Inversion-Sensitive Sorting Algorithms in Practice

AMR ELMASRY
Max-Planck Institut für Informatik
and
ABDELRAHMAN HAMMAD
Alexandria University

We study the performance of the most practical inversion-sensitive internal sorting algorithms. Experimental results illustrate that adaptive AVL sort consumes the fewest number of comparisons unless the number of inversions is less than 1%; in such case Splaysort consumes the fewest number of comparisons. On the other hand, the running time of Quicksort is superior unless the number of inversions is less than 1.5%; in such case Splaysort has the shortest running time. Another interesting result is that although the number of cache misses for the cache-optimal Greedysort algorithm was the least, compared to other adaptive sorting algorithms under investigation, it was outperformed by Quicksort.

Categories and Subject Descriptors: E.0 [**Data**]: General; E.1 [**Data Structures**]: Arrays; Lists, stacks and queues; Trees; E.2 [**Data Storage Representations**]: Contiguous representations; Linked representations; E.5 [**Files**]: Sorting/searching; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Sorting and searching; G.3 [**Probability and Statistics**]: Experimental design

General Terms: Algorithms, Design, Experimentation, Performance, Theory

Additional Key Words and Phrases: Adaptive sorting, Inversions

ACM Reference Format:

Elmasry, A. and Hammad, A. 2008. Inversion-sensitive sorting algorithms in practice. ACM J. Exp. Algor. 13, Article 1.11 (December 2008), 18 pages. DOI = 10.1145/1412228.1455267 http://doi.acm.org/10.1145/1412228.1455267

1. INTRODUCTION

A sorting algorithm is considered adaptive if it performs better for sequences having a high degree of existing order. Such algorithms require less comparisons

Authors' addresses: A. Elmasry, Max-Planck Institut für Informatik, Saarbrücken, Germany. Supported by an Alexander von Humboldt Fellowship; email: elmasry@mpi-inf.mpg.de.; A. Hammad, Department of Computer and Systems Engineering, Alexandria University, Alexandria, Egypt.

A preliminary version of the paper appeared in the 4th International Workshop on Efficient and Experimental Algorithms (WEA) 2005.

and running time to perform the sorting for such input instances. In the literature, many adaptive sorting algorithms have been proposed and many measures of disorder have been considered. Mannila [1985] formalized the concept of presortedness; he studied several measures of presortedness and introduced the concept of optimality with respect to these measures.

One of the main measures of presortedness is the number of inversions in the input sequence [Knuth 1998]. The number of inversions is the number of pairs in the wrong order. More precisely, for an input sequence $X = \langle x_1, x_2, \ldots, x_n \rangle$, the number of inversions is defined as

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

An adaptive sorting algorithm that runs in $O(n \log \frac{Inv(X)}{n} + n)$ is considered inversion-optimal. This follows from the fact that the information-theoretic lower bound for any sorting algorithm with respect to the parameters n and Inv is $\Omega(n \log \frac{Inv(X)}{n} + n)$ [Guibas et al. 1977].

The finger tree [Guibas et al. 1977] was the first data structure that utilized the existing presortedness in the input sequence. The fact that the underlying data structure is pretty complicated, and requires a lot of pointer manipulations, makes finger trees impractical. Afterwords, Mehlhorn [1979] introduced a sorting algorithm that achieves the above bound. The other inversion-optimal sorting algorithms include Blocksort [Levcopoulos and Petersson 1996], which runs in place, and the tree-based Mergesort [Moffat et al. 1998], which is optimal with respect to several other measures of presortedness. Among the inversion-optimal sorting algorithms, Splitsort [Levcopoulos and Peterson 1991] and adaptive Heapsort [Levcopoulos and Peterson 1993] are known to be promising from the practical point of view; both algorithms require at most $2.5n \log n$ comparisons. Splaysort, sorting by repeated insertion in a splay tree [Sleator and Tarjan 1985], was also proved to be inversion-optimal [Cole 2000]. Moffat et al. [1996] performed experiments showing that Splaysort is practically efficient. See Estivill-Castro and Wood [1992] and Moffat and Petersson [1992] for a nice survey of adaptive sorting algorithms.

Later studies oriented toward improving the number of comparisons of inversion-optimal sorting algorithms include: Binomialsort [Elmasry 2002], adaptive AVL sort [Elmasry 2004], and the algorithms in Elmasry and Fredman [2008], which achieve the lower bound of $n \log_2 \frac{Inv(X)}{n} + O(n)$ comparisons. This bound on the number of comparisons can be also achieved utilizing the multipartite queues of [Elmasry et al. in press; Elmasry et al. 2004] in the adaptive Heapsort algorithm.

In another line of research [Brodal et al. 2005b], I/O-optimal cache-aware and cache-oblivious adaptive sorting algorithms were introduced; one of these cache-oblivious algorithms is Greedysort. These algorithms, in addition to being inversion-optimal, perform asymptotically optimal number of cache misses even when the cache size is not given as a parameter. Expectedly, they would work nicely in practice especially when the cache size is a bottleneck.

On the other hand, Quicksort introduced by Hoare [1961] is considered the most practical sorting algorithm. Several empirical studies illustrated that Quicksort is very efficient in practice [Dromey 1984; Wainwrigh

1985]. The Quicksort algorithm and its variants also demonstrated an efficient performance when applied to nearly sorted lists [Cook and Kim 1980]. Brodal et al. [2005a] showed that the expected number of swaps performed by randomized Quicksort is $O(n \log \frac{Inv(X)}{n} + n)$, indicating that it is inversion-optimal with respect to the number of swaps.

In this article, we are interested in demonstrating a practical study of such inversion-optimal sorting algorithms. Our objective is to conclude which of these algorithms would be a good candidate in practice, and which one we would use under different circumstances. To perform our experimentations, we select the algorithms that we think are the most promising from the practical point of view among those introduced in the literature; these include: adaptive Heapsort, Splitsort, Greedysort, Splaysort, and adaptive AVL sort. We also compare the performance of these adaptive algorithms with randomized Quicksort.

This article is organized as follows: in the next section we show how our experiments are performed, illustrating the way we generate the input, set the test data, and the way the output is presented. Next, we proceed by exploring alternatives for each of the studied algorithms, together with different settings for each of them, selecting the best settings for each. Finally, we conclude the paper with a comparison between our tuned main algorithms, and give our final comments and conclusions.

