Meldable Heaps and Boolean Union-Find

(extended abstract)

Haim Kaplan
School of computer science
Tel Aviv University
Tel Aviv, Israel
haimk@cs.tau.ac.il

Nira Shafrir
School of computer science
Tel Aviv University
Tel Aviv, Israel
shafrirn@cs.tau.ac.il

Robert E. Tarjan
Dept. of Computer Science
Princeton University
Princeton, NJ, and Compaq
Computer Corp. Palo Alto, CA
ret@cs.princeton.edu

ABSTRACT

In the classical meldable heap data type we maintain an item-disjoint collection of heaps under the operations *find-min*, *insert*, *delete*, *decrease-key*, and *meld*. In the usual definition *decrease-key* and *delete* get the item and the heap containing it as parameters. We consider the modified problem where *decrease-key* and *delete* get only the item but not the heap containing it. We show that for this problem one of the operations *find-min*, *decrease-key*, or *meld* must take non-constant time. This is in contrast with the original data type in which data structures supporting all these three operations in constant time are known (both in an amortized and a worst-case setting).

To establish our results for meldable heaps we consider a weaker version of the union-find problem that is of independent interest, which we call *Boolean union-find*. In the Boolean union-find problem the find operation is a binary predicate that gets an item x and a set A and answers positively if and only if $x \in A$. We prove that the lower bounds which hold for union-find in the cell probe model hold for Boolean union-find as well.

We also suggest new heap data structures implementing the modified meldable heap data type that are based on redundant binary counters. Our data structures have good worst-case bounds. The best of our data structures matches the worst-case lower bounds which we establish for the problem. The simplest of our data structures is an interesting generalization of binomial queues.

Categories and Subject Descriptors

E.1 [**Data**]: Data Structures

General Terms

Theory

STOC'02, May 19-21, 2002, Montreal, Quebec, Canada.

1. INTRODUCTION

The classical meldable heap data type maintains an item-disjoint[1] set of heaps subject to the following operations.

make-heap: Return a new, empty heap.
$\mathbf{insert(i, h)}$: Insert a new item i with predefined key into heap h.
$\mathbf{find\text{-}min(h)}$: Return an item of minimum key in heap h. This operation does not change h.
$\mathbf{delete\text{-}min(h)}$: Delete an item of minimum key from heap h and return it.
$\mathbf{meld(h_1, h_2)}$: Return the heap formed by taking the union of the item-disjoint heaps h_1 and h_2. This operation destroys h_1 and h_2.
$\mathbf{decrease\text{-}key}(\Delta, \mathbf{i}, \mathbf{h})$: Decrease the key of item i in heap h by subtracting the nonnegative real number Δ.
$\mathbf{delete(i, h)}$: Delete item i from heap h.

In the amortized setting Fibonacci heaps [10] support delete in $O(\log n)$ time and all other operations in $O(1)$ time.

Achieving these time bounds in the worst-case turned out to be harder. Shortly after Fredman and Tarjan introduced Fibonacci heaps Driscoll et al. [8] described a meldable heap data structure they call *run-relaxed heaps*. Run-relaxed heaps support all operations within the same time bounds as Fibonacci heaps but in the worst-case except *meld*, which takes $O(\log n)$ worst-case time. More recently, Brodal [4] described a different data structure that supports meld in $O(1)$ worst-case time but the time bound for decrease-key is $O(\log n)$ in the worst-case. The time bound for all other operations are as of Fibonacci heaps but worst-case. Ultimately, Brodal [5] gave a data structure matching the time bounds of Fibonacci heaps for all operations. This data structure, is very complicated however, much more complicated than Fibonacci heaps and the other meldable heap data structures that we mentioned.

Notice that in the data type for meldable heaps the operations *decrease-key* and *delete* get the target item and the heap containing it as parameters. Therefore in order to use a data structure implementing this data type one has to keep track of which heap contains an item while heaps undergo melds. This means, in any reasonable application, that we have to maintain some external union-find data structure whose sets correspond to the heaps.

It follows that the definition of the classical meldable heap data type leaves the union-find aspect external to the prob-

[1]Items may have the same key.

lem. The focus of this paper is on a modified data type in which the union-find aspect is integrated into the problem. Specifically, we modify the definition of *decrease-key* and *delete* so that they do not get the heap containing the target item as a parameter. The new definitions of *decrease-key*, and *delete* are as follows. The definitions of the other operations remain as in the original data type.

decrease-key$(\Delta, \mathbf{i})$: Decrease the key of item i in the heap that contains it by subtracting the nonnegative real number Δ.

delete$(\mathbf{i})$: Delete item i from the heap that contains it.

Of course one way to obtain a data structure implementing this new data type is to combine a data structure for the original data type with an external union-find data structure. We maintain a one-to-one correspondence between sets of the union-find data structure and heaps in the meldable heap data structure. When we meld two heaps we unite the corresponding sets, and when we need to figure out which heap contains an item (in order to decrease its key or delete it) we perform a find operation on the item. Since the meldable heap data type allows deletions we also need to be able to delete an item from a set in the union-find data structure. (Otherwise the space utilization and the time bounds of the union-find data structure may become very large and are not proportional to the number of items in the data structure). Classical union-find data structures [13, 14] do not support a delete operation, but the data structure [12] does.

A union-find with deletions data structure allows the following operations on a collection of disjoint sets.

make-set$(\mathbf{x})$: Create a set containing the single element x.

union$(\mathbf{A}, \mathbf{B}, \mathbf{C})$: Join the sets A and B into a new set C, destroying sets A and B.

find$(\mathbf{x})$: Find the set that contains x.

delete$(\mathbf{x})$: Delete x from the set that contains it.

For the amortized case we describe in [12] a union-find with deletions data structure that supports *delete(x)* and *find(x)* in $O(\alpha(m+n, n, \log l))$ time where n is the total number of elements, l is the size of the set containing x, and m is the total number of find operations. We define $\alpha(m,n,l) = \min\{k | A_k(\lceil \frac{m}{n} \rceil) > l\}$ where $A_k(j)$ is the k^{th} row of Ackermann's function. If we combine this data structure with Fibonacci heaps, we obtain a data structure implementing our new data type for meldable heaps with the following performance. Decreasing the key of an element residing in a heap of size l takes $O(\alpha(m+n, n, \log l))$ time, *meld* and *insert* take $O(1)$ time, and *delete* of an element from a set of size l takes $O(\log l + \alpha(m+n, n, \log l)) = O(\log l)$ time.

In the worst-case setting we describe in [12] a union-find with deletions data structure that supports *find* and *delete* in $O(\log_k n)$ time, *union* in $O(k)$ time, and *insert* in $O(1)$ time (k is fixed and known to the algorithm). If we combine this data structure with the data structure of Brodal [5], we get a data structure implementing our new meldable heap data type that supports *meld* in $O(k)$ time, *decrease-key* in $O(\log_k n)$ time, *delete* in $O(\log n)$ time and *insert* and *find-min* in $O(1)$ time.

Our results: The first problem which we address in this paper is whether the time bounds achieved by the data structures described above for the modified meldable heaps are optimal. In particular we are interested in the tradeoff between *decrease-key*, *meld*, and *find-min*. Can one implement *decrease-key*, *meld* and *find-min* all in $O(1)$ time ? Notice that there is no obvious reduction from the union-find problem to our modified meldable heap problem. So although the data structure described above uses a union-find data structure explicitly one may wonder whether this is really necessary.

We show in Section 2 that indeed non-trivial lower bounds exist. In fact we prove that the same tradeoff between *union* and *find* proved as a lower bound for the union-find problem [1] holds between *meld* and *decrease-key* (or *find-min*) for meldable heaps. We establish this result both in the worst-case and in the amortized case in the cell probe model of computation. Specifically, for the worst-case we show that if *meld* takes $O(k)$ time then either *decrease-key* or *find-min* must take $\Omega(\log_k n)$ time. For the amortized case we prove that a sequence of at most $n-1$ *meld*s and $O(m)$ *decrease-key* and *find-min* operations may take $\Omega(m\alpha(m, n, \log n))$ time.

