DIMACS Technical Report 2001-24
August 2001

Optimal Adaptive Sorting

by

Amr Elmasry
Computer Science Department, Rutgers University
elmasry@paul.rutgers.edu

ABSTRACT

We introduce Binomialsort, an adaptive sorting algorithm that is optimal with respect to the number of inversions. The number of comparisons performed by Binomialsort, on an input sequence of length n that has I inversions, is at most $2n \log \frac{I}{n} + O(n)$. The bound on the number of comparisons is further reduced to $\frac{3}{\log 3} n \log \frac{I}{n} + O(n)$ by using a new structure, which we call trinomial queues. Experimental results show that our sorting algorithm is practical, efficient and easy to implement.

1 Introduction

In many applications the lists to be sorted do not consist of randomly distributed elements, but are already partially sorted. An adaptive sorting algorithm benefits from the existing presortedness in the input sequence. One of the main measures of presortedness is the number of inversions in the input sequence [5]. The number of inversions is the number of pairs in the wrong order. More precisely, for an input sequence X of length n, the number of inversions, $Inv(X)$, is defined

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

We assume that x_i is to the left of x_j, whenever i is less than j.

Finger trees [4, 2, 10] was the first data structure that utilizes the existing presortedness in the input sequence. Using a finger tree, the time required to sort a sequence X of length n is $O(n \log \frac{Inv(X)}{n} + n)$ which is asymptotically optimal [4]. These algorithms are not practical, however. Mehlhorn [9] introduced a sorting algorithm that achieves the above bound but his algorithm is not fully practical, as well. Among the sorting algorithms which are optimal with respect to the number of inversions, Splitsort [6] and Adaptive Heapsort [7] are the most promising from the practical point of view. Both algorithms require $2.5n \log n$ comparisons in the worst case. Other algorithms include Blocksort [8] which runs in-place, and the Tree-based Mergesort [12]. Splaysort, sorting by repeated insertion in a splay tree [13], was proved to be optimal with respect to the number of inversions [3]. Moffat et al. [11] performed experimental results showing that Splaysort is practically efficient.

We introduce Binomialsort, an optimal sorting algorithm with respect to the number of inversions. The number of comparisons performed by Binomialsort is at most $2n \log \frac{Inv(X)}{n} + O(n)$. Then, we introduce trinomial queues, an alternative to binomial queues, and use it to reduce the bound on the number of comparisons performed by the algorithm to $\frac{3}{\log 3} n \log \frac{Inv(X)}{n} + O(n)$. Experiments show that our sorting algorithm outperforms Splaysort when there are few inversions.

2 Adaptive transformations on binomial queues

A binomial tree [1, 14] of rank r is constructed recursively by making the root of a binomial tree of rank $r - 1$ the rightmost child of the root of another binomial tree of rank $r - 1$. A binomial tree of rank 0 is a single node. The following properties follow from the definition:

- The rank of an n-node (assume n is a power of two) binomial tree is $\log n$. [1]

[1] All logarithms are to the base two.

- The root of a binomial tree with rank r has r sub-trees each of which is a binomial tree, having respective ranks 0,1,...,r-1 from left to right.
- Other than the root, there are 2^i nodes with rank $\log n - i - 1$, for all i from 0 to $\log n - 1$.

A binomial queue is a heap-ordered binomial tree. It satisfies the additional constraint that every node contains a data value smaller than or equal to those stored in its children. We use a binary implementation of binomial queues. In such an implementation, every node has two pointers, One pointing to its right sibling and the other to its rightmost child. The right pointer of a rightmost child points to the leftmost child to form a circular list. Given a pointer to a node, both its rightmost and leftmost children can be accessed in constant time. The list of its children can be sequentially accessed from left to right.