2. EXPERIMENTAL SETTINGS

In all our experiments, the input sequence is randomly generated in a way that the expected number of inversions is controlled. We start with a sorted sequence $\langle 1, 2, \ldots, n \rangle$ and perform two phases of permutations; we call the inversions resulting from these two phases local and global inversions. Given a parameter m, the goal is to permute the sorted sequence such that the expected number of the resulting inversions is $\Theta(n \cdot m)$. For local inversions, the sorted sequence is broken into $\lceil n/m \rceil$ consecutive blocks of m elements each (except possibly the last block), and the elements of each block are randomly permuted. For global inversions, the resulting sequence is broken into m consecutive blocks of $\lceil n/m \rceil$ elements each (except possibly the last block). From each block one element is selected at random, and these m elements are randomly permuted. A value of $m = 0$ means that the input sequence is sorted in ascending order. A small value of m, with respect to n, means that the input sequence is almost sorted. A value of m, which is as big as n, means that the input sequence becomes random.

Note that the way we produce the input guarantees that the elements are distinct. Allowing for duplicates would introduce another factor of presortedness in the input; for example a list with a single repeated element is obviously sorted. Note that any input that has duplicates can be mapped to one of the $n!$ permutations of our input sample space, when assuming that an element that comes before its duplicate in the input must come before it in the output as well. A stable sorting algorithm will behave exactly the same for both inputs. An unstable sorting algorithm will find the job even easier when dealing with duplicate inputs. Another issue is that we only applied the algorithms to 4-byte integers. The performance may change if we apply the algorithms to other data types. For other structured data types, we expect the running time to be more influenced by the number of comparisons performed and by memory bottlenecks.

We fixed the value of n at 2^{22}. All the experiments are performed on an Intel® Pentium® 4, 2.8 GHz machine, with 8 Kb L1 cache memory, 512 Kb L2 cache memory, and 1 Gb RAM, running Windows XP platform. With such memory capacity, there was no need to use virtual memory. We implemented all the algorithms in C++ using Borland C++ Builder version 5.

Three primary outcomes are measured; the running time, the number of comparisons performed, and the number of cache misses. The measured running times are CPU times, ignoring the time for I/O operations. This was done using Windows APIs to get the CPU times for both the kernel and user modes. The number of cache misses are measured using Intel® Pentium® 4 performance counters and Intel® VTune™ Performance Analyzer 7.2.

In all graphs, we plot one of the measures versus $\log_2 m$. For the plots demonstrating the comparisons or the running times, each point is the average of 100 sample runs after omitting the most biased ten values out of 110 values. For the plots demonstrating the cache misses, the average of ten sample runs was taken.

We first proceed by investigating different alternatives and implementation issues for each algorithm, and conclude by selecting the best implementation for each algorithm and compare these with each other.

3. ADAPTIVE HEAPSORT

Basic Algorithm

Given a sequence $X = \langle x_1, \ldots, x_n \rangle$, the corresponding Cartesian tree [Vuillemin 1980] is a binary tree with root $x_k = \min(x_1, \ldots, x_n)$, its left subtree is the Cartesian tree for $\langle x_1, \ldots, x_{k-1} \rangle$ and its right subtree is the Cartesian tree for $\langle x_{k+1}, \ldots, x_n \rangle$.

The first phase of the algorithm is to build a Cartesian tree from the input sequence in linear time. The Cartesian tree is initialized with x_1. Then, the input sequence is scanned in order and every element is inserted into the Cartesian tree. To insert a new element x_i into the Cartesian tree, the nodes along the rightmost path of the tree are traversed bottom-up, until an element x_j that is smaller than x_i is found. The right subtree of x_j is made the left subtree of x_i, and x_i is then made the right child of x_j.

The algorithm proceeds by moving the smallest element of the Cartesian tree into a heap. The minimum of the heap is printed and deleted, and the children of the node corresponding to this element in the Cartesian tree are detached and inserted into the heap. The above step is repeated until the Cartesian tree is empty. The algorithm then continues like Heapsort by repeatedly printing and deleting the minimum element from the heap.

The work done by the algorithm is: n insertions and n minimum-deletions in the heap, plus the linear work for building and querying the Cartesian tree.

Implementation Considerations

Levcopoulos and Peterson [1993] proved that the running time of the algorithm is $O(n \log \frac{Inv(X)}{n} + n)$ by showing that, when x_i is the minimum of the heap, the size of the heap is $O(Inv(x_i))$, where $Inv(x_i)$ is the number of elements that are larger than x_i and appear before x_i with respect to the input sequence. They suggested implementing the algorithm using a binary heap. In such case, the above algorithm requires at most $3n \log_2 \frac{Inv(X)}{n} + O(n)$ comparisons. The storage required by this implementation is $3n$ extra pointers (for the Cartesian tree, two child pointers and a parent pointer that is reused for the binary heap), in addition to the n pointers to the input elements.

The following improvement to the algorithm is suggested in Levcopoulos and Peterson [1993]. Since there are at least $\lfloor n/2 \rfloor$ of the minimum-deletions, each immediately followed by an insertion (deleting a node that is not a leaf of the Cartesian tree must be followed by an insertion), each of these minimum-deletions together with the insertion that follows are implemented by replacing that minimum with the new element and proceeding to maintain the heap property from the top down. This will cost at most $2 \log_2 r$ comparisons (r is the heap size) for both the deletion and insertion, for a total of at most $2.5n \log_2 \frac{Inv(X)}{n} + O(n)$ comparisons. We call this variation the improved Heapsort algorithm.

Pushing the idea of reducing the number of comparisons for binary heaps to the extreme, implementations resulting in better bounds can be used. The best possible bounds are: $O(\log \log r)$ comparisons per insertion and $\log_2 r + O(\log \log r)$ comparisons per minimum-deletion [Gonnet and Munro 1986]. This implies that the Heapsort algorithm requires at most $n \log_2 \frac{Inv(X)}{n} + O(n \log \log \frac{Inv(X)}{n})$ comparisons. We have not implemented this variation as we expect it to be impractical.

Another possibility is to use a binomial queue instead of the binary heap. This would result in a total of at most $2 \log_2 \frac{Inv(X)}{n} + O(n)$ comparisons (every insertion takes constant amortized time, and every minimum-deletion requires at most $2 \log_2 r$ comparisons). For this implementation, we need $2n$ more pointers (every binomial-tree node points to: its first child, its next sibling, the corresponding element; but we can still use the parent pointers of the Cartesian tree).

