Worst-Case Efficient External-Memory Priority Queues*

Gerth Stølting Brodal[1,**] and Jyrki Katajainen[2,***]

[1] Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany
[2] Datalogisk Institut, Københavns Universitet, Universitetsparken 1, DK-2100 København Ø, Denmark

Abstract. A priority queue Q is a data structure that maintains a collection of elements, each element having an associated priority drawn from a totally ordered universe, under the operations INSERT, which inserts an element into Q, and DELETEMIN, which deletes an element with the minimum priority from Q. In this paper a priority-queue implementation is given which is efficient with respect to the number of block transfers or I/Os performed between the internal and external memories of a computer. Let B and M denote the respective capacity of a block and the internal memory measured in elements. The developed data structure handles any intermixed sequence of INSERT and DELETEMIN operations such that in every disjoint interval of B consecutive priority-queue operations at most $c \log_{M/B} \frac{N}{M}$ I/Os are performed, for some positive constant c. These I/Os are divided evenly among the operations: if $B \geq c \log_{M/B} \frac{N}{M}$, one I/O is necessary for every $B/(c \log_{M/B} \frac{N}{M})$th operation and if $B < c \log_{M/B} \frac{N}{M}$, $\frac{c}{B} \log_{M/B} \frac{N}{M}$ I/Os are performed per every operation. Moreover, every operation requires $O(\log_2 N)$ comparisons in the worst case. The best earlier solutions can only handle a sequence of S operations with $O(\sum_{i=1}^{S} \frac{1}{B} \log_{M/B} \frac{N_i}{M})$ I/Os, where N_i denotes the number of elements stored in the data structure prior to the ith operation, without giving any guarantee for the performance of the individual operations.

1 Introduction

A *priority queue* is a data structure that stores a set of elements, each element consisting of some *information* and a *priority* drawn from some totally ordered universe. A priority queue supports the operations:

* The full version has appeared as Technical Report 97/25, Department of Computer Science, University of Copenhagen, Copenhagen, 1997.

** Supported by the Carlsberg foundation under grant No. 96-0302/20. Partially supported by the ESPRIT Long Term Research Program of the EU under contract No. 20244 (project ALCOM-IT). Email: brodal@mpi-sb.mpg.de.

*** Supported partially by the Danish Natural Science Research Council under contract No. 9400952 (project Computational Algorithmics). Email: jyrki@diku.dk.

INSERT(x): Insert a new element x with an arbitrary priority into the data structure.

DELETEMIN(): Delete and return an element with the minimum priority from the data structure. In the case of ties, these are broken arbitrarily. The precondition is that the priority queue is not empty.

Priority queues have numerous applications, a few listed by Sedgewick [28] are: sorting algorithms, network optimization algorithms, discrete event simulations and job scheduling in computer systems. For the sake of simplicity, we will not hereafter make any distinction between the elements and their priority.

In this paper we study the problem of maintaining a priority queue on a computer with a two-level memory: a fast *internal memory* and a slow *external memory*. We assume that the computer has a processing unit, the *processor*, and a collection of hardware, the *I/O subsystem*, which is responsible for transferring data between internal and external memory. The processor together with the internal memory can be seen as a traditional random access machine (RAM) (see, e.g., [3]). In particular, note that the processor can only access data stored in internal memory. The capacity of the internal memory is assumed to be bounded so it might be necessary to store part of the data in external memory. The I/O subsystem transfers the data between the two memory levels in blocks of a fixed size.

The behavior of algorithms on such a computer system can be characterized by two quantities: *processor performance* and *I/O performance*. By the processor performance we mean the number of primitive operations performed by the processor. Our measure of processor performance is the number of element comparisons carried out. It is straightforward to verify that the total number of other (logical, arithmetical, etc.) operations required by our algorithms is proportional to that of comparisons. Assuming that the elements occupy only a constant number of computer words, the total number of primitive operations is asymptotically the same as that of comparisons. Our measure of I/O performance is the number of block transfers or *I/Os* performed, i.e., the number of blocks read from the external memory plus the number of blocks written to the external memory. Our main goal is to analyze the total work carried out by the processor and the I/O subsystem during the execution of the algorithms.

The *system performance*, i.e., the total elapsed execution time when the algorithms are run on a real computer, depends heavily on the realization of the computer. A real computer may have multiple processors (see, e.g., [18]) and/or the I/O subsystem can transfer data between several disks at the same time (cf. [2, 25, 30]), the processor operations (see, e.g., [27]) and/or the I/Os (cf. [19]) might be pipelined, but the effect of these factors is not considered here. It has been observed that in many large-scale computations the increasing bottleneck of the computation is the performance of the I/O subsystem (see, e.g., [15, 26]), increasing the importance of I/O efficient algorithms.

