The Complexity of Implicit and Space Efficient Priority Queues

Christian W. Mortensen* and Seth Pettie**

[1] IT University of Copenhagen
cworm@itu.dk
[2] Max Planck Institut für Informatik
pettie@mpi-sb.mpg.de

Abstract. In this paper we study the time-space complexity of implicit priority queues supporting the *decreasekey* operation. Our first result is that by using *one* extra word of storage it is possible to match the performance of Fibonacci heaps: constant amortized time for insert and decreasekey and logarithmic time for deletemin. Our second result is a lower bound showing that that one extra word really is necessary. We reduce the decreasekey operation to a cell-probe type game called the *Usher's Problem*, where one must maintain a simple data structure without the aid of any auxiliary storage.

1 Introduction

An *implicit* data structure on N elements is one whose representation consists simply of an array $A[0..N-1]$, with one element stored in each array location. The most well known implicit structure is certainly Williams's binary heap [26], which supports the priority queue operations *insert* and *delete-min* in logarithmic time. Although the elements of Williams's heap are conceptually arranged in a fragment of the infinite binary tree, the tree edges are not explicitly represented. It is understood that the element at $A[i]$ is the parent of $A[2i+1]$ and $A[2i+2]$. The practical significance of implicit data structures is that they are, in certain circumstances, maximally space efficient. If the elements can be considered *atomic* then there is no better representation than a packed array.

A natural suspicion is that by insisting on an implicit representation one may be sacrificing asymptotic time optimality. After 40 years of sporadic research on implicit structures [26, 17, 21, 27, 20, 5, 8, 7, 28, 9, 10, 15] we can say that this suspicion is almost completely misguided. In various dictionary & priority queue problems, for instance, there are either optimal implicit structures or ones that can be made optimal with *a couple extra words* of storage.

* This work was performed while visiting the Max-Planck-Institut für Informatik as a Marie Curie Doctoral Fellow.
** Supported by an Alexander von Humboldt Postdoctoral Fellowship.

In this paper we study the complexity of implicit priority queues that support the decrease-key operation. Our main positive result is that given *one extra word of storage* there is an implicit priority queue matching the performance of Fibonacci heaps. It supports delete-min in logarithmic time and insert and decrease-key in constant time, all amortized. That one extra word obviously has no practical consequences but it is a thorn in our side. We propose a variation on our data structure that uses no additional storage and supports decrease-key in $O(\log^* n)$ time, without affecting the other operations. This $O(\log^* n)$ bound is a theoretical burden. Is it natural? We prove that it is, in the following sense: if any implicit priority queue uses zero extra space and supports decrease-key in $o(\log^* n)$ amortized time then the amortized cost of insert/delete-min jumps dramatically, from logarithmic to $\Omega(n^{1/\log^{(k)} n})$, for any k.

We reduce the decrease-key operation to the *Absent Minded Usher's Problem*, a game played in a simplified cell-probe model. Imagine an usher seating indistinguishable theater patrons one-by-one in a large theater. The usher is equipped with two operations: he can *probe* a seat to see if it is occupied or not and he can *move* a given patron to a given unoccupied seat. (Moving patrons after seating them is perfectly acceptable.) The catch is this: before seating each new patron we wipe clean the usher's memory. That is, he must proceed without any knowledge of which seats are occupied or the number of patrons already seated. We prove that any deterministic ushering algorithm must seat m patrons with $\Omega(m \log^* m)$ probes and moves, and that this bound is asymptotically tight.

Our lower bound proof attacks a subtle but fundamental difficulty in implicit data structuring, namely, orchestrating the movement of elements within the array, given little auxiliary storage. In its present form the ushering problem is limited to proving small time-space tradeoffs. However it is likely that generalizations of the ushering method could yield more impressive lower bounds.

Organization. In the remainder of this section we define what an implicit priority queue is, survey previous work and discuss our contributions. In Section 2 we present our new data structure. Section 3 is devoted to the Usher's Problem and its relationship with implicit priority queues.

Implicit Priority Queues. We first give a specification for an abstract implicit priority queue which is suitable for theoretical analysis but impractical. We then propose a particularly space efficient method for implementing such a data structure.

An implicit priority queue of size n consists of an array $A[0..n-1]$ (plus, possibly, a little extra storage) where $A[0], \ldots, A[n-1]$ contain distinct elements (or *keys*) from a total order. We also use A to denote the set of elements in the priority queue. The following operations are supported.

insert(κ)	: $A := A \cup \{\kappa\}$
deletemin$()$	: Return $\min A$ and set $A := A \backslash \{\min A\}$
decreasekey(i, κ)	: Set $A[i] := \min\{A[i], \kappa\}$

An operation decides what to do based on the auxiliary information and any comparisons it makes between elements. Before returning it is free to alter the auxiliary information and permute the contents of A, so long as its n elements lie in $A[0, \ldots, n-1]$. We assume that "n" is known to all operations.

