Priority queues and sorting for read-only data

Asano, Tetsuo; Elmasry, Amr; Katajainen, Jyrki

Priority Queues and Sorting for Read-Only Data[*]

Tetsuo Asano[1], Amr Elmasry[2], and Jyrki Katajainen[3]

[1] School of Information Science, Japan Advanced Institute for Science and
Technology
Asahidai 1-1, Nomi, Ishikawa 923-1292, Japan
[2] Department of Computer Engineering and Systems, Alexandria University
Alexandria 21544, Egypt
[3] Department of Computer Science, University of Copenhagen
Universitetsparken 1, 2100 Copenhagen East, Denmark

Abstract. We revisit the random-access-machine model in which the
input is given on a read-only random-access media, the output is to
be produced to a write-only sequential-access media, and in addition
there is a limited random-access workspace. The length of the input
is N elements, the length of the output is limited by the computation
itself, and the capacity of the workspace is $O(S + w)$ bits, where S is a
parameter specified by the user and w is the number of bits per machine
word. We present a state-of-the-art priority queue—called an adjustable
navigation pile—for this model. Under some reasonable assumptions, our
priority queue supports *minimum* and *insert* in $O(1)$ worst-case time and
extract in $O(N/S + \lg S)$ worst-case time, where $\lg N \le S \le N/\lg N$. We
also show how to use this data structure to simplify the existing optimal
$O(N^2/S + N \lg S)$-time sorting algorithm for this model.

1 Introduction

Problem Area. Consider a sequential-access machine (Turing machine) that
has three tapes: input tape, output tape, and work tape. In space-bounded com-
putations the input tape is read-only, the output tape is write-only, and the aim
is to limit the amount of space used in the work tape. In this set-up, the the-
ory of language recognition and function computation requiring $O(\lg N)$ bits[1]
of working space for an input of size N is well established; people talk about
log-space programs [25, Section 3.9.3] and classes of problems that can be solved
in log-space [25, Section 8.5.3]. Also, in this set-up, trade-offs between space and
time have been extensively studied [25, Chapter 10]. Although one would seldom
be forced to rely on a log-space program, it is still interesting to know what can
be accomplished when only a logarithmic number of extra bits are available.

---

[1] Throughout the paper we use $\lg x$ as a shorthand for $\log_2(\max\{2, x\})$.

In this paper we reconsider the space-time trade-offs in the random-access machine. Analogous with the sequential-access machine, we have a read-only array for input, a write-only array for output, and a limited workspace that allows random access. Over the years, starting by a seminal paper of Munro and Paterson [20], the space-time trade-offs have been studied in this model for many problems including: sorting [4, 12, 24], selection [11, 12], and various geometric tasks [2, 3, 7]. The practical motivation for some of the previous work has been the appearance of special devices, where the size of working space is limited (e.g. mobile devices) and where writing is expensive (e.g. flash memories).

An algorithm (or a data structure) is said to be *memory adjustable* if it uses $O(S)$ bits of working space for a given parameter S. Naturally, we expect to use at least one word, so $\Omega(w)$ is a lower bound for the space usage, w being the size of the machine word in bits. Sorting is one of the few problems for which the optimal space-time product has been settled: Beame showed [4] (see also [25, Theorem 10.13.8]) that $\Omega(N^2)$ is a lower bound, and Pagter and Rauhe showed [24] that an $O(N^2/S + N \lg S)$ running time is achievable for any S when $\lg N \leq S \leq N/\lg N$.

Model of Computation. We assume that the elements being manipulated are stored in a read-only media. Throughout this paper we use N to denote the number of elements burned on the read-only media. Observe that N does *not* need to be known beforehand. If an algorithm must do some outputting, this is done on a separate write-only media. When something is printed to this media, the information cannot be read or rewritten again.

In addition to the input media and the output media, a limited random-access workspace is available. The data on this workspace is manipulated wordwise as on the word RAM [14]. We assume that the word size is at least $\lceil \lg N \rceil$ bits and that the processor is able to execute the same arithmetic, logical, and bitwise operations as those supported by contemporary imperative programming languages—like C [17]. It is a routine matter [19, Section 7.1.3] to store a bit vector of size n such that it occupies $\lceil n/w \rceil$ words and any string of at most w bits can be accessed in $O(1)$ worst-case time. That is, the *time complexity* is proportional to the number of the primitive operations plus the number of element accesses and element comparisons performed in total.