To prove these lower bounds we define a restricted version of the union-find problem that is of independent interest. We call this problem the *Boolean union-find* problem. In the Boolean union-find problem the goal is to maintain disjoint sets under the following two operations.

union$(\mathbf{A}, \mathbf{B}, \mathbf{C})$: Join the sets A,B into a new set C, destroying sets A and B.

find$(\mathbf{x}, \mathbf{A})$: Returns TRUE if x is in the set A and FALSE otherwise.

The Boolean union-find problem is weaker than the classical union-find problem in the sense that we can answer a Boolean query using a single regular query, but the converse is not true. Our main result in the first part of the paper is a proof that the lower bounds established for the classical union-find hold for Boolean union-find as well. In our proof we use and extend some of the techniques developed to establish a lower bound for the marked ancestor problem [2].

Once we have the lower bound for Boolean union-find we prove the lower bound for meldable heaps by a straightforward reduction from Boolean union-find to meldable heaps. We believe that the Boolean union-find problem may be useful to establish lower bounds for other problems as well.

In the second part of the paper we address the question of whether we can find a simple data structure (simpler, say, than the one of Brodal [5] together with an external union-find data structure) that matches the lower bounds for the worst-case. Since with the modified data type we must allow either *meld*, *decrease-key*, or *find-min* to take non-constant time there is hope for a simple and intuitive data structure. (Recall that with the original data type we were after a data structure in which all these operations take $O(1)$ time.)

Our main result in this part of the paper is a data structure simpler than Brodal's that achieves these bounds. Specifically, for any fixed k, *insert* and *find-min* take $O(1)$ worst case time, *meld* takes $O(k)$ worst-case time, *decrease-key* takes $O(\log_k n)$ time and *delete* and *delete-min* take $O(\log n)$ time. The building blocks of this data structure are simpler implementations of heaps with inferior time bounds which are of independent interest. All these data structures are built upon redundant binary counters, which proved useful in developing several recent data structures [7, 5, 11]. Redundant binary counters are described in Section 3.

The most simplest data structure which we present is the *fat heap*. A fat heap is an interesting and simple generalization of the classical binomial queue [6] that uses redundant binary counters rather than a regular one. Fat-heaps support *insert*, *find-min*, and *decrease-key* in $O(1)$ worst-case time, and *delete* in $O(\log n)$ time. Although our use of fat heaps does not require them to support meld one can make them support meld in $O(\log n)$ time. Thereby they match the time bounds of the run-relaxed-heaps of Driscoll et al. and provide an alternative to run-relaxed-heaps in all their applications [8]. Fat heaps are described in Section 4.

Using fat heaps we then build a data structure that almost matches the worst case lower bounds. The only gap is in the worst-case time bound for meld, which is $O(\log_k n+k)$ worst-case time rather than the desired $O(k)$. At a high level this data structure is a redundant counter of fat heaps. It is described in Section 5. Finally to remove the $O(\log_k n)$ additive factor in the running time of *meld* we use an additional level of recursion together with a union-find with deletions data structure. This data structure is described in Section 6.

2. LOWER BOUNDS

In this Section we prove the lower bounds stated in the following theorem for the Boolean union-find problem.

THEOREM 2.1. *Let $0 < \epsilon \leq 1$. Let t_u and t_q be the worst case costs of* union *and* find, *respectively, in the cell probe model with word length of b bits.*
If $t_u \geq \max\{(\frac{140b}{\log n})^{\epsilon}, 140^{\epsilon}\}$, then t_q is $\Omega(\frac{\log n}{\frac{\epsilon+1}{\epsilon}\log t_u})$

Remark: We can prove a weaker lower bound of $\Omega(\frac{\log n}{\log(t_u b \log n)})$ on the worst case time for (Boolean) find via a reduction from a special case of the decremental existential marked ancestor problem described in [2].

For the amortized case we prove the following.

THEOREM 2.2. *Any algorithm that solves the Boolean union-find problem in the cell probe model with word length $\log n$ requires $\Omega(m\alpha(m,n,\log n))$ time to execute $n-1$ unions and m finds.*

THEOREM 2.3. *Any algorithm that solves the Boolean union-find problem on n elements, with amortized query cost $\bar{t}_q \leq k, k \geq 2$, and word length $\log n$ bits requires amortized update time $\bar{t}_u = \Omega(\alpha_{192k+1}(n))$ to execute n-1 unions and m finds where $\alpha_k(n) = \min\{j | A_k(j) > n\}$.*

From these results we derive the lower bounds for meldable heaps stated in Section 1 using the following reduction. Assume we have a data structure for maintaining disjoint sets that supports the operations $meld(h_1,h_2)$, *decrease-key*(x,Δ) , and *find-min(h)* defined according to our new meldable heap data type. Then we can construct a data structure for Boolean union-find as follows. We maintain each set as a heap in the meldable heap data structure. When we perform *make-set(x)* we create a new heap containing x with an arbitrary initial key. We perform $Union(A,B,C)$ by performing $C = Meld(A,B)$. We implement $find(x,A)$ as follows.

- Let $v =$ *find-min*(A)
- If $v = x$ then return TRUE
- If $(key(v) > key(x))$ return FALSE
- *decrease-key*$(x, key(x) - key(v) + 1)$ and let $u =$ *find-min*(A)
- If $u = x$ then return TRUE and otherwise return FALSE.

To prove Theorems 2.1, 2.2, and 2.3 we use the same *update scheme* used to prove the lower bounds for the classical union-find problem in [9] and [3].[2] In the rest of this section we describe the proof of Theorem 2.1. The ideas in the proofs of Theorem 2.2 and 2.3 are similar and are omitted from this extended abstract.

For the worst-case lower bound (Theorem 2.1) the *update scheme*, Σ, is the set of all operation sequences containing unions according to the following pattern. They start with n singleton sets, and consist of $r = \frac{1}{2}\log n$ update rounds.[3] Update round k ($1 \leq k \leq r$) consists of $n/2^k$ union operations which combine pairs of sets of size 2^{k-1} into sets of size 2^k. We use the notation $n[k] = \frac{n}{2^{k-1}}$. Note that $n[k]$ is the number of sets at the beginning of round k. We also need the following definitions.

Let τ be a prefix of an operation sequence which includes $j-1$ update rounds. After performing the operations in τ we get $n[j]$ sets, each of size 2^{j-1}. We refer to these sets as old sets. Let $G_{j,r}(\tau)$ be all possible distinct groupings of the old sets into sets of size 2^r each containing $n[j]/n[r+1]$ old sets. It is easy to see that the cardinality of $G_{j,r}(\tau)$ is independent of τ. Therefore we denote the cardinality of $G_{j,r}(\tau)$ by $G_{j,r}$. Using the fact that $(N/4)^N < N! \leq N^N$ for every $N > 0$ one can easily prove that

$$G_{j,r} = \binom{n[j]}{\frac{n[j]}{n[r+1]} \cdots \frac{n[j]}{n[r+1]}} > \frac{(n[r+1])^{n[j]}}{4^{n[j]}} = 2^{(\log n[r+1]-2)n[j]} \tag{1}$$

We define $M_{j,r}(\tau)$ to be the set of operation sequences whose prefix is τ that result in a certain distinct grouping of the old sets. Again the cardinality of $M_{j,r}(\tau)$ is independent of τ and we denote it by $M_{j,r}$. Let $C_{j,r}(\tau)$ be the set of operation sequences whose prefix is τ. I.e. $C_{j,r}(\tau) = \{\tau w \mid \tau w \in \Sigma\}$. Clearly for every τ, $|C_{j,r}(\tau)| = M_{j,r} * G_{j,r}$.