The definition above sweeps under the rug a few issues that are crucial to an efficient and useful implementation. First, applications of priority queues store not only elements from a total order but *objects* associated with those elements. For example, Dijkstra's shortest path algorithm repeatedly asks for the *vertex* with minimum tentative distance; the tentative distance alone is useless. Any practical definition of a priority queue must take the application's objects into account. The second issue relates to the peculiar arguments to decreasekey. The application must tell decreasekey the *index* in A of the element to be decreased, which is necessarily a moving target since the data structure can permute the elements of A at will.

We propose a priority queue interface below that addresses these and other issues. Let us first sketch the normal (real world) interaction between application and priority queue. To insert the object v the application passes the priority queue an identifier $\mathit{id}(v)$ of its choosing, together with $\mathit{key}(v)$, drawn from some total order. In return the priority queue gives the application a $\mathit{pq\_id}(v)$, also of its choosing, which is used to identify v in later calls to decreasekey. When v is removed from the queue, due to a deletemin, the application receives both $\mathit{id}(v)$ and $\mathit{key}(v)$.

In our interface we give the data structure an extra degree of freedom, without placing any unreasonable demands on the governing application. Whereas the standard interface forces the data structure to assign *pq_id*s once and for all, we let the data structure update *pq_id*s as necessary. We also let the application maintain control of the keys. This is for two reasons, both concerning space. First, the application may not want to explicitly represent keys at all if they can be deduced in constant time. Second, the application can now use the same key in multiple data structures without producing a copy for each one. Below $\mathcal{Q}$ represents the contents of the data structure, which is initially empty. (Observe that this interface is modular. Neither the application nor the data structure needs to know any details about the other.)

The priority queue implements:

insert$(\mathit{id}(v))$	: Sets $\mathcal{Q} := \mathcal{Q} \cup \{\mathit{id}(v)\}$ and returns a $\mathit{pq\_id}(v)$
deletemin$()$	: Return, and remove from $\mathcal{Q}$, the $\mathit{id}(v)$ minimizing $\mathit{key}(v)$
decreasekey$(\mathit{pq\_id}(v))$	: A notification that $\mathit{key}(v)$ has been reduced

The application implements:

update$(\mathit{id}(v), x)$	: Set $\mathit{pq\_id}(v) := x$
compare$(\mathit{id}(v), \mathit{id}(w))$	: True iff $\mathit{key}(v) < \mathit{key}(w)$

Using this interface it is simple to implement an abstract implicit priority queue. The data structure would consist of an array of *id*s and we maintain, with appropriate update operations, that $\mathit{pq\_id}(v)$ indexes the position of $\mathit{id}(v)$ in A. For example, if we implemented a d-ary heap [17] with this interface every priority queue operation would end with at most $\log_d n$ calls to update, which is the maximum number of elements that need to be permuted in A.

Our interface should be contrasted with a more general solution formalized by Hagerup and Raman [14], in which *pq_id*s would be fixed once and for all.

In their schema the application communicates with the data structure through an intermediary *quasidictionary*, which maps each *pq_id* (issued by the quasidictionary) to an identifier that can be updated by the data structure. Since saving space is one of the main motivations for studying implicit data structures we prefer our interface to one based on a quasidictionary.

Defining "Extra Storage." All the priority queues cited in this paper store n *id*s and n *key*s, and if decreasekey is supported, n *pq_id*s as well. We consider any further storage *extra*. For the sake of simplicity we ignore temporary space used by the individual operations and any overhead involved in memory allocation. Here "memory allocation" means simulating the array A, whose length varies as elements are inserted and deleted. Brodnik et al. [3] proved that the standard solution—array doubling/halving—can be improved so that only $\Theta(\sqrt{n})$ extra words of space are used, where n is the current size of the array. In pointer-based structures (like Fibonacci heaps) the cost of memory allocation is more severe. There is a measurable overhead for each allocated block of memory.

Previous work. Much of the work on implicit data structures has focussed on the dictionary problem, in all its variations. The study of dynamic implicit dictionaries in the comparison model was initiated by Munro & Suwanda [21]. They gave a specific partial order (à la Williams's binary heap) that allowed inserts, deletes, and searches in $O(\sqrt{n})$ time, and showed, moreover, that with any partial order $\Omega(\sqrt{n})$ time is necessary for some operation. Munro [20] introduced a novel pointer encoding technique and showed that all dictionary operations could be performed in $O(\log^2 n)$ time, a bound that stood until just a few years ago. After a series of results Franceschini & Grossi [9] recently proved that all dictionary operations could be performed in worst-case logarithmic time with no extra storage. Franceschini & Grossi [10] also considered the static dictionary problem, where the keys consist of a vector of k characters. Their implicit representation allows for searches in optimal $O(k + \log n)$ time, which is somewhat surprising because Andersson et al. [1] already proved that optimal search is impossible if the keys are arranged in *sorted order.*

