trees [9, 17], though not practical, requires $c = 2$. Trinomialsort [6] requires $c = 1.89$. Recently, Elmasry and Fredman [7] introduced an adaptive sorting algorithm with $c = 1$, and hence achieving the information theoretic lower bound for the number of comparisons, with respect to the number of inversions in the input sequence.

The task of achieving the optimal number of comparisons is therefore accomplished, still with the practicality issue being open. The algorithm in [7] uses near optimal trees [1], which is a practically complicated structure with an overhead in the running time for the maintenance of these trees. The other operations performed by the algorithm in [7] (splits, combines, coalescing and reduction operations) contribute with another overhead factor, making the algorithm not fully practical. Namely, the time bound for the operations, other than the comparisons, performed by the algorithm in [7] is $\Theta(n \lg \frac{Inv(X)}{n})$.

Among the adaptive sorting algorithms Splitsort [13], Adaptive Heapsort [14] and Trinomialsort [6] are the most promising from the practical point of view. As a consequence of the dynamic finger theorem for splay trees (see Cole [5]), the splay trees of Sleator and Tarjan [21] provide a simplified substitute for finger trees to achieve the same asymptotic run time. Moffat et al. [19] performed experiments showing that Splaysort is efficient in practice.

We introduce a new adaptive sorting algorithm that uses AVL trees [2]. Our new algorithm requires $c = 1.44$ in the worst case, while it achieves a value of c

very close to 1 (optimal) from the practical point of view [10, 12]. This result is a direct consequence of the nature of the well-known search trees known as AVL trees. The worst-case behavior of the AVL trees is achieved when the tree is a Fibonacci tree, a case that rarely pops-up in practice. The space utilized by our algorithm is $O(n)$.

The contribution of this paper is to introduce a practically efficient adaptive sorting algorithm, and to show that apart from the comparisons, the other operations performed by this algorithm take total time $O(n)$; a fact that does not hold for other efficient adaptive sorting algorithms. For example, Trinomialsort, Adaptive Heapsort and Splitsort would perform a non-linear number of moves. We expect our new algorithm to be efficient, fast in practice, and easy to implement.

Other methods that use AVL trees to implement adaptive sorting algorithms are the algorithm of Mehlhorn [18], and the finger trees of Tsakalidis [23]. These two algorithms require augmenting the AVL trees with extra information that makes the implementation non-practical, with a larger constant for the number of comparisons.

Several authors have proposed other measures of presortedness and proposed optimal algorithms with respect to these measures [8, 4, 14, 15]. Mannila [16] formalized the concept of presortedness. He studied several measures of presortedness and introduced the concept of optimality with respect to these measures. Petersson and Moffat [20] related all of the various known measures in a partial order and established new definitions with respect to the optimality of adaptive algorithms.

2 The algorithm

Consider the following method for inserting y into a sorted sequence, $x_1 < x_2 < \cdots < x_{n-1}$. For a specified value r, we first perform a linear search among the items, $x_r, x_{2r}, \cdots, x_{r\lfloor n/r \rfloor}$, to determine the interval of length r among $x_1 < x_2 < \cdots < x_{n-1}$ into which y falls. Next, we perform a binary search within the resulting interval of length r to determine the precise location for y. If y ends up in position i, then $\frac{i}{r} + \lg r + O(1)$ comparisons suffice for this insertion. Using the strategy of successively inserting the items $x_1, \cdots, x_n$ in reverse order (into an initially empty list), the total number of comparisons required to sort would be bounded by $\sum_i (\frac{i}{r} + \lg r + O(1)) = \frac{Inv(X)}{r} + n \lg r + O(n)$. If we use $r = \frac{Inv(X)}{n}$, the required bound on the number of comparisons follows. Unfortunately, we cannot use this value of r, since the number of inversions is not known beforehand. Instead, we choose r to be a dynamic quantity that is maintained as insertions take place; r

is initially chosen to be $r_1 = 1$, and during the kth insertion, $k > 1$, r is given by $r_k = \frac{1}{k-1}\sum_{1\le j<k} i_j$, where i_j is the final position of the element j after the jth insertion. The quantity r_k is at least 1. For completeness, we give the proof of the following lemma, which is in [7].

lemma 1 *The preceding insertion sort algorithm performs at most* $n\lg\frac{Inv(X)}{n} + O(n)$ *comparisons to sort an input* X *of length* n.