When expressing the performance of the priority-queue operations, we use the following parameters:

B: the number of elements per block,

M: the number of elements fitting in internal memory, and
N: the number of elements currently stored in the priority queue; more specifically, the number of elements stored in the structure just prior to the execution of INSERT or DELETEMIN.

We assume that each block and the internal memory also fit some pointers in addition to the elements, and $B \geq 1$ and $M \geq 23B$. Furthermore, we use $\log_a n$ as a shorthand notation for $\max(1, \ln n / \ln a)$, where ln denotes the natural logarithm.

Several priority-queue schemes, such as implicit heaps [33], leftist heaps [12, 20], and binomial queues [9, 31] have been shown to permit both INSERT and DELETEMIN with worst-case $O(\log_2 N)$ comparisons. Some schemes, such as implicit binomial queues [10] guarantee worst-case $O(1)$ comparisons for INSERT and $O(\log_2 N)$ comparisons for DELETEMIN. Also any kind of balanced search trees, such as AVL trees [1] or red-black trees [16] could be used as priority queues. However, due to the usage of explicit or implicit pointers the performance of these structures deteriorates on a two-level memory system. It has been observed by several researchers that a d-ary heap performs better than the normal binary heap on multi-level memory systems (see, e.g., [22, 24]). For instance, a B-ary heap reduces the number of I/Os from $O(\log_2 \frac{N}{B})$ (cf. [4]) to $O(\log_B \frac{N}{B})$ per operation [24]. Of course, a B-tree [8, 11] could also be used as a priority queue, with which a similar I/O performance is achieved. However, in a virtual-memory environment a B-ary heap seems to be better than a B-tree [24].

When a priority queue is maintained in a two-level memory, it is advantageous to keep the small elements in internal memory and the large elements in external memory. Hence, due to insertions large elements are to be moved from internal memory to external memory and due to deletions small elements are to be moved from external memory to internal memory. Assuming that we maintain two buffers of B elements in internal memory, one for INSERTs and one for DELETEMINs, at most every Bth INSERT and DELETEMIN will cause a buffer overflow or underflow. Several data structures take advantage of this kind of buffering. Fishspear, developed by Fischer and Paterson [14], can be implemented by a constant number of push-down stacks, implying that any sequence of S INSERT and DELETEMIN operations requires at most $O(\sum_{i=1}^{S} \frac{1}{B} \log_2 N_i)$ I/Os, where N_i denotes the size of the data structure prior to the ith operation. Wegner and Teuhola [32] realized that a binary heap, in which each node stores B elements, guarantees $O(\log_2 \frac{N}{B})$ I/Os for every Bth INSERT and every Bth DELETEMIN operation in the worst case.

The above structures assume that the internal memory can only fit $O(B)$ elements, i.e., a constant number of blocks. Even faster solutions are possible if the whole capacity of the internal memory is utilized. Arge [5, 6] introduced an (a,b)-tree structure that can be used to carry out any sequence of S INSERT and DELETEMIN operations with $O(\frac{S}{B} \log_{M/B} \frac{S}{M})$ I/Os. Fadel et al. [13] gave a heap structure with a similar I/O performance but their bound depends on the size profile, not on S. Their heap structure can handle any sequence of S operations with $O(\sum_{i=1}^{S} \frac{1}{B} \log_{M/B} \frac{N_i}{M})$ I/Os, where N_i denotes the size of the data structure

prior to the ith operation. The number of comparisons required when handling the sequence is $O(\sum_{i=1}^{S} \log_2 N_i)$. When this data structure is used for sorting N elements, both the processor and I/O performance match the well-known lower bounds $\Omega(\frac{N}{B} \log_{M/B} \frac{N}{M})$ I/Os [2] and $\Omega(N \log_2 N)$ comparisons (see, e.g., [20]), which are valid for all comparison-based algorithms.

To achieve the above bounds—as well as our bounds—the following facilities must be provided:

1. we should know the capacity of a block and the internal memory beforehand,
2. we must be able to align elements into blocks, and
3. we must have a full control over the replacement of the blocks in internal memory.

There are operating systems that provide support for these facilities (see, e.g., [17, 21, 23]).

The tree structure of Arge and the heap structure of Fadel et al. do not give any guarantees for the performance of individual operations. In fact, one INSERT or DELETEMIN can be extremely expensive, the cost of handling the whole sequence being an upper bound. Therefore, it is risky to use these structures in on-line applications. For large-scale real-time discrete event simulations and job scheduling in computer systems it is often important to have a guaranteed worst-case performance.