Yao [27] considered a static version of the dictionary problem where the keys are integers in the range $\{1, \ldots, m\}$. He proved that if m is sufficiently large relative to n then no implicit representation can support $o(\log n)$ time queries. In the same paper Yao proved that *one-probe* queries are possible if $m \leq 2n-2$. Fiat et al. [8, 7] gave an implicit structure supporting constant time queries for any universe size, provided only $O(\log n + \log\log m)$ bits of extra storage. Zuckerman [28], improving a result of [8, 7], showed that $O(1)$ time queries are possible with zero extra storage, for m slightly superpolynomial in n. In the integer-key model we need to qualify the term "extra storage." An implicit representation occupies $n\lceil\log m\rceil$ bits, which is roughly $n\log n$ more than the information bound of $I = \lceil\log\binom{m}{n}\rceil$ bits. See [4, 22, 23, 24] for $I + o(I)$ space dictionaries.

Implicit Priority Queues. Williams's binary heap [26] uses zero extra storage and supports inserts and deletemins in worst-case logarithmic time. It was generalized by Johnson [17] to a d-ary heap, which, for any fixed d, supports inserts and decreasekeys in $O(\log_d n)$ time and deletemins in $O(d \log_d n)$ time. Carlsson et al.'s implicit binomial heap [5] supports inserts in constant time and deletemins in logarithmic time, both worst case. Their data structure uses $O(1)$ extra words of storage. Harvey and Zatloukal's postorder heap [15] also supports insert in $O(1)$ time; the time bound is amortized but they use zero extra storage. Their data structure can be generalized to a postorder d-ary heap.

General Priority Queues. The d-ary implicit heap has an undesirable tradeoff between decreasekeys and deletemins. In many applications of priority queues the overall performance depends crucially on a fast decreasekey operation. Fredman and Tarjan's Fibonacci heap [13] supports all operations in optimal amortized time: $O(\log n)$ for deletemin and $O(1)$ for the rest. Aside from the space taken for *keys*, *ids*, and *pq_ids*, Fibonacci heaps require $4n$ pointers and $n(\log\log n + 2)$ bits. Each node keeps a $(\log\log n + 1)$-bit rank, a mark bit, and references its parent, child, left sibling, and right sibling. Kaplan and Tarjan [18] shaved the space requirements of Fibonacci heaps by n pointers and n bits. The Pairing heap [12] can be represented with $2n$ pointers though it does not handle decreasekeys in constant time [11].

Our Contributions. In this paper we show that it is possible to match the performance of Fibonacci heaps using *one* extra word of storage, and that no deterministic implicit priority queue can achieve the same bounds without one extra word; see Figure 1. Our data structure may be of separate interest because it uses a completely new method for supporting decreasekeys in constant amortized time. Whereas Fibonacci-type heaps [13, 6, 18, 25] organize the elements in heap-ordered trees and link trees based on their *ranks*, our priority queue is conceptually composed of a set of unordered lists, whose elements adhere to a particular partial order. We do not tag elements with ranks. The primitives of our data structure are simply the concatenation of lists and the division of lists, using any linear-time selection algorithm [2].

	Decreasekey	Deletemin	Extra Storage	Ref
Fibonacci	$O(1)$	$O(\log n)$	$4n$ ptrs, $n(\log\log n + 2)$ bits	[13]
Thin	$O(1)$	$O(\log n)$	$3n$ ptrs, $n(\log\log n + 1)$ bits	[18]
Pairing	$\Omega(\log\log n)$	$O(\log n)$	$2n$ ptrs, n bits	[11]
	$O(\log n)$			[16]
Post. d-ary	$O(\log_d n)$	$O(d \log_d n)$	zero	[15]
New	$O(1)$	$O(\log n)$	1 ptr	
New	$O(\log^* n)$	$O(\log n)$	zero	
New l.b.	if $o(\log^* n)$	$\Omega(n^{1/\log^{(k)} n})$	zero	

Fig. 1. All priority queues support inserts in amortized constant time. The Fibonacci & Thin Heaps support amortized constant time melds. See [19] for results on worst-case bounds

Our lower bounds reinforce a theme in implicit data structures [27, 5, 8, 15], that it takes only a couple extra words of storage to alter the complexity of a problem. Although our results depend on small amounts of extra memory, the *ushering* technique is new and abstract enough to be applied elsewhere.

2 A New Priority Queue

In this section we design an *abstract* implicit priority queue. We cover the high-level invariants of the structure, some very low-level encoding issues, then sketch the operational "flow" of the priority queue. Many details are omitted; see [19].

