# An Experimental Study of Priority Queues in External Memory[1]

Klaus Brengel
and
Andreas Crauser
and
Paolo Ferragina[2]
and
Ulrich Meyer
Max-Planck-Institut für Informatik, 66123 Saarbrücken, Germany

---

In this paper we compare the performance of eight different priority queue implementations: four of them are explicitly designed to work in an external-memory setting, the others are standard internal-memory queues available in the LEDA library [Mehlhorn and Näher 1999]. Two of the external-memory priority queues are obtained by engineering known internal-memory priority queues with the aim of achieving effective performance on external storage devices (i.e., Radix heaps [Ahuja et al. 1990] and array heaps [Thorup 1996]). Our experimental framework includes some simple tests, like random sequences of insert or delete-minimum operations, as well as more advanced tests consisting of intermixed sequences of update operations and "application driven" update sequences originated by simulations of Dijkstra's algorithm on large graph instances. Our variegate spectrum of experimental results gives a good picture of the features of these priority queues, thus being helpful to anyone interested in the use of such data structures on very large data sets.

---

## 1. INTRODUCTION

A *priority queue* is a data structure that stores a set of items, each one consisting of a tuple which contains some *satellite information* plus a *priority* value (also called *key*) drawn from a totally ordered universe. A priority queue supports the following operations on the processed set: `access_minimum` (returns the item in the

---

[1]Research partially supported by EU ESPRIT LTR Project Nr. 20244 (ALCOM-IT), and by Italian MURST project "Algorithms for Large Data Sets: Science and Engineering", 1999. The software used to produce our experimental results is available at http://www.mpi-sb.mpg.de/~crauser/leda-sm.html

[2]Paolo Ferragina's present address: Dipartimento di Informatica, Università di Pisa, Italy.

---

set having the minimum key), `delete_min` (returns and deletes the item in the set having the minimum key) and `insert` (inserts a new item into the set). Priority queues (hereafter PQs) have numerous important applications: combinatorial optimization (e.g., Dijkstra's shortest path algorithm [Dijkstra 1959]), time forward processing [Chiang et al. 1995], job scheduling, event simulation, online sorting, just to name a few. Many PQ implementations currently exist for small data sets fitting into the *internal memory* of the computer, e.g., $k$–ary heaps [William 1964], Fibonacci heaps [Fredman and Tarjan 1987], radix heaps [Ahuja et al. 1990], and some of them are also publicly available to programmers (see, e.g., the LEDA library [Mehlhorn and Näher 1999]). However, in large-scale event simulations or on instances of very large graph problems (as they have recently occured, e.g., in geographical information systems), the performance of these *internal-memory* PQs may significantly deteriorate due to paging in virtual memory, thus being a bottleneck for the overall underlying application. In fact, as soon as parts of the PQ do not fit entirely into the internal memory of the computer, but reside in its *external memory* (e.g., on the hard disk), we may observe a heavy paging activity of the external storage devices because the pattern of memory accesses is not tuned to exhibit any locality of reference. Due to the technological features of current disk systems [Ruemmler and Wilkes 1994], this situation may determine a slowdown of a factor from $10^5$ to $10^6$ in the final performance of each PQ-operation. Consequently, the design of PQs must take explicitly into account the physical properties of disks in order to achieve efficient I/O-performance that allow these data structures to be successfully plugged in software libraries.

A lot of work has already been done in designing I/O-efficient variants of well-known internal-memory data structures (see, e.g., [Vitter 1998] for a survey). In all these cases, the model adopted to evaluate their efficiency is the *Parallel Disk Model* proposed by Shriver and Vitter [Shriver and Vitter 1994]. Here the memory of a computer is abstracted to consist of *two levels*: a fast and small internal memory of size $M$, and a slow and arbitrarily large external memory physically partitioned into $D$ independent disks. Data between the internal memory and the disks is transfered in blocks of size $B$ (called *disk pages*). Each disk access transfers a total of $DB$ items by retrieving one page from each disk. Algorithmic performance is measured in this model (i) by counting the number of disk accesses performed (abbreviated as I/Os), (ii) by measuring the internal CPU time, and (iii) by counting the number of occupied disk pages. Recently [Farach et al. 1998], this accounting scheme has been refined by separating between consecutive (bulk) I/Os and I/Os to random locations. Bulk I/Os are known to be faster than random I/Os because they can profit from the prefetching and caching policies of the underlying disk system. We will use the concept of bulk I/O in the experimental section to refine our analysis of the I/O-behavior of the data structures tested.

## 1.1 Previous Work

It has been observed by several researchers that $k$-ary heaps perform better than classical binary heaps on multi-level memory systems [Naor et al. 1991; LaMarca and Ladner 1997]. Consequently, a variety of known external PQs follow this design paradigm by using a *multi-way tree* as a basic structure. Buffer trees [Arge 1995; Hutchinson et al. 1997] and $M/B$–ary heaps [Fadel et al. 1999; Kumar and Schwabe

1996] achieve optimal $O((1/B)\ \log_{M/B} N/B)$ amortized I/Os per operation[3]. Unfortunately, most of these data structures are quite complex in implementation (the simplest proposal is given in [Fadel et al. 1999]), and the constants hidden in the space and I/O bounds are not negligible. For instance, all these data structures require the maintenance of some kind of child pointers and some rebalancing information which induce significant space overhead and impose writing complex rebalancing code. Recently, starting from an idea of Thorup [Thorup 1996] for RAM PQs, Brodal and Katajainen [Brodal and Katajainen 1998] designed an external-memory PQ consisting of a *hierarchy of sorted lists* that are merged upon level– or internal-memory overflows. Although I/O-optimal in the worst case, their proposal is much too complicated and hence not attrative for practical applications. In this paper, we remove this deficit by providing a variation of the *hierarchical approach* which combines attractive amortized bounds with simplicity of programming.

### 1.2 This Paper

We study two novel external-memory heaps designed to offer effective I/O-performance in the practical setting. The first PQ proposal concerns an adaptation of the *two-level radix heap* [Ahuja et al. 1990] to the external memory. This external PQ supports monotone insertions and manages integer keys in a range size $C$. It achieves an amortized I/O-bound of $O(1/DB)$ for the `insert` and $O((1/DB)\log_{\frac{M}{(DB \log C)}} C)$ for the `delete_min` operation. The space requirement is optimal, and the I/O-constants are actually very small, thus driving us to conjecture good performance in practice. Our second PQ proposal is a simplification of [Brodal and Katajainen 1998], carefully adapted to exploit the simplicity of a collection of fixed-size lists over balanced tree structures. The resulting array-based PQ is easy to implement, is I/O-optimal in the amortized sense, does not impose any constraints on the priority values (cf. radix heaps) and involves small constants in the I/O, time and space bounds. Consequently, this structure turns out to be very promising in the practical setting and thus deserves a careful experimental analysis to validate its conjectured superiority over the other PQs.

In the second part of the paper, we will perform an extensive set of experiments comparing the implementation of four external-memory PQs: one based on Buffer Trees [Arge 1995], another based on B-Trees [Bayer and McCreight 1972], and our new proposals R-Heaps and Array-Heaps. Additionally, we will compare these PQs with four internal-memory priority queues: Fibonacci heaps, $k$-ary heaps, pairing heaps and internal radix heaps. Our experimental framework includes some simple tests which are used to determine the actual I/O-behavior of `insert` and `delete_min` operations, as well as more advanced tests aimed at evaluating the CPU-speed of the internally used data structures. As a final experiment, we will test our PQs in a real setting by considering sequences of `insert/delete_min` operations which were traced from runs of Dijkstra's shortest path algorithm. These sequences will allow us to check the behavior of our PQs on a "non-random", but application driven, pattern of disk accesses. We think that our variegate spectrum of experimental results gives a good picture of the I/O-properties of these external-PQs, thus being helpful to anyone interested in using such data structures on very

[3]We use the notation $\log_x(n)$ to denote $\log_2(n)/\log_2 x$ and $\log n$ for $\log_2(n)$

large data sets.

## 2. EXTERNAL RADIX HEAPS

### 2.1 One-disk model

Our first external-heap proposal consists of a simple and fast data structure based on two-level radix heaps [Ahuja et al. 1990] (hereafter *R–heaps*). Let $C$ be a positive integer constant and assume that the element priorities are positive integers. R–heaps work under the following condition:

CONDITION 2.1 (R-HEAPS). *Upon insertion, any priority value must be an integer in the range* $[\mathtt{m}, \mathtt{m}+\mathtt{C}]$*, where* m *is the priority value of the last element removed from the heap via a* `delete_min` *operation* ($\mathtt{m} = 0$ *if no* `delete_min` *was performed up to now).*

Hence the queue is *monotone* in the sense that the priority values of the deleted elements form a *nondecreasing* sequence. This requirement is fulfilled in many applications, e.g., in Dijkstra's shortest path algorithm. The need for integer priorities is not severe since interpreting the binary representation of a floating point number as an integer does not change the ordering relation [IEEE 1987]. R–heaps with $C = 2^{64}$ can therefore be used also to manage 64 bit floating point numbers.

The structure of the *external* version of the R-heap data structure is defined as follows. Let $r$ be an arbitrary positive integer (also called *radix*) and choose the parameter $h$ to be the minimum integer so that $r^h > C$ (i.e. $h = \lceil \log_r(C+1) \rceil$). Let $k$ be the priority of an arbitrary element in the queue and let $k_v \dots k_h k_{h-1} \dots k_0$ be its representation in base $r$. Similarly, let $m_v \dots m_h m_{h-1} \dots m_0$ be the representation in base $r$ of the current minimum priority min. According to Condition 2.1, we know that if an element with priority $k$ belongs to the queue then $k - \mathtt{min} < r^h$. Consequently $k_v = m_v, k_{v-1} = m_{v-1}, \dots, k_{h+1} = m_{h+1}$ and either $k_h = m_h$ or $k_h = (m_h + 1) \mod r$. The latter case only occurs if $m_h = r - 1$ and $k_h = 0$. Namely, if the two priorities differ, they must differ in their least significant $h$ digits.

