External heaps combined with effective buffering

Ramzi Fadel
Department of Computer Science
University of Copenhagen
Universitetsparken 1
DK-2100 Copenhagen East, Denmark
ramzi@diku.dk

Kim Vagn Jakobsen
Department of Computer Science
University of Copenhagen
Universitetsparken 1
DK-2100 Copenhagen East, Denmark
kvj@vki.dk

Jyrki Katajainen
Department of Computer Science
University of Copenhagen
Universitetsparken 1
DK-2100 Copenhagen East, Denmark
jyrki@diku.dk

Jukka Teuhola
Department of Computer Science
University of Turku
Lemminkäisenkatu 14 A
FIN-20520 Turku, Finland
teuhola@cs.utu.fi

Abstract

An external heap structure is suggested that tries to make the best use of the available buffer space in main memory. The heap is a multi-way balanced tree, with blocks of records as nodes, satisfying a generalized heap property. The root is buffered, whereas other nodes are kept on secondary storage. A special property of the tree is that the nodes may be partially filled, as with B-trees. The structure is complemented with priority queue operations insert and delete-max. When handling a sequence of these operations, the amortized number of page accesses per operation is shown to be $O((1/P)\log_{(M/P)}(N/P))$, where P denotes the number of records fitting into a page, M the capacity of the buffer space in records, and N the largest number of records in the heap ($4P < M < N$). This results in optimal external heapsort that performs $O((N/P)\log_{(M/P)}(N/P))$ page accesses in the worst case, when sorting N records.

Keywords External-storage data structures, external sorting, priority queues, heaps, heapsort.

1 Introduction

The traditional data structure for implementing *priority queues* is the *heap* (see, e.g., [7]). It is a complete binary tree with the *heap property*: The priority of a parent is always higher than or equal to the priorities of its children. Thus the root contains the maximum. Of course, the order can also be the opposite — one may talk about *max-heaps* and *min-heaps*. The two important priority queue operations (in addition to creation) against a max-heap are (1) *insert*, which inserts a record with an arbitrary priority into the heap, and (2) *delete-max*, which extracts the record with the highest priority from the heap. In both cases, the heap property should be restored. Perhaps the best-known application of the heap structure is *heapsort* [8, 13] which is one of the few *in-place* sorting methods guaranteeing an $O(N \log_2 N)$ worst case, when sorting N records in main memory.

However, there are some applications, for example, large *minimum spanning tree* problems and extremely large *sorting* tasks, where the data collection may be too large to fit in main memory. On secondary storage, the typical measure of complexity is the number of *page accesses*. For this reason, the internal algorithms are not applicable as such. Our intention is to generalize the heap into an effective external data structure. In part, this was already done by Wegner and Teuhola in their *external heapsort* [12]. Their heap had the same structure as the internal heap, namely a balanced binary tree, but the nodes were extended to whole pages, and node comparisons were replaced by node merges. A clear advantage of external heapsort over external mergesort is that the former operates in *minimum space*. Another "in-situ" sorting algorithm was presented in [10], based on quicksort.

The external heapsort in [12] cannot be improved if we assume that the buffer space in main memory has a fixed size. What happens, if we express the complexity as a function of both problem size N (in records) and buffer-space capacity M (in records), keeping the page size P fixed? We could keep the top part of

Proceedings of CATS'97 (Computing: The Australasian Theory Symposium), Sydney, Australia, February 3–4 1997.

the heap always in main memory, resulting in $O((N/P)\log_2(N/M))$ page accesses. This is, however, asymptotically worse than the best possible bound $\Theta((N/P)\log_{(M/P)}(N/P))$, obtained by external (M/P)-way mergesort [1].

The behavior of (internal) heapsort in virtual memory environment was studied in [4], where it was noticed to require $O(N\log_2(N/P))$ page accesses for N records. Thus, this approach is not competitive with tailored external heapsort.

