ELSEVIER

Theoretical Computer Science 220 (1999) 345–362

Theoretical Computer Science

www.elsevier.com/locate/tcs

Heaps and heapsort on secondary storage ☆

R. Fadel[a], K.V. Jakobsen[a], J. Katajainen[a,1], J. Teuhola[b,*]

[a] *Department of Computing, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen East, Denmark*

[b] *Department of Computer Science, University of Turku, Lemminkäisenkatu 14 A, FIN-20520 Turku, Finland*

Abstract

A heap structure designed for secondary storage is suggested that tries to make the best use of the available buffer space in primary memory. The heap is a complete multi-way tree, with multi-page blocks of records as nodes, satisfying a generalized heap property. A special feature of the tree is that the nodes may be partially filled, as in B-trees. The structure is complemented with priority-queue operations insert and delete-max. When handling a sequence of S operations, the number of page transfers performed is shown to be $O(\sum_{i=1}^{S}(1/P)\log_{(M/P)}(N_i/P))$, where P denotes the number of records fitting into a page, M the capacity of the buffer space in records, and N_i the number of records in the heap prior to the ith operation (assuming $P \geqslant 1$ and $S > M \geqslant c \cdot P$, where c is a small positive constant). The number of comparisons required when handling the sequence is $O(\sum_{i=1}^{S}\log_2 N_i)$. Using the suggested data structure we obtain an optimal external heapsort that performs $O((N/P)\log_{(M/P)}(N/P))$ page transfers and $O(N\log_2 N)$ comparisons in the worst case when sorting N records. © 1999 Elsevier Science B.V. All rights reserved.

Keywords: Secondary storage; Priority queues; Heaps; Sorting; Heapsort

1. Introduction

The traditional data structure for implementing a *priority queue* is the *heap* (see, e.g., [8]). 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

☆ A preliminary version of this work appeared as "External heaps combined with effective buffering" in: *Proceedings of the Computing: The Australasian Theory Symposium, Australian Computer Science Communications* **19**, 2 (1997), 72–78.

[1] Partially supported by the Danish Natural Science Research Council under contracts 9400952 (Project "Computational Algorithmics") and 9701414 (Project "Experimental Algorithmics").

* Corresponding author. Fax: +358-2-333 8600.

E-mail addresses: teuhola@cs.utu.fi (J. Teuhola), jyrki@diku.dk (J. Katajainen) 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 a 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* [10, 20] which is one of the few *in-place* sorting methods guaranteeing an $O(N \log_2 N)$ worst-case complexity[2] when sorting N records in the primary memory of a computer.

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 primary memory. In a two-level memory model, the typical measure of complexity is the number of *pages transferred* between fast primary memory and slow secondary storage. 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* [19]. Their heap had the same structure as the internal heap, namely a complete 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 [15], based on quicksort.

The external heapsort in [19] cannot be improved if we assume that the buffer space in primary memory is of 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 (in records) fixed? We could keep the top part of the heap always in primary memory, resulting in $O((N/P)\log_2(N/M))$ page transfers. This is, however, asymptotically worse than the best possible bound $\Theta((N/P)\log_{(M/P)}(N/P))$, obtained by external $O(M/P)$-way mergesort [1].

Our intention is to create an external heap organization that tries to make the best use of the available primary memory. Especially, we try to achieve the same complexity for external heapsort as for multi-way 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 external 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. This is confirmed by the experiments in Section 6.

[2] In this article, we use $\log_a x$ as a shorthand notation for $\max(1, \ln x / \ln a)$.

In a virtual-memory environment, where the user has no control over the page-replacement policy, the best utilization of the physical resources is not possible. For instance, Alanko et al. [4] noticed that (internal) heapsort sorts N records with $\mathrm{O}(N \log_2(N/P))$ page transfers in such an environment. The behaviour of several priority-queue structures in virtual memory was studied by Naor et al. [14]. In their experiments a P-way heap was superior to the B-tree [5] and to the splay tree [16]. They also observed that a P-way heap ($P \geqslant 2$) supports both the insert and delete-max operations with $\mathrm{O}(\log_P N)$ page transfers (even though the delete-max operation has the internal cost of $\mathrm{O}(P \log_P N)$). The key observation in the present paper is that an even more efficient heap structure is obtained by letting the fanout be $\mathrm{O}(M/P)$ and storing $\mathrm{O}(M/P)$ pages in every node. Now, however, we must ourselves control the movement of pages to and from secondary storage. Some operating systems actually provide this facility for the users (see, e.g., [11–13, 21]).

The performance of our heap structure is as follows. When handling an intermixed sequence of insert and delete-max operations, starting from an empty heap, the number of page transfers is $\mathrm{O}(\sum_{i=1}^{S}((1/P)\log_{(M/P)}(N_i/P)))$ and the number of comparisons $\mathrm{O}(\sum_{i=1}^{S}\log_2 N_i)$. Here P denotes the number of records fitting into a page, M the capacity of the buffer space in records, N_i the number of records in the heap prior to the ith operation, and S is the number of operations ($P \geqslant 1, S > M \geqslant c \cdot P, c \approx 2$). This results in external heapsort that performs $\mathrm{O}((N/P)\log_{(M/P)}(N/P))$ page transfers and $\mathrm{O}(N \log_2 N)$ comparisons in the worst case when sorting N records.