The external R–heap consists of three parts (see also Figure 1):

(1) A collection of $h$ arrays, each of size $r$. Every array entry is a linear list of blocks called a *bucket*. Let $\mathcal{B}(i, j)$ denote the bucket associated with the $j$–th entry of the $i$–th array, for $0 \le i < h$, $0 \le j < r$. Each bucket keeps its first block (disk page) in internal memory. Bucket $\mathcal{B}(0, m_0)$ stores the first two blocks in internal memory.
(2) A special bucket $\mathcal{N}$ that also stores its first block in main memory. The internal space requirement of point 1 and 2 constrain $r$ to satisfy the relation $(h \cdot r + 2) \cdot B \le M$.
(3) An internal memory priority queue $\mathcal{Q}$ containing all indices of the non-empty buckets. These indices are ordered lexicographically, i.e., $(i, j) < (i', j')$ if either $i < i'$ or, $i = i'$ and $j < j'$. $\mathcal{Q}$ never stores more than $h \cdot r$ indices.

All elements of the R-heap reside in any of the buckets, either in bucket $\mathcal{N}$ or in a bucket $\mathcal{B}(i, j)$. An element with priority $k$ is stored according to the difference between its r-ary representation and the r-ary representation of min. Let $k =$

$k_v, \ldots, k_0$ and `min`$= m_v, \ldots, m_0$ be the corresponding r-ary representations. The element with priority $k$ is stored in bucket $\mathcal{N}$ if $k_h = (m_h + 1) \bmod r$, in all other cases it is stored in bucket $\mathcal{B}(i, j)$ where $i = \max(\{l \,|\, m_l \neq k_l, 0 \leq l < h\} \cup \{0\})$ and $j = k_i$. The elements inside a bucket are not sorted. Note that an element with priority equal to `min` is stored in bucket $\mathcal{B}(0, m_0)$. Furthermore, all elements that are stored in bucket $\mathcal{B}(0, j)$ must have the same priority value $j$ and the buckets $\mathcal{B}(0, j)$, with $j < m_0$, are empty.

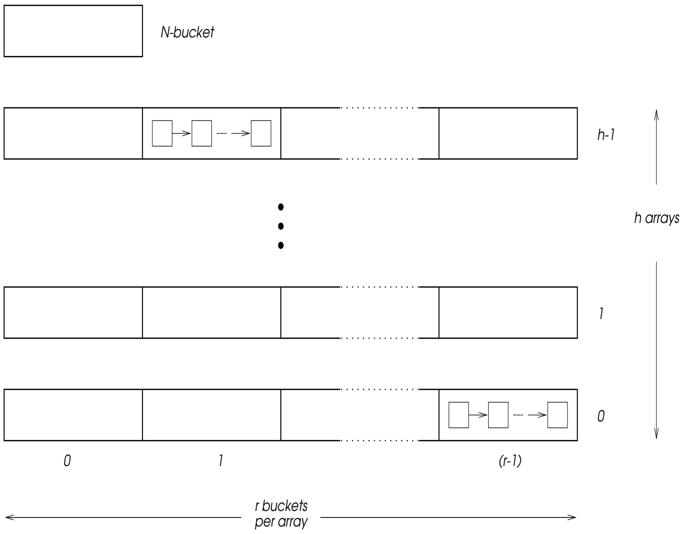


Fig. 1. The structure of the external radix heap.

**Insert:** In order to insert a new element with priority $k$ in the external R–heap, we first compute the least significant $h+1$ digits of $k$ in base $r$, thus taking $O(h)$ time and no I/Os. Then we insert that element into the bucket $\mathcal{N}$ whenever $k_h = (m_h + 1) \bmod r$; otherwise, we insert it into the bucket $\mathcal{B}(i, k_i)$, where $i = \max(\{l \mid m_l \neq k_l, 0 \leq l < h\} \cup \{0\})$. This takes $O(1)$ time. If bucket $\mathcal{B}(i, k_i)$ is empty before the `insert` operation, we also insert the index $(i, k_i)$ into $\mathcal{Q}$ taking $O(\log(rh))$ time. Notice that the insertion of an element with priority $k$ into the first block of bucket $\mathcal{B}(i, k_i)$ might completely fill it. In this case, we write that block to the disk and link it with the previous heading block of that bucket, i.e., the disk block is inserted at the front of the linked list. Bucket $B(0, m_0)$ is treated in a special way. We keep two blocks in main memory, thus managing it as an external stack. When the second block is filled by newly inserted elements, we move it to disk and move the first block into the second block.

**Delete_min:** If the bucket $\mathcal{B}(0, m_0)$ is not empty, we delete an arbitrary element from this bucket. Notice that all elements in bucket $\mathcal{B}(0, m_0)$ have the same priority and they are all equal to `min`. It may be necessary to load a block from disk as a previously occuring `delete_min` operation may have completely deleted both internal blocks of bucket $\mathcal{B}(0, m_0)$. As bucket $\mathcal{B}(0, m_0)$ is treated as an external stack, we are able to guarantee $O(1/B)$ I/Os and $O(1)$ time per operation in an intermixed sequence of inserts and delete operations on this bucket.

If instead the bucket $\mathcal{B}(0, m_0)$ is empty, we then access the internal priority queue $\mathcal{Q}$ and determine the lexicographically first non-empty bucket which is either some bucket $\mathcal{B}(i, j)$ or the bucket $\mathcal{N}$. In both cases, we scan the selected bucket and determine the new value for `min`. Since the minimum has changed, we need to reorganize the elements in the current bucket $\mathcal{B}(i, j)$. These elements are redistributed according to the rule exploited for the `insert` operation, thus they are moved to buckets in lower levels. These buckets were empty before the redistribution, so that their filling induces a change in the internal priority queue $\mathcal{Q}$. It is crucial to observe that all buckets $\mathcal{B}(i', j')$, with $i' > i$ or $i' = i$ and $j' > j$, including the bucket $\mathcal{N}$, do not need to be updated because the new `min` has changed *only* in the least significant $i$ digits with respect to the old `min`. Therefore, these elements remain in their buckets and overall no other bucket except $\mathcal{B}(i, j)$, must be reorganized.

We point out that each element in the R–heap can be redistributed at most $h$ times, namely once for each of the $h$ arrays. In fact, each time an element is redistributed, it moves from a bucket $\mathcal{B}(i, j)$ to a bucket $\mathcal{B}(i', j')$ with $i' < i$. The redistribution process touches each element of a bucket twice, when it is read out of its current bucket and when it is written to its new bucket according to the redistribution rule. This amounts to two scans of the current bucket, and thus $O(1/B)$ amortized I/Os and $O(\log(hr))$ time (due to the update of $\mathcal{Q}$) per redistributed element. Consequently the cost for each element, from its insertion up to its deletion (as minimum), is $O(h \log(hr))$ time and $O(h/B)$ I/Os.

It remains to determine the appropriate values for $r$ and $h$ that allow the R–heap data structure to work efficiently. The constraint we previously imposed on these parameters was that $(r \cdot h + 2) \cdot B + O(r \cdot h) \leq M$. This ensured that the first block of every bucket, and the internal queue $\mathcal{Q}$, fit in internal memory. The second additive term is nearly negligible because $B$ is large, and thus will not be taken into account. Now, since $h = \Theta(\log_r C)$, it suffices to choose the maximum value of $r$ such that (setting $m = M/B$):

$$r \log_r C \leq m \;\Rightarrow\; \tfrac{r}{\log r} \leq \tfrac{m}{\log C} \;\Rightarrow\; r = \Theta(\tfrac{m}{\log C} \log \tfrac{m}{\log C})$$

From the previous discussions, we derive the following result:

THEOREM 2.2. *Let $m = M/B$ and let $r = \Theta(\frac{m}{\log C} \log \frac{m}{\log C})$. An* `insert` *into an external R–heap takes $O(1/B)$ amortized I/Os and $O(\log(r \log_r C))$ amortized CPU time. A* `delete_min` *takes $O((1/B) \log_r C)$ amortized I/Os and $O(\log_r C \cdot \log(r \log_r C))$ amortized CPU time.*

PROOF. According to the discussion above, we have that $O(h \log(hr))$ time and $O(h/B)$ amortized I/Os are required to manage a single element from its insertion to its deletion from the queue (as minimum element). Upon insertion, each element gets $O(h/B)$ credits. Operation `insert` needs $O(1/B)$ credits. The cost of the redistribution process is charged to the `delete_min` operation and accounts for $O(h/B)$ credits. The bounds follow by setting $h = \Theta(\log_r C)$. □

We note that the amortized cost of the `delete_min` is larger than the one of the `insert`, but this is just a matter of accounting. Nonetheless the `delete_min` operation is so simple that it will be very fast in practice, as we experimentally prove in Section 4. Notice that for typical values of the parameters $M$, $B$ and $C$, the value of $\log_r C$ is about two or three, and thus negligible in the final performance. As far as the radix $r$ is concerned, we observe that its setting depends on $M$, $B$ and $C$. To better evaluate the relationship among all these values we plot them by assuming $m = M/B = 2^x$ and $C = 2^y$ (hence $r = (2^x/y) \cdot (x - \log(y))$). See Figure 2 for details.