Experimental Findings

Comparing the results for the basic implementation versus the improved implementation suggests that the latter is more sensitive to inversions, and that the ratio between the comparisons performed by the two is not always 3:2.5 as suggested by the theoretical worst-case analysis. The reason is that the basic implementation performs comparisons spanning the entire depth of the heap for both deletions and insertions. The improved implementation, while working top-down, saves time when the final position of the newly inserted element is close to the root of the heap; this situation is expected when the number of inversions is small. Empirical results suggest that the ratio is about 3:2.2 for $m = 100$. On the other hand, for a large number of inversions, it is not the case that the ratio is 3:2.5 either. In such case, the elements that have children in

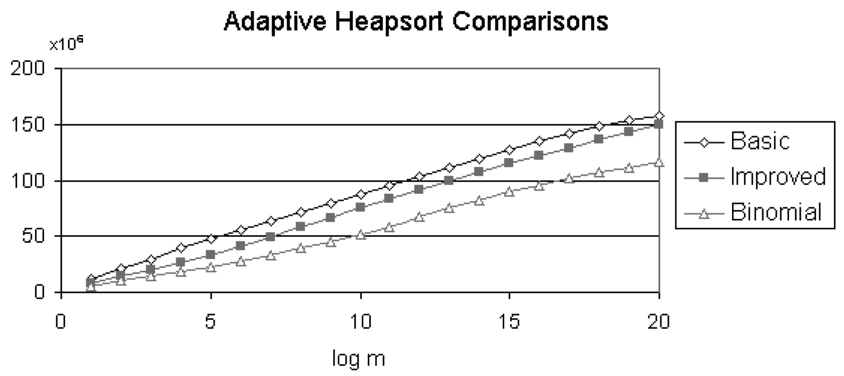

Fig. 1. Adaptive Heapsort: number of comparisons.

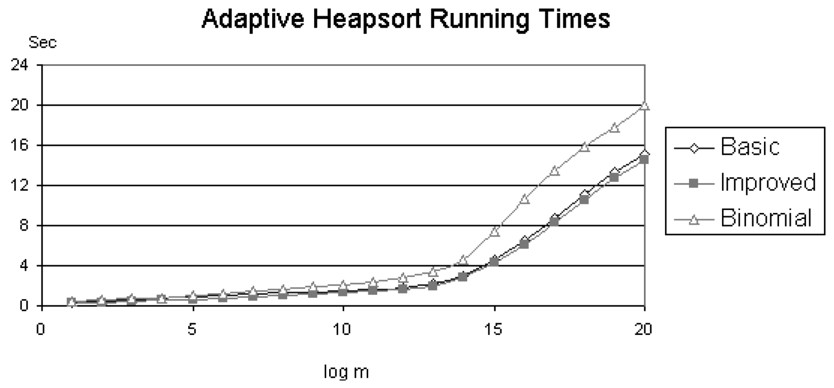

Fig. 2. Adaptive Heapsort: running times.

the Cartesian tree are deleted when the heap is still small, while the leaves of the Cartesian tree will be deleted when the heap has its maximum depth. This makes the weight of a deletion that is followed by an insertion less than the weight of a deletion that is not followed by an insertion. Empirical results suggest that the ratio is about 3:2.7 for $m = 64{,}000$.

As indicated by Figure 1, the implementation that uses binomial queues performs the least number of comparisons. However, as Figure 2 shows, this comparison reduction was accompanied by a noticeable increase in the running time. This is expected as binomial queues require many pointer manipulations.

As a consequence of the Heapsort algorithm being inversion-optimal, and since the number of inversions follows the value of m, it was expected that the running time curve would be smoothly linear with $\log m$. However, this is not the case, as indicated by the sharp bends in Figure 2. We relate this inconsistency to cache misses. When the number of inversions, and hence m, is below some threshold value, the number of cache misses tends to be very small because the heap fits into cache memory. After the number of inversions exceeds that threshold value, the heap size becomes larger than cache memory, and so cache misses begin to occur and increase with $\log m$ as Figure 3 illustrates.

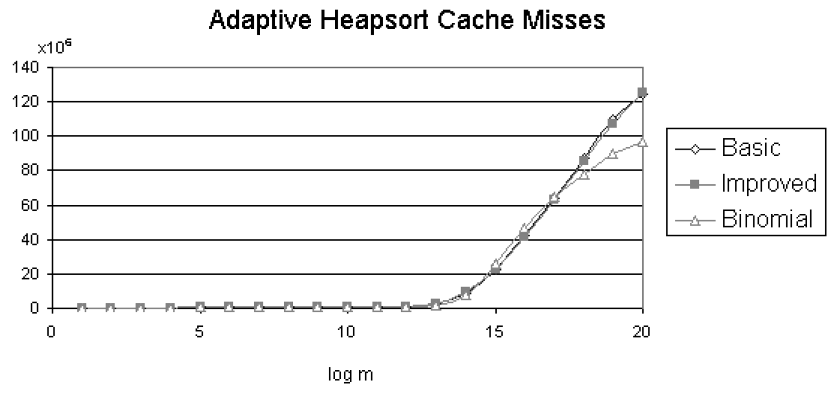

Fig. 3. Adaptive Heapsort: cache misses.

An interesting observation, as shown by Figure 3, is that the implementation that uses a binomial queue performs less cache misses although it requires more storage. This is explained by noting that, when using a binary heap, the heap operations are performed along the depth of the heap. This requires accesses to different locations of the array of the input elements by consecutive operations. On the other hand, when using a binomial queue, the heap operations are performed on the roots of the binomial trees. This introduces a sort of locality when accessing the same nodes in the near future.

4. SPLITSORT AND GREEDYSORT

Basic Algorithms

Splitsort [Levcopoulos and Peterson 1991] works by scanning the input sequence X to get an up-sequence (ascending subsequence) X_u. When an element x_i is scanned, it is placed in one of three sequences X_u, X_s, or X_g as follows. If x_i is larger than the tail of X_u, it is appended to the end of X_u. Otherwise, x_i is appended to X_s, and the tail element of X_u is detached and inserted in X_g in a way that the order of the elements in X_g is the same as their original order in X. After scanning the entire input sequence, we get a sorted up-sequence X_u and two sequences, X_s and X_g. Both X_s and X_g are sorted recursively, then the three sequences are merged.