A data structure, called *buffer tree*, with a similar performance as ours has been developed by Arge [2, 3]. His structure is an (a, b)-tree which also supports off-line search and delete operations. In measuring performance, the basic difference from our approach is that he expresses the (amortized) complexity of the operations as a function of P, M, and S, but not N_i. In sorting this difference is not essential, since the total number of operations and the maximum size of the structure are about the same. The buffer tree is quite complicated whereas the heap structure explored in this paper is conceptually simple and practical, as confirmed by the experiments.

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 the results of the simulation experiments are reported. Finally, in Section 7 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

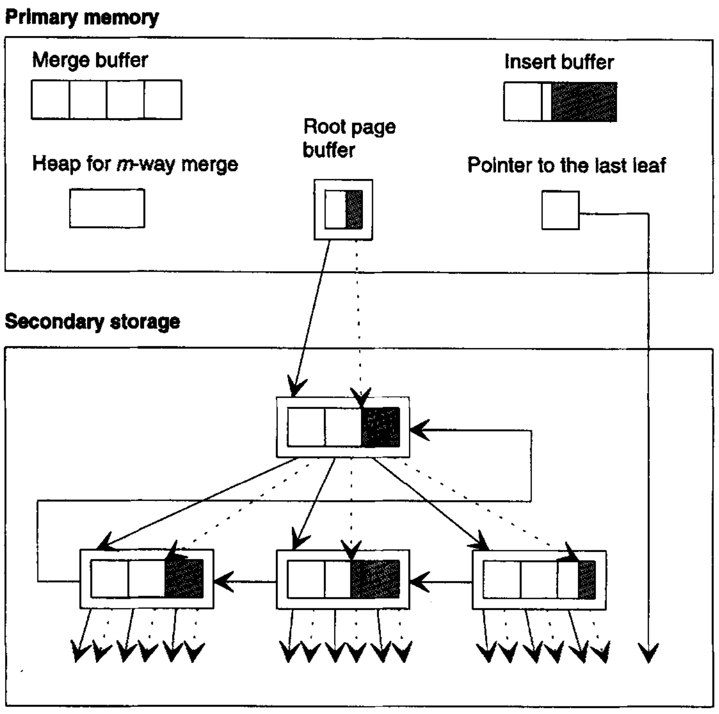

Fig. 1. The internal and external data structures for $m = 3$.

variable-size records would complicate the presentation. In the formal description of the heap data structure we use two parameters:

- P is the number of records fitting into a page. We assume that $P \geqslant 1$, i.e., a page should be at least as large as a record. Every page might contain some header information, O(1) pointers, but P is measured in records, not including the space used by these pointers.
- m denotes the (maximum) fanout of the heap nodes and also the number of pages storing records in a node. We assume that $m \geqslant 2$.

The value of m, which should be as large as possible, is determined by the amount of space available in primary memory. Due to efficiency reasons, primary memory must accommodate $m + O(1)$ pointers and $2m + 2$ pages storing records. Hence, the value of m depends on the application in question and the environment where the application is run. In general, m is $\Theta(M/P)$ in which M denotes the amount of primary memory available, measured in records.

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

- Each node is composed 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 are 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) the order number of the node among its siblings (needed in delete-max).

- 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 higher than or equal to the priority of y.
- The heap is otherwise complete, except that the lowest level can be incomplete: its nodes are arranged to the left, as in a normal binary heap. Therefore, the position of the *last* node of the heap is uniquely defined, and we can maintain a pointer to it. We keep this pointer in primary memory. The parent of the last node is the only internal node whose degree may be between 1 and m, all other internal nodes have m children.
- Each node, except the last leaf, is at least half full, i.e., contains 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 last page of the root, containing the highest-priority records, is always kept buffered in primary memory.
- 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 last root page, primary memory contains two other 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 added to the heap as a batch. The records in the insert buffer are organized as a normal (binary) heap because we have to look for a record with the highest priority. As in [19], moving records up or down requires *merging* of blocks. Here we need an auxiliary *merge buffer* of $m+1$ pages. Furthermore, a priority queue (heap) of m pointers is kept in primary memory, to support m-way merging effectively.

3. Priority-queue operations

In this section we describe how the heap data structure supports the operations insert and delete-max. The operation find-max, which inspects (but does not remove) a record with the highest priority, is often included in the repertoire of priority-queue operations but, since it does not involve any page transfers, it is uninteresting for us.

3.1. 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" [8]), records are *sifted up* as follows. We merge the block of the last leaf with that of its parent (using the merge area in primary memory). Assume that the merged sequence has r records and that h of these have priority higher than or equal to the minimum priority in the parent before the merge. Let $k = \max(r - h, \lceil Pm/2 \rceil)$. Allocate $r - k$ highest-priority records to the parent, and the rest k to the child. It can be easily verified that this choice maintains the load condition and restores the heap condition between the parent and all its children. However, the sift-up must be repeated for the parent and its grandparent, etc., up to the root, or until the heap condition is found to hold.

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 changed in the swap.

3.2. Delete-max

Due to the heap property, a record with the highest priority is either in the root or in the insert buffer. Since the root is ordered, its last page buffered, and the insert buffer is an internal heap, this record is easily found and extracted. If delete-max makes the buffer page empty, another is read in from the root, namely the page that logically precedes the one that became empty. If the root becomes imperfect, i.e. its load drops below $\lceil Pm/2 \rceil$, we have to *refill* it after delete-max. If the children contain at least $\lceil Pm/2 \rceil$ records, we move precisely $\lceil Pm/2 \rceil$ of them with the highest priorities to the root. Note that no grandchildren need be touched because all nodes (except the last leaf) must contain this amount of records. If there is only one child and it contains less than $\lceil Pm/2 \rceil$ records, we move all of them to the root, which now becomes the only node of the heap.