Encoding Bits. We encode bits using the standard technique: for two given indices i, j we equate $A[i] < A[j]$ with zero and $A[j] < A[i]$ with one. In this way we can encode small integers and pointers with $O(\log n)$ elements.

Junk Elements. All elements are tagged either *normal* or *junk*, where the tags are represented implicitly. We divide the entire array A into consecutive *triplets*. The first two elements of any triplet are junk, and their relative order encodes whether the third element is junk or normal. A junk (normal) triplet is one whose third element is junk (normal). We maintain the invariant that the minimum element in the queue is normal.

L-lists and I-lists. At a high level the data structure consists of a sequence of $O(\log n)$ L-lists and a set of $O(\log^2 n)$ I-lists, each of which is associated with a distinct interval of L-lists. For any list T we let $T^\star$ denote the set of normal elements in T and $|T|$ its length in triplets, including junk triplets. The relation $S < T$ holds when $\max S^\star < \min T^\star$, or if either $S^\star$ or $T^\star$ is empty.

The list L_{ij} belongs in slot j of zone i, where $i \in [0, \log_4 n]$, $j \in [0, 6]$, and the length of L_{ij} is roughly exponential in i. The L-lists are internally unordered but, as a whole, in sorted order. That is, if $ij < kl$ (lexicographically) then $L_{ij} < L_{kl}$. Loosely speaking, the list $I_{ij,kl}$ contains elements that, *were* they to be included in L-lists, could be assigned to some list in the interval $L_{ij}, \ldots, L_{kl}$. We let $L_{s(ij)}$ and $L_{p(ij)}$ be the non-empty successor and predecessor of list L_{ij}, respectively. The L- and I-lists obey the following *order* and *size* invariants. Some invariants refer to parameters $N \geq n$, $\omega = \lceil \log N \log\log N \rceil$ and $\gamma = \log^4 N$. In addition to L- and I-lists there is a *buffer* B which is discussed later.

Inv. O1 If $ij < kl$ then $L_{ij} < L_{kl}$.
Inv. O2 If $ij \leq kl$ then $L_{p(ij)} < I_{ij,kl} < L_{s(kl)}$.
Inv. S1 $|L_{ij}| \in [\gamma 4^i, 2\gamma 4^i]$ and $|L_{ij}|$ is a multiple of ω. $|L_{ij}|$ is non-empty only if $L_{i0}, \ldots, L_{i(j-1)}$ are also non-empty, i.e., L-lists are packed in each zone.
Inv. S2 For $ij > 00$, $|L^\star_{ij}| \geq \frac{1}{2}|L_{ij}|$.
Inv. S3 For any $ij \leq kl$, $|I_{ij,kl}|$ is a multiple of ω and $I_{ij,kl}$ is non-empty only if L_{ij} and L_{kl} are also non empty. $I_{00,kl}$ is empty for all kl.
Inv. S4 $|B| = |L_{00}| = 2\gamma$, and $|L^\star_{00}| \geq 1$.

Assuming that the minimum element is normal, it follows from Invariants O1, O2, S3, and S4 that the minimum always lies in L_{00} or B. In our data structure

L_{00} and B are treated as fixed size mini priority queues that support findmin in constant time, deletemin in $O(\log n)$ time, and decreasekey in either $O(1)$ or $O(\log^* n)$ time, depending on whether we are allowed to store one extra pointer.

Periodic Rebuilding. We use a few mechanisms to guarantee that $\min A$ is normal. First, any element that is inserted or decreasekey'd is made normal. Every insertion takes two otherwise unused junk elements to form a normal triplet and every decreasekey can require up to three unused junk elements. Our data structure rebuilds itself whenever there may be a shortage of unused junk elements. First it finds the $n/6$ smallest elements, designates them normal, and assigns them to triplets. The remaining $n/2$ junk elements constitute the *junk reservoir.* We maintain an implicit operation counter that is initially set to $n/6$. Every operation decrements the counter and when it reaches zero (i.e, when the junk reservoir may be empty) the data structure is rebuilt in linear time. It follows that the minimum element is always normal. We charge the cost of rebuilding the structure to the $n/6$ preceeding operations. Some parameters are w.r.t. an $N \geq n$. Upon rebuilding we let $N = 2^{\lceil \log(7n/6) \rceil}$. Since n (the size of the queue) does not uniquely determine N we dedicate two junk triplets to indicate the correct value.

Block Structure. The entire array A is divided into a sequence of *blocks*, each containing $\omega = \lceil \log N \log\log N \rceil$ triplets. The primary purpose of blocks is to allow a dynamic implicit representation of L- and I-lists, which are circular and doubly-linked. We dedicate $O(\log N)$ triplets of each block to represent successor and predecessor pointers. Thus, given two blocks in different L- or I-lists it is possible to splice them together in $O(\log N)$ time by altering 4 pointers. Blocks also contain other counters and pointers; see [19].