Our intention is to create an external heap organization that tries to make the best use of the available main memory. Especially, we try to achieve the same complexity for external heapsort as for mergesort. We will adopt some features from *B-trees* [5], which have become the standard comparison-based external search structure. Their virtues are *balance*, *large fanout* (implying short paths from root to leaf), and *flexibility*, due to the "slack" allowed in the loading factor of pages (usually between 0.5 and 1). It turns out that all these properties can be transferred to heaps. One may wonder, how a B-tree would manage as a priority queue. The maximum is easily found from the rightmost leaf (which could be buffered). Inserting (as well as deleting) records is quite efficient. However, a more careful study reveals that the B-tree cannot compete with the heap to be described. The B-tree contains "too much" order, and maintaining that order does not pay off.

The performance of our heap structure is as follows. When handling any sequence of insert and delete-max operations the amortized number of page accesses per operation is $O((1/P)\log_{(M/P)}(N/P))$, where P denotes the number of records fitting into a page, M the capacity of the buffer space in records, and N the largest number of records in the heap ($4P < M < N$). This results in external heapsort that performs $O((N/P)\log_{(M/P)}(N/P))$ page accesses in the worst case, when sorting N records.

A data structure with a similar performance as ours has been independently developed by Arge [2, 3]. The basic difference is that he expresses the complexity of the priority queue operations as a function of P, M, and N_0, where N_0 denotes the accumulated number of operations carried out on the structure. In sorting this difference is not essential, since the total number of operations and the maximum size of the structure are about the same. However, his data structure is quite complicated and mainly of theoretical value, whereas the heap structure explored in this paper is practical. The experimental results will be reported in the full version of this paper.

The rest of the paper is organized as follows. The new data structure is described in Section 2. In Section 3 the procedures for accomplishing the two priority queue operations, insert and delete-max, are presented. The external and internal complexities of these operations, as well as that of external heapsort, are analysed in Sections 4 and 5, respectively. In Section 6 some conclusions are drawn and extensions to the repertoire of operations are discussed.

2 Data structure

We assume that the elements to be stored in the heap are fixed-size records, each having a *priority* attribute. Priorities need not be unique; ties are broken arbitrarily in delete-max. The fixed-size assumption is not absolutely necessary, but allowing variable-size records would complicate the presentation. The following notations will be used:

- N: the total number of records,
- P: page size (the number of records fitting into a page),
- $n = \lceil N/P \rceil$ (the minimum number of pages to store N records),
- m: the number of pages per block (a collection of pages kept in a node); also the fanout of heap nodes,
- M: the capacity of available main memory in records; $M = cPm$, where constant c is around 3.

Notice that M is determined first, after which we can calculate, how big m we can afford. Hereafter we assume that $cP < M < N$ and that the size of a record is larger than the size of a pointer.

The main part of the data structure (see Fig. 1) consists of a heap with the following properties:

- The degree (fanout) of the nodes is m.
- Each node consists of six parts: (a) a *block* of m pages, containing records in ascending order of priority; (b) m pointers to its children; (c) m pointers to the last *records* of the children, that is, a page and an offset inside this page is specified; (d) a pointer to its parent, (e) a pointer to its predecessor with respect to the normal numbering of nodes in a heap; and (f) a pointer to the corresponding successor.
- The *generalized heap property* holds: For any record x in a node v and any record y in a child of v, the priority of x is larger than or equal to that of y.
- The heap is completely balanced: The nodes on the lowest level are arranged to the left, as