We describe a new data structure that gives worst-case guarantees for the cost of individual operations. Basically, our data structure is a collection of sorted lists that are incrementally merged. This idea is borrowed from a RAM priority-queue structure of Thorup [29]. Thorup used two-way merging in his internal data structure but we use multi-way merging since it behaves better in an external-memory environment. As to the processor and I/O performance, our data structure handles any intermixed sequence of operations as efficiently as the heap structure by Fadel et al. [13]. In every disjoint interval of B consecutive priority-queue operations our data structure requires at most $c \log_{M/B} \frac{N}{M}$ I/Os, for some positive constant c. These I/Os are divided evenly among the operations. If $B \geq c \log_{M/B} \frac{N}{M}$, one I/O is necessary for every $B/(c \log_{M/B} \frac{N}{M})$th priority-queue operation, and if $B < c \log_{M/B} \frac{N}{M}$, $\frac{c}{B} \log_{M/B} \frac{N}{M}$ I/Os are performed per every priority-queue operation. Moreover, every operation requires $O(\log_2 N)$ comparisons in the worst case.

2 Basic Data Structure

The basic components of our priority-queue data structure are sorted lists. When new elements arrive, these are added to a list which is kept in internal memory and sorted incrementally. If the capacity of internal memory is exceeded due to insertions, a fraction of the list containing the recently inserted elements is transferred to external memory. To bound the number of lists in external memory we merge the existing lists. This merging is related to the merging done by the external mergesort algorithm [2]. One particular list that is kept in

internal memory contains the smallest elements. If this list is exhausted due to deletions, new smallest elements are extracted from the lists in external memory. Because we are interested in worst-case bounds the merging is accomplished incrementally. A similar idea has been applied by Thorup [29] to construct RAM priority queues but instead of two-way merging we rely on multi-way merging.

Before giving the details of the data structure, let us recall the basic idea of external mergesort which sorts N elements with $O(\frac{N}{B}\log_{M/B}\frac{N}{M})$ I/Os [2]. First, the given N elements are partitioned into $\Theta(N/M)$ lists each of length $\Theta(M)$. Second, each of the lists are read into internal memory and sorted, requiring $O(N/B)$ I/Os in total. Third, $\Theta(M/B)$ sorted lists of shortest length are repeatedly merged until only one sorted list remains containing all the elements. Since each element takes part in $O(\log_{M/B}\frac{N}{M})$ merges, the total number of I/Os is $O(\frac{N}{B}\log_{M/B}\frac{N}{M})$.

Our data structure consists of two parts: an *internal part* and an *external part*. The data structure takes two parameters K and m, where K is a multiple of B, $9K + 5B \leq M$, and $m = K/B$. The internal part of the data structure stores $O(K)$ elements and is kept all the time in internal memory. The external part is a priority queue which permits the operations:

BATCHINSERT$_K(X)$: Insert a sorted list X of K elements into the external-memory data structure.

BATCHDELETEMIN$_K()$: Delete the K smallest elements from the data structure in external-memory.

Both of these operations require at most $O(\frac{K}{B}\log_m\frac{N}{K})$ I/Os and $O(K\log_2\frac{N}{K})$ comparisons in the worst case. For every Kth operation on the internal part we do at most one batch operation involving K elements on the external part of the data structure.

The internal part of the data structure consists of two sorted lists *MIN* and *NEW* of length at most $3K$ and $2K$, respectively. We represent both *MIN* and *NEW* as a balanced search tree that permits insertions and deletions of elements with $O(\log_2 K)$ comparisons. The rôle of *MIN* is to store the current at most $3K$ smallest elements in the priority queue whereas the intuitive rôle of *NEW* is to store the at most $2K$ recently inserted elements. All elements in *MIN* are smaller than the elements in *NEW* and the elements in the external part of the data structure, i.e., the overall minimum element is the minimum of *MIN*.

The external part of the data structure consists of sorted lists of elements. Each of these lists has a *rank*, which is a positive integer, and we let R denote the *maximum rank*. In Sect. 3.4 we show how to guarantee that $R \leq \log_m\frac{N}{K}+2$. The lists with rank i, $i \in \{1,\ldots,R\}$, are $L_i^1, L_i^2, \ldots, L_i^{n_i}$, $\overline{L}_i^1, \overline{L}_i^2, \ldots, \overline{L}_i^{\overline{n}_i}$, and $\overline{L}_i$.