Given a forest F, each node of which contains a single value, the sequence of values obtained by a pre-order traversal of F is called the corresponding sequence of F and denoted by $Pre(F)$. Our traversal gives precedence to the trees of the forest in left-to-right order. Also, the precedence ordering of the sub-trees of a given node proceeds from left to right. A transformation on F that results in a forest F' shall be considered *Inv-adaptive* provided that $Inv(Pre(F')) \leq Inv(Pre(F))$. Our sorting algorithm will be composed of a series of *Inv-adaptive* transformations, applied to an initial forest consisting of a sequence of singleton-node trees (the input sequence). The idea is to maintain the presortedness in the input sequence, by keeping the number of inversions in the corresponding sequence of the forest, non-increasing with time.

The following transformations on binomial queues are needed throughout the algorithm:

Heapifying a binomial queue

Given an n-node binomial queue T, with the exception that the value at its root is not the smallest value, we want to restore the heap property. Applying the standard *heapify* operation will do. Moreover, this transformation is *Inv-adaptive*. Recall that the *heapify* operation proceeds by finding the node, say x, with the smallest value among the children of the root and swapping its value with that of the root. This step is repeated with the node x as the current root, until either a leaf or a node that has a value smaller than or equal to all the values of its children is reached. Consider any two nodes whose values are swapped during the *heapify* operation. Call them y and z, and assume that y appears before (to the left of) z in $Pre(T)$. We conclude that the value of z was smaller than the value of y, otherwise their values would have not been swapped. Therefore, this swap will not increase the number of inversions.

It follows that the *heapify* operation is *Inv-adaptive.*

It remains to investigate how the *heapify* operation will be implemented. Finding the minimum value within a linked list requires linear time. This may lead to an $O(\log^2 n)$ time for the *heapify* operation. We can do better, however, by maintaining with every node an extra pointer that points to the node with the smallest value among all its left siblings, including itself. Call this pointer *sl* pointer. The *sl* pointer of the rightmost child of a node will, therefore, point to the node with the smallest value among all the children of the father node. To maintain the correct values in the *sl* pointers, whenever the value of a node is updated all the *sl* pointers of its right siblings, including itself, have to be updated. This is accomplished by proceeding from left-to-right; the *sl* pointer of a given node is updated to point to the smaller of the value at this node and the value at the node pointed to by the *sl* pointer of its left sibling. A *heapify* at a node with rank $r1$ reduces to a *heapify* at its child with the smallest value whose rank is $r2$, after one comparison plus at most $r1 - r2$ comparisons to maintain the *sl* pointers. Let $H(r)$ be the number of comparisons used by the above algorithm to *heapify* a binomial queue whose root has rank r, then

$$H(0) = 0$$

$$H(r1) \leq H(r2) + r1 - r2 + 1$$

The total number of comparisons is, therefore, at most $2 \log n$.

The number of comparisons can still be reduced as follows. First, the path from the root to a leaf, where every node has the smallest value among its siblings, is determined by utilizing the *sl* pointers. No comparisons are required for this step. Next, the value at the root is compared with the values of the nodes of this path bottom up, until the correct position of the root is determined. The value at the root is then inserted at this position, and all the values at the nodes above this position are shifted up. The *sl* pointers of the nodes which moved up and those of all their right siblings are updated. The savings are due to the fact that at each level of the queue, either a comparison with the root takes place, or the *sl* pointers are updated, but not both. We then have

$$H(r1) \leq H(r2) + r1 - r2$$

And the total number of comparisons is at most $\log n$. In the sequel it is to be understood that our binomial queues will be augmented with *sl* pointers as described above.

Lemma 1 *The total number of comparisons needed to heapify an n-node binomial queue is at most* $\log n$.

Inv-adaptive construction of a binomial queue

Given an input sequence of length n, we want to build a corresponding binomial queue *Inv-adaptively.* This is done by recursively building a binomial queue with the rightmost $\frac{n}{2}$ elements, and another binomial queue with the leftmost $\frac{n}{2}$ elements. The root of the right queue is then linked to the root of the left queue as its rightmost child. If the value at the root of the right queue was smaller than the value at the root of the left queue, the two values are swapped. A *heapify* operation is then called to maintain the heap property for the right queue and the *sl* pointer of the root of the right queue is updated. Note that swapping the values at the roots of the two queues is an *Inv-adaptive* transformation. Let $B(n)$ be the number of comparisons needed to build a queue of n nodes as above, then

$$B(2) = 1$$

$$B(n) \leq 2B(\frac{n}{2}) + \log n + 1$$

The following lemma follows

Lemma 2 *Given an input sequence of length n, a binomial queue can be Inv-adaptively built with less than $2n$ comparisons.*