Greedysort [Brodal et al. 2005b] works in a similar manner, except that it uses a different division protocol. When a scanned element x_i is larger than the tail of X_u, it is appended to the end of X_u similar to the case of Splitsort. Otherwise, x_i is appended to one of two sequences, X_1 or X_2. The algorithm uses an odd-even approach (elements are alternatively inserted in X_1 and X_2) to guarantee that X_1 and X_2 have balanced sizes; i.e., $|X_2| \leq |X_1| \leq |X_2| + 1$. After scanning the entire input sequence, we get a sorted up-sequence X_u and two sequences, X_1 and X_2. Both X_1 and X_2 are sorted recursively, then the three sequences are merged.

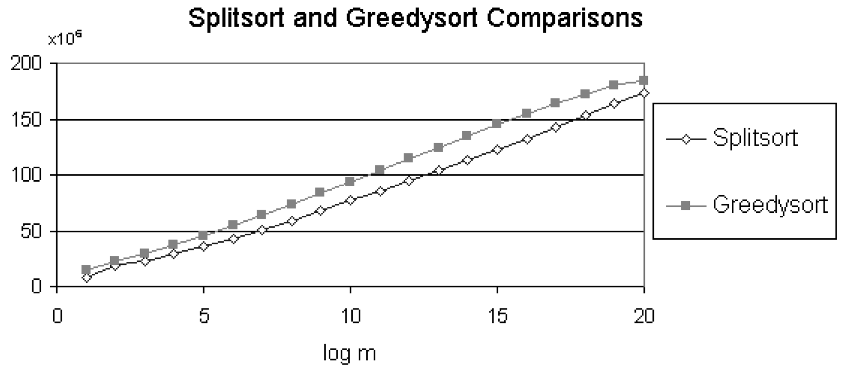

Fig. 4. Splitsort and Greedysort: number of comparisons.

Implementation Considerations

Analysis of Splitsort shows that it is inversion-optimal, and that it performs at most $4.6n \log \frac{Inv(x)}{n} + O(n)$ comparisons. In the worst case, it requires $2.5n \log n + \frac{n}{2}$ comparisons [Levcopoulos and Peterson 1991]. Greedysort, besides being inversion-optimal, is also cache-optimal, meaning that it performs $O(\frac{n}{B} \log_{\frac{M}{B}} \frac{Inv(X)}{n})$ cache misses, where B is the block size and M is the cache size (using the tall-cache assumption $M = \Omega(B^2)$) [Brodal et al. 2005b].

In addition to the array-based Greedysort, we implemented two variations of Splitsort: one array-based and one pointer-based.

Our first implementation for Splitsort is array-based, similar to the one in Levcopoulos and Peterson [1991]. We make use of an auxiliary array of n pointers. The elements in the up-sequence are linked together using such pointers keeping track of the current tail of the up-sequence. The other entries of the auxiliary array are used for indicating which of X_g and X_s the corresponding elements belong to. After the first phase, we scan the auxiliary array writing into each entry the final position of the corresponding element. Finally, we rearrange the input array according to the auxiliary array. The same auxiliary array is used for merging. First, we merge the two shorter sequences, then we merge the result with the longest sequence.

Our second implementation for Splitsort relies on linked structures. Each of the three subsequences is implemented as a linked list. In order to keep the order of the elements of X_g the same as their order in X, in addition to the pointers of the linked lists, we maintain an extra pointer with every node of X_u that points to a position in X_g. If this node is to be moved to X_g, the extra pointer is utilized to locate the correct insertion point. This requires $2n$ extra pointers.

Experimental Findings

As illustrated by Figure 4, Splitsort performs fewer comparisons than Greedysort. The difference between the two algorithms is smaller for very small values of m as well as large values of m. For an explanation, consider the case of $m = 0$ (the input sequence is already sorted). In such case, the two algorithms

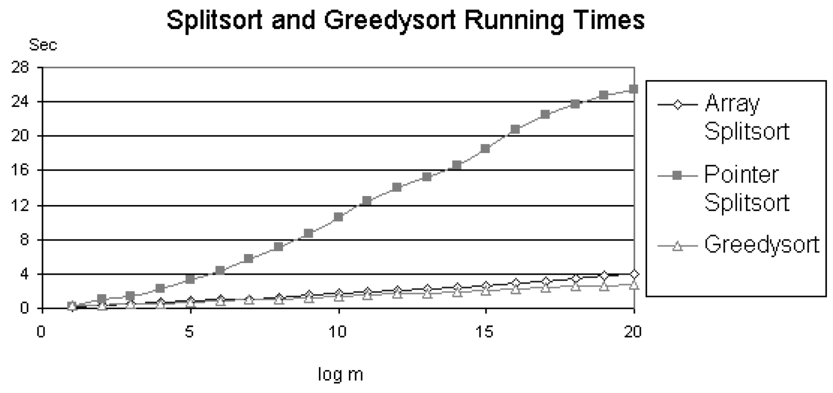

Fig. 5. Splitsort and Greedysort: running times.

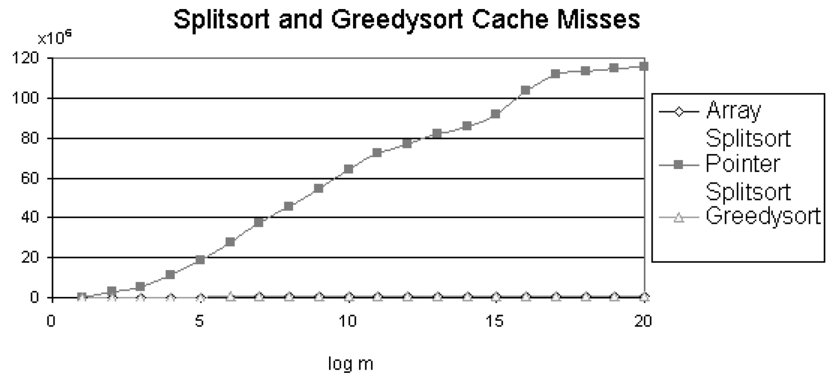

Fig. 6. Splitsort and Greedysort: cache misses.

will perform exactly the same where all the elements will be appended to X_u. On the other hand, consider the case when the input sequence is inversely sorted. In such case, the two algorithms will again perform exactly the same: the sequence X_u will always be empty, while the elements will be alternatively distributed on the other two sequences in the same reversed order.