After refilling the root, it may happen that one or more of its children have become imperfect and must be refilled, in turn. For internal nodes, refilling is done exactly as for the root. As a result of the refill, the internal node may remain internal or it may become a leaf. In the latter case, it may still remain imperfect. Of course, any previous leaf can also become imperfect, after giving part of its records to the parent. An imperfect leaf, say X, is refilled as follows. If X is the last leaf, then we do not have to do anything; this is the exception to the load condition. Otherwise, we have to "steal" records from the last leaf, denoted L. Let $|X|$ denote the number of records in leaf 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 $Pm - |X|$ highest-priority records from L to X, and sift-up X.
(2) If $\lceil Pm/2 \rceil \leqslant s \leqslant Pm$, 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, obtained by following the related pointer) and

repeat the process for X and L'. This is guaranteed to succeed because either $X = L'$ or $|L'| \geqslant \lceil Pm/2 \rceil$. After filling X, it can be sifted up.

From the above discussion it is obvious that we can steal from a certain last leaf only twice, whereafter it becomes empty and ceases to exist.

Let us now study the refill procedure of a single node. How do we find the $\lceil Pm/2 \rceil$ records with the highest priorities? The records in blocks are arranged in ascending order. Moreover, we maintain pointers from the parent to the last record of each of its children. Therefore, we can merge the blocks of the m children *from back to front*, until the required amount of records is obtained. We call this a *partial* merge. To make the m-way merge internally efficient, we use a priority-queue structure in primary memory (see Fig. 1), so that the next record with the highest priority is always obtained with $O(\log_2 m)$ comparisons, instead of m.

Notice that, in merging, most of the front pages in the children's blocks need not be touched at all; this is important in respect of the complexity. On the other hand, the lifted records are put to the *front* of the parent's block, so all parent pages have to be touched, to make room for the new ones.

The refilling process is *recursive*; we can proceed, e.g., in depth-first order. However, we want to avoid the recursion stack because its size depends on the height of the heap and, hence, on the number of records in it. An iterative traversal of the heap in depth-first order is enabled by parent–child and child–parent pointers. After backtracking from a child, the pointer to the next child is obtained from the parent immediately because the order number among siblings is stored in each child.

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 their description is omitted.

4. External complexity

The external costs are measured in terms of page transfers (reads and writes). The complexity will be determined only for an intermixed sequence of insert and delete-max operations – 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, in external heapsort, only the overall cost counts.

As for pointers, we make a very pessimistic assumption that a pointer access costs as much as a page access. This could be improved, but it would complicate the proofs considerably. Also, the data structures and algorithms should have been described in greater detail. Our cost estimate means that the resulting transfer count will be about twice as high as necessary, since each pointer access is normally followed by a page/block access.

4.1. Heaps

As usual, we define the *depth* of the root of the heap to be zero and that of any other node one plus the depth of its parent. Moreover, we say that a node is on *level* i if its depth is i. The *height* of the tree is the largest depth of any node.

Lemma 1. *The height of an m-way heap storing N records, such that each node (except possibly the last leaf) stores at least $\lceil Pm/2 \rceil$ records, is bounded by $\log_m(N/P) + O(1)$.*

Proof. Since the number of nodes is bounded by $\left\lceil \frac{N}{\lceil Pm/2 \rceil} \right\rceil$ and $m \geqslant 2$, it is obvious that the height of the heap is at most $\log_m(N/P) + O(1)$. □

Let us next analyse the cost of basic subroutines in the insert and delete-max operations.

Lemma 2. *The merging of two sorted blocks occupying p and q pages, respectively, costs $2(p+q)$ page transfers. If the p pages reside in primary memory and p result pages can also stay in primary memory, we need $2q$ transfers.*

Proof. The results are obvious because the merging is done by a single scan over the blocks. □

Lemma 3. *Assume that an imperfect parent of (at most) m children is given, and each of the children's sorted blocks consists of at most m pages. A partial m-way merge of these blocks, gathering the $\lceil Pm/2 \rceil$ highest-priority records and merging them to their parent, requires at most $7m$ page transfers.*

Proof. It is again clear that the sorted blocks are scanned sequentially (now from back). Let us first think about the page reads. We clearly have to touch at least $\lceil m/2 \rceil$ pages. However, for each block, the first and last of the touched pages may contribute very little to the result (one and zero records may be lifted from them, in the worst case). Therefore, we get an upper bound $2m + \lceil m/2 \rceil$ for the number of page reads. The blocks are read page-wise into the merge buffer and the result is written to the front of the parent. Thus, we have to read also the pages in the parent and move the records forward (no gaps allowed). This is another $\lceil m/2 \rceil$ page reads. All pages in the parent may have to be rewritten, causing m page writes. Thus, the total number of page transfers amounts to at most $4m + 2 \leqslant 5m$. When we add the reads and writes of m pointers, we get the claimed result. □

Now we are ready to analyse the cost of a sequence of insert and delete-max operations, starting from an empty heap.