For each rank i, we will incrementally merge the lists $\overline{L}_i^1, \ldots, \overline{L}_i^{\overline{n}_i}$ and append the result of the merging to the list $\overline{L}_i$. The list $\overline{L}_i$ contains the already merged part of $\overline{L}_i^1, \ldots, \overline{L}_i^{\overline{n}_i}$, and all elements in $\overline{L}_i$ are therefore smaller than those in $\overline{L}_i^1, \ldots, \overline{L}_i^{\overline{n}_i}$. When the incremental merge of the lists $\overline{L}_i^j$ finishes, the list $\overline{L}_i$ will be promoted to a list with rank $i+1$, provided that $\overline{L}_i$ is sufficiently long, and

a new incremental merge of lists with rank i is initiated by making $L_i^1, \ldots, L_i^{n_i}$ the new $\overline{L}_i^j$ lists. The details of the incremental merge are given in Sect. 3.2.

We guarantee that the length of each of the external lists is a multiple of B. An external list L containing $|L|$ elements is represented by a single linked list of $|L|/B$ blocks, each block storing B elements plus a pointer to the next block, except for the last block which stores a null pointer. There is one exception to this representation. The last block of $\overline{L}_i$ does not store a null pointer, but a pointer to the first block of $\overline{L}_i^1$ (if $\overline{n}_i = 0$, the last block of $\overline{L}_i$ stores a null pointer). This allows us to avoid updating the last block of $\overline{L}_i$ when merging the lists $\overline{L}_i^1, \ldots, \overline{L}_i^{\overline{n}_i}$.

In the following, we assume that pointers to all the external lists are kept in internal memory together with their sizes and ranks. If this is not possible, it is sufficient to store this information in a single linked list in external memory. This increases the number of I/Os required by our algorithms only by a small constant.

In Sect. 3 we describe how BATCHINSERT$_K$ and BATCHDELETEMIN$_K$ operations are accomplished on the external part of the data structure, and in Sect. 4 we describe how the external part can be combined with the internal part of the data structure to achieve a worst-case efficient implementation of INSERT and DELETEMIN operations.

3 Maintenance of the External Part

3.1 The MERGE$_K$ Procedure

The heart of our construction is the procedure MERGE$_K(X_1, X_2, \ldots, X_\ell)$, which incrementally merges and removes the K smallest elements from the sorted lists $X_1, \ldots, X_\ell$. All list lengths are assumed to be multiples of B. After the merging of the K smallest elements we rearrange the remaining elements in the X_i lists such that the lists still have lengths which are multiples of B. We allow MERGE$_K$ to make the X_i lists shorter or longer. We just require that the resulting X_i lists remain sorted. For the time being, we assume that the result of MERGE$_K$ is stored in internal memory.

The procedure MERGE$_K$ is implemented as follows. For each list X_i we keep the block containing the current minimum of X_i in internal memory. In internal memory we maintain a heap [33] over the current minima of all the lists. We use the heap to find the next element to be output in the merging process. Whenever an element is output, it is the current minimum of some list X_i. We remove the element from the heap and the list X_i, and insert the new minimum of X_i into the heap, provided that X_i has not become empty. If necessary, we read the next block of X_i into internal memory.

After the merging phase, we have from each list X_i a partially filled block B_i in internal memory. Let $|B_i|$ denote the number of elements left in block B_i. Because we have merged K elements from the blocks read and K is a multiple of B, $\sum_{i=1}^{\ell} |B_i|$ is also a multiple of B. Now we merge the remaining elements in the

B_i blocks in internal memory. This merge requires $O(\ell B \log_2 \ell)$ comparisons. Let $\hat{X}$ denote the resulting list and let B_j be the block that contained the maximum element of $\hat{X}$. Finally, we write $\hat{X}$ to external memory such that X_j becomes the list consisting of $\hat{X}$ concatenated with the part of X_j that already was stored in external memory. Note that X_j remains sorted.

In total, $\textsc{Merge}_K$ performs at most $K/B + \ell$ I/Os for reading the prefixes of $X_1, \ldots, X_\ell$ (for each list X_i, we read at most one block of elements that do not take part in the merging) and at most ℓ I/Os for writing $\hat{X}$ to external memory. The number of comparisons required for $\textsc{Merge}_K$ for each of the $K+\ell B$ elements read into internal memory is $O(\log_2 \ell)$. Hence, we have proved

Lemma 1. *$\textsc{Merge}_K(X_1, \ldots, X_\ell)$ performs at most $2\ell + K/B$ I/Os and $O((K + \ell B) \log_2 \ell)$ comparisons. The number of elements to be kept in internal memory by $\textsc{Merge}_K$ is at most $K + \ell B$. If the resulting list is written to external memory incrementally, only $(\ell + 1)B$ elements have to be kept in internal memory simultaneously.*

3.2 Batch Insertions

To insert a sorted list of K elements into the external part of the data structure we increment n_1 by one and let $L_1^{n_1}$ contain the K new elements, and apply the procedure $\textsc{MergeStep}(i)$, for each $i \in \{1, \ldots, R\}$.