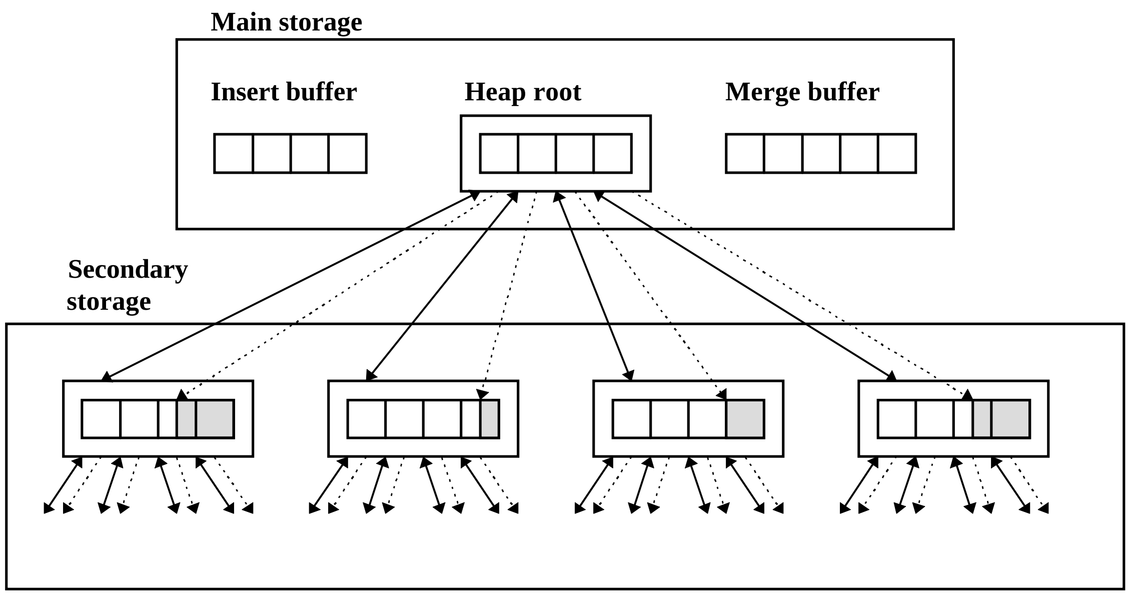

Figure 1: The internal and external data structures when $m = 4$.

in normal binary heaps. Therefore, the position of the *last* node of the heap is uniquely defined, and we can maintain a pointer to it. The parent of the last node is the only node whose degree may be between 1 and m, all other nodes have m or no children.

- Each node, except the root and the last leaf, is at least half full, i.e., they contain at least $\lceil Pm/2 \rceil$ records. This is called the *load condition*. A node with (temporarily) less records is said to be *imperfect*.
- The root is always kept in main storage. Its records are maintained in ascending order of priority, as in the other nodes.
- The pages within a block are either physically consecutive, or two-way linked, so that we can move from page to page in both directions. The latter alternative would avoid wasting storage space, because the empty pages at the end of each block could be released and reused.

In addition to the root, main storage contains two buffers. New records are not immediately inserted in the heap, but gathered in an *insert buffer* consisting of m pages. When this buffer space gets full, the contained records are stored in the heap as a batch. The records in the insert buffer are organized as a normal (binary) heap, because we have to look for its maximum priority. The other buffer is needed when the heap is manipulated. As in [12], moving records up or down requires *merging* of blocks. Here we need an auxiliary *merge buffer* of $m + 1$ pages.

3 Priority queue operations

The two operations to be executed against the described data structure are insertion of a record with any priority, and extraction of a record having the highest priority. During the operations, records will be moving up and down the heap. *Inspecting* (without removing) the highest priority is often included in the repertoire of priority-queue operations, but since it does not involve any page accesses, it is uninteresting for us.

Insert

Inserted records are stored first in the related buffer of m pages. When this buffer becomes full, it is first sorted internally (by heapsort) and then the sorted outcome is transferred to the heap as its new last leaf. To restore the heap property (also called "heapifying" [7]), records are *sifted up* as follows. We merge the block of the last leaf with that of its parent (using the merge area in main storage). If the parent had k records before the sift-up, we move k of the merged records with the *largest* priorities to the parent, and the rest to the child. Thus, *the size of the parent block does not change*. Observe also that the minimum priority in the parent can only increase in this process, and thus its children will all satisfy the heap property. However, the sift-up must be repeated for the parent and its grandparent, etc., up to the root.

One point in the above procedure needs elaboration. When defining the heap in Section 2, we stated that the last leaf (L) may be imperfect. Now, having created a new last leaf (L'), we must check whether L satisfies the load condition. If it does not, we swap the two (actually the pointers in their parents), and sift-up *both*, one at a time. The sift-up of the last leaf propagates upwards only in the case that its parent is new.

Delete-max

Due to the heap property, the maximum priority is either in the root (if not empty), or in the insert buffer. Since the root is ordered and the insert buffer is an internal heap, the maximum is easily found and extracted. If the root block is empty, we have to *refill* it before delete-max. We move at least $\lceil Pm/2 \rceil$ records with the largest priorities from the children to the root. Each of the children has at least this amount of records (except in the special case when the last leaf, possibly imperfect, is among the children, but correct operation is easily guaranteed). Due to the heap property, no lower-level node (grandchild, etc.) may contain a higher priority. After refilling the root, it may happen that one or more children have become imperfect, in turn, and must be refilled, recursively.