Note that if we substitute $r = \frac{1}{2}\log n$ into Equation (1) we obtain

$$G_{j,r} > 2^{(\frac{1}{2}\log n - 2)n[j]} \ , \tag{2}$$

and therefore

$$|C_{j,r}(\tau)| > M_{j,r} * 2^{(\frac{1}{2}\log n - 2)n[j]} \ . \tag{3}$$

DEFINITION 2.1. *Let $L_u(x_i)$ be the set containing x_i following operation sequence u, and let $A(u) = (L_u(x_1), \cdots, L_u(x_n))$. For operation sequences u and w we define $dist(A(u),A(w))$ to be the Hamming distance between $A(u)$ and $A(w)$.*

We denote by $find_u(x,Y)$ the query $find(x,Y)$ performed after the execution of operation sequence $u \in \Sigma$. We divide each operation sequence into $q = \frac{1}{2}\frac{\log n}{(1+\epsilon^{-1})\log t_u}$ epochs each consisting of $\beta = (1+\epsilon^{-1})\log t_u$[4] rounds (except for the last round which may be shorter). Epoch e extends from round $j_e = (e-1)\beta + 1$ up to round $e\beta$ (epoch q contains the last rounds of updates). To prove Theorem 2.1 we will show that

[2] We will also adopt most of the notation of [3].

[3] For simplicity we assume that $\frac{1}{2}\log n$ is integral.

[4] To simplify the notation we also assume that β is integral.

for a random update sequence u and a random item x the expected time of the query $find_u(x, L_u(x))$ is proportional to the number of epochs.

We also need the following definition.

DEFINITION 2.2. *Let X be a subset of $C_{j_e,r}(\tau)$. We call X* large *if* $|X| \geq M_{j_e,r}2^{(\frac{9}{20}\log n-2)n[j_e]-1}$. *Otherwise we call X* small.

The high-level structure of the proof is as follows. For an operation sequence u, we associate a disjoint set of registers, $S_e(u)$, with each epoch $1 \leq e \leq q$. We show that on average the query $find_u(x, L_u(x))$ has to look at registers from a constant fraction of these sets. To that end we show that for every epoch e with probability no smaller than $1/16$, $find_u(x, L_u(x))$ has to look at a register of $S_e(u)$. We do this as follows. Let τ be the prefix of u consisting of the first $e-1$ epochs.

1. Our register partition per update sequence induces a partition of $C_{j_e,r}(\tau)$ into equivalence classes. We denote the equivalence class of $u \in C_{j_e,r}(\tau)$ by $[u]_e$. For each *large* equivalence class we pick a representative $w \in [u]_e$. We show that if $[u]_e$ is *large* and the query $find_u(x, Y)$ does not read registers associated with epoch e then $find_u(x, Y) = find_w(x, Y)$.

2. We show that for a random operation sequence $u \in C_{j_e,r}(\tau)$, $[u]_e$ is *large* with probability at least $1/2$.

3. We show that if $[u]_e$ is *large* then for random x, $L_u(x) \neq L_w(x)$ with probability at least $1/8$.

Given an update sequence u we now define the sets of registers $S_e(u)$, $1 \leq e \leq q$. Let $reg_e(u)$ denote the registers written during u's updates in epoch e. Let $D^u(i)$ be the contents of register i following operation sequence u.

The sets $S_e(u)$ are defined backwards from $S_q(u)$ to $S_1(u)$ concurrently for all operation sequences $u \in \Sigma$. Assume we have already defined $S_{e+1}(u), \cdots, S_q(u)$ for each $u \in \Sigma$. Let $S_{>e}(u)$ be the set containing the registers that were assigned to epochs $e+1, \cdots, q$ for update sequence u.

$$S_{>e}(u) = \bigcup_{e+1 \leq j \leq q} S_j(u).$$

Let $R_{>e}(u)$ be the set whose items are pairs of registers from $S_{>e}(u)$ and their contents after u i.e.

$$R_{>e}(u) = \{(i, D^u(i)) \mid i \in S_{>e}(u)\}.$$

Let $find_u^e(x, Y)$ be the query $find_u(x, Y)$ performed on the data structure when the contents of the registers in $reg_e(u) \setminus S_{>e}(u)$ are restored to the time before epoch e was about to begin. Let

$$\begin{aligned} T_{>e}(u) \quad = \quad & \{(i, D^u(i)) \in R_{>e}(u) \mid \\ & \text{i was read by one of the queries } find_u^e(.,.)\}. \end{aligned}$$

Let τ be the prefix of u consisting of the $e-1$ first epochs we define

$$[u]_e = \{\tau v \in \Sigma \mid T_{>e}(\tau v) = T_{>e}(u)\}.$$

If $[u]_e$ is *small* then we define $< u >_e = \{u\}$. Otherwise ($[u]_e$ is large), we pick a representative sequence v from $[u]_e$, (say the lexicographically first in $[u]_e$), and let $< u >_e = \{u\} \bigcup \{v\}$. We define

$$S_e(u) = (\bigcup_{w \in <u>_e} reg_e(w)) \setminus (\bigcup_{k>e} S_k(u)).$$

The next lemma implies that if $< u >_e = \{u, w\}$, and $find_u(x, Y)$ doesn't read registers from $S_e(u)$, then $find_u(x, Y) = find_w(x, Y)$.

LEMMA 2.1. *Suppose a certain query $find_u(x, Y)$ doesn't read registers from the set $S_e(u)$ and $< u >_e = \{u, w\}$, then $D^u(i) = D^w(i)$ for all registers i read by the query $find_u(x, Y)$.*

PROOF. First we make the following observations.

1. *If $w \in [u]_e$ then for all elements x and sets Y, $find_u^e(x, Y)$, and $find_w^e(x, Y)$ read the same registers. Furthermore each of these registers has exactly the same content when read by $find_u^e(x, Y)$ and when read by $find_w^e(x, Y)$.*
Proof: Suppose i is read by the query $find_u^e(x, Y)$. If $(i, D^u(i)) \in T_{>e}(u)$ then since $T_{>e}(u) = T_{>e}(w)$ we obtain that $D^u(i) = D^w(i)$. If $(i, D^u(i)) \notin T_{>e}(u)$, then $(i, D^w(i)) \notin T_{>e}(w)$, and by the definition of $T_{>e}(u)$ the contents of i both in $find_u^e(x, Y)$ and in $find_w^e(x, Y)$ is a result of the updates performed in the first $e-1$ epochs which are identical in u and w (given by τ).

2. *If i is read by the query $find_u(x, Y)$ then i is also read by $find_u^e(x, Y)$ and the content of i is the same for both queries. (therefore also $find_u(x, Y) = find_u^e(x, Y)$.)*
Proof: The only thing that may differentiate $find_u(x, Y)$ and $find_u^e(x, Y)$ is the content of registers from the set $reg_e(u) \setminus S_{>e}(u)$. Since $S_e(u) \supseteq (reg_e(u) \setminus S_{>e}(u))$, these registers are not read by $find_u(x, Y)$, and thus are also not read by $find_u^e(x, Y)$.

Now let i be a register read by the query $find_u(x, Y)$. From our assumption on $find_u(x, Y)$ it follows that $i \notin S_e(u)$. If $i \in S_k(u)$ for some $k > e$ then by Observation (2), $(i, D^u(i)) \in T_{>e}(u)$. Since $T_{>e}(u) = T_{>e}(w)$ we obtain that $D^u(i) = D^w(i)$. Last assume that $i \in S_k(u)$ for some $k < e$, or $i \notin S_k(u)$ for any k. This implies that the last time i was written during u's updates was prior to the updates in epoch e. Therefore $D^u(i)$ is a function of the updates in the first $e-1$ epochs. Now consider the partition induced by w. Suppose $i \in S_l(w)$. We claim that $l < e$. This claim implies that $D^u(i) = D^w(i)$ since the updates done in the first $e-1$ epochs are the same for both u and w.

To prove the claim assume for a contradiction that $i \in S_l(w)$ for some $l \geq e$. We split into two cases.

1. $l = e$. By our definitions $< w >_e = \{w\}$, thus $reg_e(w) \supseteq S_e(w)$, and therefore $i \in reg_e(w)$. It follows from the definition of the register partition that either $i \in S_e(u)$, or $i \in S_{>e}(u)$. This is in contradiction with the assumption that $i \in S_k(u)$ for some $k < e$, or $i \notin S_k(u)$ for any k.