The Structure of B and L_{00}. Recall that both B and L_{00} have fixed size and are located in A at fixed locations depending on N. We keep the normal triplets of L_{00} packed together and arranged in a $(\log N)$-ary implicit heap. Thus L_{00} supports deletemins in $O(\log N \log_{\log N} \omega) = O(\log N)$ time and decreasekeys on elements in $L_{00}^\star$ in $O(\log_{\log N} \omega) = O(1)$ time. It allows bulk insertion of k elements in $O(k + \log \gamma)$ time, where the $\log \gamma = O(\log\log N)$ term is the time to determine $|L_{00}^\star|$. When dealing with B and L_{00} a deletemin accepts a junk element to plug up the hole and an insert returns a displaced junk element. The buffer B consists of, in this order, a triplet containing $\min B^\star$ followed by fixed size mini-buffers $B^{\log^* N}, \ldots, B^1$, where $\left|B^1\right| = \Theta(\gamma) = \Theta(\log^4 N)$ and $\left|B^\ell\right| = \Theta(\log^{(\ell)} N)$. The normal triplets of each mini-buffer are packed together, and furthermore, the normal triplets of B^1 are arranged in a $(\log N)$-ary implicit heap. To insert the element $e \notin B$ into B we tag it normal, if not already. We identify the first junk triplet of $B^{\log^* N}$ and swap e with the third element of this triplet. If $e = \min B$ we swap it with the old minimum. At this point $B^{\log^* N}$ may be full, that is, it contains only normal triplets. In general, whenever $B^{\ell+1}$ is full we perform a binary search to determine the first junk triplet in B^ℓ, say $B^\ell[j]$. We then swap the whole mini-buffer $B^{\ell+1}$ with the junk triplets in $B^\ell[j..j + \left|B^{\ell+1}\right| - 1]$, which may cause B^ℓ to be full. If $\ell = 1$ an artificial decreasekey operation is performed on each of these elements in order to restore the heap order of B^1. The cost of relieving an overflow of $B^{\ell+1}$ is $O(\left|B^{\ell+1}\right| + \log\left|B^\ell\right|) = O(\left|B^{\ell+1}\right|)$, i.e., $O(1)$ per element. Since an element appears in each buffer at most once (per insertion) the amortized cost of an insert is $O(\log^* N)$. See [19] for B's deletemin and decreasekey routines.

If B can store an extra pointer (outside of the array A) then it can support inserts and decreasekeys in $O(1)$ time. We get rid of $B^{\log^* N}, \ldots, B^2$ and let the pointer index the first junk triplet in B^1.

Memory Layout. The array A is partitioned into five parts. The first part consists entirely of junk and the first three occupy fixed locations depending on N.

1. Preamble. Contains a flag indicating N, the operation counter, and the free block pointer, which points to the location of item (5), below.
2. The Buffer B and the list L_{00}.
3. Representative blocks: one block from each L- and I-list is kept in a fixed location. If the list is empty an all-junk block is kept in its spot. The representative blocks contain additional statistics about their respective lists.
4. An array of blocks in use, by either L- or I-lists.
5. An array of unused blocks, a.k.a. the junk reservoir.

High-Level Operations. The priority queue operations only deal with L_{00} and B. However, such an operation can induce a number of low level operations if one of the invariants is violated, for instance, if $|L^\star_{00}|$ reaches zero or $|B| > 2\gamma$. The asterixes below mark places where a sequence of low level operations may be necessary. Rebuilding the structure is considered a low level operation.

Insert(κ)**:** Insert* κ into B. Put the displaced junk element at the end of the junk reservoir. Decrement* the global operation counter.

Decreasekey(i, κ)**:** If $A[i]$ lies in B or $L^\star_{00}$ then perform the decreasekey there. Otherwise insert* the element $A[i]$ into B with the new key κ, changing its status to normal if it was junk. Put the displaced junk element at $A[i]$. Decrement* the global operation counter.

Deletemin()**:** Return $\min(L^\star_{00} \cup \min B^\star)$ using the deletemin operation* provided either by B or L_{00}. Use the last junk element in the junk reservoir to plug up the hole. Decrement* the global operation counter.

A Sketch of the Rest. Ignoring other parts of the data structure, the behavior of our L-lists is straightforward. Whenever $L^\star_{00}$ becomes empty we find its successor $L' = L_{s(00)}$. If L' is in zone 0 we simply rename it L_{00} and if L' is in zone $i > 0$ we *divide* it, with a linear-time selection algorithm, into $O(i)$ shorter lists, which are distributed over zones 0 through $i - 1$. Similarly, when $L^\star_{00}$ becomes full we divide it into smaller lists, which are inserted into zone 0. In general, whenever a zone i contains more than 7 lists (a violation) we concatenate some of the lists to form one whose size is suitable to be inserted into zone $i + 1$. Our main difficulty is filing newly inserted/decreasekey'd elements into the correct L-list. Any direct method, like performing a binary search over the L-lists, is doomed to take $\Omega(\log \log n)$ time. The purpose of the I-lists is to direct unfiled elements to their correct L-lists, at constant amortized time per element.

