Theoretical Computer Science 84 (1991) 107-126
Elsevier

A pointer-free data structure for merging heaps and min-max heaps

Giorgio Gambosi and Enrico Nardelli
Istituto di Analisi dei Sistemi ed Informatica, C.N.R., Roma, Italy

Maurizio Talamo
Dipartimento di Matematica Pura ed Applicata, University of L'Aquila, L'Aquila, Italy, and Istituto di Analisi dei Sistemi ed Informatica, C.N.R., Roma, Italy

Abstract

Gambosi, G., E. Nardelli and M. Talamo, A pointer-free data structure for merging heaps and min-max heaps, Theoretical Computer Science 84 (1991) 107-126.

In this paper a data structure for the representation of mergeable heaps and min-max heaps without using pointers is introduced. The supported operations are: *Insert*, *DeleteMax*, *DeleteMin*, *FindMax*, *FindMin*, *Merge*, *NewHeap*, *DeleteHeap*. The structure is analyzed in terms of amortized time complexity, resulting in a O(1) amortized time for each operation except for *Insert*, for which a O(lg n) bound holds.

1. Introduction

The use of pointers in data structures seems to contribute quite significantly to the design of efficient algorithms for data access and management. Implicit data structures [13] have been introduced in order to evaluate the impact of the absence of pointers on time efficiency.

Traditionally, implicit data structures have been mostly studied for what concerns the dictionary problem, both in 1-dimensional [4, 5, 9, 10, 14], and in multi-dimensional space [1]. In such papers, the maintenance of a single dictionary has been analyzed, not considering the case in which several instances of the same structure (i.e. several dictionaries) have to be represented and maintained at the same time and within the same array-structured memory.

In this paper, the implicit representation of a different data structure, the mergeable heap, is studied. Heaps are traditionally considered as the first example of implicit structures, since they can be used to easily implement a priority queue by means of an array [18]: this approach can be extended to the implementation of double ended priority queues (dequeues) by means of min-max heaps [3].

A different solution to the representation of priority queues is offered in [7], where a pointer-free modification of the binomial queue introduced in [18] is given. Such a structure, which is fully implicit (that is, it uses the first $n + c$ memory location to represent a n-element priority queue) makes it possible to efficiently perform the merge between substructures required in binomial queues, but does not seem to be extendible to the management of multiple instances of mergeable priority queues.

The extension of the implicit representation of priority queues to support the merge operation has been considered in [15], where an algorithm is presented for merging two heaps represented in different arrays. For a general discussion of mergeable heaps see [2, 12, 16].

In particular, we are interested to introduce a data structure which makes it possible to represent different instances of mergeable priority queues and dequeues within the same array and at the same time. Thus, the major goal of this paper is to develop a pointer free data structure for mergeable heaps and min-max heaps under which the basic operations of *Insert*, *Deletemax*, *Deletemin*, *Findmax*, *Findmin*, *Merge*, *Newheap*, and *Deleteheap* can be performed efficiently. The approach introduced to obtain such a result is related to the techniques introduced in [6] for the dynamization of decomposable searching problems.

Time complexity will be considered within the paper in an amortized sense [17], i.e. time complexity will be analyzed by averaging a worst case sequence of operations over time.

We will first consider a data structure for mergeable heaps and then extend it to manage also mergeable min-max heaps, thus settling, in the amortized analysis framework, the open question posed in [3], i.e. whether it is possible to devise a data structure for min-max heaps which allows an efficient management of merge operations.

The paper is organized as follows: in Section 2 the proposed data structure is presented; operations on the structure are described in Section 3 and analyzed in Section 4. In Section 5 extensions and further remarks are given.

2. Description of the data structure

As stated above, we are interested in dynamic data structures, in which the number of data items represented by the structure may change with time. In particular, we are interested in extending the approaches introduced in the framework of implicit data structures design to the simultaneous representation of multiple instances of dynamic data structures, which are subject to the operations of melding, creation

and deletion. That is, we are interested to deal at the same time with both a (time varying) set of instances of a data structure and a (time varying) set of elements to be represented in such structures. In the following, we will denote as n and $m \leq n$ the current number of elements and the current number of instances, respectively.

Our basic model is a 1-dimensional array $\mathbb{M}$ of locations $\mathbb{M}[1], \mathbb{M}[2], \ldots, \mathbb{M}[i], \ldots$, in which the elements of the represented set can be stored. In order to make it possible to efficiently represent several instances of the same structure at the same time, we relax the definition in [13] to include also data organizations in which elements are not stored in a set of contiguous locations.

Given m different instances of the structure containing n items, we represent them by storing data in $n + c_1 m + c_2$ memory locations, where c_1 and c_2 are suitable constants, using no additional space for updating the overall structure (i.e., merging instances and inserting and deleting both items and instances). We do not allow the presence of pointers, that is, we do not allow explicit representation of structural information.

Note that, in the approach introduced in [15], updating the structure in the case of merging heaps, requires a $O(n)$ additional space. Moreover the analysis given for space complexity does not consider this additional space as space used for the element representation.

Let us now start with the definition of the structure: let $k = \lceil \lg n \rceil + 1$. (All logarithms are to base 2 unless stated otherwise.)