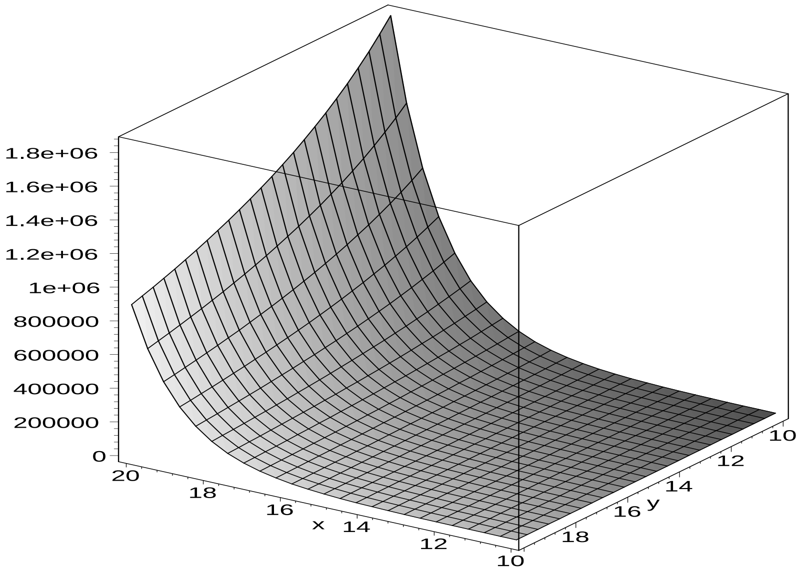


Fig. 2. Relationship between $r, M, B$ and $C$. $M/B = 2^x$ and $C = 2^y$. The z-axis shows the dependancy of radix $r$ on $x$ and $y$.

LEMMA 2.3. *An external R–heap storing $N$ elements occupies no more than $N/B$ disk pages.*

PROOF. Only one disk page can be non-full in each bucket, and this page does not reside on disk but is located in internal memory. □

External radix heaps are efficient in applications where $C$ is small. This typically occurs in time-scheduling problems, where $C$ is an upper bound to the time after which an event takes place, or in Dijkstra's shortest path computations, where $C$ is an upper bound on the edge weight. We investigate the practical behavior of R–heaps in Section 4.

### 2.2 D–Disk Model

Radix-heaps can be easily extended to work efficiently in the D–disk model by means of the *disk striping* technique. Disk striping on $D$ disks can be seen as using a single disk with block size $B' = D \cdot B$. If we apply this to our external R–heaps we get the result that the constraint on the memory occupancy of the internal data structures becomes $r \cdot h \cdot B' \leq M$, which is $r \cdot h \cdot B \cdot D \leq M$. Hence we can choose $r = \Theta((m/\log C)\log(m/\log C))$ with $m = M/DB$.

THEOREM 2.4. *Let* $m = M/(DB)$ *and let* $r = \Theta(\frac{m}{\log C}\log\frac{m}{\log C})$. *An* `insert` *into an external R–heap takes amortized* $O(1/DB)$ *I/Os and a* `delete_min` *takes* $O((1/DB)\log_r C)$ *amortized I/Os.*

We point out that the bound for `delete_min` is not optimal because it is not $D$ times smaller than the bound for the one-disk case. Actually, $D$ occurs in the base $r$ of the logarithm.

## 3. EXTERNAL ARRAY-HEAPS

Our second proposal is a simplification of the data structure introduced in [Brodal and Katajainen 1998] and consists of two parts: an internal heap $\mathcal{H}$ and an external collection $\mathcal{L}$ of sorted arrays of different lengths. The collection $\mathcal{L}$ is in turn subdivided into $L$ levels $\mathcal{L}_i$, each consisting of $\mu = (cM/B) - 1$ arrays (called *slots*) having length $l_i = (cM)^i/B^{i-1}$, where $c < 1$. Each slot of $\mathcal{L}_i$ is either empty or contains a sorted sequence of at most $l_i$ elements (see Figure 3). The following property is easy to prove:

PROPERTY 3.1. *The total size of* $(\mu + 1)$ *slots, each containing* $l_i$ *elements, is equal to* $l_{i+1}$.

Elements are inserted into $\mathcal{H}$; if $\mathcal{H}$ gets full, then $l_1 = cM$ of these elements are moved to the disk and stored in sorted order into a free slot of $\mathcal{L}_1$. If there is no such free slot (i.e., an *overflow* occurs), all the slots of $\mathcal{L}_1$ are merged with the elements coming from $\mathcal{H}$, thus forming a sorted list which is then moved to the next level $\mathcal{L}_2$. If no free slot of $\mathcal{L}_2$ exists, the overflow process is repeated on $\mathcal{L}_2$ by searching for a free slot in $\mathcal{L}_3$. This process continues until a free slot is eventually found.

The `delete_min` operation maintains the invariant that the smallest element always resides in the internal heap $\mathcal{H}$. This is ensured by refilling $\mathcal{H}$ whenever it becomes empty via the removal of some appropriate blocks from the sorted slots of $\mathcal{L}$.

We describe now in more detail two versions of this heap; the former is a simplified non-optimal data structure intended to be fast in practice, the latter is slightly more complicated but reaches I/O-optimality.

### 3.1 A practical version

The internal data structure $\mathcal{H}$ is divided into two parts, $H_1$ and $H_2$. $H_1$ stores the newly inserted elements (at most $2cM$ in total), whereas $H_2$ stores at most the smallest $B$ elements of each non-empty slot in (any level of) $\mathcal{L}$. As a result, fewer than $cM(2 + L)$ elements reside in internal memory. Furthermore, $(\mu + 1)B = cM$

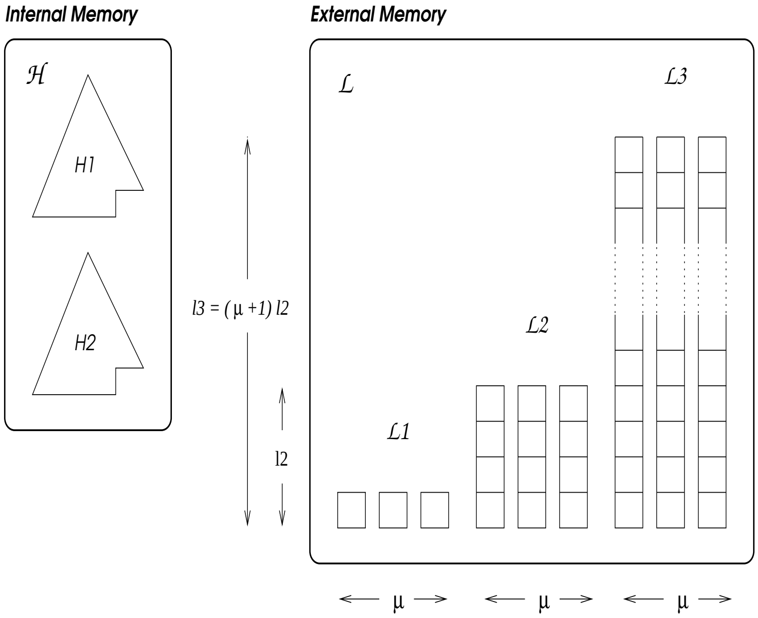


Fig. 3. The array heap

internal memory space is needed upon level overflow, in order to merge the $\mu$ slots of any level plus the one sequence arriving from the previous level. This imposes the requirement that $L$ must behave as a constant in order to ensure that we have enough internal memory to host the required data structures. We will comment further on this at the end of this section, and now just observe that for practical values of the system parameters, it is $L \leq 4$ so that the above relation can be fulfilled in practice by setting $c = 1/7$. For the remainder of this section we assume that $cM > 3B$.

Let us now introduce in detail four basic procedures which are used in operations `insert` and `delete_min` to manage overflows and underflows in slots.

—Merge-Level(i,S,S') produces a sorted sequence $S'$ by merging the sorted sequences in the $\mu$ slots of $\mathcal{L}_i$ (including their first blocks in $\mathcal{H}$) and the sorted sequence $S$. Every block of sequence $S$ and every block of each of the slots in $\mathcal{L}_i$ is read and written once. Therefore, the operation takes at most $2\lceil(|S| + \mu l_i)/B\rceil \leq 2\lceil(l_{i+1})/B\rceil \leq 3l_{i+1}/B$ I/Os since $l_{i+1} \geq l_1 = cM > 3B$.

—Store(i,S) assumes that $\mathcal{L}_i$ contains an empty slot and $S$ has length in the range $[l_i/2, l_i]$. $S$ is read and then stored into an empty slot of $\mathcal{L}_i$ and its smallest $B$ elements are moved to $\mathcal{H}$. This takes at most $2\lceil l_i/B\rceil \leq 3l_i/B$ I/Os since $l_i \geq l_1 = cM > 3B$.

—Load($i, j$) fetches the next $B$ smallest elements from the $j$th slot of $\mathcal{L}_i$ into the internal heap $\mathcal{H}$. It therefore takes one I/O.

—Compact(i) is executed if there are at least two slots in level $\mathcal{L}_i$ for which the sum of their elements, including those in $\mathcal{H}$, is at most $l_i$. Then these two slots (plus their first blocks in $\mathcal{H}$) are merged into a new slot, thus freeing a slot of

$\mathcal{L}_i$. The smallest $B$ elements of the new slot are moved to $\mathcal{H}$. Let $j$ and $k$ be the slots of level $\mathcal{L}_i$ that are merged during the execution of Compact(i) and let $e_j$ (resp. $e_k$) be the number of elements in slot $j$ (resp. $k$). Compact(i) needs at most $\lceil e_j/B \rceil + \lceil e_k/B \rceil + \lceil l_i/B \rceil \leq (e_j + e_k)/B + l_i/B + 3 \leq 2l_i/B + 3 \leq 3l_i/B$ I/Os because $l_i \geq l_1 = cM > 3B$.

**Insertion.** A new element is initially inserted into $H_1$. If $H_1$ gets full, the largest $cM$ elements of $H_1$ form the sequence $S$ which is then moved to the level $\mathcal{L}_1$ (the smallest $B$ elements in $S$ stay in internal memory and are copied to $H_2$). This insertion may influence all the levels of $\mathcal{L}$ in a cascading way. At a generic step, the sorted sequence $S$ has originated from level $\mathcal{L}_{i-1}$ (initially $\mathcal{L}_0 = H_1$) and must be moved to the next level $\mathcal{L}_i$. If $\mathcal{L}_i$ contains an empty slot, then $S$ is stored into it (via Store(i,S)). Otherwise a further overflow occurs at $\mathcal{L}_i$ and thus, since all slots of $\mathcal{L}_i$ contain at least $l_i/2$ elements, they are merged with $S$ forming a new sorted sequence $S'$ (Merge-Level(i,S,S')). Such a sequence $S'$ is moved to the next level and the loop above is repeated with $S'$ playing now the role of $S$. Of course, at most $L$ iterations of the loop suffice to find a free slot, because we eventually reach the last level of $\mathcal{L}$.