Theorem 4. *An intermixed sequence of S insert and delete-max operations requires at most* $26\sum_{j=1}^{S_I}((1/P)\log_m(N_{i_j}/P)) + O(S/P)$ *page transfers in total, where* $i_1, i_2, \ldots, i_{S_I}$ *are the indices of insert operations and* N_{i_j} *denotes the number of records stored in the heap prior to the execution of the* i_j*th insert operation. Especially,* $i_1 = 1$ *and* $N_1 = 0$.

Proof. To prove the result, we shall apply the standard bank-account paradigm (for example, see [18]). We assume that each page transfer costs one *euro*. To perform all the operations in the sequence, a certain amount of money, namely $26h/P + O(1/P)$ euros, are allocated to each record, where h is the height of the tree at insert time, i.e., $h \leqslant \log_m(N/P) + O(1)$. Here N, in turn, denotes the number of records in the structure before the insert. Now it is our intention to show that the allocated money is sufficient to pay all the page transfers required in the whole sequence of operations (both inserts and deletes). This will then directly give the claimed result.

We continue the metaphor by saying that the money is deposited to imaginary *accounts*, associated with various parts of the data structure. The insert buffer has an *insert account*, from which money is withdrawn to pay for the sift-ups of inserted blocks. Each record has a *delete account*, containing money to pay for the refills. In addition, each node has a *merge account*, which is needed only when the node becomes the last leaf and the money there is used for paying the merge of the last leaf with another node, plus the related sift-up. At record insert, the following amounts are deposited into the individual accounts:

(1) $6h/P + O(1/P)$ euros to the insert account,

(2) $14h/P + O(1/P)$ euros to the delete account of the record,

(3) $6h/P + O(1/P)$ euros to the merge account of the (not yet stored) node to be created next (if ever).

Let us now analyse the individual operations and steps.

(A) Insert

In most cases a record is inserted in the insert buffer, causing no page transfers. However, the associated money is deposited to the related accounts, as described above. When the insert buffer gets full (Pm new records), a new last leaf is created, resulting in one or two sift-up chains (see the algorithm). Each chain consists of parent-child merges, where the other partner can always be kept in primary memory (see Lemma 2). The accumulated amount of money in the insert account is $6hm + O(m)$ euros because the new height of the tree is at most one larger than the heights before *any* of the Pm previous inserts. Thus, there is enough money to pay for sift-ups ($4hm + O(m)$). The remaining $2hm + O(m)$ euros are used for reading $2h + O(1)$ pointers.

(B) Delete-max

Each record has, as explained, a delete account opened at insert time. Now we should show that it contains a sufficient amount of money for the record to be lifted up to the root. In fact, we can prove the following *invariant*:

Each record on level i has $14i/P + O(1/P)$ *euros in its delete account.*

First, it should be noticed that inserts do not invalidate the invariant because we can assume that, when some records are swapped between a parent and its child, also the money in their accounts is swapped! It is quite normal in the accounting method to move money around, where appropriate.

In most cases a record with the highest priority is deleted from the buffered root page (or, in special cases, from the insert buffer). When the buffer page gets empty, another is read in, namely the one preceding the earlier buffer page. This costs one access, which is $1/P$ per record, so that this cost can be included in the $O(1/P)$ term of the complexity.

When the root gets imperfect, it is refilled by (at most) $\lceil Pm/2 \rceil$ highest-priority records of its children. Refilling may then propagate in the heap arbitrarily wide. In a successful refill, it is sufficient to check that the invariant holds after the refill. A successful refill moves $\lceil Pm/2 \rceil$ records up. According to Lemma 3, a refill, together with all pointer manipulation, costs at most $7m$ transfers. Each of the lifted $\lceil Pm/2 \rceil$ records pays $14/P$ euros (withdrawn from its delete account), which together sum up to the required amount. The claimed invariant is easily seen to hold, and each record has enough money to travel all the way to the root.

If refilling does not succeed, the node either is or has become a leaf (after making its only child, i.e. the last leaf, empty). This case is handled by merging the imperfect leaf with the last leaf, resulting in a sift-up. The money that each node has in its merge account ($6hm + O(m)$ euros) is used now. As explained in the algorithm, a leaf merge can happen only twice for a certain last leaf. The cost of these two (binary) block merges is at most $5m + O(1)$, because the other partner (last leaf) contributes at most m pages to the merges altogether and each merge results in at most m pages. The cost of two sift-ups is at most $6hm + O(m)$ (see discussion on insert cost). The sift-ups thus dominate, and the money in the merge account suffices to defray the cost of the task. After two merges, the last leaf ceases to exist and its money has been spent.

In maintaining the merge accounts, we still have to consider the situation where the current last leaf is swapped (if imperfect) with the inserted new last leaf. Actually, we prove the following invariant:

Each node on level i has $6im + O(m)$ euros in its merge account, except the last leaf, which may have only $3im + O(m)$ euros if it is imperfect.

Immediately after node insert (before the swap), the invariant holds, based on the initial amount of money given to it. Assume that the heights of the heap before and after the node insert are h and h'. Obviously, either $h' = h$ or $h' = h + 1$. In case of swapping with the previous last leaf, the new (full) node exchanges half of the money in its merge account, namely $3h'm + O(m)$ euros, with the whole contents of the merge account ($3hm + O(m)$) of the imperfect last leaf. After the swap the new node (on level h) has $3hm + 3h'm + O(m) \geq 6hm + O(m)$ euros in its merge account, and the last leaf (on level h') has $3h'm + O(m)$ euros. Both quota are sufficient and the invariant holds. □

It must be emphasized that the complexity result has a tremendous slack in it. Accessing pointers counts for about half the amount. Therefore, the real constant factor would be around 13 and even that is pessimistic, i.e., computed assuming always the worst cases. Moreover, we used the insert operations to cover all the costs. If there are equally many deletes, i.e. each record is deleted sooner or later, the complexity per operation is still halved. From Theorem 4 we easily get the following:

Corollary 5. *An intermixed sequence of S insert and delete-max operations requires at most* $26\sum_{i=1}^{S}((1/P)\log_m(N_i/P))+\mathrm{O}(S/P)$ *page transfers in total, where N_i denotes the number of records stored in the heap prior to the execution of the ith operation. Especially,* $N_1=0$.