2. $l > e$. By observations (1) and (2), $find_w^e(x, Y)$ reads the same registers as $find_u(x, Y)$ so it reads register i. It follows that register $i \in S_{>e}(w)$ was read by the

query $find^e_w(x, Y)$ and therefore by the definition of $T_{>e}(w)$, $(i, D^w(i)) \in T_{>e}(w)$. Since $T_{>e}(w) = T_{>e}(u)$ we obtain that $(i, D^w(i)) \in T_{>e}(u)$ and therefore i must be in $S_{>e}(u)$. This is in contradiction with the assumption that $i \in S_k(u)$ for some $k < e$, or $i \notin S_k(u)$ for any k.

$\square$

Let u be an update sequence and let τ be the prefix of u consisting the first $e-1$ epochs. The next two lemmas imply that if $[u]_e \subseteq C_{j_e,r}(\tau)$ is large then the number of vectors $v \in [u]_e$ that are close to an arbitrary vector, $\overline{V}$, is relatively small.

LEMMA 2.2. *For any vector $\overline{V} \in R^n$ the following holds*

$$| \{u \in C_{j_e,r}(\tau) \mid dist(\overline{V}, A(u)) \leq \frac{n}{4}\} | \leq$$

$$(M_{j_e,r})2^{n[j_e](1+\frac{1}{2}\log n[r+1])} = (M_{j_e,r})2^{n[j_e](1+\frac{1}{4}\log n)}$$

PROOF. In [9], see also [3]. $\square$

Based on this lemma we next show that if $X \subseteq C_{j_e,r}(\tau)$ is *large*, then for at least half of the operation sequences $u \in X$ the Hamming distance between $A(u)$ and $\overline{V}$ is at least 1/4.

LEMMA 2.3. *If $X \subseteq C_{j_e,r}(\tau)$ is* large *and n is large enough, then for any vector $\overline{V} \in R^n$, $| \{u \in X \mid dist(\overline{V}, A(u)) \geq \frac{n}{4}\} | \geq 1/2 \mid X \mid$.*

PROOF. By lemma 2.2 for a certain $\overline{V}$, there are at least $|X| - (M_{j_e,r})2^{n[j_e](1+\frac{1}{4}\log n)}$ operation sequences $u \in X$, for which $dist(\overline{V}, A(u)) \geq n/4$.
Since $n[j_e](1 + \frac{1}{4}\log n) < n[j_e](\frac{9}{20}\log n - 2) - 2$, for large enough n, $|X| - (M_{j_e,r})2^{n[j_e](1+\frac{1}{4}\log n)} \geq (1/2)|X|$ for n large enough. $\square$

For a fixed τ, the next lemma show that there aren't too many equivalence classes $[\tau w]_e$ where $\tau w \in \Sigma$.

LEMMA 2.4. *Let τ be an update sequence for the first $e-1$ epochs. If $t_u \geq \max\{(\frac{140b}{\log n})^\epsilon, 140^\epsilon\}$ then the number of equivalence classes $[\tau w]_e$ where $\tau w \in \Sigma$ is no greater than $2^{n[j_e](\frac{1}{20}\log n)}$.*

PROOF. First we give an upper bound on $|R_{>e}(u)|$. From the definition of $S_{>e}(u)$ and $R_{>e}(u)$ it follows that $|R_{>e}(u)| \leq 2\sum_{i=e+1}^{q}$(number of memory writes in epoch i). Recall that epoch $e+1$ starts at round $e\beta+1$, where $\beta = (1+\epsilon^{-1})\log t_u$. In round i we perform $\frac{n[i]}{2} = \frac{n}{2^i}$ update operations each of which writes to at most t_u registers. Therefore by combining these observations together we obtain that

$$|R_{>e}(u)| \leq 2\sum_{i=e\beta+1}^{\frac{1}{2}\log n} \frac{n}{2^i} t_u < 2\frac{n}{2^{e\beta}} t_u = 2\frac{n[j_e]}{2^\beta} t_u$$

$$\leq 2\frac{n[j_e]}{(t_u)^{1+\epsilon^{-1}}} t_u = 2\frac{n[j_e]}{(t_u)^{1/\epsilon}}$$

We denote this upper bound on $|R_{>e}(u)|$ by x_e, i.e. $x_e = 2\frac{n[j_e]}{(t_u)^{1/\epsilon}}$. The rest of this proof is similar to the proof of the upper bound on the output variability in [1].

Let M be the memory image after the execution of τ. For $u \in C_{j_e,r}(\tau)$ let M^u be the memory image obtained from M by assigning to each of the registers in the set $S_{>e}(u)$ its value following the execution of u. Note that M^u and M differ in at most x_e registers. Note also that $find^e_u(x, Y)$ is the query $find(x, Y)$ performed on the memory image M^u.

Let Q be the query program and let n' be the number of all possible queries. We assume without loss of generality that each query takes exactly t_q probes and it does not look at the same register twice. Let $a(i, k, M^u)$ be the address accessed by the i'th query in the k'th step when the memory contents is M^u. Let $A_k(u)$ be the set of registers accessed by all queries in the k'th step if the memory contents is M^u i.e. $A_k(u) = \{a(i, k, M^u) \mid 1 \leq i \leq n'\}$. Clearly $|A_k(u)| \leq n'$. Let $B_k(u) = \{(i, D^u(i)) \in R_{>e}(u) \mid i \in A_k(u)\}$. We model a parallel execution of all possible queries on all memory images M^u, for every $u \in C_{j_e,r}(\tau)$ as a single decision tree of depth t_q as follows.

The root corresponds to the set $A_1(u)$ of up to n' cells read in the first step. Note that since it is the first step, $A_1(u) = A_1(w)$ for all $u, w \in C_{j_e,r}(\tau)$. The root has a child for each set $B_1(u)$ where $u \in C_{j_e,r}(\tau)$. We associate each update sequence $u \in C_{j_e,r}(\tau)$ with the level one node corresponding to $B_1(u)$.

If $B_1(u) = B_1(w)$ for $u, w \in C_{j_e,r}(\tau)$ then the registers read by u and w in the first step had the same content and therefore $A_2(u) = A_2(w)$. So all update sequences associated with a particular node v at level one read exactly the same registers in the second step of the queries. A level one node v has a child for each set $B_2(u)$ where u is an update sequence associated with v.

In general, each node v at level k of the decision tree corresponds to a set of update sequences $U(v) \subseteq C_{j_e,r}(\tau)$. Each pair of sequences $u, w \in U(v)$ read exactly the same registers in the first k steps of the queries and all these registers have the same contents in M^u and M^w respectively. Therefore for each such u and w, it must be the case that $A_{k+1}(u) = A_{k+1}(w)$. A level k node v has a child for each set $B_{k+1}(u)$ where $u \in U(v)$.

Notice that if u and w are associated with the same leaf at level t_q then $T_{>e}(u) = T_{>e}(w)$ so $w \in [u]_e$. Therefore the number of leaves in the tree bounds the cardinality of the set $\{[\tau w]_e \mid \tau w \in \Sigma\}$.

By the assumption that Q does not read a register twice the sets $B_i(u), 1 \leq i \leq t_q$ are disjoint. Thus $\sum_{i=1}^{t_q} |B_i(u)| \leq x_e$. So a bound on the number of leaves is

$$\sum_{j_1+\cdots+j_{t_q} \leq x_e} \binom{n'}{j_1} 2^{bj_1} \cdots \binom{n'}{j_{t_q}} 2^{bj_{t_q}} \leq$$

$$2^{bx_e} \sum_{k \leq x_e} \binom{n' t_q}{k} \leq 2^{(b+\log n'+\log t_q)x_e}$$

In our case $n' = nn[r+1] = n^{3/2}$. Note also that $t_q \leq n$.[5] So the cardinality of the set $\{[\tau w]_e | \tau w \in \Sigma\}$ is bounded by

$$2^{(b+\log n'+\log t_q)x_e} \leq 2^{(b+2.5\log n)x_e} \leq 2^{(b+2.5\log n)2\frac{n[j_e]}{(t_u)^{1/\epsilon}}} \leq$$

[5]Simple algorithms achieve $t_q \leq n$ and $t_u = O(1)$.

$$2^{7\max\{b,\log n\}\frac{n[j_e]}{(t_u)^{1/\epsilon}}} \leq 2^{n[j_e](\frac{7\max\{b,\log n\}}{(\max\{(\frac{140b}{\log n})^{\epsilon},140^{\epsilon}\})^{1/\epsilon}})} =$$

$$2^{n[j_e]\max\{(\frac{7b}{((\frac{140b}{\log n})^{\epsilon})^{1/\epsilon}}),(\frac{7\log n}{(140^{\epsilon})^{1/\epsilon}})\}} \leq 2^{n[j_e](\frac{1}{20}\log n)}$$

□

Using Lemma 2.4 we prove the next lemma that shows that with probability at least half if we pick u at random then $[u]_e$ is large and $| < u >_e | = 2$.