We do *not* assume the availability of any powerful memory-allocation routines. The workspace is an infinite array (of words), and the space used by an algorithm is the prefix of this array. Even though this prefix can have some unused zones, the length of the whole prefix specifies the *space complexity* of the algorithm.

Our Results. In our setting the elements lie in a read-only array and the data structure only constitutes references to these elements. We assume that each of the elements appears in the data structure at most once, and it is the user's responsibility to make sure that this is the case. Also, all operations are position-based; the position of an element can be specified by a pointer or an index. Since the positions can be used to distinguish the elements, we implicitly assume that the elements are distinct. Consider a priority queue Q. Recall that

Table 1. The performance of adjustable navigation piles (described in this paper) and their competitors in the read-only random-access model; N is the size of the read-only input and S is an asymptotic target for the size of workspace in bits where $\lg N \leq S \leq N/\lg N$.

Reference	Space	minimum	insert	extract
[5]	$\Theta(N \lg N)$	$O(1)$	$O(1)$	$O(\lg N)$
[16]	$\Theta(N)$	$O(1)$	$O(\lg N)$	$O(\lg N)$
[12]	$\Theta(S)$	$O(1)$	$O(N \lg N/S + \lg S)$	$O(N \lg N/S + \lg S)$
[24]	$\Theta(S)$	$O(N/S^2 + \lg S)$	$O(N/S + \lg S)$ amortized	$O(N/S + \lg^2 S)$
[this paper]	$\Theta(S)$	$O(1)$	$O(1)$	$O(N/S + \lg S)$

a *priority queue* is a data structure that stores a collection of elements and supports the operations *minimum*, *insert*, and *extract* defined as follows:

$Q.minimum()$**:** Return the position of the minimum element in Q.
$Q.insert(p)$**:** Insert the element at position p of the read-only array into Q.
$Q.extract(p)$**:** Extract the element at position p of the read-only array from Q.

In the non-adjustable set-up, any priority queue—like a binary heap [26] or a queue of binary heaps [5] (that are both in-place data structures)—could be used to store positions of the elements instead of the elements themselves.

The main result of this paper is a simplification of the memory-adjustable priority queue by Pagter and Rauhe [24] that is a precursor of all later constructions. First, we devise a memory-adjustable priority queue that we call an *adjustable navigation pile*. Compared to navigation piles [16], that require $\Theta(N)$ bits, our adjustable variant can achieve the same asymptotic performance with only $\Theta(N/\lg N)$ bits. (Another priority queue that uses $\Theta(N)$ bits in addition to the input array was given in [9].) Second, we use this data structure for sorting. The algorithm is priority-queue sort like heapsort [26]: Insert the N elements one by one into a priority queue and extract the minimum from that priority queue N times. In Table 1 we compare the performance of the new data structure to some of its competitors. Note that the stated bounds are valid under some reasonable assumptions that are made explicit in Section 2.

We encourage the reader to compare our solution to that of Pagter and Rauhe [24]. In the basic setting, Pagter and Rauhe proved that the running time of their sorting algorithm is $O(N^2/S + N \lg^2 S)$ using $O(S)$ bits of workspace. Lagging behind the optimal bound for the space-time product by a logarithmic factor when $S = \omega(N/\lg^2 N)$, they suggested using their memory-adjustable data structure in Frederickson's adjustable binary heap [12] to handle subproblems of size $N \lg N/S$ using $O(\lg N)$ bits for each. In accordance, by combining the two data structures, the treatment achieves an optimal $O(N^2/S + N \lg S)$ running time for sorting, where $\lg N \leq S \leq N/\lg N$. In our treatment we avoid the complication of plugging two data structures together.