Proof: Define $E(k)$, the excess number of comparisons performed during the first k insertions, to be the actual number performed minus $k\lg(\frac{1}{k}\sum_{1\le j\le k} i_j)$. We demonstrate that $E(k) = O(n)$ when $k = n$. We proceed to estimate $E(k+1) - E(k)$.

Let r' denote the average, $\frac{1}{k+1}\sum_{1\le j\le k+1} i_j$, and let r denote the corresponding quantity, $\frac{1}{k}\sum_{1\le j\le k} i_j$. Then

$$\begin{aligned} E(k+1) - E(k) &= \lg r + i_{k+1}/r - (k+1)\lg r' + k\lg r + O(1) \\ &= i_{k+1}/r + (k+1)(\lg r - \lg r') + O(1). \end{aligned} \qquad (1)$$

Now write $r' = (k\cdot r + i_{k+1})/(k+1) = (k/(k+1))\cdot r\cdot g$, where $g = 1 + i_{k+1}/(k\cdot r)$. Substituting into (1) this expression for r', we obtain

$$E(k+1) - E(k) = i_{k+1}/r + (k+1)(\lg(k+1)/k) - (k+1)\lg g + O(1)\,.$$

The term $(k+1)(\lg(k+1)/k)$ is $O(1)$, leaving us to estimate $i_{k+1}/r - (k+1)\lg g$. We have two cases: (i) $i_{k+1} \le k\cdot r$ and (ii) $i_{k+1} > k\cdot r$

For case (i), using the fact that $\lg(1+x) \ge x$ for $0 \le x \le 1$, we find that $i_{k+1}/r - (k+1)\lg g \le 0$ (since $\lg g \ge i_{k+1}/(k\cdot r)$ for this case). For case (ii), we bound $i_{k+1}/r - (k+1)\lg g$ from above using i_{k+1}. But the condition for case (ii), namely $i_{k+1} > k\cdot r = \sum_{1\le j\le k} i_j$, implies that the sum of these i_{k+1} estimates (over those k for which case (ii) applies) is at most twice the last such term, which is bounded by n. Since $E(1) = 0$, we conclude that $E(n) = O(n)$. □

To convert the above construction to an implementable algorithm with total running time $O(n\lg\frac{Inv(X)}{n})$, we utilize the considerable freedom available in the choice of the r_k values in the above construction, while preserving the result of the preceding Lemma. Let $\alpha \ge 1$ be an arbitrary constant. If we replace our choice for r_k in the above algorithm by any quantity s_k satisfying $r_k \le s_k \le \alpha\cdot r_k$, then the above lemma still holds; the cost $i_k/s_k + \lg s_k + O(1)$ of a single insertion cannot grow by more than $O(1)$ as s_k deviates from its initial value r_k while remaining in the indicated range.

Efficient Implementation

At each insertion point, the previously inserted items are organized into a list of consecutive bands from left to right. Every band has $1, 2,$ *or* 3 AVL trees. An AVL tree is organized as a search tree with the data items stored only in the leaves of the tree. A rank value is assigned to every band. The trees of a band with rank h will have heights equal to h, except for at most one tree that may have height equal to $h - 1$. We call a tree that has a height one less than the rank of its band a *short* tree. We call a band that has a short tree an s-band. These conditions are referred to as the rank conditions. At any time of the algorithm, the ranks of the bands form an increasing consecutive sequence $m, m + 1, m + 2, \ldots$ from left to right, with the value of m changing through the algorithm. This is referred to as the monotonicity condition.

With every insertion, the relevant tree is first identified by employing a linear search through the list of trees from left to right. After each insertion, the band list may require reorganization, though on a relatively infrequent basis. The details of this implementation is facilitated by defining the following operations:

1. Split: An AVL tree of height h can be split in constant time into two trees, one of height $h - 1$ and the other of height $h - 1$ *or* $h - 2$, by removing the root node of the given tree.

2. Combine: Two AVL trees, one of height $h - 1$ and the other of height $h - 1$ *or* $h - 2$, can be combined in constant time to form an AVL tree of height h by adding a new root node.

3. Find largest: The value of the largest member of a given tree can be accessed in constant time.

4. Tree-insertion: An insertion of a new value into an AVL tree of height h can be performed with at most h comparisons, and in time $O(h)$.