LEMMA 2.5. *Pick uniformly at random an update sequence u. Let e be a fixed epoch. Then, $Pr_{u \in \Sigma}\lceil | < u >_e | = 2 \rceil \geq 1/2$.*

PROOF. The classes $[u]_e$ form a partition of $C_{j_e,r}(\tau)$. By Lemma 2.4 there are at most $2^{n[j_e](\frac{\log n}{20})}$ classes $[u]_e$. Recall that a set is considered *small* if it contains less than $(M_{j_e,r})2^{(\frac{9}{20}\log n - 2)n[j_e]-1}$ update sequences and *large* otherwise. Let s be the total number of elements in small classes, it follows that $s \leq M_{j_e,r}2^{(\frac{1}{2}\log n-2)n[j_e]-1}$. Since by Equation (3) $|C_{j_e,r}(\tau)| \geq M_{j_e,j}2^{(\frac{1}{2}\log n - 2)n[j_e]}$ we obtain that $s \leq \frac{1}{2}|C_{j_e,r}(\tau)|$ and the lemma follows. □

Based upon these Lemmas we complete the proof of theorem 2.1 as outlined before. Pick an element x uniformly at random, and pick an operation sequence $u \in \Sigma$ uniformly at random independently of x. Perform the unions specified by u. Suppose that after the updates of u are performed, $x \in Y$. We show that on average the query $find_u(x, Y)$ has to read registers from a constant fraction of the sets $S_e(u), 1 \leq e \leq q$.

By Lemma 2.5, with probability $\geq \frac{1}{2}$, $| < u >_e = \{u, w\}| = 2$. If $| < u >_e | = 2$, then $[u]_e$ is *large* and thus by Lemma 2.3 for at least half of the operation sequences $v \in [u]_e$ the Hamming distance between $A(w)$ to $A(v)$ is at least 1/4. Thus if $[u]_e$ is *large* then with probability $\frac{1}{8}$, $find_w(x, Y) \neq find_u(x, Y)$. So $Pr_{u\in\Sigma}\lceil | < u >_e | = 2, find_{w \in <u>_e \setminus \{u\}}(x, Y) \neq find_u(x, Y)\rceil \geq \frac{1}{16}$.

Let $X_e \in \{0, 1\}$ be the indicator random variable for the event $| < u >_e | = 2$ and $find_w(x, Y) \neq find_u(x, Y)$. By the argument of the previous paragraph $X_e = 1$ with probability at least $\frac{1}{16}$ so $E(X_e) \geq \frac{1}{16}$. It follows by linearity of expectation that $E[\sum_{j=1}^{q} X_j] = \Omega(q)$. By lemma 2.1 if $X_e = 1$ then $find_u(x, Y)$ has to read registers from $S_e(u)$ in order to give a correct answer. Therefore $E[\sum_{j=1}^{q} X_j]$ upper bounds the expected number of registers read by the query $find_u(x, Y)$.

3. REDUNDANT COUNTERS

Two redundant counters form the heart of our heap structure. These counters are based on the redundant binary representation of Knuth and Clancy [7], extended to support increments and decrements of arbitrary digits. Similar counters are used by Brodal [5] and Kaplan and Tarjan [11]. In this section we describe simple pointer-based implementation of a redundant b-ary counter (for $b \geq 2$) that supports incrementing and decrementing an arbitrary digit in $O(1)$ time.

A *b-ary redundant representation* (b-ary RR) of a non-negative integer x is a sequence of digits $d_n, \ldots, d_0$, with $d_i \in \{-1, 0, 1, \ldots, b\}$, such that $x = \sum_{i=0}^{n} d_i b^i$. We call d *regular* if, between any two digits equal to b, there is a digit other than $b - 1$, and between any two digits equal to -1, there is a digit other than 0. A *fix* operation on a digit $d_i = b$ in a regular b-ary RR d increments d_{i+1} by 1 and sets d_i to 0, producing a new regular b-ary RR d' representing the same number as d. A *fix* operation on a digit $d_i = -1$ in a regular b-ary RR d decrements d_{i+1} by 1 and sets d_i to $b - 1$, producing a new regular b-ary RR d'.

To add 1 to digit d_i of a regular b-ary RR d, we proceed as follows: If $d_i = b$, fix d_i. If, following this, $d_i = b - 1$ or $d_i = b - 2$ and the least significant digit d_j with $j > i$ and $d_j \neq b - 1$ satisfies $d_j = b$, fix d_j. Add one to d_i. If $d_i = b$, fix d_i. It is straightforward to verify that this computation preserves regularity and adds b^i to the number represented by d.

We subtract 1 from digit d_i using a similar algorithm (details are omitted). To implement this scheme, we use a singly-linked list of digits, from least-significant to most-significant. In addition, each digit d_i equal to $b - 1, 0$, has a *forward pointer*. The forward pointer of digit $d_i = b-1$ indicates the least-significant digit d_j with $j > i$ and $d_j \neq b - 1$, if this digit is a b. The forward pointer of digit $d_i = 0$ indicates the least-significant digit d_j with $j > i$ and $d_j \neq 0$, if this digit is -1. In all other cases the forward pointer points to an arbitrary digit. The details of how these pointers are maintained are omitted.

4. FAT HEAPS

As a basic building block for our faster data structures we need a heap data structure that supports all operations but meld. In particular this structure has to support decrease-key in $O(1)$ time, make-heap, insert, and find-min in $O(1)$ time and delete in $O(\log n)$ time[6]. The implementation of decrease-key and delete should be according to their new definition. We also need to be able to delete an arbitrary element from a non-empty heap in constant time.

In this section we describe *fat heaps* which achieve these time bounds. Fat heaps are an interesting generalization of binomial queues [6]. A Binomial queue is essentially a binary counter of binomial trees. In contrast a fat heap is a redundant counter of fat binomial trees. In addition fat heaps maintain another counter to allow fast decrease-key.

In analogy with binomial trees, we define a *fat tree* F_k of rank k recursively, as follows. A fat tree F_0 consists of a single node. A fat tree F_k for $k > 0$ consists of *three* F_{k-1} trees linked by making the roots of two of them leftmost children of the root of the third.

A *fat heap* consists of a forest of almost heap-ordered fat trees, at most four per rank, with at most two violations of the heap order per rank. (A violation of rank k is a node of rank k whose key is less than that of its parent). We also impose regularity constraints on the trees of various ranks and the violations of various ranks, as described below. We maintain a pointer to a node containing a minimum key in the forest, called the *minimum node.* We maintain the property that the minimum node is a tree root; whenever this is not the case, we can restore this property by swapping the item in the minimum node with the item in an arbitrary root; this may eliminate a violation. For each node we maintain pointers to its leftmost child, to its left and right siblings, to its parent and to the heap that contains it. We

[6]All time bounds in the rest of the paper are worst-case.

also store its rank.

We organize the roots of the fat trees into a redundant ternary counter called the root counter. If there are t trees of rank k, the digit d_k of the counter has an implicit value of $t-1$. Instead of storing the value of d_k, we store a list of the trees of rank k. Incrementing d_k corresponds to adding a new tree of rank k to the forest. We maintain the invariant that the digit sequence of the counter is regular as defined in Section 3. The root counter supports both increment and decrement in constant time.

We organize the violations into a similar redundant binary counter called the *violation counter*, the k^{th} position of which is a list of nodes of rank k at which violations occur. Again we maintain the invariant that the digit sequence of the violation counter is regular. Incrementing the i^{th} digit of the violation counter corresponds to creating a new rank-i violation. The violation counter supports increment in constant time.