Related Models. The basic feature that distinguishes the model of computation we use from other related models is the capability of having random access

to the input data. In the context of sequential-access machines, the input is on a tape that only allows single-pass algorithms. The so-called streaming model still enforces sequential access, but allows multi-pass algorithms (Munro and Paterson [20] considered this model). For some problems, the read-only random-access model is more powerful than the multi-pass sequential-access model (for example, for selection the lower bound known for the multi-pass streaming model [6] can be bypassed in the read-only random-access model [11]).

2 Memory-Adjustable Priority Queues

Assumptions. In this section two memory-adjustable priority queues are described. The first structure is a straightforward adaptation of a tournament tree (also called a selection tree [18, Section 5.4.1]) for read-only data. For a parameter S, it uses $O(S)$ words of workspace. The second structure is an improvement of a navigation pile [16] for which the workspace is $O(S + w)$ bits, $\lg N \leq S \leq N/\lg N$, where N is the size of the read-only input and w is the size of the machine word in bits. Both data structures can perform *minimum* and *insert* in $O(1)$ worst-case time and *extract* in $O(N/S + \lg S)$ worst-case time.

When describing the data structures, we tactically assume that

1. N is known beforehand.
2. The elements are extracted from the data structure in monotonic fashion. Accordingly, at any given point of time, we keep a single element as a boundary telling that the elements smaller than or equal to that element have been extracted from the data structure. We call such an element the *latest output*, and say that an element is *alive* if it is larger than the latest output.
3. The elements are inserted into the data structure sequentially—but still insertions and extractions can be intermixed—in streaming-like fashion starting from the first element stored in the read-only input.

These assumptions are valid when a priority queue is used for sorting. Actually, in sorting all insertions are executed before extractions; a restriction that is not mandated by the data structure. At the end of this section, we show how to get rid of these assumptions. The first assumption is not critical. But, when relaxing the second assumption, the required size of workspace has to increase by N bits. In addition, when relaxing the third assumption, the worst-case running time of *insert* will become the same as that required by *extract*.

Tournament Trees. For an integer S, we use $\bar{S}$ as a shorthand for $2^{\lceil \lg S \rceil}$. It suffices that $S \leq N/\lg N$; even if S was larger, the operations would not be asymptotically faster. The input array is divided into $\bar{S}$ *buckets* and a complete binary tree is built above these buckets. Each leaf of the tree *covers* a single bucket and each branch node covers the buckets covered by the leaves in the subtree rooted at that branch. We call the elements within the buckets covered by a node the *covered range* of this node. Note that the covered range of a node is a sequence of elements stored in consecutive locations of the input array. The
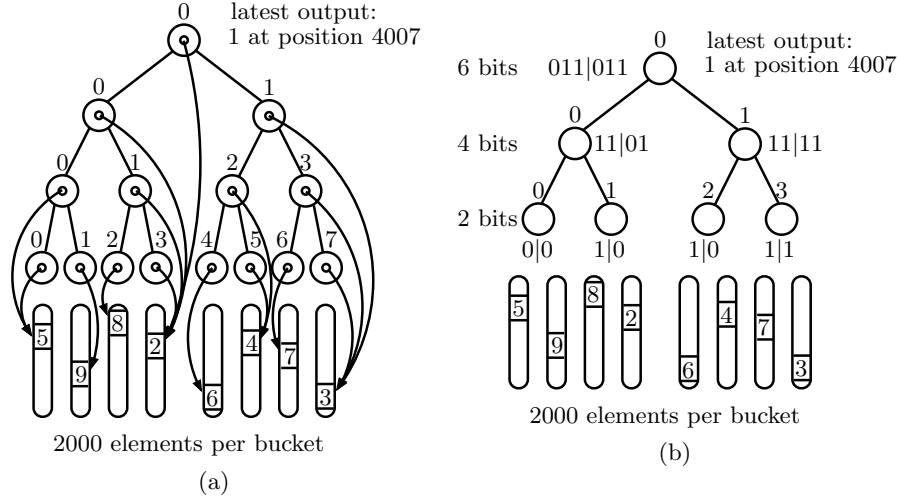
Fig. 1. (a) A tournament tree and (b) a navigation pile when $N = 16\,000$ and $\bar{S} = 8$. Only the smallest alive element in each bucket is shown.

actual data stored at each node is a pointer or an index specifying the position of the smallest alive element in the covered range of that node.