Inserts and decreasekeys are directed toward the buffer B. When B is full we divide it into geometrically increasing sets $J_0 < J_1 < J_2 \ldots$, with $|J_{\ell+1}| = 2|J_\ell|$.

Each set J_ℓ is concatenated with the narrowest possible I-list that satisfies Invariant O2. That is, we pick $O(\log n)$ normal representatives $e_{ij} \in L^\star_{ij}$ and concatenate J_ℓ with $I_{ij,kl}$ such that $e_{ij} < J_\ell < e_{kl}$, where ij is maximal and kl minimal. The procedure for dividing I-lists is identical to dividing the buffer. (We basically consider B to be a special I-list.) In particular, whenever $I_{ij,kl}$ *might* contain the minimum element—if L_{ij} is the first non-empty L-list—we divide it into J-sets of geometrically increasing size and concatenate them with the proper I-lists. It is not obvious why this method should work. Every time an I-list is processed its elements have a good chance of ending up in another I-list, which presumably covers a narrower interval of L-lists. That is, it looks like we're just implementing binary search in slow motion. The full analysis of our data structure relies on a complicated potential function; see [19] for the details.

3 The Absent Minded Usher's Problem

In [19] we show that in any implicit priority queue the decreasekey operation must be prepared to solve a version of the Absent Minded Usher's Problem, which is the focus of this section. Refer to [19] for the complete lower bound proof.

Let A be an array of infinite length, where each location of A can contain a *patron* (indistinguishable from other patrons) or be empty. An *usher* is a deterministic program to insert a new patron into A without any knowledge of its contents. That is, the usher does not know how many patrons it has already inserted. The usher can probe a location of A to see if it is occupied and move any patron to an empty location of A. We are interested in the complexity of the best ushering program, that is, the number of probes and moves needed to seat N patrons in an initially empty array. There exists a simple $O(N \log^* N)$ time usher; it is based on the same *cascading buffers* technique used in B from the previous section. We prove that any usher requires $\Omega(N \log^* N)$ time.

We imagine an infinite graph whose vertices are layed out in a grid. The x-axis corresponds to time (number of insertions) and the y-axis corresponds to the array A. An usher is partially modeled as a set of x-monotone paths through the grid, with each path representing where a particular patron was at each moment in time. We assign each edge a cost, which represents in an amortized sense the time taken to bring that patron to its current position. By reasoning about the usher's decision tree we are able to derive a recurrence relation describing the costs of edges. The solution to this recurrence is then used to lower bound the complexity of the ushering problem.

We put the patron to be inserted in the artificial position $A[-1]$. The usher's algorithm is modeled as a binary decision tree. At each internal node is an array position to be probed and at each leaf is a list of pairs of the form (j_1, j_2), indicating that the patron at $A[j_1]$ should be moved to $A[j_2]$. Each leaf is called an *operation* and the cost of executing an operation o is its depth in the decision tree, $d(o)$, plus the number of patron moves, $m(o)$.

Consider an infinite graph with vertex set $\{A_i[j] : i \geq 0, j \geq -1\}$, where $A_i[j]$ represents $A[j]$ *after* i insertions. There exists an edge $(A_{i-1}[j_1], A_i[j_2])$ exactly

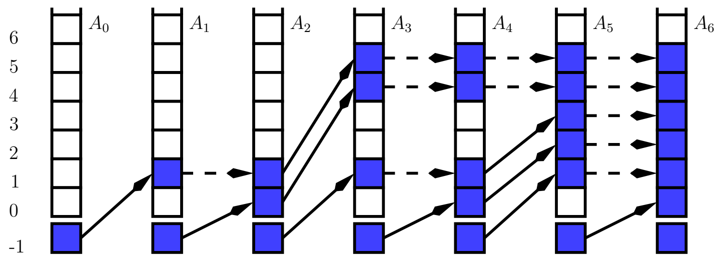

Fig. 2. This (infinite) two dimensional grid depicts the flow of patrons over time for a particular ushering algorithm. Probes made by the algorithm are not represented. Solid edges are fresh, dashed ones leftover

when $A_{i-1}[j_1]$ and $A_i[j_2]$ contain the same patron. Note that the graph is composed exclusively of paths. An edge is *leftover* if it is of the form $(A_{i-1}[j], A_i[j])$ and *fresh* otherwise, i.e., fresh edges correspond to patron movements and leftover edges correspond to unmoved patrons; see Figure 2. Let $\text{pred}(A_{i-1}[j_1], A_i[j_2])$ denote the edge $(A_{i-2}[j_3], A_{i-1}[j_1])$ if such an edge exists.