On the other hand, as shown in Figure 5, the running time of the array-based Splitsort is smaller than that of the pointer-based implementation. This is a result of the fact, indicated by Figure 6, that the number of cache misses of the array-based Splitsort is mush less than that of the pointer-based implementation.

Figure 6 indicates that Greedysort is the best with respect to the number of cache misses. Note also the tilt for the pointer-based Splitsort around $\log m = 15$. We relate this tilt to the size of the cache memory, where for smaller values of m the working data-set fits nicely in the cache memory and the algorithm starts performing much more cache misses as m increases.

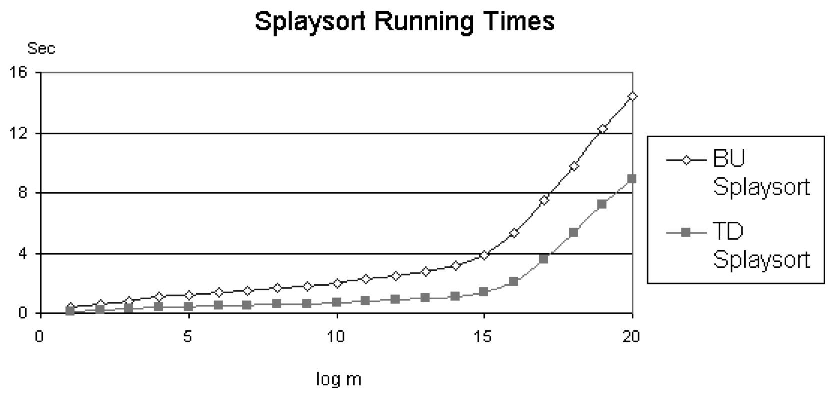

Fig. 7. Splaysort: running times.

5. SPLAYSORT

Basic Algorithm

Splaysort works by repeatedly inserting the elements into a splay tree [Sleator and Tarjan 1985]. A splay tree is a binary search tree that uses a series of edge rotations with every insertion, which results in bringing the inserted element to the root. Once the splay tree is constructed, an in-order traversal produces the sorted sequence.

Implementation Considerations

As a consequence of Cole [2000], the running time of Splaysort is $O(n \log \frac{Inv(X)}{n} + n)$.

Two variations of this algorithm are implemented: one using bottom-up splaying and the other using top-down splaying. The difference is in the way the rotations are performed along the path from the root to a leaf node. The bottom-up splaying performs the rotations after the insertion is done starting from the inserted node upwards toward the root. Both algorithms use the same data structures, except that the bottom-up (BU) splay trees need extra parent pointers. Top-down (TD) splaying performs rotations while searching the tree during the insertion from the root to leaf nodes, with no need for parent pointers.

Experimental Findings

Although the number of comparisons is the same for both implementations, the running time for TD Splaysort is much less, see Figure 7. This is because in BU Splaysort we trace the elements back from a leaf to the root, which almost doubles the pointer operations. This same conclusion was obtained in Moffat et al. [1996].

As indicated by Figure 8, TD Splaysort performs fewer cache misses than BU Splaysort. We relate this to the extra storage used by the BU Splaysort, and to the double passes performed by the BU Splaysort along the splaying path.

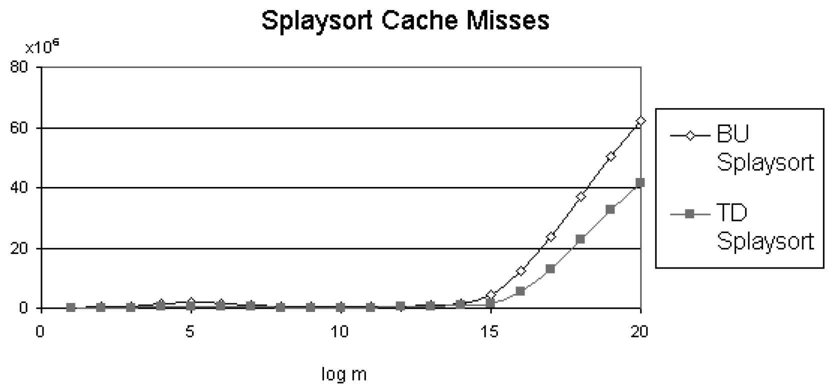

Fig. 8. Splaysort: cache misses.

Figure 8 also shows that the number of cache misses is more for sequences with a larger number of inversions. For sequences with a small number of inversions, the insertions are biased to be performed in the right subtrees. As a consequence, the splaying operations result in these right subtrees being smaller in size. This influences subsequent insertions to be performed faster, resulting in fewer cache misses.

6. ADAPTIVE AVL SORT

Basic Algorithm

AVL sort [Elmasry 2004] works by repeatedly inserting the elements into AVL trees. The AVL trees are organized into a list of consecutive bands from left to right. Every band has 1, 2, *or* 3 AVL trees. The AVL trees are implemented as search trees with the data elements stored only in the leaves while the internal nodes contain indexing information. The elements in the bands are sorted in increasing order from left to right; the same holds for trees within a band. In other words, the elements of an AVL tree are smaller than the elements of the trees to the right of this tree. After inserting all the elements, the sorted sequence is produced by an in-order traversal to the trees from left to right.

A rank value is assigned to every band. The trees in a band with rank h have heights h, except for at most one of these trees that may have height $h-1$. At any stage of the algorithm, the ranks of the bands form an increasing consecutive sequence $s, s+1, s+2, \ldots,$ from left to right, with the value of s changing through the algorithm depending on the number of inversions resulting from the already inserted elements. The value of s, while inserting the element x_k, is computed as: $s = \lceil \log_\theta r_k \rceil + 1$, where $1 < \theta \leq 2$ is a tuning parameter, $r_k = \frac{1}{k-1} \sum_{1 \leq j < k} i_j$, and i_j is the order of x_j in the sorted sequence after the j-th insertion.

With every insertion, the bands are scanned from left to right to locate the band into which the element will be inserted. Within this band, the trees are also scanned to locate the tree into which the element will be inserted. To do that efficiently, we keep track of the largest element within every band as well as within every tree.