**Deletion.** The smallest element is removed from the internal heap $\mathcal{H}$. If it was the last element coming from a slot $j$ of some level $\mathcal{L}_i$, we load the next $B$ elements from this slot into $H_2$ (Load($i,j$)). This way, the corresponding internal buffer is refilled with the smallest $B$ elements of that slot. In order to not use unreasonably many partially filled slots, we check the size of the slot affected by a load operation and possibly execute a Compact on it. Compact merges two partially filled slots into one. Consequently, every level $\mathcal{L}_i$ consists of at most one slot containing fewer than $l_i/2$ elements. The Compact operation moves the smallest $B$ elements of the newly formed sequence to internal memory, so that the block of the slot containing the largest elements might be partially filled in external memory. This partially filled block resides in external memory.

3.1.1 *Correctness and I/O-bounds.* To prove the correctness of the above operations we must show that during overflows, there is enough space in one of the next levels to store the newly arriving elements, and we must guarantee that the minimum element is always in the internal heap $\mathcal{H}$. As elements only move to higher levels, we do not lose any elements during the overflow process. Therefore elements either reside in the internal data structure $\mathcal{H}$ or in the external data structure $\mathcal{L}$.

LEMMA 3.2. *The smallest element is always stored in $\mathcal{H}$.*

PROOF. The minimum element was either newly inserted and is therefore stored in $H_1$ or it belongs to a specific slot in some level of the external $\mathcal{L}$. As the slots are linearly ordered, the smallest element of each of them is stored in its first block, which indeed resides in the internal data structure $H_2$. □

We now turn to bounding the number of levels in $\mathcal{L}$ and to estimating the overall disk space needed by our data structure.

LEMMA 3.3. *After $N$ operations, there are at most $L \leq \log_{cM/B}(N/B)$ levels.*

PROOF. In the worst case, no deletions occur and therefore each overflow moves the maximum possible number of elements from one level to another.

Thus, the number of elements in $j$ levels is $\sum_{i=1}^{j} \frac{(cM)^{i+1}}{B^i} < \frac{(cM)^{j+2}}{B^{j+1}}$. We must choose the smallest $j$ so that $(cM)(cM/B)^{j+1} \overset{!}{\geq} N$. Since $cM > 3B$ we have $j \geq log_{cM/B}(N/B) - 1$. $\square$

LEMMA 3.4. *Every level $\mathcal{L}_i$ contains at most one slot which is non-empty and consists of fewer than $l_i/2$ elements (i.e., this slot is underfilled). Hence the total number of used disk pages is bounded above by $2(X/B) + L$, where $X$ is the number of elements currently in the heap. The total required internal space is $cM(3 + L)$.*

PROOF. The $j$th slot in $\mathcal{L}_i$ may become underfilled only after the execution of Load$(i, j)$. By induction, let us assume that at most one slot in $\mathcal{L}_i$ is currently underfilled, say the $j'$-th slot. After Load$(i, j)$, the deletion operation checks the occupancy of the $j$th slot, and if underfilled it calls Compact(i) thus emptying $j'$ and refilling $j$. The index $j'$ can be easily found. After Compact(i) no other slot in $\mathcal{L}_i$ is underfilled, so that the invariant is preserved. In the case that no slot in $\mathcal{L}_i$ was underfilled before Load$(i, j)$, $j$ becomes the unique underfilled slot of $\mathcal{L}_i$.

Partially filled pages are created by the `delete_min` operation. Recall that a Load operation may trigger a Merge-Level or a Compact operation; in this case the internal-memory blocks of the slots involved by these operations are merged thus possibly causing some elements in $\mathcal{H}$ of these slots to be moved to external memory. As a consequence the smallest $B$ of these elements remain into $\mathcal{H}$, but the moved elements might create a partially filled block in external memory. In any slot (of any level), only one page can be partially filled. Additionally only one slot per level is underfilled, so that in the worst case, it can consist of only one partially filled page (thus giving the term $L$). For all the other slots, since they are at least half full, they consist of at least one full page. Therefore, the number of their top (partially filled) pages is no more than the total number of occupied pages, which is indeed $X/B$. Concerning the internal space, $H_1$ stores $2cM$ elements and $H_2$ stores one disk page per slot which is $(\frac{cM}{B} - 1) \cdot B \cdot L \leq cM \cdot L$. Further $(\mu + 1) \cdot B = cM$ internal space is needed to merge the $\mu$ slots of level $\mathcal{L}_i$ plus the sequence $S$ arriving from level $\mathcal{L}_{i-1}$. $\square$

We remark here that it is possible to achieve a (optimal) space bound of $X/B$ disk pages by properly acting on the emptying and refilling operations. Nevertheless, the result in Lemma 3.4 shows that the number of levels $L$ should behave as a constant in order to ensure the correctness of the data structure. If fact, only when $L \leq 1/c - 3$ are we guaranteed to have enough internal memory to host the heaps $H_1$ and $H_2$. This is asymptotically not true but reasonable in practice due to Lemma 3.3. Indeed, let us choose $c = 1/7$ and set $M = 256$ Mbytes and $B = 64$ kbytes as it occurs in the machine used for our experiments in Section 4. Under this setting, we derive that $L \leq 4$ for $N \leq 6.8 \cdot 10^{15}$. Actually $N$ is in the range of the Petabytes for this "small" machine !

We are therefore left to prove the I/O-bounds for the operations `insert` and `delete_min`.

THEOREM 3.5. *Let us assume that $N, M$ and $B$ are such that the data structure $\mathcal{H}$ fits in internal memory and $cM > 3B$. In a sequence of $N$ intermixed* `insert` *and* `delete_min` *operations,* `insert` *takes $18L/B$ amortized I/Os and* `delete_min` *takes $7/B$ amortized I/Os.*

Proof. We prove our amortized bounds by using an accounting argument. We associate with each non-empty slot $j$ of level $\mathcal{L}_i$ a deposit $D_{i,j}$ that accounts for $6x/B$ credits, where $x$ is the number of its empty entries. Initially, all slots are empty and thus the accounts have zero credits. Furthermore, we associate with the internal heap $\mathcal{H}$ a deposit for that we ensure to contain at least 1 credit in the case that a Load operation must be performed. This credit will be used to pay for the Load.

Each element is to be inserted with $18L/B$ credits. Each element uses $18/B$ credits from that amount everytime it moves from one level to the next one according to the Merge-Level operation. In case of a deletion, $7/B$ credits are left in the internal heap $\mathcal{H}$ (notice that the deleted element resided in $\mathcal{H}$ before deletion).

It remains to show that whenever an operation `insert` or `delete_min` is executed we have enough credits to pay for them. Recall that `insert` relies on operations Merge-Level and Store, whereas `delete_min` uses Compact and Load.

In the case of an overflow at level $\mathcal{L}_i$ triggered by an `insert` operation, we use Merge-Level(i,S,S') and then execute Store(i+1,S') to store the sequence $S'$ in a free slot of level $\mathcal{L}_{i+1}$ (if any). From above, Merge-Level(i,S,S') needs $3l_{i+1}/B$ I/Os and Store(i+1,S') needs $3l_{i+1}/B$ I/Os, so that merging and storing these elements is possible in $6l_{i+1}/B$ I/Os. Since the size of $S'$ is between $[l_{i+1}/2, l_{i+1}]$, this I/O-cost can be payed by taking $12/B$ credits from the deposit of each element affected by this moving operation. In fact, we are moving at least $l_{i+1}/2$ elements from level $\mathcal{L}_i$ to level $\mathcal{L}_{i+1}$, and thus we can employ the credits associated with their $i$–th level. We note that the slot of level $\mathcal{L}_{i+1}$, say $j$, might be partially filled after the storage of the elements and hence it is necessary to put credits in its deposit $D_{i+1,j}$ to maintain the property that $D_{i+1,j}$ accounts for the number of its empty entries. We observe that the number of empty entries $l_{i+1} - S'$ is at most $l_{i+1}/2$ so that we have to fill in $3l_{i+1}/B$ credits. Since $|S'| \geq l_{i+1}/2$ we can take $6/B$ credits from each of its elements and put them in the deposit $D_{i+1,j}$. Globally, $18/B$ credits have been used to move an element from $\mathcal{L}_i$ to $\mathcal{L}_{i+1}$, and this is the amortized cost associated with the `insert` operation for one level. In all, $18L/B$ credits suffice to pay for the cost of `insert`.

In the case of a `delete_min` operation, we leave $7/B$ credits in the deposit of $\mathcal{H}$. If a Load is required, its cost is paid by using $B(1/B) = 1$ credit from the deposit of $\mathcal{H}$. These credits were left by the $B$ elements removed before that Load. The other $B(6/B)$ credits left by these elements, are added to the deposit of the slot $j$ on which the Load operated, thus preserving the invariant on the slot account $D_{i,j}$. If the element later participates again in an overflow operation, it will re-enter the external level structure exactly in level $\mathcal{L}_i$. Therefore it still has enough credits to pay for the following Merge-Level operations.