In its basic form, the data structure is an array of $2\bar{S} - 1$ positions (indices). To make the connection to our adjustable navigation piles clear, we store the positions in breadth-first order as in a binary heap [26]. We start the numbering of the nodes at each level from 0. For the sake of simplicity, we maintain a *header* that stores the pointers to the beginning of each level (even though this information could be calculated). For a node number i, its left child has number $2i$ at the level below, the right child has number $2i + 1$ at the level below, and the parent has number $\lfloor i/2 \rfloor$ at the level above. When we know the current level and the number of a node, the information available at the header and these formulas are enough to get to a neighbouring node in constant time. In Fig. 1(a) we give an illustration of a tournament tree when $N = 16\,000$ and $\bar{S} = 8$.

To support *insert* efficiently, we partition the data structure into three components: the tournament tree, the submersion buffer, and the insertion buffer. The *submersion buffer* is the last full bucket that is being integrated with the tournament tree incrementally. The *insertion buffer* is the bucket that embraces all the new elements. Observe that one or both of these buffers can be empty. The idea is to insert the elements into a buffer and, first when the buffer gets full, integrate it with the tournament tree. This buffering technique has been used in other contexts as well (see, for example, [1, 8]). The overall minimum can be in any of the three components. To support *minimum* in $O(1)$ worst-case time, we keep track of the position of the overall minimum.

In connection with *insert*, the next element from the read-only array becomes part of the insertion buffer. If the new element is smaller than the buffer mini-

mum and/or the overall minimum, the positions of these minima are updated. If the insertion buffer becomes full, the submersion buffer must have been already integrated into the tournament tree. At this point, we treat the insertion buffer as the new submersion buffer and start a bottom-up submersion process updating the nodes of the tournament tree that cover this bucket. Every branch node inherits the position of the smaller of the two elements pointed to via its two children. As long as the submersion is not finished, each *insert* is responsible for continuing a constant amount of the submersion work. Since the work needed to update this path is $O(\lg S) = O(N/S)$, the process terminates before the insertion buffer is again full. Clearly, *insert* takes $O(1)$ worst-case time.

In *extract*, simple calculations are done to determine which bucket covers the element being extracted. The latest output is first set up to date. Since the smallest alive element of the present bucket must have been extracted, the bucket is scanned to find its new minimum. There are three cases depending on whether the present bucket is one of the buffers or covered by the tournament tree. If the present bucket is the insertion buffer, it is just enough to update the position of its minimum. If the present bucket is the submersion buffer, the submersion process is completed by recomputing the positions in the nodes of the tournament tree that cover the submersion buffer. Hereafter, the submersion buffer ceases to exist. If the present bucket is covered by the tournament tree, it is necessary to redo the comparisons at the branch nodes that cover the present bucket. In all three cases the position of the overall minimum is updated, if necessary. It is the scanning of a bucket that makes this operation expensive: The worst-case running time is $O(N/S + \lg S)$, which is $O(N/S)$ when $S \leq N/\lg N$.

Navigation Piles. In principle, a navigation pile [16] is a compact representation of a tournament tree. The main differences are (see Fig. 1(b)):

1. Only the nodes whose heights are larger than 0 have a counterpart. Hence, the number of nodes in the complete binary tree is $\bar{S} - 1$.

2. Any node only stores partial information about the position of the smallest alive element in the covered range of that node. Here our construction differs from that used in the navigation piles [16] and their precursors [24].

A branch node of height $h \in \{1, 2, \ldots, \lg \bar{S}\}$ covers 2^h buckets. As in the original navigation piles, we use h bits to specify in which bucket the smallest alive element is. A significant ingredient is the concept of a *quantile*. (A similar quantile-thinning technique was used in [24], but not in an optimal way, and later in [10].) For a branch node of height h, every covered bucket is divided into 2^h quantiles, and another h bits are used to specify in which quantile the smallest alive element is. That is, except that the last quantile can possibly be smaller, a quantile contains $\lceil N/(\bar{S} \cdot 2^h) \rceil$ elements. We need $2h$ bits per node; but if $2h \geq \lceil \lg N \rceil$, we do not use more than $\lceil \lg N \rceil$ bits (since this is enough to specify the exact position of the smallest alive element). To sum up, since there are $\bar{S}/2^h$ nodes of height h and since at each node we store $\min \{2h, \lceil \lg N \rceil\}$ bits,

the total number of bits is bounded by

$$\sum_{h=1}^{\lg \bar{S}} \frac{\bar{S} \cdot \min\{2h, \lceil \lg N \rceil\}}{2^h} < 4\bar{S}.$$