After each insertion, as a result of the above conditions, the band list would require reorganization, though on a relatively infrequent basis. This can be accomplished using the following four basic 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 its root ($h \geq 2$).
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 in an AVL tree of height h by adding a new root. The values of the left tree must not be larger than those of the right tree.
3. Find largest: The value of the largest member of a given tree can be accessed in constant time. A pointer to the largest value is maintained after each operation.
4. Tree-insertion: The insertion of a new value in an AVL tree of height h can be performed in $O(h)$ time within h comparisons [Adelson-Velskii and Landis 1962].

Implementation Considerations

Following the analysis in Elmasry [2004], AVL sort performs at most $1.44n \log \frac{Inv}{n} + O(n)$ comparisons and runs in $O(n \log \frac{Inv}{n} + n)$.

A linked list of band headers is maintained. Every band points to its first tree header and stores the number of trees in the band for faster processing. Tree headers within a band are organized in a linked list, every header pointing to its tree root.

To facilitate the find-largest operation in constant time, we store and maintain with every tree node the maximum element within its subtree. Every tree node also stores the size of its subtree. For a new insertion, when we scan the bands from left to right, the size of every passed band is accumulated. Within the band into which the element will be inserted, tree sizes are also summed up until we reach the tree into which the element will be inserted. The number of elements smaller than the inserted element in its tree is calculated using this size information, while the tree is searched for the insertion. Because every tree node stores the size of its subtree, this size must be maintained by the above operations, including rotations.

In order to recalculate the new value of s, instead of calling the time-consuming logarithm function with every insertion, this is done by saving a table for powers of θ. Once the value of θ is decided, such table is constructed and stored. The size of this table is logarithmic with respect to the maximum value of s.

The extra storage required by AVL sort is $26n + O(\log n)$ bytes; 21 bytes for each of the n internal nodes (each node holds: left, right, data, and maximum pointers, 1 byte for balancing information, and 4 bytes for its subtree size), $5n$ bytes for each of the n leaves (each node has a pointer to the data, and an extra byte to distinguish it from internal nodes), and $O(\log n)$ bytes for band and tree-header data.

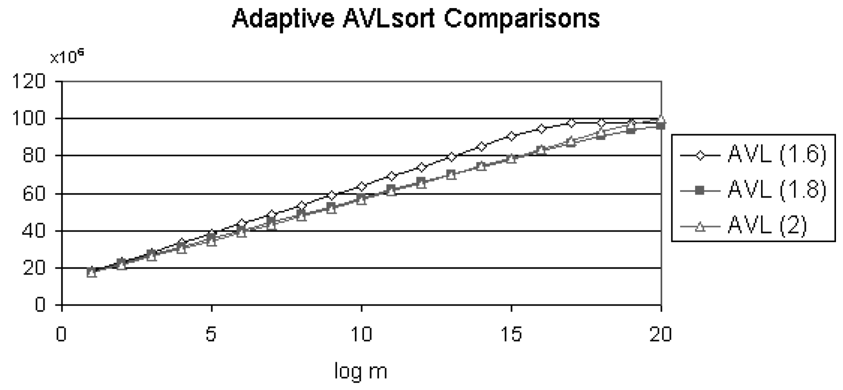

Fig. 9. AVLsort: number of comparisons.

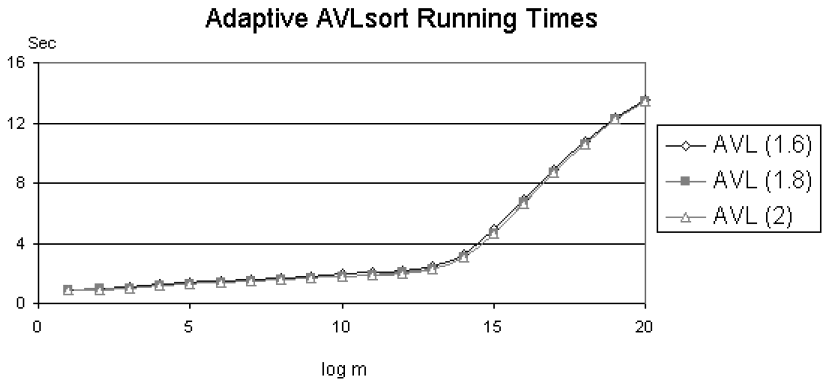

Fig. 10. AVLsort: running times.

Experimental Findings

Consider the case when the value of s, as calculated by the above formula, oscillates (increases by 1 after an insertion and decreases by 1 after the next insertion). This would result in several *split* and *combine* operations to update the rank of the leftmost band. Experimental results showed that the algorithm suffers from a degradation in performance in such situations. The conditions for changing such a rank were relaxed as follows. We still increment the rank of the leftmost band when the calculated value for s increases. But, we only decrement the rank of the leftmost band when such value for s becomes 2 less than the current value for such rank. For example, if the current leftmost rank is 7 and then after an insertion the calculated value for s becomes 6, the rank will not change. The rank is only decremented to 6 if the computed value becomes 5.

As indicated by Figures 9 and 10, tuning the parameter θ has an important influence on the performance of the AVL sort algorithm. Analytically [Elmasry 2004], to guarantee the worst-case performance, a value of θ equal to the golden ratio is used. For the average-case performance, the best value of θ would be close to 2. Experimental results show that, with less inversions, values of θ close to 2 are preferable. This is a result of the fact that for low inversions most

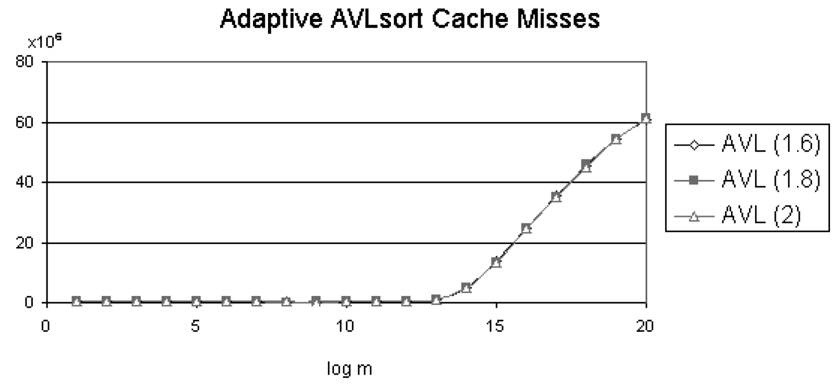

Fig. 11. AVLsort: cache misses.

insertions are performed in the left trees, while most of the other trees are produced by *combine* operations, making these AVL trees near balanced. On the other hand, with more inversions, lower values of θ are better.