Consider the cost of the single operation: inserting y into a sorted sequence $S = x_1 < x_2 < \cdots < x_{n-1}$. If S is organized, as mentioned above, in a list of trees, and y belongs to the ith tree, which is of height h, then the insertion requires no more than $i + h$ comparisons.

As a result of an insertion, the height of the trees may increase and the rank conditions are to be maintained. Such a case arises when the height of a tree, in a band of rank h, becomes $h + 1$. This tree is split into two trees. If, as a result of this split, we now have two short trees in this band, these two trees are combined (If these two trees are not adjacent, the heights of the trees in this band must have either the pattern $h-1, h, h-1$ *or* $h-1, h, h, h-1$. In either case, we split the middle trees, whose heights are h, then combine every adjacent pair of the trees. This accounts for at most 3 splits and 3 combines.). Otherwise, if the number of trees of this band becomes 4, the two right trees are combined and the combined tree is moved to become the left tree of the next higher band. This operation is referred to as a *promote* operation. This combine/promote may be repeated several times through consecutive bands. We call such a process a propagating promotion.

As in the preceding lemma, we define $r_k = \frac{1}{k-1} \sum_{1 \leq j < k} i_j$, for $k > 1$. Besides enforcing the rank conditions, we maintain the additional condition that just prior to the kth insertion the rank m of the leftmost band satisfies

$$m = \lceil \log_\theta r_k \rceil + 1, \tag{2}$$

Where θ is a parameter of the algorithm whose value will be analyzed and determined later. The value of θ should satisfy $1 < \theta \leq 2$.

If, as a result of an insertion, r_k grows such that $m = \lceil \log_\theta r_k \rceil$ (note that r_k may grow by at most 1), then a *coalescing* operation is performed. The trees of the leftmost interval are combined to form 1 *or* 2 trees that are promoted to the next band whose rank is $m+1$ (if there were 3 trees in the leftmost band, 2 of them are combined including the short tree if it exists). If there was only one short tree to be promoted, the two leftmost trees of the band whose rank is $m+1$ will have heights $m-1, m$ or $m-1, m+1$. In the first case, these two trees are combined. In the second case, the tree with height $m+1$ is first split once or twice before combines take place (for a total of at most 2 splits and 2 combines). If, as a result of this promotion, two short trees exist in the band whose rank is $m+1$, these two trees are combined (as above). If the number of trees of this band becomes 4 *or* 5, a combine/promote is performed and repeated as necessary through consecutive bands. After this coalescing operation, the rank of the leftmost band is $m+1$. As a special case that does not affect the bounds on the operations of the algorithm, the coalescing operation is skipped when there is only one band and the number of nodes is not enough to perform the coalescing operation while maintaining the rank conditions.

If r_k drops such that $m = \lceil \log_\theta r_k \rceil + 2$ (note that r_k may drop by at most 1), then a *reduction* operation is performed. A new leftmost band with rank $m-1$ is created. The leftmost tree of the old leftmost band (the band with rank m) is moved to the new band. We call this operation a *demote* operation. If this tree has height m, then it is split. If, as a result of the demotion, the band whose rank is m now has no trees, the leftmost tree of the band whose rank is $m+1$ is demoted and split if necessary. This demote/split may be repeated for several times through consecutive bands. We call such a process a propagating demotion.

Note that the reduction and the coalescing operations serve to preserve the rank conditions and the monotonicity conditions as well as (2).

Analysis

lemma 2 *Our algorithm performs at most* $n \log_\phi \frac{Inv(X)}{n} + O(n)$ *comparisons to sort an input* X *of length* n *(*ϕ *is the golden ratio* $= \frac{1+\sqrt{5}}{2}$*).*

Proof: In view of Lemma 1, it suffices to show that a given insertion, arriving in position L, requires at most

$$\log_\phi s_k + L/s_k + O(1) \tag{3}$$

comparisons, where $r_k \leq s_k \leq \alpha \cdot r_k$ and $\alpha \geq 1$ is an arbitrary constant.