Finally, we recall that Compact($i$) needs $3l_i/B$ credits, and this can be paid by using the credits associated with the deposits of the slots to be compacted. In fact, when the compaction is executed on two slots, say $j'$ and $j''$, their number of empty entries is at least $l_i$ and hence, $6l_i/B$ credits are available in the accounts $D_{i,j'}$ and $D_{i,j''}$. When we merge the two slots, slot $j''$ becomes empty and slot $j'$ contains at most $l_i/2$ empty entries. Hence we use $3l_i/B$ credits from the available $6l_i/B$ credits above to pay for the compaction, and the other $3l_i/B$ credits are used to refill the deposit of $D_{i,j'}$ (thus accounting for at most $l_i/2$ empty entries). □

### 3.2 An improved version

We introduce a slight modification in the external data structure $\mathcal{L}$ that reduces its internal space usage thus making it also useful when less stringent conditions are imposed on $L$, while still keeping its simplicity. We associate with each level $\mathcal{L}_i$ a special slot of length $l_i$, called min-buffer$_i$. It always contains the smallest elements in the slots of $\mathcal{L}_i$ as a sorted sequence of at most $l_i$ elements. We copy *only the first block* of min-buffer$_i$ into $\mathcal{H}$. Consequently $\mathcal{L}_i$ keeps no more than $B$ elements in internal memory, instead of the previous $B\mu$ elements. As a result $\mathcal{H}$ consists of at most $2cM$ elements (from $H_1$) plus $LB$ elements coming from the external structure $\mathcal{L}$.

The requirement that min-buffer$_i$ always holds the smallest elements within $\mathcal{L}_i$ can be easily maintained under overflows from level $\mathcal{L}_{i-1}$ to level $\mathcal{L}_i$: when a sequence $S$ of elements arrives from $\mathcal{L}_{i-1}$, we check whether $S$ contains elements that are smaller than the elements currently stored in min-buffer$_i$. If this is the case, we swap elements between $S$ and min-buffer$_i$ so that the smallest elements are in min-buffer$_i$. Notice that min-buffer$_i$ might run out of elements due to a sequence of `delete_min` operations. A Refill takes care of this situation by merging the $\mu$ slots in $\mathcal{L}_i$ until their smallest $l_i$ elements have been computed. The merge cost can be charged on the $l_i/2$ deletions that have been performed before min-buffer$_i$ became empty.

THEOREM 3.6. *Under the assumption* $\log_{M/B}(N/B) \leq M/B$*, the new variant of the array heap requires* $O((1/B)\log_{M/B}(N/B))$ *amortized I/Os for the* `insert` *operation and* $O(1/B)$ *I/Os for the* `delete_min` *operation. The space occupancy is* $O(N/B)$ *disk pages.*

PROOF. The number of levels is at most $O(\log_{M/B}(N/B))$ (see Lemma 3.3). Thus, the bounds can be shown by a similar credit argument as that used in the proof of Theorem 3.5. □

### 3.3 Array-Heaps in the D-Disk Model

Since our aim is to get a simple and effective heap data structure for multiple disks, we combine array-heaps with the elegant approach of [Barve et al. 1997]. We indeed stripe the blocks of each slot among the disks using a *random starting point*. We only discuss the implementation of Merge-Level(i,S,S') and Refill(i) because all the other basic tools (i.e., Store, Compact and Load) generalize optimally to the $D$-disk setting by just assuming that each slot of any level is stored in a *striped* way among the $D$ disks.

The efficiency of the randomized mergesort algorithm of [Barve et al. 1997] relies on the random selection of the starting disks for the striped runs and also on a *forecast* data structure that makes sure to have the appropriate disk blocks in internal memory when they are needed. Our main idea is therefore to use the merge-pass of that algorithm to produce just the first $\ell$ elements of the final sorted sequence, thus requiring only $O(\ell/DB)$ I/Os. For Merge-Level(i,S,S') we set $\ell \leq l_{i+1}$, for Refill(i) we set $\ell = l_i$, and thus require $O(l_{i+1}/DB)$ and $O(l_i/DB)$ *optimal I/Os*, respectively. By assuming that $D = O(B)$, we have

THEOREM 3.7. *Assume that* $\log_{M/B}(N/B) \leq M/B$*. Given a sequence of* $N$ *operations, the array heap data structure requires* $O((1/DB)\ \log_{M/B}(N/DB))$ *amor-*

*tized expected I/Os for an* `insert` *and* $O(1/DB)$ *amortized expected I/Os for a* `delete_min`*. The total occupied space is* $O(N/B)$ *disk pages.*

PROOF. According to [Barve et al. 1997], the number of runs merged at each round of randomized mergesort is $R = (M/B - 4D)/(2 + D/B)$ which is $\Theta(M/B)$ for $D = O(B)$; and this is optimal. □

## 4. EXPERIMENTS

We implemented external radix heaps, array heaps, buffer trees and B-trees using an external-memory library of algorithms and data structures called `LEDA-SM` [Crauser and Mehlhorn 1999] (an acronym for "LEDA for Secondary Memory")[4]. This library is an extension of the internal-memory library *LEDA* [Mehlhorn and Näher 1999] and is *portable*, *easy to use* and *efficient*. The library `LEDA-SM` consists of a collection of efficient data structures and algorithms explicitly designed to work in an external memory setting. `LEDA-SM` is implemented in `C++` as a set of template classes and functions. The system underlying `LEDA-SM` reflects a realistic view of the external memory model [Shriver and Vitter 1994]: The internal memory is directly provided by the internal memory of the computer, whereas the $D$ abstract disks are modeled by the file system, which also provides proper tools for implementing the low-level I/O via block transfers. Each disk is modeled with a *single file* and is divided into logical blocks (disk pages) of a fixed size $B$. The size of this file is fixed, thus modeling the fact that real disk space is bounded. If each disk file is on a different disk, `LEDA-SM` models independent disks. [5] Since `LEDA-SM` uses the file system, it explicitly takes advantage of the underlying *I/O-buffering* and *read-ahead* strategy at no additional implementation effort[6]. Computation and I/O is not directly overlapped as we do not use threading. However, the I/O system calls that write to disk overlap I/O and computation. The programmer can still keep track of the number of disk accesses (bulk, random and total) performed by the algorithms, because the system allows the explicit counting of the number of both writes and reads to the $D$ disks. Although files on file systems may not necessarily be consecutive on the underlying disk, one can guarantee consecutiveness on the file system by choosing an empty file system as modern file systems normally try to allocate disk space for files sequentially. Thus logically consecutive file locations tend to be physically consecutive on the disk. We performed some low-level tests with `LEDA-SM` which certified that bulk and random I/Os also have the conjectured effects during file I/O (see [Crauser 1998]).

### 4.1 Experimental Setup

We compare eight different PQ implementations, namely array heaps, external radix heaps, buffer trees, B-trees, Fibonacci heaps, $k$-ary heaps, pairing heaps and

[4]`LEDA-SM` can be downloaded from `http://www.mpi-sb.mpg.de/~crauser/leda-sm.html`. Other interesting external memory libraries are, e.g., TPIE [Vengroff and Vitter 1995] and `VIC*` (see `http://www.cs.dartmouth.edu/~thc/ViC`).

[5]At the moment we do not support real parallel read or write accesses via threading, this is planned in the future.

[6]As the file system is allowed to buffer disk pages, some of the disk requests can immediately be satisfied by the I/O–buffer. This can notably speed up the real performance of external-memory algorithms.

internal radix heaps. The first four priority queues are designed to work in external memory, whereas the last four are LEDA-implementations of well-known internal-memory priority queues. *B-trees* are a natural extension of $(a, b)$-trees to external memory where the fan-out of each internal node is increased to $\Theta(B)$, so that the entire node fits in a single disk page. `Insert` and `delete_min` require $O(\log_B N)$ I/Os in the worst case. B-trees are implemented as $B^+$-trees so that the keys are stored in the leaves and the internal nodes keep only some routing information. The tree structure is implemented by linking disk blocks via pointers.

*Buffer-Trees* are also an extension of $(a, b)$-trees but they are intended to reach better amortized I/O-bounds for the `insert` and `delete_min` operations, namely $O((1/B) \log_{M/B}(N/B))$ amortized I/Os. The fanout of the internal nodes is $\Theta(M/B)$ and thus the height is reduced to $O(\log_{M/B}(N/B))$ levels (notice that the case $M = O(B)$ does not occur in practice). Each internal node has associated with it a buffer of size $\Theta(M)$, which is usually kept on disk. Instead of storing a newly inserted element directly into the correct leaf, the element is put into the root's buffer. When this buffer is full, we empty it and move its elements to the appropriate buffers of the internal nodes in the next level. The smallest element resides somewhere in the buffers of the nodes on the leftmost path. The PQ now consists of two parts: one kept in internal memory and one stored on disk. The internal part keeps at most $\Theta(M)$ elements stored into a standard PQ (implemented by LEDA's sorted sequence); the external part contains the rest of the elements and is implemented by a buffer tree as indicated in [Arge 1995].[7] Newly inserted elements are checked against the ones stored in the internal part; if the internal data structure becomes full then $\Theta(M)$ of its elements are moved to the buffer tree. If the internal structure runs out of elements due to a sequence of `delete_min`s, we then empty all buffers on the leftmost path and take the smallest $\Theta(M)$ elements stored in the leftmost leaf. These are the smallest $\Theta(M)$ elements in the PQ. Internal nodes are implemented by an array of disk blocks; buffers are implemented by linked lists of disk blocks.

*Radix heaps* are implemented by means of $h$ arrays of size $r$ each. An array entry is a *bucket* and it is implemented by a linked list of disk blocks where the pointer information is kept inside the blocks themselves. A small LEDA Fibonacci heap is used to implement the internal priority queue $\mathcal{Q}$ (see Section 2). This choice comes from the fact that Fibonacci heaps are fast if the queue is small (see [Cherkassky et al. 1999]).