The navigation bits are stored in a bit vector in breadth-first order. As before, we maintain a header giving the position of the first bit at each level. The space needed by the header is $O(\lg^2 \bar{S})$ bits. Inside each level the navigation information is stored compactly side by side, and the nodes are numbered at each level starting from 0. Since the length of the navigation bits is fixed for all nodes at the same level, using the height and the number of a node, it is easy to calculate the positions where the navigation bits of that node are stored.

Let us now illustrate how to access the desired quantile for a branch node in constant time. Let the number of the branch node be x within its own level, and assume that its height is h. The first element of the covered range is in position $x \cdot 2^h \cdot \lceil N/\bar{S} \rceil$. The first h bits of the navigation information gives the desired bucket; let this bucket index be b, so we have to go $b \cdot \lceil N/\bar{S} \rceil$ positions forward. The second h bits of the navigation information gives the desired quantile inside that bucket; let this quantile index be q, so we have to proceed another $q \cdot \lceil N/(\bar{S} \cdot 2^h) \rceil$ positions forward before we reach the beginning of the desired quantile. Obviously, these calculations can be carried out in constant time.

The priority-queue operations can be implemented in a similar way as for a tournament tree. To facilitate constant-time *minimum*, we can keep a separate pointer to the overall minimum (since the root of an adjustable navigation pile does not necessarily specify a single element). One subtle difference is that, when we update a path from a node at the bottom level to the root, we have to scan the elements in the quantiles specified for the sibling nodes of the nodes along the path. After updating the navigation bits of a node y, we locate its parent x and its sibling z. The navigation bits at z are used to locate the quantile that has the minimum element covered by z. This quantile is scanned and the minimum element is found and compared with the minimum element covered by y. From the bucket number and the position of the smaller of the two elements, the navigation bits at x are then calculated and accordingly updated. If the quantile for x has only one element, the position of this single element can be stored as such. The key is that for a node of height h, the size of the quantile is $\lceil N/(\bar{S} \cdot 2^h) \rceil$, so the total work done in the scans of the quantiles of the siblings along the path is proportional to $\lceil N/\bar{S} \rceil$ as it should be. It follows that the efficiency of the priority-queue operations is the same as for a tournament tree.

Getting Rid of the Assumptions. So far we have consciously ignored the fact that the size of the buckets depends on the value of N, and that we might not know this value beforehand. The standard way of handling this situation is to rely on global rebuilding [23, Chapter V]. We use an estimate N_0 and initially set $N_0 = 8$. We build two data structures, one for N_0 and another for $2N_0$. The first structure is used to perform the priority-queue operations, but insertions and extractions are mirrored in the second structure (if the extracted element exists there). When the structure for N_0 becomes too small, we dismiss the smaller

structure in use, double N_0, and in accordance start building a new structure of size $2N_0$. We should speed up the construction of the new structure by inserting up to two alive elements into it at a time, instead of only one. This guarantees that the new structure will be ready for use before the first one is dismissed. Even though global rebuilding makes the construction more complicated, the time and space bounds remain asymptotically the same.

A possible scenario in applications is when extractions are no more monotonic. To handle this situation, we have to allocate one bit per array entry, indicating whether the corresponding element is alive or not. This increases the size of the workspace significantly if S is much smaller than N.

Since we have random-access capability to the read-only input, it is not absolutely necessary that elements are inserted into the data structure by visiting the input sequentially. We could insert the elements in arbitrary order. If this is the case, in connection with each *insert* we have to fix the information related to the present bucket as in *extract*. That is, we have to find the smallest alive element of the bucket and update the navigation information on the path from a node at the bottom level to the root. This means that the worst-case cost of *insert* becomes the same as that of *extract*, i.e. $O(N/S + \lg S)$.