Let m be the rank of the leftmost band, and $m2$ be the rank of the band that has the tree into which the newly inserted item falls, and let i be the position of this tree (number of the tree counting the trees from the left), so that the total insertion cost is at most $i + m2$. As a result of the rank and monotonicity conditions, we have $i \geq m2 - m + 1$. Next, we bound L from below as follows. Contributing to L, there is at least 1 tree in each of the bands with ranks from m to $m2 - 1$. There is another $i - d - 1$ (where $d = m2 - m \geq 0$) trees of heights at least $m - 1$. For any AVL tree, the size of a tree of height h is at least ϕ^h. It follows that:

$$\begin{aligned} L &> (i - d - 1)\phi^{m-1} + \sum_{i=m-1}^{m2-2} \phi^i \\ &> (i - d - 2)\phi^{m-1} + \phi^{m2-1} \end{aligned}$$

Choosing our parameter to be $\theta = \phi$, it follows from (2) that $\phi^{m-1} = s_k$. Hence

$$\frac{L}{s_k} > (i - d - 2) + \phi^d$$

The lemma follows by indicating the following relation, which implies (3)

$$i + m2 < \frac{L}{s_k} + m + 2d - \phi^d + 2$$

□

lemma 3 *The time spent by our algorithm, in performing operations other than comparisons, is $O(n)$.*

Proof: The primitive operations, that the algorithm performs other than the comparisons, are the *spit* and *combine* operations. Each of these operations requires a constant time. Excluding the propagating promotion and demotion, the number of splits and combines that accompany each insertion, coalescing or reduction is a constant. Hence, the only operations that need to be investigated are the propagating promotions and demotions.

We use a potential function [22] to derive the linear bounds on these operations. Let N_s be the number of s-bands and let N_{odd} be the number of bands that have 1 *or* 3 trees. Let Φ^i be the potential function after the *ith* insertion, such that $\Phi^i = c_1 N_{odd} + c_2 N_s$, where c_1 and c_2 are constants to be determined and $c_1 > c_2$. The value of Φ^0 is 0, and the value of $\Phi^n = O(n)$. What we need to show is that the difference in potential when added to the actual amount of work during the *ith* insertion is bounded by a constant.

Consider the case that during the $i + 1$ insertion a propagating promotion that involves t bands takes place. Assume first that the initiative of this propagating promotion is an insertion that causes a height of a tree to become $h + 1$ in a band of rank h. As a result of a promotion in a band, the number of trees in this band should have been 3, and becomes 2 after the promotion. This should be the case for the $t - 1$ bands that propagate the promotion, accounting for a decrease of $t - 1$ in the value of N_{odd}. In the last band, the opposite may take place and N_{odd} may increase be 1 as a result. Except for the first band, into which the newly inserted item falls, the number of s-bands may only decrease as a result of any of these promotions. Hence, the amortized cost of the propagating promotion in this case is bounded by $O(t) - c_1(t - 2) + c_2$. By selecting c_1 greater than the constant involved in the O() notation in this bound, the amortized cost of this operation is a constant. The analysis is similar if the initiative for the propagating promotion is a coalescing operation. The only difference is the first band that gets promoted trees, where the number of trees in this band may remain the same (either 2 or 3). This band may also be converted to an s-band as a result of this promotion. This leads to a bound of $O(t) - c_1(t - 3) + c_2$, which is a constant as well.

Consider the case that during the $i + 1$ insertion a reduction operation is performed. Assume that this reduction initiates a propagating demotion that involves t bands. A new band is created that may get 1 tree (increasing N_{odd} by 1), or 2 trees one of them may be short (increasing N_s by 1). For each of the next $t - 2$ bands that involves a demotion, the number of trees in each of these bands should have been 1 before this propagating demotion. If a demoted tree is not short, it is split resulting in 2 trees. This causes N_{odd} to decrease by 1, while N_s may increase by 1. On the other hand, if a demoted tree is short, the number of trees in the corresponding band remains 1 after the demotion, while the number of s-bands decreases by 1 causing N_s to decrease by 1. In the last band, the opposite may take place, and N_{odd} may increase by 1 as a result. Hence, the amortized cost of the propagating demotion is bounded by $O(t) - (c_1 - c_2)(t - k - 2) - c_2 k + 2c_1$, for some $k \leq t - 2$. By selecting c_2 and $c_1 - c_2$ greater than the constant in the O() notation in this bound, the amortized cost of this operation is a constant. □

We have thus established the following theorem.