How do we find the $\lceil Pm/2 \rceil$ largest priorities? The records in blocks are arranged in ascending order. Moreover, we maintain a pointer to the last record of each block. Therefore we can merge the m child blocks from back to front, until the parent block is filled, or one child becomes empty. In the special case that the last leaf is the only child, which is imperfect, the parent becomes a leaf, and the normal procedure for imperfect leaves is to fill them from the current last leaf. Otherwise, if the last leaf becomes empty, the process need not be stopped, and therefore we always get at least $\lceil Pm/2 \rceil$ records to the parent. Notice that most of the front pages in the child blocks need not be touched at all; this is important in respect of the complexity.

Now the question remains, what happens when a leaf X becomes imperfect. If X is accidentally the last leaf, then we do not have to do anything; this is the exception to the loading factor condition. Otherwise, we have to "borrow" records from some other leaf. There are several alternatives to do this, but we suggest the following simple procedure. Let $|X|$ denote the number of records in node X. Now we calculate the sum $S = |X| + |L|$, and depending on the value of S there are three possibilities:

1. If $S > Pm$, move $\lfloor S/2 \rfloor - |X|$ largest-priority records from L to X (i.e., an even split), and sift-up X.
2. If $S \in \{\lceil Pm/2 \rceil, \ldots, Pm\}$, then merge the blocks of X and L into X, and sift-up X (deleting L).
3. If $S < \lceil Pm/2 \rceil$, then merge the blocks of X and L into X and delete L. Find the new last leaf L' (predecessor of L), and repeat the process for X and L'. This is guaranteed to succeed, because either $X = L'$ or $|L'| \geq \lceil Pm/2 \rceil$. After filling X, it can be sifted up.

We have not explained all details in the above descriptions concerning the maintenance of pointers, the arrangement of merges, as well as the allocation and release of storage. However, the inclusion of these features is relatively straightforward, so the description of these is omitted.

4 External complexity

The external costs are measured in terms of page accesses (reads and writes). Only *amortized* costs (cf. [11]) will be determined — the worst case of a single operation can be really bad; for example in delete-max, the refilling may propagate to *all* nodes of the heap. It depends on the application, whether this is important or not. For instance, for external heapsort, only the amortized cost counts.

Due to amortizing, the cost of a specific operation does not become real until at some future restructuring of the heap. Now, since the heap may grow and shrink in between, it is not clear what the number of records N (or the number of pages n) stands for. Therefore, we specify that N is the *largest* number of records in the heap at any moment during the usage of the heap.

Theorem 1. *The amortized external cost of record insert is $O((1/P)\log_m n)$.*

Proof. The inserts are buffered in blocks of Pm records. When the insert buffer becomes full, we perform one (in a special case two) sift-up chain from a leaf to the root. In a chain, we access $h-1$ or h blocks, i.e., at most mh pages, where h is the current height of the tree. Therefore, the amortized number of page accesses per insert is at most $2mh/(Pm)$, which is $O((1/P)\log_m n)$, since $h = O(1 + \log_m(n/m))$. ■

Theorem 2. *The amortized external cost of delete-max is $O((1/P)\log_m n)$.*

Proof. Delete-max causes page accesses only when the root (in main storage) becomes empty, and its block must be refilled. For the purposes of the proof, we assume that a *fixed* number of records, namely $\lceil Pm/2 \rceil$ are moved up at each refill. This assumption corresponds to the worst case that can happen. The drawback is that the loading factor of blocks tends to be lower than in reality. In the algorithm even more than $\lceil Pm/2 \rceil$ records can be lifted up from the children, if there is room in the parent block, and no child block becomes empty. Moreover, sometimes the maximum may be found in the insert buffer, which also reduces the number of refillings. In refilling a block, at most $2m$ pages in the child blocks are touched, because the required $\lceil Pm/2 \rceil$ highest-priority records are found among those (starting from the rear pages of blocks).