A single heap H can be implemented by an array $H[1..2^k - 1]$ which is partitioned in k subarrays, called *heaptrees*. Each subarray implements a heap-structured binary tree by storing, as usual, the two sons of the element stored in location i in locations $2i$ and $2i+1$ [19]. We shall denote the heaptrees as $\mathcal{HT}_1, \mathcal{HT}_2, \ldots, \mathcal{HT}_{k-1}, \mathcal{HT}_k$, and they are made as follows: $\mathcal{HT}_1$ is the subarray indexed by [1..1], $\mathcal{HT}_2$ is the subarray indexed by [2..3], etc. (see in Fig. 1 an example for $k = 5$). In general, heaptree $\mathcal{HT}_i$ has index bounds $[2^{i-1}..2^i - 1]$.

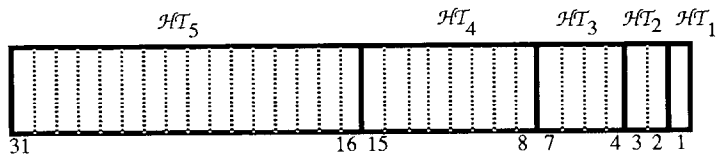

Fig. 1.

A heaptree $\mathcal{HT}_i$ has size $size(\mathcal{HT}_i) = 2^{i-1}$. We shall denote as $elem(\mathcal{HT}_i)$ the number of elements actually contained in $\mathcal{HT}_i$. If $elem(\mathcal{HT}_i) = size(\mathcal{HT}_i)$ we say that $\mathcal{HT}_i$ is *complete*. Given two heaptrees $\mathcal{HT}_i$ and $\mathcal{HT}_j$ we say that $\mathcal{HT}_i$ is *more significant* than $\mathcal{HT}_j$ if $i > j$. We assume that each of the non-empty heaptrees is complete, except for the most significant one (which we denote as the *leftmost* one). The leftmost non-empty heaptree may therefore have a number of elements less than its size (but, as we shall see, greater than half its size).

We will show later that the following invariant holds.

Invariant 1. If $\mathcal{HT}_i$ is the leftmost non-empty heaptree, then

$$\tfrac{1}{2}size(\mathcal{HT}_i) < elem(\mathcal{HT}_i) \leq size(\mathcal{HT}_i);$$

otherwise

$$elem(\mathcal{HT}_i) = size(\mathcal{HT}_i).$$

Given a heap H_i, the status of its heaptrees may be coded by a string of bits,

$$str(H_i) = b_k^i b_{k-1}^i \ldots b_2^i b_1^i,$$

where $b_j^i = 1$ means that $\mathcal{HT}_j$ of H_i is not empty and $b_j^i = 0$ means that $\mathcal{HT}_j$ of H_i is empty.

We maintain for each heap H_i (see an example in Fig. 2):
- the index $max(H_i)$ of the heaptree whose root is the element of maximum value in H_i;
- the index $left(H_i)$ of the leftmost non-empty heaptree of H_i (which corresponds to the most significant "1" in $str(H_i)$);
- the index $hleaf(H_i)$, relative to $\mathcal{HT}_{left(H_i)}$, of its leaf of highest index. Clearly, $hleaf(H_i) = size(\mathcal{HT}_{left(H_i)})$ if and only if $\mathcal{HT}_{left(H_i)}$ is complete.

In the following we will denote a generic heap as H, without any subscript, if no ambiguity is introduced by such a notation.

The overall structure of m heaps is implemented in the 1-dimensional array M by firstly representing the $m' > m$ heaptress $\mathcal{HT}_1$, then the $m' > m$ heaptrees $\mathcal{HT}_2$, and so on, always keeping adjacent equally ranked heaptrees (see in Fig. 3 an example with $m = 3$, $m' = 5$). The consequence of this approach is that, in general,

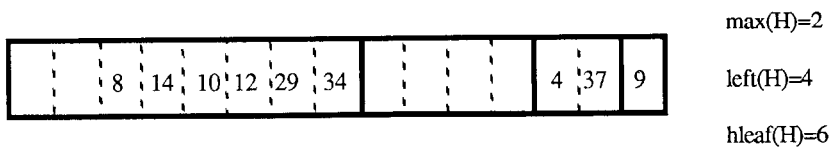

Fig. 2.

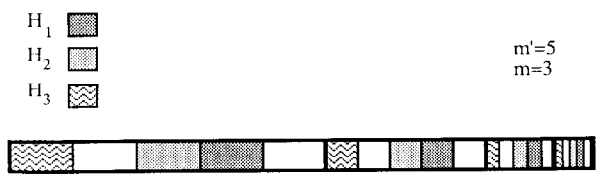

Fig. 3.

space for a certain number $m' - m$ of heaps will be available in $\mathbb{M}$: this is done in order to amortize the cost of creating or deleting a heap.