The two figures also show that changing θ is more effective on the number of comparisons than the running times. The reason is that we use integer data, and hence the comparisons are not dominating the running time.

When there are few inversions, the leftmost trees will have smaller heights and the newly inserted elements are expected to be inserted in these trees. This results in fewer cache misses. See Figure 11.

7. CONCLUSIONS AND COMMENTS

One of our main observations, with respect to the running time of the algorithms that require linked structures, is that the dynamic allocations of such structures are time consuming. We noticed that more than 15% of the time of such algorithms (Heapsort, Splaysort and AVL sort) was used for such memory allocations. Our solution, that saves around 15% of the time, was to allocate a chunk of memory at the beginning of the algorithm that is enough for the memory requirement of the algorithm throughout its execution.

Regarding the number of comparisons, all the implemented adaptive sorting algorithms were noticeably much better than Quicksort for low inversions. This behavior is expected, since Quicksort is not adaptive with respect to the number of comparisons: the expected number of comparisons performed by the implemented version of randomized Quicksort is $O(n \log n)$, regardless of the number of inversions in the input. Some of the algorithms always perform fewer comparisons than Quicksort, even for random lists. Empirical results illustrate that all the adaptive sorting algorithms under study perform at most $c \cdot n \log \frac{Inv(X)}{n} + O(n)$ comparisons, with c between 1 and 2; for AVL sort, $c \approx 1$. See Figure 12.

As illustrated by the running-time curves in Figure 13, randomized Quicksort is preferable for sequences with a larger number of inversions. TD Splaysort shows the best performance when the number of inversions in the input sequence is small. For the given input size 2^{22}, TD Splaysort is better than

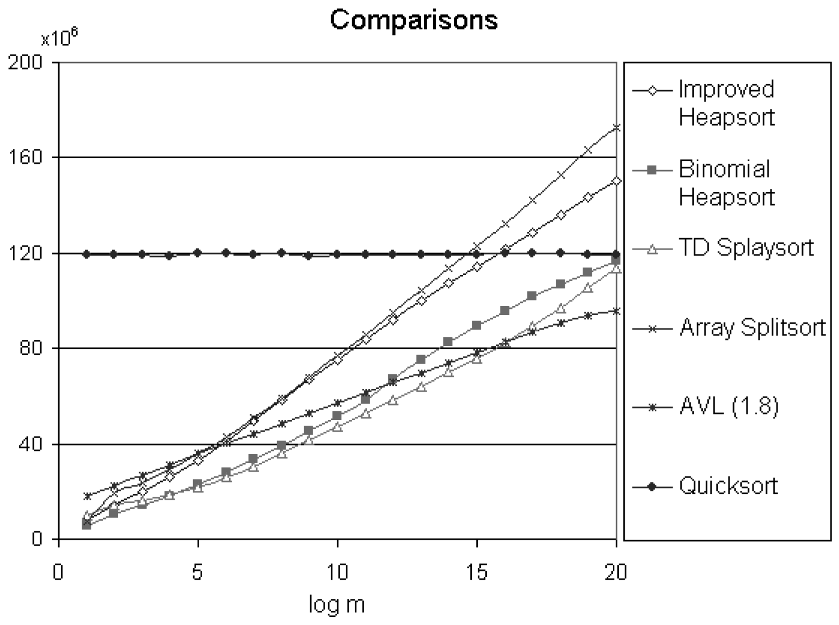

Fig. 12. All algorithms: number of comparisons.

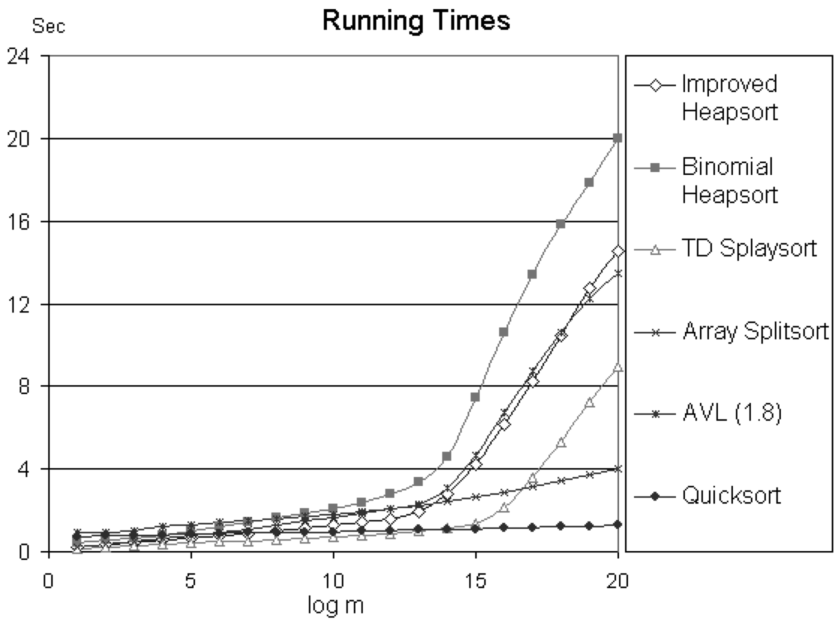

Fig. 13. All algorithms: running times.

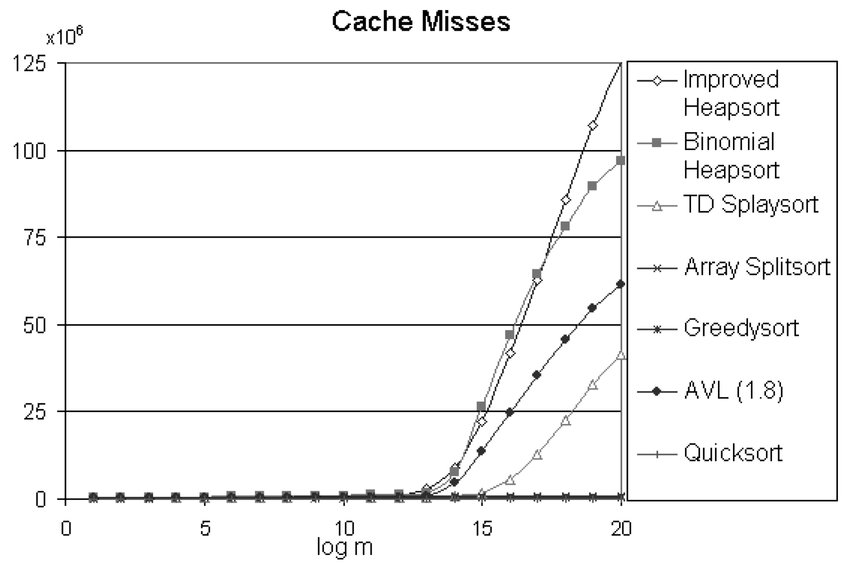

Fig. 14. All algorithms: cache misses.