The critical question is, how many child blocks have to be refilled, in turn. As mentioned above, the worst case is when all the heap blocks are half-full, and all have to be refilled. However, we claim that only one child needs to be refilled, on the average. This can be easily concluded when studying several, say k, successive refillings of the *same* parent. During this sequence, $k\lceil Pm/2 \rceil$ records are moved to the parent. Now, if the child set (regarded as a whole) were refilled more often, say $(1+\epsilon)k$ times, then the number of records in the child blocks would rise to at least $m\lceil Pm/2 \rceil + k\epsilon\lceil Pm/2 \rceil$, which is larger than Pm^2 when $k > m/\epsilon$, i.e., the capacity of child blocks would be exceeded, resulting in contradiction.

Thus, it must hold that $\epsilon = 0$, implying that the number of parent refills is asymptotically the same as the number of refills for the whole child set. This can be considered an application of *Kirchhoff's law* of confluent flows (for other applications of Kirchhoff's law in algorithms, see [9]). For one root refill, we perform one refill per each level, on the average. The asymptotic amortized refill cost per record is proportional to

$$\frac{2m \log_m n}{\lceil Pm/2 \rceil} = O((1/P)\log_m n).$$

It should be emphasized that insert operations do not invalidate the above deduction, because the sift-up operations do not change the sizes of blocks on the way up. The size of the last leaf may change, but only increase, so reducing the need for refill. ∎

Our starting point was the external heapsort by Wegner and Teuhola [12], the complexity of which was shown to be $O(n \log_2 n)$ accesses for n pages and $O(1)$ buffer pages. Now we get a stronger result:

Corollary 3. *Given N records stored on n pages, external heapsort can sort these with $O(n \log_m n)$ page accesses by using $O(n/m)$ extra pages for records and $O(m)$ buffer pages.*

Proof. External heapsort sorts the given records by performing first N insert operations and then N delete-max operations. The organization of the computation is as in internal heapsort. The costs of N insert and N delete-max operations add up to $O((N/P)\log_m n + (N/P)\log_m n) = O(n \log_m n)$ accesses. As to the space complexity, each node requires some space for pointers and it can contain at most one half-full page of records (the linked organization of block pages is used here, avoiding empty pages at the block rear). Since the number of nodes is $O(n/m)$, the space bound follows. Our calculations do not take into account the space consumed by the pointers. However, it is not difficult to modify the heap organization — without effecting the time bound — such that only a constant amount of pointers is used at each page. We leave the elaboration of these details for an interested reader. ∎

This result is important in the sense that the external complexity is the same as for m-way external mergesort, which is known to be optimal [1]. However, the space complexity of external heapsort is a bit worse than that of external mergesort, since the latter requires only $O(m)$ buffer pages plus a constant number of pointers within each page.

5 Internal complexity

The internal costs of insert and delete-max are counted as the number of priority comparisons. Notice that the number of record moves cannot be higher than a constant times the number of comparisons, because the decision about record movement is done only after its priority has been compared with some other. Altogether, the total number of all internal operations performed is proportional to that of comparisons.

Theorem 4. *The amortized internal cost of insert is $O(\log_2(Pm) + \log_m n)$.*

Proof. We can divide the comparisons carried out into two parts:

1. Comparisons in the insert buffer: Since the insert buffer is organized as an internal heap, an insert costs $O(\log_2(Pm))$ comparisons.
2. Comparisons during sift-up operations: First, the insert buffer is sorted internally using $O(Pm \log_2(Pm))$ comparisons. A sift-up step means that parent and child blocks are merged, using the merge buffer, and then the records are returned back to the two blocks. This is a linear operation, and therefore, deducing from Theorem 1, the number of comparisons is $O(Pm \log_m n)$, in the whole sift-up chain. When this is amortized over $\Theta(Pm)$ inserted records, we get $O(\log_2(Pm) + \log_m n)$ comparisons, on the average, completing the proof. ∎

Theorem 5. *The amortized internal cost of delete-max is $O(\log_2(Pm) + \log_2 n)$.*

Proof. Again we inspect two sorts of comparisons:

1. Comparisons in the buffers: The maximum priority is found either from the root, or from the insert buffer, with one comparison. However, keeping the insert buffer (heap) in shape costs $O(\log_2(Pm))$ comparisons.