Let N denote the maximum number of records ever stored in the heap. Since $N_i \leqslant N$ for all i and $N \leqslant S$, we have two weaker results:

Corollary 6. *An intermixed sequence of S insert and delete-max operations requires at most* $26(S/P)\log_m(N/P)+\mathrm{O}(S/P)$ *page transfers in total.*

Corollary 7. *An intermixed sequence of S insert and delete-max operations requires at most* $26(S/P)\log_m(S/P)+\mathrm{O}(S/P)$ *page transfers in total.*

4.2. Heapsort

Our starting point was the external heapsort by Wegner and Teuhola [19], which sorts N records with $\mathrm{O}((N/P)\log_2(N/P))$ page transfers. Now we improve on this. In principle, we could first build the heap by repeating the insert operation for each record to be sorted and then extract the records in sorted order by repeating the delete-max operation. However, we obtain a better constant factor to the complexity by using a faster heap-building procedure.

Theorem 8. *Given N records stored compactly on $\lceil N/P \rceil$ pages, an external heap can be built with* $\mathrm{O}(N/P)$ *page transfers.*

Proof. The claimed complexity is obtained, e.g., by the following algorithm. First, compute the number N_ℓ of records to be assigned on level ℓ in a complete external heap will full nodes:

$$N_\ell = \begin{cases} Pm \cdot m^\ell & \text{for } \ell = 0,\dots,\ell_{max}-1, \\ N - \sum_{\ell=0}^{\ell_{max}-1} N_\ell & \text{for } \ell = \ell_{max}, \end{cases}$$

where $\ell_{max} = \lceil \log_m(N/P) \rceil - 1$. Second, partition the set of all records into subsets $R_0,\dots,R_{\ell_{max}}$ such that, for $\ell = 0,\dots,\ell_{max}$, $|R_\ell| = N_\ell$ and, for any record x in R_i, any record y in R_j, and $i<j$, the priority of x is larger than or equal to the priority of y. Third, assign these subsets to the heap nodes on their respective levels, after internal sort. Clearly, this will produce a legitimate heap.

Partitionings are performed bottom-up, so that we first determine the leaf level, then the next higher level, and so on. In order to extract the records of level ℓ, we have to solve a *selection* problem, where we determine the highest-priority record belonging to level ℓ. In this selection, the records on levels $\ell+1, \ell+2, \ldots$ are already excluded. Selection of the highest-priority record on level ℓ can be done with $O(\sum_{i=0}^{\ell}(N_i/P))$ page transfers by adapting the linear selection algorithm developed for primary memory [6]. This is straightforward because the essential parts of the algorithm are linear scans, otherwise the processing can be done in primary memory. Partitioning makes also (trivially) $O(\sum_{i=0}^{\ell}(N_i/P))$ page transfers. By summing up and assuming that $m \geqslant 2$, we obtain that the total number of page transfers performed is bounded by

$$\begin{aligned}
O\left(\sum_{\ell=0}^{\ell_{max}-1}\sum_{i=0}^{\ell}(N_i/P)+N_{\ell_{max}}/P\right) &= O\left(\sum_{\ell=0}^{\ell_{max}-1}\sum_{i=0}^{\ell}(Pm\cdot m^i/P)+N_{\ell_{max}}/P\right)\\
&\leqslant O\left(\sum_{\ell=0}^{\ell_{max}-1}(2\cdot Pm\cdot m^{\ell}/P)+N_{\ell_{max}}/P\right)\\
&\leqslant O(4\cdot Pm\cdot m^{\ell_{max}-1}/P+N_{\ell_{max}}/P)\\
&= O(N/P+N_{\ell_{max}}/P)\\
&= O(N/P). \quad \square
\end{aligned}$$

Theorem 9. *Given N records stored compactly on $\lceil N/P\rceil$ pages, external heapsort can sort these with at most $14(N/P)\log_m(N/P)+O(N/P)$ page transfers.*

Proof. Referring to the proof of Theorem 4, we note that the insert account is not needed now because the heap is built off-line with $O(N/P)$ page transfers as described in Theorem 8. Also, the merge account can be avoided; we can let any leaf be imperfect, not just the last one, because the height of the heap does not grow after building it. We only need the delete account of records, which got an initial deposit of $14h/P+O(1/P)$ euros each. Altogether we make at most

$$\begin{aligned}
O(N/P)+\sum_{i=1}^{N}(14h_i/P+O(1/P)) &\leqslant O(N/P)+\sum_{i=1}^{N}(14\log_m(N/P)/P+O(1/P))\\
&= 14(N/P)\log_m(N/P)+O(N/P)
\end{aligned}$$

page transfers. $\square$

Compared to external n-way mergesort, the complexity of which is known to be only $2(N/P)\log_n(N/P)+O(N/P)$, our algorithm seems clearly inferior. Observe that here n can be larger than m since mergesort uses less internal space than heapsort. It is, however, obvious that our constant factor is highly exaggerated. Pointer manipulation costs are overestimated and many approximations were overly pessimistic. Experimental comparison between the two sorting methods is reported in Section 6.