The procedure $\textsc{MergeStep}(i)$ does the following. If $\overline{n}_i = 0$, the incremental merge of lists with rank i is finished, and we make $\overline{L}_i$ the list $L_{i+1}^{n_{i+1}+1}$, provided that $|\overline{L}_i| \geq Km^i$. Otherwise, we let $\overline{L}_i$ be the list $L_i^{n_i+1}$ because the list is too short to be promoted. Finally, we initiate a new incremental merge by making the lists $L_i^1, \ldots, L_i^{n_i}$ the new $\overline{L}_i^j$ lists. If $\overline{n}_i > 0$, we concatenate $\overline{L}_i$ with the result of $\textsc{Merge}_K(\overline{L}_i^1, \ldots, \overline{L}_i^{\overline{n}_i})$, i.e., we perform K steps of the incremental merge of $\overline{L}_i^1, \ldots, \overline{L}_i^{\overline{n}_i}$. Note that, by writing the first block of the merged list on the place occcupied earlier by the first block of $\overline{L}_i^1$, we do not need to update the pointer in the previous last block of $\overline{L}_i$.

The total number of I/Os performed in a batched insertion of K elements is K/B for writing the K new elements to external memory and by Lemma 1 at most $2(\overline{n}_i + K/B)$ for incrementally merging the lists with rank i. The number of comparisons for rank i is $O((\overline{n}_i B + K) \log_2 \overline{n}_i)$. The maximum number of elements to be stored in internal memory for batched insertions is $(\overline{n}_{\max} + 1)B$, where $\overline{n}_{\max} = \max\{\overline{n}_1, \ldots, \overline{n}_R\}$. To summarize, we have

Lemma 2. *A sorted list of K elements can be inserted into the external part of the data structure by performing $(1 + 2R)K/B + 2\sum_{i=1}^{R} \overline{n}_i$ I/Os and performing $O(\sum_{i=1}^{R} (\overline{n}_i B + K) \log_2 \overline{n}_i)$ comparisons. At most $(\overline{n}_{\max} + 1)B$ elements need to be stored in internal memory.*

3.3 Batch Deletions

The removal of the K smallest elements from the external part of the data structure is carried out in two steps. In the first step the K smallest elements are located. In the second step the actual deletion is accomplished.

Let $\mathcal{L}$ be one of the lists L_i^1 or $\overline{L}_i$, for some i, or an empty list. We will guarantee that the list $\mathcal{L}$ contains the K smallest elements of the lists considered so far. Initially $\mathcal{L}$ is empty. By performing $L_i^1 \leftarrow \textsc{Merge}_K(L_i^1, \ldots, L_i^{n_i}) \cdot L_i^1$, L_i^1 now contains the K smallest elements of $L_i^1, \ldots, L_i^{n_i}$. The procedure $\textsc{SplitMerge}_K$ takes two sorted lists as its arguments and returns (the name of) one of the lists. If the first argument is an empty list, then the second list is returned. Otherwise, we require that the length of both lists to be at least K and we rearrange the K smallest elements of both lists as follows. The two prefixes of length K are merged and split among the two lists such that the lists remain sorted and the length of the lists remain unchanged. One of the lists will now have a prefix containing K elements which are smaller than all the elements in the other list. The list with this prefix is returned. For each rank $i \in \{1, \ldots R\}$, we now carry out the assignments $L_i^1 \leftarrow \textsc{Merge}_K(L_i^1, \ldots, L_i^{n_i}) \cdot L_i^1$, $\mathcal{L} \leftarrow \textsc{SplitMerge}_K(\mathcal{L}, L_i^1)$, and $\mathcal{L} \leftarrow \textsc{SplitMerge}_K(\mathcal{L}, \overline{L}_i)$.

It is straightforward to verify that after performing the above, the prefix of the list $\mathcal{L}$ contains the K smallest elements in the external part of the data structure. We now delete the K smallest elements from list $\mathcal{L}$, and if $\mathcal{L}$ is $\overline{L}_i$ we perform $\textsc{MergeStep}(i)$ once.