We now define a cost function c. If the edge (u,v) does not exist then $c(u,v) = 0$. Any edge of the form $(A_{i-1}[j], A_i[j])$ has cost $c(\text{pred}(A_{i-1}[j], A_i[j]))$: leftover edges inherit the cost of their predecessor. Let $C_i = \sum_{j_1,j_2} c(A_{i-1}[j_1], A_i[j_2])$ be the total cost of edges into A_i, and let $P_i = \sum_{j_1 \neq j_2} c(\text{pred}(A_{i-1}[j_1], A_i[j_2]))$ be the cost of the *predecessors* of the fresh edges into A_i. Let o_i be the operation performed at the ith insertion. (For example, in the ushering algorithm partially depicted in Figure 2, $o_2 = o_4 = o_6$ and o_1, o_3, and o_5 are distinct.) Each fresh edge e between A_{i-1} and A_i is assigned the same cost:

$$c(e) \overset{\mathbf{def}}{=} \frac{P_i + d(o_i) + m(o_i)}{m(o_i)}$$

That is, the total cost assigned to these $m(o_i)$ fresh edges is their inherited cost, P_i, plus the actual time of the operation o_i: $d(o_i) + m(o_i)$. It follows from the definitions that C_i is exactly the time to insert i patrons. Let $T(m)$ be the *minimum* possible cost of a fresh edge associated with an operation performing m movements. We will show $T(m) = \Omega(\log^* m)$ and that this implies the amortized cost of N insertions is $\Omega(N \log^* N)$.

For the remainder of this section we consider the ith insertion. Suppose that the operation o_i moves patrons *from* locations $A_{i-1}[j_1, j_2, \ldots, j_{m(o_i)}]$. The patrons in these locations were placed there at various times in the past. Define $i_q < i$ as the index of the insertion that *last* moved a patron to $A[j_q]$.

Lemma 1. *Let p, q be indices between 1 and $m(o_i)$. If $i_p \neq i_q$ then $o_{i_p} \neq o_{i_q}$.*

We categorize all operations in the patron's decision tree w.r.t. $m = m(o_i)$—recall that i is fixed in this section. An operation o is *shallow* if $d(o) < \lfloor \log m \rfloor / 2$ and *deep* otherwise. It is *thin* if $m(o) < \left\lfloor \sqrt{\log m} \right\rfloor$ and *fat* otherwise.

Lemma 2. $\left|\{q : o_{i_q} \textit{ is shallow and thin}\}\right| < \frac{1}{2}\sqrt{m \log m}$

Lemma 3. *If (u, v) is a fresh edge, where $v \in A_k$ and o_k is deep and thin, then $c(u, v) \geq \frac{1}{2}\sqrt{\log m}$.*

In summary, Lemma 2 implies that at least $m - \sqrt{m \log m}/2$ of the fresh edges between A_{i-1} and A_i can be traced back to earlier edges that are either in deep and thin operations or fat operations. The cost of edges in deep and thin operations is bounded by Lemma 3 and the cost of edges in fat operations is bounded inductively. Recall that $T(m)$ is the minimum cost of a fresh edge associated with an operation performing m moves.

Lemma 4. $T(m) \geq (\log^* m)/4$

Proof. Let e be a fresh edge into A_i with $m = m(o_i)$ and let $\beta = \min\{\frac{1}{2}\sqrt{\log m}, T(\sqrt{\log m}), T(\sqrt{\log m}+1), \ldots\}$. Then:

$$c(e) \; = \; \frac{m(o_i) + P_i + d(o_i)}{m(o_i)} \; \geq \; \frac{m + (m - \frac{1}{2}\sqrt{m \log m}) \cdot \beta}{m}$$

Since the only property of e that we used in the above inequalities is $m(o_i) = m$, any lower bound on $c(e)$ implies the same lower bound on $T(m)$. We assume inductively that $T(r) \geq \frac{1}{4}\log^* r$, which holds for $r \leq 2^{16}$ since $T(r) \geq 1$ and $\log^* 2^{16} = 4$. For $m > 2^{16}$ we have:

$$\begin{aligned} T(m) &\geq 1 + \left(1 - \sqrt{\log m/4m}\right) \cdot \beta \; \geq \; 1 + \left(1 - \sqrt{\log m/4m}\right) \cdot \frac{\log^*(\sqrt{\log m})}{4} \\ &> \left(1 - \sqrt{\log m/4m}\right)\left(\frac{\log^* m - 2}{4} + 1\right) \; > \; \frac{\log^* m}{4} \end{aligned}$$