We maintain an array indexed by rank, whose k^{th} position points to the k^{th} digit of the root counter. We maintain a similar array for accessing the violations counter. Since the lengths of the counters are functions of the current size of the heap, these arrays must be extensible. (It is well-known that extensible arrays can be obtained from ordinary arrays by array doubling and incremental copying.)[7]

4.1 Operations

We describe how to perform *make-heap*, *find-min*, *insert*, *decrease-key delete* and *delete-min*, and how to delete an arbitrary element from a non empty heap in constant time.

To perform *make-heap*, we return pointers to an empty root counter and an empty violation counter. To perform *find-min*(h), we return the item in the minimum node of h. To perform *insert*(i, h), we put the new item i in the single node of a new F_0 tree, and insert this tree into the forest by incrementing the least-significant digit of the root counter. To increment a digit of the root counter, we may need to fix one or more digits of the root counter. To fix a digit $d_i = 3$, we link three of the four rank-i trees in the tree list to form one rank-$(i+1)$ tree, by making the two roots with higher keys leftmost children of the root with the smallest key, breaking a tie arbitrarily.

We perform *decrease-key*(Δ, i) as follows. Let h be the heap pointed by i. Let x be the node containing i. Subtract Δ from the key of i. If the new key of i is smaller than the minimum key of h, swap i with the minimum key. Let r be the rank of x. If x is a violating node, add x as a new rank-r violation by incrementing the r^{th} digit d_r of the violation counter.

The increment of d_r may require fixing one or more digits of the violation counter. Such a fix, say of $d_i = 2$, corresponds to converting two rank-i violations into a rank-$(i+1)$ violation, which we do as follows. Arrange the two violations to have the same parent, by swapping the subtree rooted at the violating node whose parent has smaller key with the subtree rooted at the rank-i sibling of the violating node whose parent has larger key. It is easy to verify that no new violations are created by this swap. Let y be the common parent of the two violating nodes after the swap. If the rank of y is $i+1$, let F be an F_i tree obtained from y by deleting the two violating nodes from its list of children, and let F' and F'' be the F_i trees rooted at the two violating nodes. Link F, F', and F'' to form an F_{i+1} tree whose root z is a node with smallest key among the roots of F, F' and F''. If y was not a tree root, replace it by z in the list of children of the old parent of y; and, if z turns out to be violating, increment d_{i+1}. Otherwise, replace the tree rooted at y by the new tree rooted at z. If, on the other hand, the rank of y is bigger than $i+1$, then, by the regularity condition on the violation counter, y must have a child w of rank $i+1$ that is not violating, and the two rank-i children of w must also be non-violating. Replace the two violating rank-i children of y by the two good rank-i children of w. Then link the two F_i trees rooted at the two violating nodes and the one rooted at w after cutting off its two rank-i children to form a new F_{i+1} tree. The root z of the newly formed F_{i+1} tree replaces w as a child of y. If z turns out to be violating, increment d_{i+1}.

We perform *delete-min*(h) as follows. Delete the subtree rooted at the minimum node from the forest. Decrement the digit in which the subtree was stored by one. To decrement a digit of the root counter, we may need to fix one or more digits of the root counter. To fix a digit $d_i = -1$, we remove one of the rank-$i+1$ trees from digit $i+1$ of the root counter, and cut off its two rank-i children. If its children were violating we decrease the value of the ith digit in the violation counter. We store the three F_i trees in digit i. We discard the minimum node, and insert the trees rooted at its children one-by-one into the forest by incrementing the appropriate digits of the root counter. The new minimum key is either at the root of a tree in the forest or in a violating node. Search the roots of the trees in the forest and the violating nodes for the new minimum m'. If m' is in a violating node, swap it with the element stored in a root of a tree in the forest. The node that previously stored m' may no longer be a violating node, and if this happens decrease the corresponding digit in the violation counter. After the swap, the new minimum is at a root of a tree in the forest. Let this root be the new minimum node.

We perform *delete*(i) by performing *decrease-key*(∞, i) and then *delete-min*(h).

delete of an arbitrary element: By regularity condition the value of the least significant digit, (digit 0), of the counter is not -1. If digit 0 contains an element which is not the minimum element of h we remove it and decrement digit 0 of the counter. If digit 0 contains just the minimum element of h x, and the heap contains more than one element, we decrement digit 0 of the counter. As a result digit 0 contains three items beside x. We remove one of these three items.

Based on the implementation described above, it is straightforward to obtain the following result.

THEOREM 4.1. *Fat heaps support* find-min, insert, decrease-key *and* delete of an arbitrary element: *in* $O(1)$ *worst-case time, and* delete, *and* delete-min *in* $O(\log n)$ *worst-case time.*

Remark: We can add a *meld* operation that runs in $O(\log n)$ time to fat heaps. However when we do that the cost of *decrease-key* increases to $O(\log n)$. The main difference between meldable fat heaps and nonmeldable fat heaps is that in the former we do not maintain a pointer from each item to the heap containing it. Therefore when we perform *decrease-key* we first have to locate the heap containing the target item (to find its violation counter). This

[7]Alternatively we can maintain in each node v of the heap pointers to the nodes of the counters that correspond to the rank of v.

makes decrease-key cost $O(\log n)$. The details of this modification are omitted from this extended abstract.

5. SIMPLE MELDABLE STRUCTURE

For a fixed integer $k > 1$ we describe a heap data structure H that supports *insert* and *find-min* in $O(1)$ time, *decrease-key* in $O(\log n/\log k)$ time, *meld* in $O(\frac{\log n}{\log k} + k)$ time, and *delete* in $O(\log n)$ time. For $n < k^k$, this heap is optimal up to a constant factor. In section 6 we will augment the *Simple Meldable Structure*, in order to get a heap which is optimal for all values of n.

We define a *heap of type* T_0 to be a fat heap containing at most k elementary items. For every $i \geq 1$ we define an *heap of type* T_i to be a fat heap storing at most k items each of which is a pair $p = (h_1, h_2)$ where h_1 and h_2 are heaps of type T_{i-1}. The first heap of each pair contains exactly k items, and the second heap of each pair contains at most k items. The key of a pair $p = (h_1, h_2)$ in a heap of type T_i for $i \geq 1$ is the smaller among the minimum key of an item in h_1, and the minimum key of an item in h_2. As each T_i is a fat heap, each of its items, (which may be a pair), points to T_i. Each heap which is a component of a pair points to the pair of which it is a component. We call a fat heap with exactly k elements *full*.

Our heap data structure is based on a redundant binary counter that supports increment and decrement of any digit. We implement this counter as described in Section 3 using the digits -1,0,1,2. With digit i of the counter we store 3 heaps of type T_i, at least one of which is not empty. We denote by h_1^i, h_2^i, h_3^i the three heaps of type T_i associated with digit i. Items are stored in h_1^i, h_2^i, h_3^i such that h_2^i is not empty only if h_1^i is full and h_3^i is not empty only if both h_1^i and h_2^i are full. If i is not the most significant digit of the counter then the total number of elements in h_1^i,h_2^i, and h_3^i is at least k and at most $3k$. Hence h_1^i is always full if i is not the most significant digit. We have a pointer from each digit of the counter to the heap. In addition to the redundant binary counter we also keep a pointer to the minimum element in the heap. The minimum element in the heap is contained in a heap of type T_0 which is either attached to digit 0 of the counter or is a component of a pair inside some heap T_1 for $i > 0$ (which in turn can be a component of a pair inside a heap of type T_2 etc.).

The value of the i'th digit of the counter, d_i, corresponds to the number of elements stored in h_1^i, h_2^i, and h_3^i. Digit d_i is -1 if there are at most k elements stored at h_1^i, and the heaps h_2^i and h_3^i are both empty. (Note that if i is not the most significant digit than there cannot be less than k elements in h_1^i.) Digit d_i is 0 if h_1^i is full, and there is only one element in h_2^i. Digit d_i is 1 if h_1^i and h_2^i are full, and there are $k-1$ elements in h_3^i. Digit d_i is 2 if h_1^i, h_2^i, and h_3^i are all full. In the remaining case where there are between $k+2$ and $3k-2$ elements in h_1^i, h_2^i, and h_3^i, d_i can be either 0 or 1.