3 Adaptive pairing

For any two queues $q1$ and $q2$, where $q1$ is to the left of $q2$, a pairing operation is defined as follows. If the value of the root of $q2$ is the smaller, perform a left-link by making $q1$ the leftmost child of the root of $q2$. Otherwise, perform a right-link by making $q2$ the rightmost child of the root of $q1$. See Figure 1. It is straightforward to verify that this way of pairing is *Inv-adaptive.* Consider the corresponding sequence of the heap before and after the pairing. This sequence will not change by a right-link. A left-link changes the position of the root of the right queue, by moving it before (to the left of) all the elements of the left queue. Since the value of the root of the right queue is smaller than all the elements of the left queue, the number of inversions decreases. The *Inv-adaptive* property follows.

Given a forest of priority queues, we combine these queues into a single queue as follows. Starting from the rightmost queue, each queue is paired with the result of the linkings of the queues to its right. We refer to this right-to-left sequence of node pairings as a *right-to-left incremental pairing* pass. The intuition behind doing the pairing from right to left is to keep the total number of right-links throughout the algorithm linear, as will be clear later.

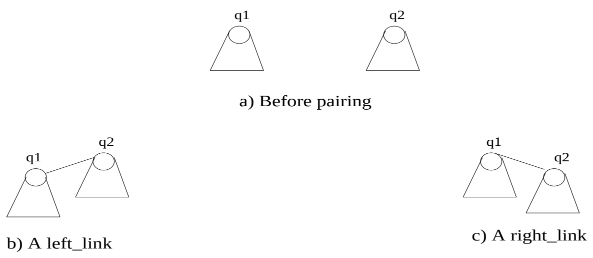

Figure 1: Adaptive Pairing

4 The Binomialsort algorithm

Given an input sequence X of length n. If n is not a power of 2 we split the input list into sub-lists whose lengths are powers of 2 as follows. The first sub-list will have the leftmost $2^{\lceil \log n \rceil - 1}$ elements. We repeat splitting the rest of the elements in the same way, until no elements are left. The sorting algorithm is then applied on each sub-list separately. We then do a merging phase to combine the sorted sub-lists. We start with the rightmost (smallest) two sub-lists and merge them. Then, the result of the merging is repeatedly merged with the next list to the left (next list in size), until all the elements are merged. The total number of comparisons spent in the merging phase is at most $\sum_i \frac{n}{2^i} i$, which is less than $2n$.

Throughout the algorithm, we assume without loss of generality that n is a power of 2. The algorithm in a nutshell is:

Build an initial binomial queue *Inv-adaptively.*
While *the heap is not empty* **do**
  Delete the root of the heap and output its value.
  Do a *fix-up* step.
  Do a *right-to-left incremental pairing* pass.
enddo

Informally, the *fix-up* step is to assure the invariant that, before the pairing pass, the sizes of the trees of the heap increase exponentially from left to right. To do that efficiently, we associate a rank with every node. After building the initial binomial queue, the rank of a node is equal to the number of its children. A property we want to retain is that the rank of a node whose sub-tree is of size s is $O(\log s)$. We adopt the invariant that, before the pairing pass, the rank of the root of every tree is strictly greater than the rank of the root of the tree to its left (its left neighbor).

For a given node, call the children that this node had in the initial binomial queue, the initial children. Call the children that were left-linked to that node, the left children. Call the, at most one, child that may be right-linked to that node, the right child. (We will show later that, as a result of our pairing strategy, any node can gain at most one right child). Consider any node at any point throughout the algorithm. The rank of any initial child is exactly one more than the rank of the initial child to its left. Two other properties will hold as a result of the above invariant. First, the rank of the right child will be the largest rank among all the other children. Second, the rank of any left child will be strictly greater than the rank of its left neighbor. In other words, if a node gains two children by left-links, the one that was gained first will have a strictly larger rank than that of the second. When time comes for a node to leave the heap, the ranks of its children from left to right form at most two runs of strictly increasing values. One is formed from its left children and the other is formed from its initial children and its right child. The purpose of the *fix-up* step is to reconstruct such a forest of trees to retain the invariant.