By always keeping the prefix of $\mathcal{L}$ in internal memory the total number of I/Os for the deletion of the K smallest elements (without the call to $\textsc{MergeStep}$) is $(4R-1)(K/B) + 2\sum_{i=1}^{R} n_i$ because, for each rank i, $n_i + 2(K/B)$ blocks are to be read into internal memory and all blocks except the K/B blocks holding the smallest elements should be written back to external memory. The number of comparisons for rank i is $O((K + Bn_i)\log_2 n_i)$. The additional call to $\textsc{MergeStep}$ requires at most $K/B + \overline{n}_i$ additional block reads and block writes, and $O((K + B\overline{n}_i)\log_2 \overline{n}_i)$ comparisons. Let $n_{\max} = \max\{n_1, \ldots, n_R\}$ and $\overline{n}_{\max} = \max\{\overline{n}_1, \ldots, \overline{n}_R\}$. The maximum number of elements to be stored in internal memory for the batched minimum deletions is $2K + B\max\{n_{\max}, \overline{n}_{\max}\}$.

Lemma 3. *The K smallest elements can be deleted from the external part of the data structure by performing at most $4R(K/B) + 2\sum_{i=1}^{R} n_i + \overline{n}_{\max}$ I/Os and $O(\sum_{i=1}^{R}(K + n_i B)\log_2 n_i + (K + B\overline{n}_{\max})\log_2 \overline{n}_{\max})$ comparisons. At most $2K + B\max\{n_{\max}, \overline{n}_{\max}\}$ elements need to be stored in internal memory.*

3.4 Bounding the Maximum Rank

We now describe a simple approach to guarantee that the maximum rank R of the external data structure is bounded by $\log_m N/K + 2$. Whenever insertions cause the maximum rank to increase, this is because of $\textsc{MergeStep}(R-1)$ has finished an incremental merge resulting a list of length Km^{R-1}, which implies

that $R \leq \log_m \frac{N}{K} + 1$. The problem we have to consider is how to decrement R when deletions are performed.

Our solution is the following. Whenever MERGESTEP(R) finishes the incremental merge of lists with rank R, we check if the resulting list $\overline{L}_R$ is very small. If $\overline{L}_R$ is very small, i.e., $|\overline{L}_R| < Km^{R-1}$, and there are no other list of rank R, we make $\overline{L}_R$ a list with rank $R-1$ and decrease R.

To guarantee that the same is done also in the connection with batched minimum deletions, we always call after each BATCHDELETEMIN$_K$ operation, described in Sect. 3.3, MERGESTEP(R) k times (for $m \geq 4$ it turns out that $k = 1$ is sufficient, and for $m = 3$ or $m = 2$ it is sufficient to let $k = 2$ or $k = 3$). It can be proved that this guarantees $R \leq \log_m \frac{N}{K} + 2$.

3.5 Resource Bounds for the External Part

In the previous discussion we assumed that n_i and $\overline{n}_i$ where sufficiently small, such that we could apply MERGESTEP to the L_i^j and $\overline{L}_i^j$ lists. Let m' denote a maximum bound on the merging degree. It can be proved that $m' \leq 5 + 2m$, for $m \geq 2$. Because $m = K/B$ it follows that the maximum rank is at most $\log_{K/B} \frac{N}{K} + 2$ and that the maximum merging degree is $5 + 2K/B$. From Lemmas 2 and 3 it follows that the number of I/Os required for inserting K elements or deleting the K smallest elements is at most $O(\frac{K}{B} \log_{K/B} \frac{N}{K})$ and the number of comparisons required is $O(K \log_2 \frac{N}{K})$. The maximal number of elements to be stored in internal memory is $4K + 5B$.

4 Internal Buffers and Incremental Batch Operations

We now describe how to combine the buffers *NEW* and *MIN* represented by binary search trees with the external part of the priority-queue data structure. We maintain the invariant that $|MIN| \geq 1$, provided that the priority queue is nonempty. Recall that we also required that $|MIN| \leq 3K$ and $|NEW| \leq 2K$.

We first consider INSERT(x). If x is less than or equal to the maximum of *MIN* or all elements of the priority queue are stored in *MIN*, we insert x into *MIN* with $O(\log_2 K)$ comparisons. If *MIN* exceeds its maximum allowed size, $|MIN| = 3K + 1$, we move the maximum of *MIN* to *NEW*. Otherwise, x is larger than the maximum of *MIN* and we insert x into *NEW* with $O(\log_2 K)$ comparisons. The implementation of DELETEMIN deletes and returns the minimum of *MIN*. Both operations require at most $O(\log_2 K)$ comparisons.