Theorem 1. *Any usher seating N patrons must perform $\Omega(N \log^* N)$ operations. For some patron it must perform $\Omega(\log N)$ operations.*

See [19] for the proof of Theorem 1. Our lower bound on implicit priority queues shows that the decreasekey operation can be forced to behave like an usher, seating m patrons for *some* m of its choosing. If the data structure has no extra storage and if $m > \log^{(k)} n$ (for some fixed k), then by Theorem 1 the amortized cost per decreasekey is $\Omega(\log^*(\log^{(k)} n)) = \Omega(\log^* n - k)$. If m is smaller, i.e., decreasekeys were performed quickley, then we show that a further sequence of $O(m \log^* n)$ decreasekeys, inserts, and deletemins must take $\Omega(n^{1/\log^{(k)} n})$ time.

Acknowledgment. We thank Rasmus Pagh for many helpful comments.

References

1. A. Andersson, T. Hagerup, J. Hastad, and O. Petersson. Tight bounds for searching a sorted array of strings. *SIAM J. Comput.*, 30(5):1552–1578, 2000.
2. M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *J. Comput. Syst. Sci.*, 7(4):448–461, 1973.

3. A. Brodnik, S. Carlsson, E. Demaine, J. I. Munro, and R. Sedgewick. Resizable arrays in optimal time and space. In *WADS*, pages 37–??, 1999.
4. A. Brodnik and J. I. Munro. Membership in constant time and almost-minimum space. *SIAM J. Comput.*, 28(5):1627–1640, 1999.
5. S. Carlsson, J. I. Munro, and P. V. Poblete. An implicit binomial queue with constant insertion time. In *SWAT*, pages 1–13, 1988.
6. J. R. Driscoll, H. N. Gabow, R. Shrairman, and R. E. Tarjan. Relaxed heaps: an alternative to Fibonacci heaps with applications to parallel computation. *Comm. ACM*, 31(11):1343–1354, 1988.
7. A. Fiat and M. Naor. Implicit $O(1)$ probe search. *SIAM J. Comput.*, 22(1):1–10, 1993.
8. A. Fiat, M. Naor, J. P. Schmidt, and A. Siegel. Nonoblivious hashing. *J. ACM*, 39(4):764–782, 1992.
9. G. Franceschini and R. Grossi. Optimal worst-case operations for implicit cache-oblivious search trees. In *Proc. 8th WADS*, 2003.
10. G. Franceschini and R. Grossi. No sorting? better searching! In *Proc. 45th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 491–498, 2004.
11. M. L. Fredman. On the efficiency of pairing heaps and related data structures. *J. ACM*, 46(4):473–501, 1999.
12. M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan. The pairing heap: a new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986.
13. M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.
14. T. Hagerup and R. Raman. An efficient quasidictionary. In *SWAT*, 2002.
15. N. J. A. Harvey and K. Zatloukal. The post-order heap. In *Proc. FUN*, 2004.
16. J. Iacono. Improved upper bounds for pairing heaps. In *SWAT*, pages 32–43, 2000.
17. D. B. Johnson. Priority queues with update and finding minimum spanning trees. *Info. Proc. Lett.*, 4(3):53–57, 1975.
18. H. Kaplan and R. E. Tarjan. New heap data structures. Technical Report TR-597-99, Computer Science Dept., Princeton University, March 1999.
19. C. W. Mortensen and S. Pettie. The complexity of implicit and space-efficient priority queues. Manuscript, `http://www.mpi-sb.mpg.de/ pettie/`, 2005.
20. J. I. Munro. An implicit data structure supporting insertion, deletion, and search in $O(\log^2 n)$ time. *J. Comput. Syst. Sci.*, 33(1):66–74, 1986.
21. J. I. Munro and H. Suwanda. Implicit data structures for fast search and update. *J. Comput. Syst. Sci.*, 21(2):236–250, 1980.
22. Rasmus Pagh. Low redundancy in static dictionaries with constant query time. *SIAM J. Comput.*, 31(2):353–363, 2001.
23. R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *SODA*, pages 233–242, 2002.
24. R. Raman and S. S. Rao. Succinct dynamic dictionaries and trees. In *Proc. 30th Int'l Colloq. on Automata, Languages and Programming*, pages ??–??, 2003.
25. T. Takaoka. Theory of 2–3 heaps. *Discrete Appl. Math.*, 126(1):115–128, 2003.
26. J. W. J. Williams. Algorithm 232 (heapsort). *Comm. ACM*, 7:347–348, 1964.
27. A. C. C. Yao. Should tables be sorted? *J. ACM*, 28(3):615–628, 1981.
28. D. Zuckerman. *Computing Efficiently Using General Weak Random Sources.* Ph.D. Thesis, The University of California at Berkeley, August 1991.