Subsequent to the initial construction of the binomial queue, node ranks will be updated during the *fix-up* step as illustrated below.

The fix-up step

Let w be the node that has just been deleted before the *fix-up* step. The *fix-up* step works as follows. We start with the leftmost tree and traverse the roots of the trees from left to right, until a node whose rank is greater than or equal to the rank of its right neighbor is encountered (if at all such a node exists). Call this node x and its rank r_x. This node must have been the first left child linked to w. The above invariant implies that the rank of w was strictly greater than r_x. Hence, w must have an initial child whose rank is r_x. It also follows that if w has a right child, the rank of this child must be greater than $r_x + 1$. Starting from the right neighbor of x, we continue traversing the roots of the trees from left to right, while the ranks of the traversed nodes are less than or equal to $r_x + 1$. Call the forest of trees defined by these nodes the working-forest. The above analysis implies that the roots of the trees of the working-forest were initial children of w. There are two cases, depending on the number of trees in the working-forest.

a) Combine

If the number of trees of the working-forest is more than one, we perform a *combine* step. We select a node in the leftmost tree of the working-forest in a process called *root-finding*, and promote it. The *root-finding* is done by starting at the root of this tree and repeatedly checking the leftmost child of the current node until either a leaf or a node that has no siblings is encountered. Cut this node. If it is a leaf, update all the *sl* pointers of its right siblings. If it is an internal node that has no siblings, its parent inherits its children as well as its rank. Next, promote this node by changing its rank to $r_x\!+\!1$, and making the roots of the working-forest children of this promoted node. Finally, perform a *heapify* operation on the combined queue. See Figure 2.

The way the *root-finding* is done insures that the *combine* step is *Inv-adaptive*. Consider the position of the new promoted root of the working-forest within the

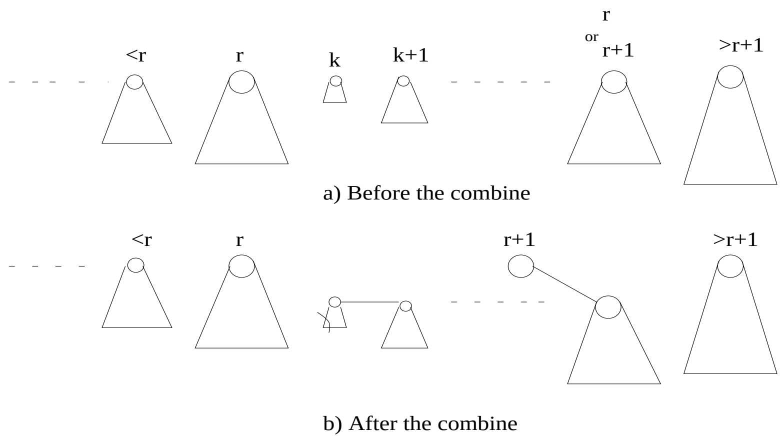

Figure 2: The combine step
Symbols beside nodes represent ranks. k<r

corresponding sequence of the heap before and after the *combine* step. When the new root is promoted, it becomes the first node of the corresponding sequence of the working-forest. After the *heapify* operation, this promoted node returns back to either its original position or to a position to the right of its original position. This follows from the fact that the promoted node was initially a node on the left spine of the leftmost tree of the working-forest. Hence, the number of inversions does not increase.

b)Fuse

If there is only one tree in the working-forest, we perform a *fusion* step. Call the root of this tree y. Let z be the rightmost child of x. Note that the rank of y is equal to r_x. This follows from the fact that there must have been an initial child of w whose rank is r_x. The tree of y is fused with the tree of x. The *fusion* is done by comparing the value of y with the value of x. If the value of y is smaller, the two values are swapped and a *heapify* step is performed on the tree of y. The rank of x is incremented by 1. If the rank of z is less than r_x, we make the tree of y the rightmost child of x. If the rank of z is equal to r_x (a case that happens only if x was promoted as a root of a combined queue), we compare the value of y with the value of z. If the value of y is smaller, the two values are swapped and a *heapify* step is performed on the tree of y. The tree of y becomes the rightmost child of z, and the rank of z is incremented by 1. Since the *heapify* operation is *Inv-adaptive*, it follows that the *fusion* step is *Inv-adaptive*, as well.