2. Comparisons during refillings: Refilling of one block is an m-way merge of $O(Pm)$ records. This costs $O(Pm \log_2 m)$ comparisons. Gathering up the refill costs on the path down, deducing from Theorem 2, we get $O(Pm \log_2 m \log_m n) = O(Pm \log_2 n)$. The amortized cost per deleted record is thus $O(\log_2 n)$.

This is $O(\log_2(Pm) + \log_2 n)$ in total. Refilling a leaf causes a sift-up, but according to Theorem 4, the cost of this is $O(\log_2(Pm) + \log_m n)$, which does not increase the overall complexity. ■

Applying the above two results, we get

Corollary 6. *External heapsort sorts N records in $O(N \log_2 N)$ internal time.*

Proof. From Theorems 4 and 5 we see that the delete-max cost dominates the insert cost. The direct multiplication of N by the amortized cost per delete-max gives $O(N \log_2 N)$. ■

The internal running time of external heapsort is asymptotically the same as that required by internal heapsort. Actually, if $N \leq Pm$, only the root remains, and the method degenerates to normal internal heapsort: The heap is built into the insert buffer and then sorted (by heapsort) into the root.

6 Conclusion and further work

We have described an external priority queue organization, which is a natural generalization of the traditional heap organization in main memory. Multi-page nodes with a large fanout imply a very low height for the heap, which keeps the number of page accesses low. The key point is an effective utilization of the main memory. The obtained complexities for the two priority queue operations can be considered to be satisfactory for two reasons:

- The two external complexities are balanced.
- The operations guarantee an optimal external heapsort.

Either of the two operations can, of course, be made more efficient, at the cost of the other. Also, our frame of reference includes only comparison-based techniques. For special priority distributions better results may be obtained by other means.

It would be of interest to develop efficient algorithms for maintaining some special types of priority queues on secondary storage. The applications we have had in mind are that of finding a minimum spanning tree in an undirected graph and that of computing a shortest path tree in a directed graph. The standard solutions for these problems (see, e.g., [7]) use a priority queue which, in addition to insert and delete-min, supports an operation for decreasing priority values. This presupposes that the records also contain a *unique key* (or address) and that there exists a search mechanism for the records by this key. However, we have not been able to develop a data structure which could be used to solve, for example, the minimum-spanning-tree problem faster than by the method of Chiang et al. [6].

Acknowledgement

This work was partially supported by the Danish Natural Science Research Council under contract No. 9400952 (Project "Computational Algorithmics").

References

[1] A. Aggarwal and J.S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, Volume 31, pages 1116–1127, 1988.

[2] L. Arge. External-storage data structures for plane-sweep algorithms. Technical Report A94 RS-94-16, Department of Computer Science, University of Aarhus, Århus, 1994.

[3] L. Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *Proceedings of the 4th Workshop on Algorithms and Data Structures*, Lecture Notes in Computer Science 955, pages 334–345, Springer, Berlin, 1995.

[4] T.O. Alanko, H.H.A. Erkiö and I.J. Haikala. Virtual memory behavior of some sorting algorithms. *IEEE Transactions on Software Engineering*, Volume SE-10, pages 422–431, 1984.

[5] R. Bayer, and E.M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, Volume 1, pages 173–189, 1972.

[6] Y.-J. Chiang, M.T. Goodrich, E.F. Grove, R. Tamassia, D.E. Vengroff, and J.S. Vitter. External-memory graph algorithms. In *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 139–149, ACM, New York and SIAM, Philadelphia, 1995.

[7] T.H. Cormen, C.E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, 1990.

[8] R.W. Floyd. Algorithm 245, Treesort 3. *Communications of the ACM*, Volume 7, page 701, 1964.

[9] D.E. Knuth. *The Art of Computer Programming*, Volume 1: *Fundamental Algorithms*. Addison-Wesley Publishing Company, Reading, 1968.

[10] H.W. Six, and L. Wegner. Sorting a random access file *in situ*. *The Computer Journal*, Volume 27, pages 270–275, 1984.

[11] R.E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, Volume 6, pages 306-318, 1985.

[12] L.M. Wegner and J.I. Teuhola. The external heapsort. *IEEE Transactions on Software Engineering*, Volume 15, pages 917–925, 1989.

[13] J.W.J. Williams. Algorithm 232, Heapsort. *Communications of the ACM*, Volume 7, pages 347–348, 1964.