Theorem 1 *The preceding insertion sort algorithm sorts an input X of length n in time $O(n \log \frac{Inv(X)}{n})$, and performs at most $n \log_\phi \frac{Inv(X)}{n} + O(n)$ comparisons. The space requirement for the algorithm is $O(n)$.*

Tuning the parameter

In the above analysis, to prove the bound for the number of comparisons, we have chosen the parameter $\theta = \phi$. In practice, this value is a too conservative value to insure the worst-case behavior. For random presorted sequences the performance of AVL trees is very efficient, and empirical data shows that the average height of an AVL tree of n nodes is about $1.02 \log n$ [10, 12]. This motivates using a larger value of θ. Knowing that the constant factor in the height of an average AVL tree is close to 1, the parameter θ can be chosen to be closer to 2. The proof of Lemma 2 works for the average case to prove the 1.02 constant factor, by using $\log_\theta 2 = 1.02$.

References

[1] A. Andersson and T. W. Lai. *Fast updating of well-balanced trees.* In Proc. Scandinavian Workshop on Algorithm Theory, Springer Verlag (1990), 111–121.

[2] G. Adelson-Velskii and E. Landis. *On an information organization algorithm.* Doklady Akademia Nauk SSSR, 146(1962), 263-266.

[3] M. Brown and R. Tarjan. *Design and analysis of data structures for representing sorted lists.* SIAM J. Comput. 9 (1980), 594–614.

[4] S. Carlsson, C. Levcopoulos and O. Petersson. *Sublinear merging and natural Mergesort.* Algorithmica 9 (1993), 629–648.

[5] R. Cole. *On the dynamic finger conjecture for splay trees. Part II: The proof.* SIAM J. Comput. 30 (2000), 44–85.

[6] A. Elmasry. *Priority queues, pairing and adaptive sorting.* 29th International Colloquium for Automata, Languages and Programming. In LNCS 2380 (2002), 183-194.

[7] A. Elmasry and M. Fredman. *Adaptive sorting and the information theoretic lower bound.* Symposium on Theoretical Aspects of Computer Science. In LNCS 2607 (2003), 654-662.

[8] V. Estivill-Castro and D. Wood. *A new measure of presortedness.* Infor. and Comput. 83 (1989), 111–119.

[9] L. Guibas, E. McCreight, M. Plass and J. Roberts. *A new representation of linear lists.* ACM Symp. on Theory of Computing 9 (1977), 49–60.

[10] L. Guibas and R. Sedgewick. *A dichromatic framework for balanced trees.* In Foundations of Computer Science (1978), 8-21.

[11] D. Knuth. *The art of Computer programming. Vol III: Sorting and Searching.* Addison-wesley, second edition (1998).

[12] P. Karlton, S. Fuller, R. Scroggs and E. Kaehler. *Performance of height-balanced trees.* Information Retrieval and Language Processing 19(1) (1976), 23-28.

[13] C. Levcopoulos and O. Petersson. *Splitsort - An adaptive sorting algorithm.* Information Processing Letters 39 (1991), 205–211.

[14] C. Levcopoulos and O. Petersson. *Adaptive Heapsort.* J. of Algorithms 14 (1993), 395–413.

[15] C. Levcopoulos and O. Petersson. *Exploiting few inversions when sorting: Sequential and parallel algorithms.* Theoretical Computer Science 163 (1996), 211–238.

[16] H. Mannila. *Measures of presortedness and optimal sorting algorithms.* IEEE Trans. Comput. C-34 (1985), 318–325.

[17] K. Mehlhorn. *Data structures and algorithms. Vol.1. Sorting and Searching.* Springer-Verlag, Berlin/Heidelberg. (1984)

[18] K. Mehlhorn. *Sorting presorted files.* Proc. Of the 4th GI Conference on Theory of Computer Science. LNCS 67 (1979), 199–212.

[19] A. Moffat, G. Eddy and O. Petersson. *Splaysort: fast, verstile, practical.* Softw. Pract. and Exper. Vol 126(7) (1996), 781–797.

[20] O. Petersson and A. Moffat. *A framework for adaptive sorting.* Discrete App. Math. 59 (1995), 153–179.

[21] D. Sleator and R. Tarjan. *Self-adjusting binary search trees.* J. ACM 32(3) (1985), 652–686.

[22] R. Tarjan. *Amortized computational complexity.* SIAM J. Alg. Disc. Meth. 6 (1985), 306-318.

[23] A. Tsakalidis. *AVL-trees for localized search.* Information and Control 67 (1985), 173–194.