Analysis

Lemma 3 *The total number of right-links in all the pairing passes is at most* $n - \log n$*.*

Proof: The *right-to-left incremental pairing* pass insures that a node can gain at most one right child. When a node gains a right child it becomes a node on the right spine of the rightmost tree. Hence, it will never gain another right child in any pairing pass. The $\log n$ nodes, which form the right spine of the rightmost tree in the initial binomial queue, will not gain any right children. □

Lemma 4 *The total number of comparisons involved in all the root-findings is at most* $n - \log n$*.*

Proof: When a leaf is cut during the *root-finding* process, some comparisons are required to update the *sl* pointers of its right siblings. Consider any node, x, whose rank was r_x in the initial binomial queue. The *sl* pointer of x is updated at most once when one of its left siblings is cut and promoted. Since x had r_x left siblings in the initial binomial queue, this may happen at most r_x times. Then, the total number of comparisons involved to update the *sl* pointers, for all the nodes, is at most $n - \log n$.
□

Lemma 5 *The total number of comparisons involved in all the fusion steps is at most* $\frac{3}{4}n$*.*

Proof: When two trees are involved in a *fusion* step, the right tree must have been a sub-tree of the initial binomial queue. We call the root of the right tree an f-node. If the rank of the two trees is r, there must be at most two nodes with rank r on the right spine of the left tree. Note that when a node is promoted to become the root of a combined queue, it may have the same rank as its rightmost child. In the worst case, one comparison and a *heapify* step on the right tree are performed at each of these two nodes. The *heapify* step on a sub-tree of the initial binomial queue whose rank is r requires at most r comparisons, for a total of at most $2(r+1)$ comparisons. Consider an f-node whose rank was r before the *fusion*. After the *fusion* and until this node leaves the heap, the rank of its left siblings will be strictly smaller than r. Hence, it will not be involved in another *fusion* step as an f-node. The cost of the *fusion* can then be charged to this node, for a total of at most $\sum_{i=0}^{\log n - 2} \frac{n}{2^{i+1}} 2(i+1)$ comparisons, which is less than $4n$. A more careful analysis (omitted) will reduce the constant to $\frac{3}{4}$. □

Lemma 6 *The rank of a node that has* s *descendants is less than* $\log s + 1$*.*

Proof: After building the initial binomial queue, the rank of a node that has s descendants is $\log s$. This establishes the base case.

During the *fusion* step, the number of descendants of a node, whose rank is incremented from r to $r+1$, increases by a sub-tree of size 2^r.

During the *combine* step, the *root-finding* process insures that a node of rank r will not lose any of its rightmost child's 2^{r-1} descendants before losing all its other descendents. When a node, whose rank is r, has only one remaining child (the rightmost child), the *root-finding* process promotes the child, and this node inherits its 2^{r-1} descendants, as well as its rank which is $r-1$. When a node is promoted to become the new root of the working-forest, it is given a new rank, say $r+1$. The size of this node's new rightmost sub-tree (a sub-tree of the initial binomial queue whose rank was either r or $r+1$) is either 2^r or 2^{r+1}. □

Lemma 7 *The total number of comparisons involved in the heapify operations in all the combine steps is less than* $n \log \frac{Inv(X)}{n} + 2n$.

Proof: Consider the moment when a node w is leaving the heap. Let f_w be the number of descendents of the left children of w. Each of these nodes represents an element that is greater than w and was appearing before it (to the left of it) in the input sequence. Since all the steps of the algorithm are *Inv-adaptive*, then $\sum_x f_x \leq Inv(X)$.