3 Sorting

Priority-Queue Sort. To sort the given N elements, we create an empty adjustable navigation pile, insert the elements into this pile by scanning the read-only array from beginning to end, and then repeatedly extract the minimum of the remaining elements from the pile. See the pseudo-code in Fig. 2.

Analysis. From the bounds derived for the priority queue, the asymptotic performance can be directly deduced: The worst-case running time is $O(N^2/S + N \lg S)$ and the size of workspace is $O(S + w)$ bits. It is also easy to count the number of element comparisons performed during the execution of the algorithm. When inserting the N elements into the data structure, $O(N)$ element comparisons are performed. We can assume that after these insertions, the buffers are integrated into the main structure. In each *extract* we have to find the minimum

procedure: *priority-queue-sort*
input: A: read-only array of N elements; S: space target
output: stream of elements produced by the *print* statements
$P \leftarrow navigation\text{-}pile(A, S)$
for $x \in \{0, 1, \ldots, N - 1\}$
  $\mid$ $P.insert(x)$
repeat N times
  $\mid$ $y \leftarrow P.minimum()$
  $\mid$ $P.extract(y)$
  $\mid$ $print(A[y])$

Fig. 2. Priority-queue sort in pseudo-code; the position of an element is its index.

of a single bucket which requires at most $N/\bar{S}$ element comparisons. In addition, we have to update a single path in the complete binary tree. At each level, the minimum below the current node is already known and we have to scan the quantiles of the sibling nodes. During the path update, we have to perform at most $N/\bar{S} + \lg \bar{S}$ element comparisons. We know that $S \leq \bar{S} \leq 2S$. Hence, the total number of element comparisons performed is bounded by $2N^2/S + N \lg S + O(N)$.

4   Concluding Remarks

Summary. In the construction of adjustable navigation piles three techniques are important: 1) node numbering with implicit links between nodes, 2) bit packing and unpacking, and 3) quantile thinning. In addition to the connection to binary heaps [26], we pointed out the strong connection to tournament trees. We made the conditions explicit for when a succinct implementation of a priority queue requiring $o(N)$ bits is possible (when extractions are monotonic), so that algorithm designers would be careful when using the data structure.

Our sorting algorithm for the read-only random-access model is a heapsort algorithm [26] that uses an adjustable navigation pile instead of a binary heap. In spite of optimality, one could criticize the model itself since the memory-access patterns may not always be friendly to contemporary computers; and we are not allowed to move the elements. Navigation piles are slow [15] for two reasons: 1) The bit-manipulation machinery is heavy and index calculations devour clock cycles. 2) The cache behaviour is poor because the memory accesses lack locality. Unfortunately, the situation is not much better for adjustable navigation piles; the buckets are processed sequentially, but the quantiles lie in different buckets.

Other Data Structures for Read-Only Data. In our experience, very few data structures can be made memory adjustable as elegantly as priority queues. A stack is another candidate [3]. For example, a dictionary must maintain a permutation of a set of size N; this means that it is difficult to manage with much less than $N \lceil \lg N \rceil$ bits. However, when the goal is to cope with about N bits, a bit vector extended with rank and select facilities (for a survey, see e.g. [21]) is a relevant data structure. Two related constructions are the wavelet stack used in [11] and the wavelet tree introduced in [13] (for a survey, see [22]).

References

1. Alstrup, S., Husfeldt, T., Rauhe, T., Thorup, M.: Black box for constant-time insertion in priority queues. ACM Trans. Algorithms 1(1), 102–106 (2005)
2. Asano, T., Buchin, K., Buchin, M., Korman, M., Mulzer, W., Rote, G., Schulz, A.: Memory-constrained algorithms for simple polygons. E-print arXiv:1112.5904, arXiv.org, Ithaca (2011)
3. Barba, L., Korman, M., Langerman, S., Sadakane, K., Silveira, R.: Space-time trade-offs for stack-based algorithms. E-print arXiv:1208.3663, arXiv.org, Ithaca (2012)
4. Beame, P.: A general sequential time-space tradeoff for finding unique elements. SIAM J. Comput. 20(2), 270–277 (1991)