To compare the methods with respect to their space usage, we first have to fix the assumed implementation of mergesort. A pointer-free implementation would require

O(N/P) extra pages to keep the intermediate results. It is thus more economic to apply a pointer-based solution, where a page slot can be reused as soon as its contents have been read to the internal buffer. This version is comparable to our external heapsort; both require a constant number of pointers per page. The external heap has the additional cost, due to fragmentation, that every node can have an almost-empty last page, resulting in O(N/Pm) extra pages.

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 either (1) the decision about record movement is done only after its priority has been compared with some other, or (2) a block move is accompanied by a corresponding number of comparisons. Also the number of pointer manipulations is at most of the same order as the number of priority comparisons. Altogether, the total number of all internal operations performed is proportional to that of comparisons. However, here we refrain from giving upper bounds to constant factors.

5.1. Heaps

Again we compute the complexity for a sequence of insert and delete-max operations, starting from an empty heap. The following theorem gives an asymptotic complexity which was proved the best possible in [17].

Theorem 10. *An intermixed sequence of S insert and delete-max operations requires* $\mathrm{O}(\sum_{i=1}^{S}\log_2 N_i)$ *priority comparisons in total, where N_i denotes the number of records stored in the heap prior to the execution of the ith operation. Especially,* $N_1 = 0$.

Proof. Assume that the indices of insert operations are $i_1, i_2, \ldots, i_{S_I}$ and the indices of delete-max operations $j_1, j_2, \ldots, j_{S_D}$.

Let us first analyse the costs of inserts. The i_kth insert in a non-full insert buffer costs $\mathrm{O}(\log_2 B_{i_k})$ comparisons, where B_{i_k} is the number of records currently in the buffer. Since $B_{i_k} \leqslant N_{i_k}$, the cost is $\mathrm{O}(\log_2 N_{i_k})$ comparisons. When the insert buffer becomes full, it is sorted, using $\mathrm{O}(Pm\log_2(Pm))$ comparisons. Let $b_{\lfloor Pm/2\rfloor+1}, \ldots, b_{Pm}$ denote the indices of insert operations for the last $\lceil Pm/2\rceil$ records in the buffer. At the inserts of these records, the total number of records in the heap has been at least $\lfloor Pm/2\rfloor$. Therefore, the sorting cost can be estimated by

$$\mathrm{O}(Pm\log_2(Pm)) \leqslant \mathrm{O}(Pm\log_2(2N_{b_k})) \text{ for } k = b_{\lfloor Pm/2\rfloor+1}, \ldots, b_{Pm}.$$

The cost per insert b_k is $\mathrm{O}(2\log_2 2N_{b_k})$ or $\mathrm{O}(\log_2 N_{b_k})$. This means that half of the records in the insert buffer "pay" the sorting, in the amortized sense. In a sift-up, binary parent-child merges are performed, using the merge buffer. The cost *per page* is O(P)

comparisons. Now we utilize Theorem 4, from which we get an upper bound for the number of pages handled. By multiplying this by the number of comparisons per page we get $O(P \cdot \sum_{k=1}^{S_I}((1/P)\log_m(N_{i_k}/P)))$ comparisons, which is at most $O(\sum_{k=1}^{S_I}\log_2 N_{i_k})$, as required.

Let us now consider delete-max. A record with the highest priority is found either from the root or from the insert buffer with one comparison, but keeping the latter in shape costs $O(\log_2 B_{j_k})$ comparisons (for B_{j_k} records in the buffer), which is $O(\log_2 N_{j_k})$. In case of an imperfect root, one or more refills are required. In a refill, an m-way merge is performed, the complexity of which is not any more linear, because an internal priority queue must be maintained. The cost per page is $O(P\log_2 m)$ comparisons. Merging leaves and the following sift-up cost $O(P)$ comparisons per page, as in insert. Again, by Theorem 4, we get $O((P+P\log_2 m)\cdot\sum_{k=1}^{S_I}((1/P)\log_m(N_{i_k}/P)))$ comparisons, which is at most $O(\sum_{k=1}^{S_I}\log_2 N_{i_k})$. In other words, again the inserts cover both insert and delete costs. This completes the proof of the theorem. □

5.2. Heapsort

Using Theorem 10, it is trivial to derive the internal complexity of external heapsort.

Theorem 11. *External heapsort sorts N records with $O(N\log_2 N)$ priority comparisons.*

Proof. If $N \leqslant Pm$, we can do the whole job in the insert buffer, i.e., we use internal heapsort which requires $O(N\log_2 N)$ comparisons. Now, assume that $N > Pm$ and think first of a simplified version of external heapsort, implemented as N inserts followed by N delete-max operations. We can apply Theorem 10 directly: we replace N_i by the upper bound N and S by $2N$, and obtain $O(\sum_{i=1}^{2N}\log_2 N) = O(N\log_2 N)$ comparisons. The more advanced heap-building algorithm of Theorem 8 makes only $O(N\log_2 M)$ comparisons, including the internal sort of nodes. This, added to the last N terms in the above sum (corresponding to delete-max operations), improves the constant factor in $O(N\log_2 N)$. □

As for the internal complexity, external heapsort is satisfactory since its performance is asymptotically the same as that of internal heapsort.