There are two problems with the implementation of INSERT and DELETEMIN. Insertions can cause *NEW* to become too big and deletions can make *MIN* empty. Therefore, for every Kth priority-queue operation we perform one batch insertion or deletion. If $|NEW| \geq K$, we remove K elements from *NEW* one by one and perform BATCHINSERT$_K$ on the removed elements. If $|NEW| < K$ and $|MIN| \leq 2K$, we instead increase the size of *MIN* by moving K small elements to *MIN* as follows. First, we perform a BATCHDELETEMIN$_K$ operation to extract the K least elements from the external part of the data structure. The

K extracted elements are inserted into NEW one by one, using $O(K \log_2 K)$ comparisons. Second, we move the K smallest elements of NEW to MIN one by one. If $|NEW| < K$ and $|MIN| > 2K$, we do nothing but delay the batch operation until $|MIN| = 2K$ or $|NEW| = K$. Each batch operation requires at most $O(\frac{K}{B} \log_{K/B} \frac{N}{K})$ I/Os and at most $O(K(\log_2 \frac{N}{K} + \log_2 K)) = O(K \log_2 N)$ comparisons.

By doing one of the above described batch operations for every Kth priority-queue operation it is straightforward to verify that $|NEW|+(3K-|MIN|) \leq 2K$, provided that the priority queue contains at least K elements, implying $|NEW| \leq 2K$ and $|MIN| \geq K$, because each batch operation decreases the left-hand side of the equation by K.

The idea is now to perform a batch operation incrementally over the next K priority-queue operations. Let N denote the number of elements in the priority queue, when the corresponding batch operation is initiated. Notice that N can at most be halved while performing a batch operation, because $N \geq 2K$ prior to the batch operation. Because $|MIN| \geq K$ when a batch operation is initiated, it is guaranteed that MIN is nonempty while incrementally performing the batch operation over the next K priority-queue operations.

Because a batch operation requires at most $O(\frac{K}{B} \log_{K/B} \frac{N}{K})$ I/Os and at most $O(K \log_2 N)$ comparisons, it is sufficient to perform at most $O(\log_2 N)$ comparisons of the incremental batch operation per priority-queue operation and operations if $B \geq c \log_{M/B} \frac{N}{M}$, one I/O for every $B/(c \log_{M/B} \frac{N}{M})$th priority-queue operation and if $B < c \log_{M/B} \frac{N}{M}$, $\frac{c}{B} \log_{M/B} \frac{N}{M}$ I/Os for every priority-queue operation, for some positive constant c, to guarantee that the incremental batch operation is finished after K priority-queue operations.

Because $|MIN| \leq 3K$, $|NEW| \leq 2K$, and a batched operation at most requires $4K + 5B$ elements to be stored in internal memory, we have the constraint that $9K + 5B \leq M$. Let now $K = \lfloor (M - 5B)/9 \rfloor$. Recall that we assumed that $M \geq 23B$, and therefore, $K \geq 2B$. Since $M > K$, $O(\log_{M/B} \frac{N}{M}) = O(\log_{K/B} \frac{N}{M})$. Hence, we have proved the main result of this paper.

Main theorem *There exists an external-memory priority-queue implementation that supports* Insert *and* DeleteMin *operations with worst-case* $O(\log_2 N)$ *comparisons per operation. If* $B \geq c \log_{M/B} \frac{N}{M}$, *one I/O is necessary for every* $B/(c \log_{M/B} \frac{N}{M})$*th operation and if* $B < c \log_{M/B} \frac{N}{M}$, $\frac{c}{B} \log_{M/B} \frac{N}{M}$ *I/Os are performed per every operation, for some positive constant* c.

5 Concluding Remarks

We have presented an efficient priority-queue implementation which guarantees a worst-case bound on the number of comparisons and I/Os required for the individual priority-queue operations. Our bounds are comparison based.

If the performance bounds are allowed to be amortized, the data structure can be simplified considerably, because no list merging and batch operation is required to be incrementally performed. Then no $\overline{L}_i$ and $\overline{L}_i^j$ lists are required, and

we can satisfy $1 \leq |MIN| \leq K$, $|NEW| \leq K$, and $n_i < m$ by always (completely) merging exactly m lists of equal rank, the rank of a list L being $\lfloor \log_m \frac{|L|}{K} \rfloor$.

What if the size of the elements or priorities is not assumed to be constant? That is, express the bounds as a function of N and the length of the priorities. How about the priorities having variable lengths? Initial research in this direction has been carried out by Arge et al. [7], who consider the problem of sorting strings in external memory.