Assume that the rank of the first left child gained by w is r. Using Lemma 6, then $r < \log f_w + 1$. During the *fix-up* step that follows the deletion of w, if a *combine* step is performed, the accompanying *heapify* operation is performed on a sub-tree of the initial binomial queue whose rank is at most $r+1$, for a total of at most $r+1$ comparisons.

Let CH be the total number of comparisons involved in all the *heapify* operations during the *combine* steps. Using Jensen's inequality, then

$$\begin{aligned} CH &< \sum_x (\log f_x + 2) \\ &\leq n \log \tfrac{Inv(X)}{n} + 2n \end{aligned}$$ □

Lemma 8 *The total number of left-links in all the pairing passes is at most* $n \log \frac{Inv(X)}{n} + n$.

Proof: Consider the moment when a node w is leaving the heap. Let f_w be as defined in Lemma 7. Let l_w be the number of nodes linked to w by a left-link in all the pairing passes. Using the fact that the ranks of the left children of w strictly increase from left to right, and applying Lemma 6 on each of the sub-trees defined by these nodes, then $f_w > \sum_{i=1}^{l_w-1} 2^{i-1} + 1$, and hence $l_w \le \log f_w + 1$. Let L be the total number of left-links. Using Jensen's inequality, then

$$\begin{aligned} L &\le \sum_x (\log f_x + 1) \\ &\le n \log \tfrac{Inv(X)}{n} + n \end{aligned} \qquad \square$$

The following theorem follows, as a consequence of the above lemmas.

Theorem 1 *Binomialsort sorts a sequence X of length n, in at most $2n \log \frac{Inv(X)}{n} + O(n)$ comparisons and $O(n \log \frac{Inv(X)}{n} + n)$ time.*

5 Trinomial queues

A trinomial tree of rank r will be constructed recursively using 3 trinomial trees of rank $r-1$. The first is linked as the rightmost child of the second. Then, the second is linked as the rightmost child of the third. A trinomial tree of rank 0 consists of a single node. The following properties hold for trinomial trees: (Assuming that the number of nodes n is a power of 3)

- The rank of an n-node trinomial tree is $\frac{\log n}{\log 3}$.
- The root of a trinomial tree with rank r has r children, having respective ranks 0,1,...,r-1 from left to right.
- Other than the root, there are 2.3^i nodes with rank $\frac{\log n}{\log 3} - i - 1$, for all i from 0 to $\frac{\log n}{\log 3} - 1$.

A trinomial queue is a trinomial tree that has the additional constraint that every node contains a data value smaller than or equal to those stored in its children.

Applying the same techniques we used for *heapifying* a binomial queue on a trinomial queue, the number of comparisons needed to do the *heapify* on a trinomial queue of rank r is at most $2r$. In a similar fashion, building an initial trinomial queue *Inv-adaptively* can be done with a linear number of comparisons.

Using trinomial queues instead of binomial queues throughout the algorithm, it can be easily verified that the total number of comparisons involved in all the *heapify* operations during the *fix-up* steps is at most $\frac{2}{\log 3} n \log \frac{Inv(X)}{n} + O(n)$ and the total number of left-links is at most $\frac{1}{\log 3} n \log \frac{Inv(X)}{n} + O(n)$. All the other operations requires a linear number of comparisons, for a total of at most $\frac{3}{\log 3} n \log \frac{Inv(X)}{n} + O(n)$ comparisons.

Theorem 2 *Trinomialsort sorts a sequence X of length n, in at most $\frac{3}{\log 3} n \log \frac{Inv(X)}{n} + O(n)$ comparisons and $O(n \log \frac{Inv(X)}{n} + n)$ time.*

References

[1] M. Brown. *Implementation and analysis of binomial queue algorithms.* SIAM J. Comput. 7 (1978), 298-319.

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

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

[4] 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.

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

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

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

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

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

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

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

[12] A. Moffat, O. Petersson and N. Wormald *A tree-based Mergesort.* Acta Informatica, Springer-Verlag (1998), 775-793.

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

[14] J. Vuillemin. *A data structure for manipulating priority queues.* Comm. ACM 21(4) (1978), 309-314.