6. Experimental results

A number of test runs were performed with the suggested external heap structure, in order to investigate its usefulness in practice. Here we report only a few results and restrict ourselves to recording the external behaviour. A more detailed experimental study can be found in [9].

Actually, the tests were only simulations of the real operations. The operating system was not trusted; we wanted to have full control of all page transfers. Two simulators were implemented. The first simulates external memory usage, offering operations PageRead and PageWrite for the use of the programmer. The number of these operations is counted. The second simulates virtual memory and applies the LRU (Least Recently Used) page-replacement algorithm. In this environment the programmer can use the memory as if it were an internal array. The simulator keeps track of the pages in primary memory and counts the number of page transfers. The simulator is simplified so that a replaced page is rewritten to secondary storage also in a case where the page was not changed.

In the experiments only page transfers of interest were measured, that is, the transfers of the pages containing records only. The page transfers performed when accessing pointers, program segments, temporary variables, or other such data structures are not included in the counts.

Our first experiment compared the performance of three priority-queue structures:

(1) An m-way heap as described in the preceding sections. A technical difference from the theoretical description was that the whole root was buffered, for simplicity.
(2) A P-way (internal) heap implemented using the virtual-memory simulator, and
(3) a P-way B-tree used as a priority queue, with highest-priority records kept in a buffer of M/P pages, and the page transfers controlled by explicit PageRead and PageWrite commands.

Notice that both insert and delete-max have logarithmic external complexity also for the B-tree, but the base of logarithm is P. Moreover, the amortized complexity of delete-max has a factor $1/P$ since the rightmost leaf is buffered.

The test setting was such that we first inserted a certain number N of records into the structure, and then started to execute insert and delete-max operations randomly, both having 50% probability. The measurements were taken during this latter period. The total number of page transfers for subsequent N operations is shown in Fig. 2, for page size $P = 50$ and primary memory size $M = 50P$. As expected, the external heap is by far the best of the three data structures tested. The observation that the B-tree is slightly better than the P-way heap in virtual memory is not quite in agreement with the results obtained in [14]. Apparently, the difference results from the fact that we did not try to take advantage of page alignments in the virtual-memory simulator. This increases the number of page transfers to about the double, compared to the optimal alignment of sibling sets in the P-way heap.

The second experiment concerned sorting. Our external heapsort was, naturally, compared with external mergesort, which is the de facto standard in practice. Moreover, the theoretical complexities of the two are asymptotically the same, as well as the buffer sizes (up to a constant factor). The outdegree m of the external heap and the order n of merging were somewhat smaller than M/P, due to the auxiliary structures in the primary memory. Comparison with hillsort presented in [19] would have been unfair because it uses only a constant number of buffer pages.

In the external mergesort, the sorting was carried out bottom-up, without recursion. The initial sorted lists of size $\frac{2}{3}m$ pages were created by internal mergesort. The algorithm had also an extra workspace, equal to the input size, in secondary storage.

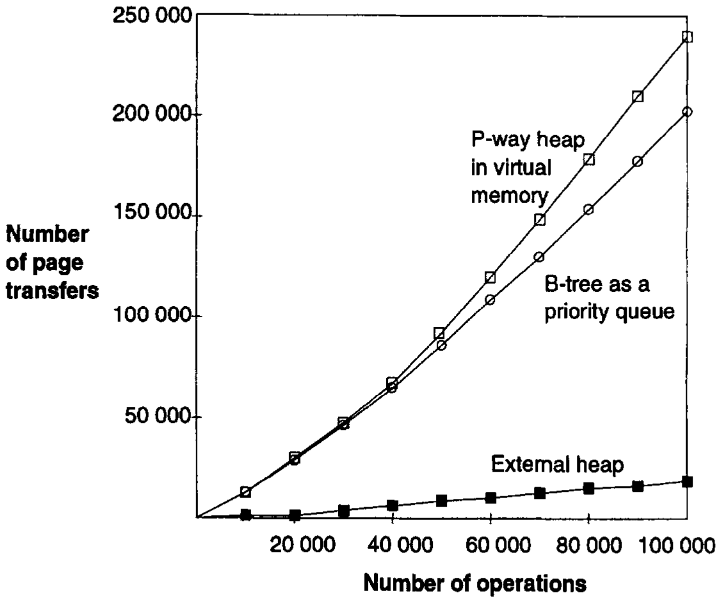

Fig. 2. The average number of page transfers per insert/delete-max operation when $P=50$ and $M/P=50$.

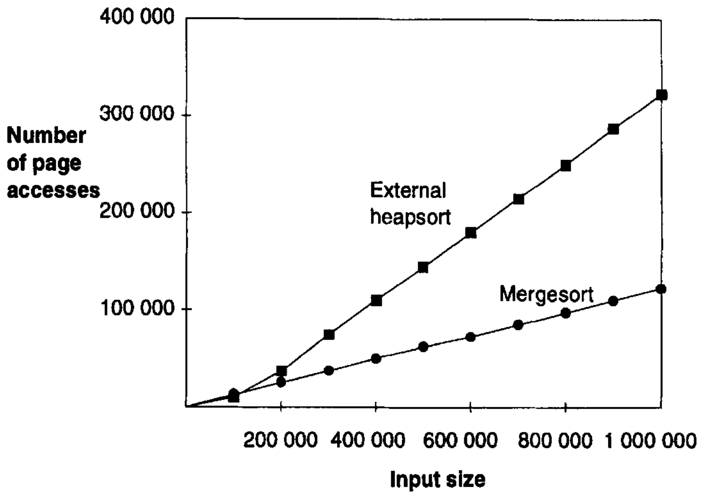

Fig. 3. The number of page transfers for the two external sorting programs when $P=50$ and $M/P=50$.