We now show how to perform a fix operation on the redundant counter while maintaining the relation between the value of d_i and the number of items in h_1^i, h_2^i, and h_3^i.

We show how to perform a fix operation on the redundant counter while maintaining the relation between the value of d_i and the number of items in h_1^i, h_2^i, and h_3^i. To fix a digit $d_i = 2$, we take the full heap h_3^i and insert it as a pair whose second component is empty into the first among h_1^{i+1}, h_2^{i+1}, and h_3^{i+1} which is not full. By the regularity condition that the counter satisfies we know that $d_{i+1} \neq 2$. Therefore at least one of h_1^{i+1}, h_2^{i+1}, h_3^{i+1} is not full and we can insert an element into it. If the total number of elements in h_1^{i+1}, h_2^{i+1}, and h_3^{i+1} after the insertion is $3k$, or $k+1$ we increment d_{i+1} to two, or zero, respectively. If the total number of elements in h_1^{i+1}, h_2^{i+1}, and h_3^{i+1} after the insertion is $3k-1$, and $d_{i+1} = 0$ then we increment d_{i+1} to one. We also set d_i to be 0. We fix a digit $d_i = -1$ by deleting a pair $p = (h_1, h_2)$ from either h_2^{i+1}, or h_3^{i+1}. We omit the details from this extended abstract.

5.1 Heap operations

We now describe how to perform the heap operations on the simple meldable heap data structure. We denote by $h(v)$ the fat heap that contains the element v, and by $v.key$ the key of the element v.

Find-min(h)**:** We follow the pointer to the minimum item of the heap and return this item.

insert(x, h)**:** By the regularity condition satisfied by the counter one of h_1^0, h_2^0, and h_3^0 is not full. We insert x into the first among h_1^0, h_2^0, and h_3^0 which is not full. If the total number of elements in h_1^0, h_2^0, and h_3^0 has increased from k to $k+1$, or from $3k-1$ to $3k$ we increment the counter. Also if the total number of elements in h_1^0, h_2^0, and h_3^0 has increased from $3k-2$ to $3k-1$ and the value of d_0 is 0 we increment the corresponding digit of the counter. We also update the pointer to the minimum element of the heap if the key of new element is smaller than the previous minimum of the heap.

meld(H_1, H_2)**:** Suppose the last digit of the counter of H_1 is j_1, and the last digit of the counter of H_2 is j_2. Suppose without loss of generality that $j_1 \leq j_2$. If $j_1 > 0$ We traverse the counters of H_1 and H_2 from the least significant digit to digit $j_1 - 1$. When traversing digit d_i we pair the heaps h_1^i, h_2^i, and h_3^i of H_1 into one or two pairs such that in each pair at least one of the heaps is full. We insert the first pair into a non full heap of type T_{i+1} stored with d_{i+1} of H_2. Then we insert the second pair of h_1^i, h_2^i, and h_3^i if there is one into a non full heap of type T_{i+1} of H_2. Finally, we insert the heaps (of which there are at most 3k), of type T_{j_1} stored in digit j_1 of H_1, one by one to digit d_{j_1} of H_2. When inserting an element to digit i of H_2, we increment digit i of the counter of H_2 in the following cases: the value of digit i was 2, the number of elements in digit i has increased from $3k-1$ to $3k$ or from k to $k+1$, the number of elements stored in d_i has increased from $3k-2$ to $3k-1$ and the value of digit d_i was 0. We also update the minimum of the heap if the minimum element in H_1 is smaller than the minimum element in H_2.

decrease-Key(Δ, x)**:**

Decrease-key uses a recursive procedure, $DecKey$. $DecKey$ gets as parameters an element and a non-negative value. DecKey(w, Δ') is defined as follows:

1. Perform decrease-key(w, Δ'), on the fat heap $h(w)$.
2. If as a result, the minimum value of $h(w)$ has decreased to $w.key$, and $h(w)$ is contained in a pair p such that $p.key > w.key$, call $DecKey(p, p.key - w.key)$

To implement decrease-key(x, Δ), we call the procedure $DecKey(x, \Delta)$. We also update the pointer to the minimum element of the heap, if the minimum element has changed.

delete(x)**:** We follow the pointer from x to the heap of type T_0, $h(x)$, that contains it. If that heap is part of a pair inside a heap of type T_1, we follow the pointer from the pair to that heap. We continue this way till we get to the top level counter. We delete an element y from the last among the heaps h_1^0, h_2^0, and h_3^0 stored in digit 0, which is not empty. If the total number of elements in h_1^0, h_2^0, and h_3^0 has decreased from $k+1$ to k, or from $3k$ to $3k-1$ we decrement the counter. Also if the total number of elements in h_1^0, h_2^0, and h_3^0 has decreased from $k+2$ to $k+1$ and the value of d_0 is 1 we decrement digit 0 of the counter. We delete x from the fat heap h, and insert y to h. We distinguish two cases:

$y.key < x.key$: If as a result, the minimum value of $h(y)$ has decreased, and $h(y)$ is part of a pair p whose minimum value is bigger than $y.key$, we call $DecKey(p, p.key - y.key)$. $DecKey$ is the recursive procedure used to define the decrease-key operation.

$y.key > x.key$: We continue recursively as follows: Let h' be the fat heap that contains the element whose value has changed. (at first $h' = h(y)$). If as a result the minimum value of h' has increased and h' is part of a pair p whose minimum value should also increase, delete p from the heap that contains it, update its key and re-insert it to that heap. Continue in the same manner, applying the same logic to $h' = h(p)$.

In both cases, if x was the minimum element then we should traverse the counter digits to find the new minimum element.

delete-min(H)**:** We perform Delete(Find-min(H)).

5.2 Analysis

The next lemma shows that the length of the counter of a heap which contains n elements is $O(\log n/\log k)$. The proof is straightforward and omitted from this extended abstract.

LEMMA 5.1. *A non empty heap whose counter consists of j digits contains $\Omega(k^{j-1})$ elementary items.*

The following Theorem summarizes the performance of the simple meldable heaps.

THEOREM 5.1. *Simple meldable heaps support insert and find-min in $O(1)$, decrease$-$key(x, Δ) in $O(\frac{\log n}{\log k})$, delete(x) in $O(\log n)$ and meld in $O($length of the shorter structure$+k)$ $= O(\frac{\log n}{\log k} + k)$, where n is the number of items in the heap or heaps on which the operation takes place.*

PROOF. It is straightforward to see that insert and find-min take $O(1)$ time. To analyze the performance of decrease-key and delete, we first notice that it takes $O(i)$ steps to reach the top level counter from an elementary item stored in a heap of type T_i attached to digit i of the top level counter. Furthermore, by Lemma 5.1, $i \leq \frac{\log n}{\log k}$. Therefore decrease-key requires at most $O(\frac{\log n}{\log k})$ decrease-key operations on fat-heaps of size at most k, each such operation costs $O(1)$. So decrease-key takes $O(\frac{\log n}{\log k})$ time. Delete requires at most $O(\frac{\log n}{\log k})$ delete and insert operations (when $y > x$) on fat heaps of size k. So delete takes $O(\log n)$ time.

Meld consists of two stages. In the first stage we traverse all but the most significant digit of H_1. Per each such digit we perform a constant number of insert operations into heaps hanging off the counter of H_2 as well as no more than a constant number of increment operations on the corresponding digit of H_2. Thus the time it takes to perform this stage is proportional to the length of the smaller counter. In the second stage we insert at most 3k heaps stored in the most significant digit of the counter of H_1 to the appropriate heap hanging off the counter of H_2. We also perform the appropriate increment operations on the counter of H_2. Thus this stage takes $O(k)$ time and the theorem follows. □

6. RECURSIVE MELDABLE HEAPS

We describe a heap data structure H that supports *insert* and *find-min* in $O(1)$ time, *decrease-key* in $O(\log n/\log k)$ time, *meld* in $O(k)$ time, and *delete* in $O(\log n)$ time. The heap builds upon the *simple meldable structure*, described in section 5. It also uses a union-find with deletions data structure which supports for a given k, *union* in $O(k)$, *find* and *delete* in $O(\log n/\log k)$, and *insert* in $O(1)$.