*Array heaps* are implemented as follows. $H_1$ is implemented as a binary heap of fixed size. We choose an array-based implementation so that we can extract the biggest $M/2$ elements by just sorting the array via the effective LEDA array-sorting code. $H_2$ is simply a small array storing the first block of each slot in every level. Operations `insert` and `delete_min` are implemented on top of this structure to add the necessary priority queue functionality. The external part consists of $L$ levels where a level is formed of $\mu + 1$ slots of size $l_i$. Each slot is a linked list of disk blocks. The linked disk blocks are allocated by `LEDA-SM` in such a way

[7] Starting from the original search tree implementation, we added some functionality so that the buffer tree can work as a priority queue.

that they are consecutive on the disk (like an array of disk blocks). Keeping the blocks additionally linked allows us to support an easy and efficient `delete_min` operation. During the `Load` operation (see Section 3.1), we can indeed use the pointer information to easily identify the interesting disk block. As `LEDA-SM` is able to allocate single disk blocks and as there is a general management structure of used and free disk blocks (empty pages can be reused), we are able to achieve the space bounds of Lemma 3.4. For the implementation of the internal-memory priority queues we refer to [Mehlhorn and Näher 1999].

We use the following three tests to analyze the performance of the PQs:

**Insert-All-Delete-All.** We perform $N$ `insert` followed by $N$ `delete_min` operations. This test allows us to compare the raw I/O-performance of the various PQs. The keys are `integers`, randomly drawn from the range $[0, 10^7]$.

**Intermixed insertions and deletions.** This test is used to check the I/O- and the CPU-speed of the internal-memory part of the external PQs. We first insert 20 million keys in the queue and then randomly perform `insert` and `delete_min` operations. An `insert` occurs with probability 1/3 and a `delete_min` occurs with probability 2/3.

**Dijkstra's shortest-path algorithm.** We simulate Dijkstra's algorithm in internal memory on a large graph (using a computer with a very large internal memory, a Sun Enterprise 10000 with 20 Gbytes of internal memory). Dijkstra's algorithm uses `decrease_priority` operations, but unfortunately, no external PQ is yet able to efficiently support such operations. Typically the external version of the Dijkstra's algorithm is obtained by "simulating" `decrease_priority` operations only via `insert` and `delete_min` plus some auxiliary data structures. A similar trick was first used by [Chiang et al. 1995] for external depth-first search. Our idea is therefore to simulate this variant of Dijkstra's algorithm on some instances of large graphs and record the sequence of `insert` and `delete_min` operations performed. These sequences of operations are then used to test our external PQs, thus allowing us to study the behavior of the external heaps in the case of "non-random", but application driven, update operations.

The tests were performed on a SPARC ULTRA 1/143 with 256 Mbytes of internal memory and a local 9 Gbytes fast-wide SCSI disk. The internal memory priority queues use swap-space on the local disk to provide the necessary working space and a page size of 8 kbytes. `LEDA-SM` used one single disk, i.e., all data structures were tested in the one disk case. We measured wall clock time together with number of executed I/Os. The external memory priority queues used a disk block size of 32 kbytes (this value is optimal for request time versus data throughput). Each external PQ uses for its in-core part about 16 Mbytes out of 256 Mbytes available. Thus there is enough space for the buffer cache of the file system. In order to have a better picture of the I/O-behavior of the experimented data structures, we decided to also estimate the actual "distribution" of the I/Os so to understand their degree of "randomness". This is a very important parameter to be considered since it is well known [Ruemmler and Wilkes 1994] that accessing one page from the disk in most cases decreases the cost of accessing the page succeeding it, so that *bulk* I/Os are less expensive per page than *random* I/Os. This difference becomes much more prominent if we also consider the reading-ahead/caching optimizations which are common in current disks and operating systems. In order to take into account this technical feature, without introducing new parameters that would make the analysis much more complex, we follow [Farach et al. 1998] by choosing a reasonable value for the *bulk load size* and by counting the number of random and total I/Os. Since the transfer rates are more or less stable (and currently large) while seek times are highly variable and costly (because of their mechanical nature), we choose a value for the bulk size which allows the *extra cost* induced by the seek step when data are fetched to be hidden. For the disk used in our experiments, the average `t_seek` is 11 msecs, the `disk_bandwidth` is 7 Mbytes/sec. We have then chosen `bulk_size` = 8 disk pages for a total of 256 kbytes. It follows that `t_seek` is 24% of the total transfer time needed for a bulk I/O. Additionally, the bulk size of 256 kbytes allows us to achieve 81% of the maximum data transfer rate of our disk while keeping the service time of the requests low. Using a page size of 32 kbytes, we still keep the service time for random blocks reasonably small and the throughput rate reasonably high.

### 4.2 Experimental Results

We comment below on the experimental results obtained for the tested priority queues. Table 1 summarizes the running time in seconds and the number of I/Os on the insert-all-delete-all test. Figure 4 shows a graphical plotting of the running times for the insert-all-delete-all test and Figure 5 shows the average time per `insert/delete_min` operation in microseconds.

4.2.1 *Results for Buffer Trees.* Our implementation follows an idea proposed in [Hutchinson et al. 1997] and uses big unbounded buffers instead of buffers of size $M$ [Arge 1995]. This allows for saving much time in the costly rebalancing process which does not occur as often as in the standard implementation. Another speed-up in our implementation is induced by the use of a simple in-core data structure (`sorted sequence` of LEDA) which reduces the CPU time. As a result, the `insert` operation is very fast as shown in Table 1. However, `delete_min` is slower than that of radix heaps and array heaps, and this is due to the costly rebalancing step. Earlier experiments [Crauser and Mehlhorn 1999; Hutchinson et al. 1997] have already shown that the internal buffer size heavily influences the performance of buffer trees. Unfortunately, the optimal value of the buffer size is machine dependent and must be found experimentally.

Summing the time required for the `insert` and `delete_min` operations, we conclude that buffer trees are three times slower than array heaps and six times slower than radix heaps (see Table 1). They perform approximately four (resp. three) times the number of I/Os executed by radix heaps (resp. array heaps). 75% of these I/Os are random I/Os that are mainly executed during the `delete_min` operation and are triggered by the costly rebalancing code. Overall buffer trees are the second best "general-purpose" PQ, when no restriction is imposed on the priority value.

| Insert/Delete_min time performance of the external queues (in secs) | | | | |
|---|---|---|---|---|
| N [$*10^6$] | radix heap | array heap | buffer tree | B-tree |
| 1 | 6/24 | 18/11 | 56/34 | 11287/259 |
| 5 | 17/97 | 74/63 | 148/309 | 66210/1389 |
| 10 | 35/178 | 353/89 | 201/882 | - |
| 25 | 85/372 | 724/295 | 311/2833 | - |
| 50 | 164/853 | 1437/645 | 445/6085 | - |
| 75 | 246/1416 | 2157/1005 | 569/9880 | - |
| 100 | 325/1957 | 2888/1408 | * | - |
| 150 | 478/3084 | 4277/2297 | * | - |
| 200 | 628/4036 | 5653/3234 | * | - |

| Insert/Delete_min time performance of the internal queues (in secs) | | | | |
|---|---|---|---|---|
| N [$*10^6$] | Fibonacci heap | $k$-ary heap | pairing heap | radix heap |
| 1 | 3/32 | 4/33 | 3/19 | 3/11 |
| 2 | 6/73 | 8/75 | 6/45 | 5/27 |
| 5 | 17/208 | 21/210 | 14/126 | 11/71 |
| 7.5 | $172800^*$/- | 32/344 | 22/207 | 18/124 |
| 10 | -/- | 43/482 | 30/291 | 23/162 |
| 20 | -/- | $172800^*$/- | $172800^*$/- | $172800^*$/- |

| Random/Total I/Os for external queues | | | | |
|---|---|---|---|---|
| N [$*10^6$] | radix heap | array heap | buffer tree | |
| 1 | 44/420 | 24/720 | 228/668 | |
| 5 | 422/3550 | 120/4560 | 16722/21970 | |
| 10 | 1124/8620 | 168/9440 | 35993/47297 | |
| 25 | 2780/21820 | 570/29520 | 93789/123285 | |
| 50 | 7798/56830 | 1288/66160 | 190147/249955 | |
| 75 | 12466/89370 | 2016/102480 | 286513/376625 | |
| 100 | 17736/124740 | 2776/139760 | * | |
| 150 | 27604/192500 | 4216/210080 | * | |
| 200 | 38284/211570 | 5712/284320 | * | |

Table 1. *Experimental results on the insert-all-delete-all test. In the first two tables, the notation a/b indicates that a (resp. b) seconds were taken to perform N* `insert` *(resp. N* `delete_min`*) operations. In the last table, a/b refers to the number of random resp. total I/Os executed for the N* `insert` *plus the N* `delete_min` *operations. A '-' indicates that the test was not started, $x^*$ indicates that the test was stopped after x seconds and a $*$ indicates that the test was not started due to insufficient disk space.*

4.2.2 *Results for B-Trees.* B-Trees are not developed to support effective PQ-operations. In fact, the performance of $O(\log_B N)$ I/Os leads to a large number of I/Os, large running time, and hence to an overall poor practical performance. `delete_min` might be speeded up by caching the path leading to the leftmost child. However, because of the random nature of the I/Os, the final performance is not yet comparable to any of the other external PQs.

4.2.3 *Results for R-heaps.* External radix heaps are the fastest integer-based PQ. Their simple algorithmic design allows very fast `insert` and `delete_min` operations to be supported. There is no need to maintain an internal data structure for the minimum elements as is required for array heaps and buffer trees. This obviously reduces the CPU time. If we sum the time required for `insert` and `delete_min` operations, radix heaps are about 2.5 times faster than array heaps and six times faster than buffer trees (see Table 1). They execute the smallest number of I/Os and only 15% of them are random. These random I/Os mainly occur during the `delete_min`. Unfortunately the use of radix heaps is restricted to manage integer keys and to implement monotone PQs.