The merging passes were done from the initial area to the working space and back in alternating order.

The number of page transfers performed by the two sorting methods is depicted in Fig. 3 for $P=50$ and $M/P=50$. It seems that at least our current (non-optimized) version of external heapsort does not quite reach the efficiency of mergesort. The heap-building procedure was implemented using normal insert operations. The faster off-line procedure could probably improve the results. Anyway, for practical purposes, we suggest the m-way external heap to be used mainly as a priority queue, not for external sorting. The latter application is at least theoretically interesting, due to its optimality, up to constant factors.

7. Conclusion and further work

We have described an external priority-queue organization, which is a natural generalization of the traditional heap organization in primary memory. Multi-page nodes with a large fanout imply a very small height for the heap, which keeps the number of page transfers low. The key point is an effective utilization of the primary memory. The obtained complexity for priority-queue operations can be considered satisfactory because it guarantees asymptotically optimal performance for external heapsort, in respect of both external and internal complexity. Our frame of reference includes only comparison-based techniques. For special distributions or restricted domains of priorities, 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 to these problems (for example, see [8]) 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. [7].

Acknowledgements

We would like to thank Jesper Bojesen who detected a severe error in an earlier version of the sift-up algorithm.

References

[1] A. Aggarwal, J.S. Vitter, The input/output complexity of sorting and related problems, Comm. ACM 31 (1988) 1116–1127.
[2] L. Arge, The buffer tree: a new technique for optimal I/O-algorithms, Proc. 4th Workshop on Algorithms and Data Structures, Lecture Notes in Computer Science, vol. 955, Springer, Berlin, 1995, pp. 334–345.
[3] L. Arge, Efficient external-memory data structures and applications, BRICS Dissertation DS-96-3, Department of Computer Science, University of Aarhus, Århus, 1996.
[4] T.O. Alanko, H.H.A. Erkiö, I.J. Haikala, Virtual memory behavior of some sorting algorithms, IEEE Trans. Software Eng. SE-10 (1984) 422–431.

[5] R. Bayer, E.M. McCreight, Organization and maintenance of large ordered indexes, Acta Inform. 1 (1972) 173–189.
[6] M. Blum, R.W. Floyd, V. Pratt, R.L. Rivest, R.E. Tarjan, Time bounds for selection, J. Comput. System Sci. 7 (1973) 448–461.
[7] Y.-J. Chiang, M.T. Goodrich, E.F. Grove, R. Tamassia, D.E. Vengroff, J.S. Vitter, External-memory graph algorithms, Proc. 6th Annual ACM–SIAM Symp. on Discrete Algorithms, ACM, New York and SIAM, Philadelphia, 1995, pp. 139–149.
[8] T.H. Cormen, C.E. Leiserson, R.L. Rivest, Introduction to Algorithms, The MIT Press, Cambridge, 1990.
[9] R. Fadel, K.V. Jakobsen, Data structures and algorithms in a two-level memory, M.Sc. Thesis, Department of Computing, University of Copenhagen, Copenhagen, 1996.
[10] R.W. Floyd, Algorithm 245, Treesort 3, Comm. ACM 7 (1964) 701.
[11] K. Harty, D.R. Cheriton, Application-controlled physical memory using external page-cache management, Proc. 5th Internat. Conf. on Architectural Support for Programming Languages and Operating Systems, ACM SIGPLAN Notices 27 (1992) 187–197.
[12] K. Krueger, D. Loftesness, A. Vahdat, T. Anderson, Tools for the development of application-specific virtual memory management, Proc. 8th Annual Conf. on Object-Oriented Programming Systems, Languages, and Applications, ACM SIGPLAN Notices 28 (1993) 48–64.
[13] D. McNamee, K. Amstrong, Extending the Mach external pager interface to accommodate user-level page replacement policies, Technical Report 90-09-05, Department of Computer Science and Engineering, University of Washington, Seattle, 1990.
[14] D. Naor, C.U. Martel, N.S. Matloff, Performance of priority queue structures in a virtual memory environment, Comput. J. 34 (1991) 428–437.
[15] H.W. Six, L. Wegner, Sorting a random access file in situ, Comput. J. 27 (1984) 270–275.
[16] D.D. Sleator, R.E. Tarjan, Self-adjusting binary search trees, J. ACM 32 (1985) 652–686.
[17] D.D. Sleator, R.E. Tarjan, Self-adjusting heaps, SIAM J. Comput. 15 (1986) 52–69.
[18] R.E. Tarjan, Amortized computational complexity, SIAM J. Algebraic Discrete Meth. 6 (1985) 306–318.
[19] L.M. Wegner, J.I. Teuhola, The external heapsort, IEEE Trans. Software Engineering 15 (1989) 917–925.
[20] J.W.J. Williams, Algorithm 232, Heapsort, Comm. ACM 7 (1964) 347–348.
[21] Y. Yokote, The Apertos reflective operating system: the concept and its implementation, Proc. 7th Annual Conf. on Object-Oriented Programming Systems, Languages, and Applications, ACM SIGPLAN Notices 27 (1992) 414–434.