A heap H is a pair (x, C_1) where x is the minimum item in H, and C_1 is a *simple meldable heap*, (which we will refer to as the "counter" structure). The counter part of the pair in H may be empty. The elementary items in the counter (items stored in T_0), are themselves pairs of an element and a counter structure. Along with H we keep a doubly linked list of pairs whose counter structure isn't empty. The list doesn't include the top level pair. Each pair whose counter is not empty, has a pointer to its position in that list.

To each heap we also maintain a corresponding set in the union-find with deletions data structure. Each heap points to its corresponding set and vice versa. Also each item has a pointer to its node in the union-find with deletions data structure.

6.1 Heap operations

In this section we describe how to perform the heap operations on the recursive meldable heap data structure.

find-min(H)**:** Return the value in the element part of the pair H.

insert(x,H): We insert x to the set (of the union-find with deletions structure), attached to H. If H is empty, we create a pair containing x and an empty counter and let H point to this pair. Otherwise, Let y be the minimum item stored in H. If $x < y$, then we let the element part of H point to x instead of y. We create a pair p containing x (if $x \geq y$), or y (if $x < y$), in its element part and an empty counter, and insert p to the counter of H.

decrease-key(x, Δ)**:** Let $p = (x, C)$ be the pair which contains x. We decrease the key of x by Δ. If p is not contained in any counter, then x is the minimum element and we finish. If p is contained in a counter C_1 we perform *decrease-key* of p in C_1. Let $p_1 = (y, C_1)$ be the pair that contains C_1.

If the new value of x is now smaller than the value of y (in which case after the decrease-key of p, (x, C) must be the minimum element of C_1), and p_1 is the top level pair, then we replace x and y in their pairs and update the values in C_1 appropriately. (The minimum values of the pairs and the heaps in C_1 that contained (x, C) and now contain (y, C), are updated to the key of y. The element (y, C) becomes the new minimum of C_1 after this update).

If the value of x is smaller than the value of y and p_1 is not the top level counter we continue as follow. We remove C_1 from p_1 and remove p_1 from the list of pairs with a non-empty counter. We perform find on the element corresponding to x in the union-find with deletions data structure to get the heap $H = (z, C_2)$ that contains x. We merge the top level counter C_2 of H with C_1 by performing $meld(C_1, C_2)$ on the counter structures. If the minimum element of H is now in the counter then it must be (x, C). In this case we switch the old minimum z with the new minimum x in their pairs, updating the counter data structure as described in the previous case.

meld(H_1, H_2)**:** Let $H_1 = (x_1, C_1)$ and let $H_2 = (x_2, C_2)$. Assume w.l.o.g that $x_1 \leq x_2$. If the length of C_1 or the length of C_2 is at most $k+1$, we merge the two counters by performing meld on the counter structures. Let C be the merged counter, $(C = Meld(C_1, C_2))$. We create a pair p containing x_2 in its element part, and an empty counter and insert p to the counter structure C. The resulting heap H is the pair (x_1, C).

If the length of both C_1 and C_2 exceeds $k+1$, then we perform $insert((x_2, C_2), C_1)$ (inserting the pair (x_2, C_2) to the counter structure C_1). We add the pair (x_2, C_2) to the list of pairs whose counter is not empty in H_1.

In both cases we also concatenate the lists of pairs whose counters are not empty, and perform union of the corresponding sets.

delete-min(H): Let $H = (x_1, C_1)$. We delete x_1 from the union-find data structure, and remove x_1 from the pair representing H. If C_1 is not empty then let $p = (x_2, C_2)$ be the minimum element in the counter structure C_1. We perform $delete\text{-}min(C_1)$, deleting p from the counter structure. If the counter part of p, C_2 is not empty, we merge C_2 with the top level counter by performing $C = meld(C_1, C_2)$ on the counters structures, and extract p from the list of pairs whose counter is not empty. If C_2 is empty, we remove a pair $p' = (x_3, C_3)$ from the list of pairs whose counter is not empty (if the list is not empty). We perform $C = meld(C_3, C_1)$. The resulting heap is the pair $H = (x_2, C)$.

delete(x): is carried out by executing $dcrease\text{-}key(x, \infty)$ and then $delete\text{-}min(H)$. (Using the union-find structure to find H).

6.2 Analysis

Notice that both *decrease-key* and *delete* may perform *meld* on the counter structures of some pairs. The next lemma shows that if a heap of n elements contains pairs whose counter is not empty then $n > k^k$, which means that $\log n / \log k > k$. Therefore the meld of the counter structures, performed by *delete* and *decrease-key* costs $O(\log n / \log k)$ which is $O(k + \log n / \log k)$ for these values of n.

LEMMA 6.1. *Let n be the number of items in a heap H. Let C be the counter component of H. If $n \leq k^k$ then the counter component of all the pairs stored in C is empty.*

PROOF. (sketch) We show by induction on the operation sequence that if a certain heap H contains $j > 0$ pairs, (beside the top level pair), whose counter component is not empty, then the number of elements in H is at least $(j+1)k^k$.

The nontrivial cases are the meld and delete-min operations. Consider a *delete-min* on a heap H that contains $j > 0$ pairs whose counter is not empty. By the induction hypothesis $|H| = n \geq (j+1)k^k$. During the operation the counter structure of a certain pair is merged into the top level counter. Thus the number of such pairs is decremented by one. So following the delete-min operation, H contains at most $j-1$ pairs whose counter part is not empty, and $|H| = n - 1 \geq (j+1)k^k - 1 \geq (j)k^k$. Thus the claim holds for this case. We omit the details of the other cases. □

The following theorem which follows from Lemma 6.1 summarizes the performance of recursive meldable heaps.

THEOREM 6.1. *For a given k, the heap structure supports* find-min *and* insert *in $O(1)$,* meld *in $O(k)$* decrease-key *in $O(\log n / \log k)$, and* delete *and* delete-min *in $O(\log n)$.*

7. REFERENCES

[1] S. Alstrup, A. Ben-Amram, and T. Rauhe. Worst-case and amortised optimality in union-find. In *Proceedings of the thirty-first annual ACM Symposium on Theory of Computing (STOC)*, pages 499–506, 1999.

[2] Stephen Alstrup, Thore Husfeldt, and Theis Rauhe. Marked ancestor problems. In *IEEE Symposium on Foundations of Computer Science*, pages 534–544, 1998.

[3] A. M. Ben-Amram and Z. Galil. A generalization of a lower bound technique due to Fredman and Saks. *Algorithmica*, 30:34–66, 2001.

[4] Gerth Stolting Brodal. Fast meldable priority queues. In *Workshop on Algorithms and Data Structures*, pages 282–290, 1995.

[5] G.S. Brodal. Worst-case priority queues. In *Proc. 7th annual ACM-SIAM Symposium on Discrete Algorithms (SODA 96)*, pages 52–58. ACM Press, 1996.

[6] M. R. Brown. Implementation and analysis of binomial queue algorithms. *SIAM J. Computing*, 7(3):298–319, 1978.

[7] M. J. Clancy and D. E. Knuth. A programming and problem-solving seminar. Technical Report STAN-CS-77-606, Department of Computer Science, Stanford University, Palo Alto, 1977.

[8] James R. Driscoll, Harold N. Gabow, Ruth Shrairman, and Robert E. Tarjan. Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation. *Communications of the ACM*, 31(11):1343–1354, 1988.

[9] M. L. Fredman and M. E. Saks. The cell probe complexity of dynamic data structures. In *Proc. 21st ACM Symposium on Theory of Computing*, pages 345–354, 1989.

[10] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[11] H. Kaplan and R. E. Tarjan. Purely functional representations of catenable sorted lists. In *Proceedings of the 28th Annual ACM Symposium on Theory of Computing*, pages 202–211. ACM Press, 1996.

[12] Haim Kaplan, Nira Shafrir, and Robert E. Tarjan. Union-find with deletions. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, 2002.

[13] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22:215–225, 1975.

[14] R. E. Tarjan and J. Van Leeuwen. Worst case analysis of set union algorithms. *Journal of the ACM*, 31:245–281, 1984.