Table I. Extra storage used by the implemented algorithms

Algorithm	Implementation	Extra memory (bytes)
Heapsort	binary heaps	$16n$
	binomial queues	$24n + O(\log n)$
Splaysort	bottom-up	$16n$
	top-down	$12n$
Splitsort	array-based	$4n + O(\log n)$
	pointer-based	$8n + O(\log n)$
AVL sort		$26n + O(\log n)$
Quicksort		$O(\log n)$

randomized Quicksort, as long as the number of inversions is less than 2^{15}. Splitsort, though not the best with low inversions, is better than TD Splaysort for high inversions, while it is not much worse than Splaysort with low inversions. It is also interesting to demonstrate that the running time of Quicksort is slightly better for a smaller number of inversions; this is a consequence of the result of Brodal et al. [2005a] that randomized Quicksort is adaptive with respect to the number of swaps.

The most effective factor among our performance measures is the cache misses. Although some algorithms, like AVL sort, have some advantage regarding the number of comparisons, cache misses degrade their performance causing their running time to be worse than the algorithms that perform fewer cache misses. Both Quicksort and Greedysort perform the fewest number of cache misses. See Figure 14.

The extra storage used by each of the algorithms is given in Table I (a pointer is stored in 4 bytes).

REFERENCES

Adelson-Velskii, G., and Landis, E. 1962. On an information organization algorithm. *Doklady Akademia Nauk SSSR*, *146*, 263–266.

Brodal, G., Fagerberg, R., and Moruz, G. 2005. On the adaptiveness of Quicksort. *Proceedings of the 7th Workshop on Algorithm Engineering and Experiments*, Vancouver, B.C., Canada, January 22, pp. 130–140, SIAM.

Brodal, G., Fagerberg, R., and Moruz, G. 2005. Cache-aware and cache-oblivious adaptive sorting. *Proceedings of the 32nd International Colloquium on Automata, Languages and Programming*, Lisboa, Portugal, July 11–15, Lecture Notes on Computer Science 3580, pp. 576–588, Springer Verlag.

Cole, R. 2000. On the dynamic finger conjecture for splay trees. Part II: The proof. *SIAM Journal on Computing*, *30*, 44–85.

Cook, R. and Kim, J. 1980. Best sorting algorithms for nearly sorted lists. *Communications of the ACM*, *23*, 620–624.

Dromey, P. 1984. Exploiting partial order with Quicksort. *Software: Practice and Experience*, *14*, 509–518.

Elmasry, A. 2004. Adaptive sorting with AVL trees. *18th IFIP World Computer Congress, Proceedings of the 3rd International Conference on Theoretical Computer Science*, Toulouse, France, August 23–26, pp. 315–324.

Elmasry, A., Jensen, C., and Katajainen, J. Multipartite priority queues. *ACM Transactions on Algorithms*. To appear.

Elmasry, A., Jensen, C., and Katajainen, J. 2004. A framework for speeding up priority queue operations. *CPH STL TR 2004-3*, Department of Computing, University of Copenhagen.

Elmasry, A. 2002. Priority queues, pairing and adaptive sorting. *Proceedings of the 29th International Colloquium on Automata, Languages and Programming*, Malaga, Spain, July 8–13, Lecture Notes on Computer Science 2380, pp. 183–194, Springer Verlag.

Elmasry, A. and Fredman, M. 2008. Adaptive sorting: an information theoretic perspective. *Acta Informatica*, *45*(1), 33–42.

Estivill-Castro, V., and Wood, D. 1992. A survey of adaptive sorting algorithms. *ACM Computer Surveys*, *24*(4), 441–476.

Gonnet, G., and Munro, J. I. 1986. Heaps on heaps. *SIAM Journal on Computing*, *15*, 964–971.

Guibas, L., Mccreight, E., Plass, M., and Roberts, J. 1977. A new representation for linear lists. *Proceedings of the 9th Annual ACM Symposium on Theory of Computing*, Atlanta, GA, USA, April 30–May 2, pp. 49–60.

Hoare, C. 1961. Algorithm 64: Quicksort. *Communications of the ACM*, *4*(7), 321.

Knuth, D. 1998. The Art of Computer Programming. Vol III: Sorting and Searching. Addison-wesley, second edition.

Levcopoulos, C., and Peterson, O. 1996. Exploiting few inversions when sorting: Sequential and parallel algorithms. *Theoretical Computer Science*, *163*, 211–238.

Levcopoulos, C., and Peterson, O. 1993. Adaptive Heapsort. *Journal of Algorithms*, 14, 395–413.

Levcopoulos, C., and Peterson, O. 1991. Splitsort—An adaptive sorting algorithm. *Information Processing Letters*, *39*, 205–211.

Mannila, H. 1985. Measures of presortedness and optimal sorting algorithms. *IEEE Transactions on Computers*, C-34, 318–325.

Mehlhorn, K. 1979. Sorting presorted files. *Proceedings of the 4th GI Conference on Theoretical Computer Science*, Aachen, Germany, March 26–28, Lecture Notes on Computer Science 67, pp. 199–212, Springer Verlag.

Moffat, A., Eddy, G., and Petersson, O. 1996. Splaysort: Fast, versatile, practical. *Software: Practice and Experience*, *126*(7), 781–797.

Moffat, A. and Petersson, O. 1992. An overview of adaptive sorting. *Australian Computer Journal*, *24*, 70–77.

Moffat, A., Petersson, O., and Wormald, N. 1998. A tree-based Mergesort. *Acta Informatica*, *35*(9), 775–793.

Sleator, D., and Tarjan, R. 1985. Self-adjusting binary search trees. *Journal of the ACM*, *32*(3), 652–686.

Vuillemin, J. 1980. A unifying look at data structures. *Communications of the ACM*, 23, 229–239.

Wainwright, R. 1985. A class of sorting algorithms based on Quicksort. *Commuications of the ACM*, *28*(4), 396–402.