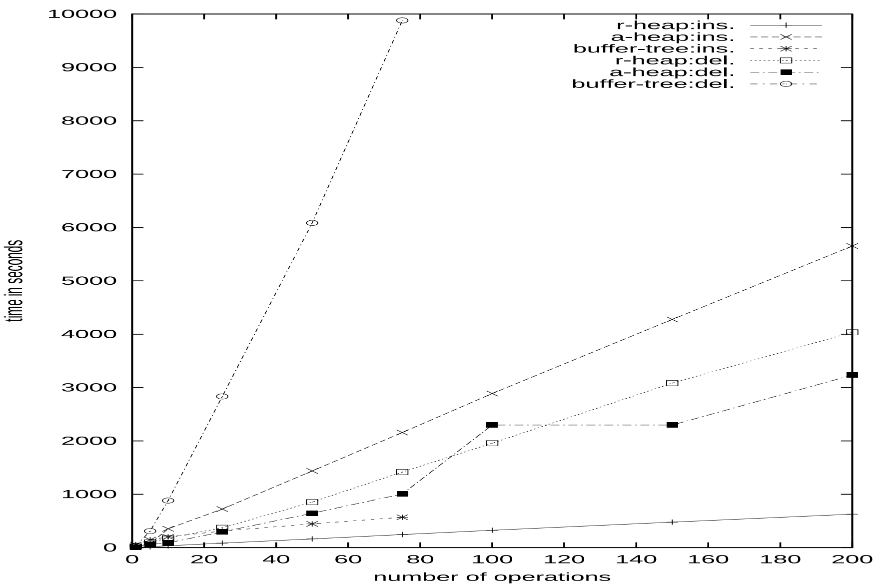


Fig. 4. Time performance of the insert-all-delete-all test for array heaps, radix heaps and buffer trees

4.2.4 *Results for Array Heaps.* The array heap obviously needs a tricky in-core data structure to differentiate between newly inserted elements (structure $H_1$ is implemented by a modified binary heap) and minimum elements coming from the external slots (structure $H_2$ is implemented by a set of arrays). This leads to a slowdown in the CPU-time for the operation `insert`, which is actually substantial. Indeed `insert` in array-heaps is up to ten times slower than `insert` in radix heaps or in buffer trees even if array-heaps may execute a smaller number of both random and total I/Os. The management of the binary heap plus the merging of the levels is more complicated than the `insert` operation for radix heaps where only the $r$-ary representation is computed and the element is inserted into its bucket.

Hence on our machine the I/O-behavior of `insert` in array-heaps is not I/O-bounded, and the CPU-cost is predominant. On the other hand, array heaps achieve a slightly better performance on the `delete_min` operation than radix heaps, and they are up to nine times faster than buffer trees. In this case, there is no costly rebalancing operation as it instead occurs in buffer trees. In conclusion, if we sum the time for both update operations, we find that array heaps are more than three times faster than buffer trees and 2.5 times slower than radix heaps. Consequently, array heaps are the fastest general-purpose priority queue among the ones we have tested. If we look at the I/O-behavior, we see that array heaps carry out only slightly more I/Os than radix heaps, and only about 2% of them are random. These random I/Os are quite evenly distributed between `insert` and `delete_min` operations.

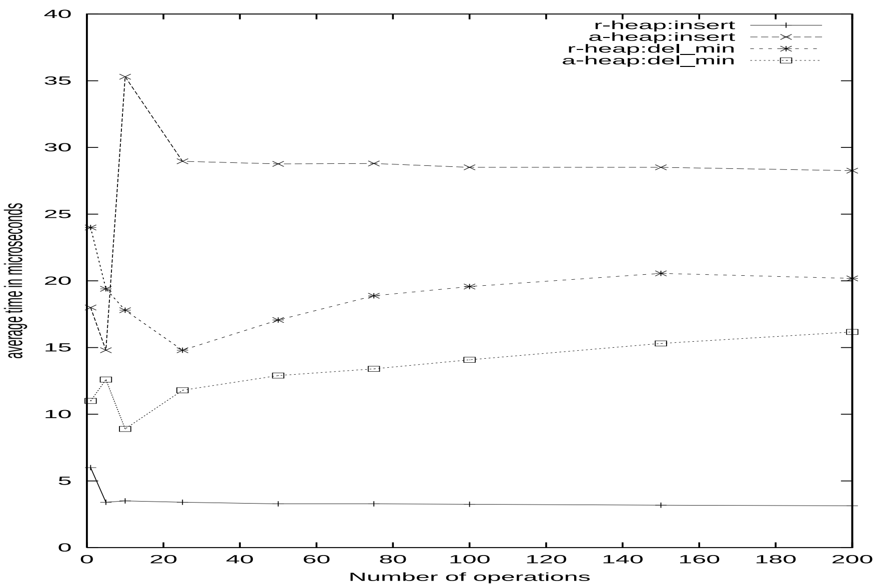


Fig. 5. Average time per operation in microseconds of the insert-all-delete-all test for array heaps, radix heaps and buffer trees.

4.2.5 *Internal-Memory Priority Queues.* Our tests considered standard PQs whose implementations are available in LEDA. All these data structures are very fast when running in internal memory, and among them the best choice is again the radix heap. However, there is a dramatic degradation of their performance when the item set is so large that it cannot be fit into the internal memory of the computer (see Table 1). In this case, the PQs make heavy use of the swap space on disk. This jump occurs between 5 and 7.5 million items for Fibonacci heaps, or between 10 and 20 million items for radix heaps, pairing heap and $k$-ary heap. The early breakdown for Fibonacci heaps is due to the fact that a single item occupies a large amount of space: about 40 bytes for a 4 byte priority value and a 4 byte information value. As we expected, none of the tested internal memory PQs is a good choice for large data sets: the heavy use of pointers causes these data structures to access the external memory in random way, so that any locality can be hardly exploited by the underlying caching/prefetching policies of the operating system.

4.2.6 *Mixed Operations.* The test on an intermixed sequence of `insert` and `delete_min` operations is used to check the speed of the internal data structures used by the best external PQs. The results of this test are summarized in Table 2. We see again that radix heaps are superior to array heaps because they do not need to manage a complicated internal data structure, which reduces CPU-time, and also they execute many fewer random (and fewer total) I/Os. This leads to a speed up of a factor 1.5. Buffer trees are more complicated and rebalancing is costly. The buffer tree is about six times slower than array heaps and seven times slower than radix heaps. It executes about nine times more I/Os and nearly all of them are random.

| Time performance on mixed operations | | | |
|---|---|---|---|
| N[*10^6] | radix heap | array heap | buffer-tree |
| 50 | 544 | 770 | 4996 |
| 75 | 609 | 945 | 5862 |
| 100 | 619 | 1027 | 6029 |
| Random/Total I/Os on mixed operations | | | |
| 50 | 2935/19615 | 22325/26997 | 153321/177201 |
| 75 | 5128/26752 | 24256/28384 | 171615/196647 |
| 100 | 5782/30094 | 24220/28380 | 171658/196578 |

Table 2. *Time and I/Os (random/total) for the mixed operation test*

| Time performance on the Dijkstra's test (graph fanout 3) | | | |
|---|---|---|---|
| N[*10^6] | radix heap | array heap | buffer tree |
| 1 | 55 | 139 | 263 |
| 3 | 208 | 375 | 1350 |
| 5 | 423 | 597 | 2480 |
| 7 | 507 | 988 | 3579 |
| 10 | 708 | 1435 | 5331 |
| 15 | 1103 | 2202 | 8415 |
| Total/Random I/Os | | | |
| 1 | 136/1232 | 32/1040 | 2973/3557 |
| 3 | 598/5110 | 136/5360 | 32174/37710 |
| 5 | 1096/9016 | 256/10000 | 6406/72733 |
| 7 | 1558/12950 | 328/14240 | 95051/107483 |
| 10 | 2246/18774 | 488/24160 | 141016/159496 |
| 15 | 3500/28868 | 832/41440 | 221567/250279 |

| Time performance on the Dijkstra's test (graph fanout 5) | | | |
|---|---|---|---|
| N[*10^6] | radix heap | array heap | buffer tree |
| 1 | 95 | 211 | 657 |
| 3 | 422 | 755 | 2598 |
| 5 | 621 | 1313 | 4709 |
| 7 | 878 | 1811 | 6958 |
| 10 | 1376 | 2642 | 10694 |
| 15 | 2030 | 4115 | * |
| Total/Random I/Os | | | |
| 1 | 328/3080 | 80/3120 | 16156/19380 |
| 3 | 1028/9716 | 240/11040 | 66599/76887 |
| 5 | 1774/16758 | 456/3760 | 124935/143575 |
| 7 | 2546/23730 | 728/36560 | 185329/214537 |
| 10 | 3574/32998 | 1128/56800 | 275748/318268 |
| 15 | 6916/58772 | 1816/90800 | */* |

Table 3. *Time and I/Os (random/total) for the Dijkstra's test applied on a random d-regular graph consisting of $N$ nodes and $dN$ edges, $d \in \{3, 5\}$. An $*$ indicates that the test was not started due to insufficient disk space.*

4.2.7 *Tests for Dijkstra's Algorithm.* We considered $d$-regular graphs with $N$ nodes and $dN$ edges, where $d \in \{3, 5, 7\}$. Edge weights are integers drawn randomly and uniformly from the interval $[1, 1000]$. Figure 6 shows the priority queue behavior of Dijkstra's algorithm on a random graph with 7 million nodes and 49 million edges ($d = 7$). The algorithm performs 96 million of priority queue operations. At the beginning, it performs more `insert` than `delete_min` operations, until the queue reaches its maximum number of elements. After that, more elements are deleted

| Time performance on the Dijkstra's test (graph fanout 7) | | | |
|---|---|---|---|
| N[$*10^6$] | radix heap | array heap | buffer tree |
| 1 | 232 | 425 | 1147 |
| 3 | 747 | 1384 | 4069 |
| 5 | 1262 | 2368 | 7332 |
| 7 | 1897 | 3309 | 11029 |
| Total/Random I/Os | | | |
| 1 | 464/4648 | 128/5040 | 25486/30510 |
| 3 | 1310/13118 | 392/19280 | 97596/112620 |
| 5 | 2210/22106 | 752/38480 | 182328/211248 |
| 7 | 3646/33334 | 1136/56880 | 267891/314395 |

Table 4. *Time and I/Os (random/total) for the Dijkstra's test applied on a random d-regular graph consisting of $N$ nodes and $7N$ edges ($d = 7$).*

than new elements come into the queue. This behavior is typical of Dijkstra's algorithm on random $d$-regular graphs (see also [Crauser et al. 1998]).