References

1. G. M. Adel'son-Vel'skiĭ and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics*, volume 3, pages 1259–1263, 1962.
2. 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.
3. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company, Reading, 1974.
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. 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, Springer, Berlin/Heidelberg, pages 334–345, 1995.
6. L. Arge. Efficient external-memory data structures and applications. BRICS Dissertation DS-96-3, Department of Computer Science, University of Aarhus, Århus, 1996.
7. L. Arge, P. Ferragina, R. Grossi, and J. S. Vitter. On sorting strings in external memory. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, ACM Press, New York, pages 540–548, 1997.
8. R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, volume 1, pages 173–189, 1972.
9. M. R. Brown. Implementation and analysis of binomial queue algorithms. *SIAM Journal on Computing*, volume 7, pages 298–319, 1978.
10. S. Carlsson, J. I. Munro, and P. V. Poblete. An implicit binomial queue with constant insertion time. In *Proceedings of the 1st Scandinavian Workshop on Algorithm Theory*, Lecture Notes in Computer Science 318, Springer-Verlag, Berlin/Heidelberg, pages 1–13, 1988.
11. D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, volume 11, pages 121–137, 1979.
12. C. A. Crane. Linear lists and priority queues as balanced trees. Technical Report STAN-CS-72-259, Computer Science Department, Stanford University, Stanford, 1972.
13. R. Fadel, K. V. Jakobsen, J. Katajainen, and J. Teuhola. Heaps and heapsort on secondary storage. To appear in *Theoretical Computer Science*.
14. M. J. Fischer and M. S. Paterson. Fishspear: A priority queue algorithm. *Journal of the ACM*, volume 41, pages 3–30, 1994.
15. G. A. Gibson, J. S. Vitter, J. Wilkes *et al.* Strategic directions in storage I/O issues in large-scale computing. *ACM Computing Surveys*, volume 28, pages 779–793, 1996.

16. L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, IEEE, New York, pages 8–21, 1978.
17. K. Harty and D. R. Cheriton. Application-controlled physical memory using external page-cache management. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems, ACM SIGPLAN Notices*, volume 27, number 9, pages 187–197, 1992.
18. J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley Publishing Company, Reading, 1992.
19. B. H. H. Juurlink and H. A. G. Wijshoff. The parallel hierarchical memory model. In *Proceedings of the 4th Scandinavian Workshop on Algorithm Theory*, Lecture Notes in Computer Science 824, Springer-Verlag, Berlin/Heidelberg, pages 240–251, 1994.
20. D. E. Knuth. *The Art of Computer Programming*, volume 3/ *Sorting and Searching*. Addison-Wesley Publishing Company, Reading, 1973.
21. K. Krueger, D. Loftesness, A. Vahdat, and T. Anderson. Tools for the development of application-specific virtual memory management. In *Proceedings of the 8th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, ACM SIGPLAN Notices*, volume 28, number 10, pages 48–64, 1993.
22. A. LaMarca and R. E. Ladner. The influence of caches on the performance of heaps. *The ACM Journal of Experimental Algorithmics*, volume 1, article 4, 1996.
23. D. McNamee and K. Amstrong. Extending the Mach external pager interface to accommodate user-level block replacement policies. Technical Report 90-09-05, Department of Computer Science and Engineering, University of Washington, Seattle, 1990.
24. D. Naor, C. U. Martel, and N. S. Matloff. Performance of priority queue structures in a virtual memory environment. *The Computer Journal*, volume 34, pages 428–437, 1991.
25. M. H. Nodine and J. S. Vitter. Large-scale sorting in parallel memories. In *Proceedings of the 3rd ACM Symposium on Parallel Algorithms and Architectures*, ACM Press, New York, pages 29–39, 1991.
26. Y. N. Patt. Guest editor's introduction: The I/O subsystem — A candidate for improvement. *IEEE Computer*, volume 27, number 3, pages 15–16, 1994.
27. D. A. Patterson and J. L. Hennessy. *Computer Organization & Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers, San Francisco, 1994.
28. R. Sedgewick. *Algorithms*. Addison-Wesley Publishing Company, Reading, 1983.
29. M. Thorup. On RAM priority queues. In *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, ACM, New York and SIAM, Philadelphia, pages 59–67, 1996.
30. J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, volume 12, pages 110–147, 1994.
31. J. Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, volume 21, pages 309–315, 1978.
32. L. M. Wegner and J. I. Teuhola. The external heapsort. *IEEE Transactions on Software Engineering*, volume 15, pages 917–925, 1989.
33. J. W. J. Williams. Algorithm 232, Heapsort. *Communications of the ACM*, volume 7, pages 347–348, 1964.