5. Carlsson, S., Munro, J.I., Poblete, P.V.: An implicit binomial queue with constant insertion time. In: Karlsson, R., Lingas, A. (eds.) SWAT 1988. LNCS, vol. 318, pp. 1–13. Springer, Heidelberg (1988)

6. Chan, T.M.: Comparison-based time-space lower bounds for selection. ACM Trans. Algorithms 6(2), 26:1–26:16 (2010)

7. De, M., Nandy, S.C., Roy, S.: Convex hull and linear programming in read-only setup with limited work-space. E-print arXiv:1212.5353, arXiv.org, Ithaca (2012)

8. Edelkamp, S., Elmasry, A., Katajainen, J.: Weak heaps engineered (submitted for publication)

9. Elmasry, A.: Three sorting algorithms using priority queues. In: Ibaraki, T., Katoh, N., Ono, H. (eds.) ISAAC 2003. LNCS, vol. 2906, pp. 209–220. Springer, Heidelberg (2003)

10. Elmasry, A., He, M., Munro, J.I., Nicholson, P.K.: Dynamic range majority data structures. In: Asano, T., Nakano, S.-i., Okamoto, Y., Watanabe, O. (eds.) ISAAC 2011. LNCS, vol. 7074, pp. 150–159. Springer, Heidelberg (2011)

11. Elmasry, A., Juhl, D.D., Katajainen, J., Satti, S.R.: Selection from read-only memory with limited workspace. In: Du, D.Z., Zhang, G. (eds.) COCOON 2013. LNCS, Springer, Heidelberg (2013)

12. Frederickson, G.N.: Upper bounds for time-space trade-offs in sorting and selection. J. Comput. System Sci. 34(1), 19–26 (1987)

13. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: SODA 2003. pp. 841–850. SIAM, Philadelphia (2003)

14. Hagerup, T.: Sorting and searching on the word RAM. In: Morvan, M., Meinel, C., Krob, D. (eds.) STACS 1998. LNCS, vol. 1373, pp. 366–398. Springer, Heidelberg (1998)

15. Jensen, C., Katajainen, J.: An experimental evaluation of navigation piles. CPH STL Report 2006-3, Department of Computer Science, University of Copenhagen, Copenhagen (2006)

16. Katajainen, J., Vitale, F.: Navigation piles with applications to sorting, priority queues, and priority deques. Nordic J. Comput. 10(3), 238–262 (2003)

17. Kernighan, B.W., Ritchie, D.M.: The C Programming Language. Prentice Hall, Englewood Cliffs, 2nd edn. (1988)

18. Knuth, D.E.: Sorting and Searching, The Art of Computer Programming, vol. 3. Addison-Wesley, Reading, 2nd edn. (1998)

19. Knuth, D.E.: Combinatorial Algorithms: Part 1, The Art of Computer Programming, vol. 4A. Addison-Wesley, Boston (2011)

20. Munro, J.I., Paterson, M.S.: Selection and sorting with limited storage. Theoret. Comput. Sci. 12(3), 315–323 (1980)

21. Navarro, G., Providel, E.: Fast, small, simple rank/select on bitmaps. In: Klasing, R. (ed.) SEA 2012. LNCS, vol. 7276, pp. 295–306. Springer, Heidelberg (2012)

22. Navarro, G.: Wavelet trees for all. In: Kärkkäinen, J., Stoye, J. (eds.) CPM 2012. LNCS, vol. 7354, pp. 2–26. Springer, Heidelberg (2012)

23. Overmars, M.H.: The Design of Dynamic Data Structures, LNCS, vol. 156. Springer, Heidelberg (1983)

24. Pagter, J., Rauhe, T.: Optimal time-space trade-offs for sorting. In: FOCS 1998. pp. 264–268. IEEE Computer Society, Los Alamitos (1998)

25. Savage, J.E.: Models of Computation: Exploring the Power of Computing (2008), `http://cs.brown.edu/~jes/book/home.html`, the book is released in electronic form under a CC-BY-NC-ND-3.0-US license

26. Williams, J.W.J.: Algorithm 232: Heapsort. Commun. ACM 7(6), 347–348 (1964)