Again radix heaps execute fewer total I/Os than both the other priority queues, and result in the fastest PQ (see Tables 3 and 4). Array heaps perform the fewest random I/Os but they are two times slower than radix heaps because they perform approximately double the number of total I/Os and their `insert` operation is CPU-intensive. Buffer trees are four times slower than array heaps and eight times slower than radix heaps; they perform a huge number of I/Os (compared to the other heaps) and most of these I/Os are random. Although buffer trees have some nice applications (see e.g. [Arge 1995]), they are not designed to work well in a priority queue setting. We also see from this example that external radix heaps fit perfectly in Dijkstra's shortest path algorithm because they can profit from bounded edge weights (see also [Ahuja et al. 1990]).

## 5. CONCLUSIONS AND FURTHER RESEARCH

We have compared eight different priority queue implementations: four of them were explicitely designed for external memory whereas the others were standard internal PQs available in LEDA. As we expected, all in-core PQs failed to offer acceptable performance when managing large data sets. The fastest PQ turns out to be the radix heap: its simple structure allows effective I/O and time performance. Unfortunately, radix heaps are restricted to work on integer keys and monotone heaps. Therefore, in the general case, array heaps become the natural choice. However, in the light of their good performance, it would be interesting to re-engineer this data structure concentrating the attention on its internal part in order to speed up the operation `insert` (recall that it is CPU-bounded).

Another interesting topic of future research is to experimentally investigate the best choice for the parameters involved in the buffer-tree design (see also [Hutchinson et al. 1997]). As well, it would be interesting to investigate the applicability of the array-heap data structure to the efficient construction of *suffix arrays* [Manber and Myers 1993] on very large text collections. This is a crucial problem nowadays due to the huge increase of textual information available electronically and the poor I/O-performance of the currently known suffix-array construction algorithms. Recently in [Crauser and Ferragina 2000] some novel algorithms have been presented which could benefit the features of array heaps, i.e. management of ar-

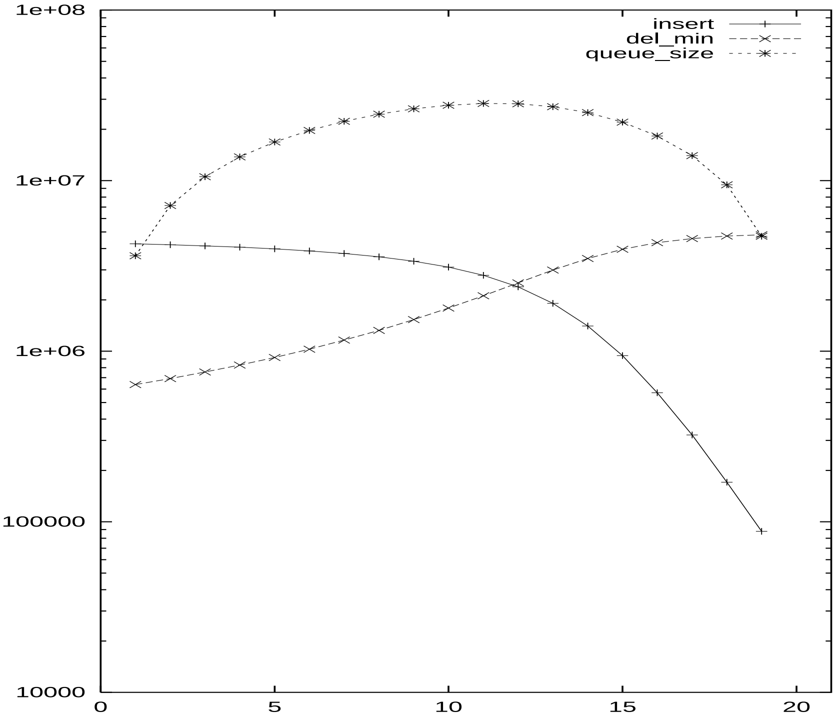


Fig. 6. *PQ behavior on a random graph with 7 million nodes and 49 million edges (d = 7). The y-axis counts the number of elements that are inserted (*`insert`*), deleted (*`del_min`*) or currently in the queue (*`queue_size`*). The x-axis counts the number of rounds, where each round consists of* 1/20*-th of the total PQ-operations.*

bitrary keys (there, the keys are tuples), reduced space occupancy and effective I/O-performance.

## REFERENCES

AHUJA, R., MEHLHORN, K., ORLIN, J., AND TARJAN, R. 1990. Faster algorithms for the shortest path problem. *Journal of the ACM 37*, 213–223.

ARGE, L. 1995. The buffer tree: A new technique for optimal I/O-algorithms. In *Proceedings of the 4th International Workshop on Algorithms and Data Structures*, Lecture Notes in Computer Science 955 (1995), pp. 334–345.

BARVE, R. D., GROVE, E. F., AND VITTER, J. S. 1997. Simple randomized mergesort on parallel disks. *Parallel Computing 23*, 4, 601–631.

BAYER, R. AND MCCREIGHT, E. 1972. Organization and maintenance of large ordered indices. *Acta Informatica 1*, 173–189.

BRODAL, G. S. AND KATAJAINEN, J. 1998. Worst-case efficient external-memory priority queues. In *Proceedings of the 6th Scandinavian Workshop on Algorithmic Theory*, Lecture Notes in Computer Science 1432 (1998), pp. 107–118.

CHERKASSKY, B. V., GOLDBERG, A. V., AND SILVERSTEIN, C. 1999. Buckets, heaps, lists, and monotone priority queues. *SIAM Journal on Computing 28*, 4, 1326–1346.

CHIANG, Y., GOODRICH, M. T., GROVE, E. F., TAMASSIA, R., VENGROFF, D. E., AND VITTER,

J. S. 1995. External-memory graph algorithms. In *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms* (1995), pp. 139–149.

CRAUSER, A. 1998. On the performance of LEDA-SM. Technical report, Max-Planck Institut für Informatik.

CRAUSER, A. AND FERRAGINA, P. 2000. A theoretical and experimental study on the construction of suffix arrays in external memory. *Algorithmica*, to appear. Also *Proceedings of the European Symposium on Algorithms, LNCS 1643, pp. 224–235.*

CRAUSER, A. AND MEHLHORN, K. 1999. LEDA-SM, extending LEDA to secondary memory. In *Proceedings of the 3rd International Workshop on Algorithmic Engineering*, Lecture Notes in Computer Science 1668 (1999), pp. 228–242.

CRAUSER, A., MEHLHORN, K., MEYER, U., AND SANDERS, P. 1998. A parallelization of dijkstra's shortest path algorithm. In *Proceedings of the 23rd International Symposium on Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science 1450 (1998), pp. 722–731.

DIJKSTRA, E. 1959. A note on two problems in connection with graphs. *Num. Math. 1*, 269–271.

FADEL, R., JAKOBSEN, K. V., KATAJAINEN, J., AND TEUHOLA, J. 1999. Heaps and heapsort on secondary storage. *Theoretical Computer Science 220*, 2, 345–362.

FARACH, M., FERRAGINA, P., AND MUTHUKRISHNAN, S. 1998. Overcoming the memory bottleneck in suffix tree construction. In *Proceedings of the 39th IEEE Symposium on Foundations of Computer Science* (1998), pp. 174–185.

FREDMAN, M. AND TARJAN, R. 1987. Fibonacci heaps and their use in improved network optimization algorithms. *Journal of the ACM 34*, 596–615.

HUTCHINSON, D., MAHESHWARI, A., SACK, J., AND VELICESCU, R. 1997. Early experiences in implementing buffer trees. In *Proceedings of the 2nd International Workshop on Algorithmic Engineering* (1997), pp. 92–103.

IEEE. 1987. IEEE standard 754-1985 for binary floating-point arithmetic. reprinted in SIGPLAN 22.

KUMAR, V. AND SCHWABE, E. J. 1996. Improved algorithms and data structures for solving graph problems in external memory. In *Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing* (1996), pp. 169–177.

LAMARCA, A. AND LADNER, R. 1997. The influence of caches on the performance of heaps. *ACM Journal of Experimental Algorithmics 1*, article 4.

MANBER, U. AND MYERS, G. 1993. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing 22*, 5, 935–948.

MEHLHORN, K. AND NÄHER, S. 1999. *LEDA: A Platform of Combinatorial and Geometric Computing.* Cambridge University Press, Cambridge, England.

NAOR, D., MARTEL, C., AND MATLOFF, N. 1991. Performance of priority queues in a virtual memory environment. *The Computer Journal 34*, 5, 428–437.

RUEMMLER, C. AND WILKES, J. 1994. An introduction to disk drive modeling. *IEEE Computer 27*, 3, 17–29.

SHRIVER, E. AND VITTER, J. 1994. Optimal algorithms for parallel memory i: Two-level memories. *Algorithmica 12*, 2-3, 110–147.

THORUP, M. 1996. On RAM priority queues. In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms* (1996), pp. 59–67.

VENGROFF, D. E. AND VITTER, J. S. 1995. Supporting I/O-efficient scientific computation in TPIE. In *Proceedings of the IEEE Symposium on Parallel and Distributed Processing* (1995), pp. 74–77.

VITTER, J. 1998. External memory algorithms. In *ACM Symposium on Principles of Database Systems* (1998). Also invited paper in European Symposium on Algorithms, 1998.

WILLIAM, J. 1964. Algoritm 232 (heapsort). *Communication of the ACM 7*, 347–348.