3. Manipulation of the data structure

We define the following operations:

- *Findmax*(H): the value of the element with maximum value in heap H is returned.
- *Deleteheap*(H): delete an empty heap H. This is only done in correspondence with the *Deletemax* operation which leaves H empty.
- *Newheap*(H): create a new heap H. This is only done in correspondence with the first *Insert* operation performed on H.
- *Insert*(e, H): a new element e is inserted in heap H.
- *Deletemax*(H): the element with maximum value is deleted from heap H.
- *Merge*(H_1, H_2): a new heap is created from the merging of H_1 and H_2. The resulting heap is stored in H_2, while H_1 is deleted.

Operations are executed as described below. We assume the existence of a threshold variable T (used by *Deleteheap* operations) and of a counter variable P (used by *Newheap* and accessed also by *Deleteheap*) which are both initialized to 1. The role of T is to drive the compaction of the structure by signaling that "enough" *Deleteheap* has been made, while the role of P is to provide a reference to a place where a newly created heap can be stored. Moreover, we assume the existence of a specific value which will be stored in heaptree $\mathcal{HT}_1$ of a heap H in order to mark H as unused.

Findmax(H).

- The value of the array cell corresponding to the root of $\mathcal{HT}_{max(H)}$ is returned.

Deleteheap(H).

- Mark heap H as unused;
- Set $T := T - 1$ and $m := m - 1$;
- IF $T = 0$

  THEN
  - for the heaptree $\mathcal{HT}_1$ of each used heap H_i, determine the number δ_i of heaptrees $\mathcal{HT}_1$ of deleted heaps contained in locations of smaller index;
  - move each element e contained in a heap H_i

$$\delta_i \cdot size(\mathcal{HT}(e)) + \left\lfloor \frac{J}{2} \right\rfloor \cdot (size(\mathcal{HT}(e)) - 1)$$

    positions to the right (i.e. towards smaller indices), where $J = m' - m$ denotes the number of unused heaps in the structure;
  - Set $T := \lceil n/2 \rceil$, $P := m + 1$, and $m' := m' - \lfloor J/2 \rfloor$.

Newheap(H).

- IF $P = m' + 1$ (i.e. there is no space available in $\mathbb{M}$ for a new heap)
  THEN move each element e in the overall structure $n(size(\mathcal{HT}(e)) - 1)$ positions to the left (i.e. towards greater indices), where $\mathcal{HT}(e)$ denotes the heaptree to which e belongs;
- set $m' := m' + n$;
- allocate the heap whose first location is $\mathbb{M}[P]$;
- set $P := P + 1$ and $m := m + 1$.

Insert(e, H).

- IF H does not exist THEN *Newheap*(H);
- find the index j of the rightmost empty heaptree in H;
- IF $j = 1$
  THEN $\mathcal{HT}_1 := [e]$
  ELSE IF $j = left(H) + 1$ and $\mathcal{HT}_{left(H)}$ is not complete
    THEN e is added to $\mathcal{HT}_{left(H)}$, and its maximum is adjusted
    ELSE $\mathcal{HT}_j$ is built from scratch, using element e plus all the elements contained in $\mathcal{HT}_{j-1} \ldots \mathcal{HT}_1$;
- the new maximum of H is derived by comparing the old maximum with element e.

Deletemax(H).

- $n := n - 1$;
- IF $hleaf(H) = 1$
  THEN
  - delete the element in $\mathcal{HT}_1$;
  - *Deleteheap*(H)
  ELSE
  - The root of $\mathcal{HT}_{max(H)}$ is deleted, the element corresponding to $hleaf(H)$ is moved from $\mathcal{HT}_{left(H)}$ to the root of $\mathcal{HT}_{max(H)}$, $hleaf(H)$ is decreased, and a new maximum for $\mathcal{HT}_{max(H)}$ is determined;
  - IF $elem(\mathcal{HT}_{left(H)}) = size(\mathcal{HT}_{left(H)})/2$
    THEN IF $\mathcal{HT}_{left(H)\ 1}$ is empty
      THEN $\mathcal{HT}_{left(H)}$ is moved to $\mathcal{HT}_{left(H)-1}$ and $left(H)$ is decreased
      ELSE $\mathcal{HT}_{left(H)}$ is rebuilt from scratch using elements in $\mathcal{HT}_{left(H)-1}$ and in $\mathcal{HT}_{left(H)}$ itself;
  - The new overall maximum is decided by comparing roots of non-empty heaptrees and $max(H)$ is updated in consequence.

Merge(H_1, H_2).

It can be assumed, without loss of generality, that $elem(H_1) \leq elem(H_2)$.

Let $\mathcal{HT}_{left(H_1)}(\mathcal{HT}_{left(H_2)})$ be the leftmost heaptree in H_1 (H_2). By hypothesis, $left(H_1) \leq left(H_2)$. Let $h = size(\mathcal{HT}_{left(H_1)}) - elem(\mathcal{HT}_{left(H_1)})$ and let S be the set of items in H_2 in locations $2^{left(H_2)-1} + hleaf(H_2), \ldots, 2^{left(H_2)-1} + hleaf(H_2) - (h-1)$.

Note that, by hypothesis and by Invariant 1, $h < hleaf(H_2)$ and all the items in S belong to $\mathcal{HT}_{left(H_2)}$.

The two strings $str(H_1) = b_k^1 b_{k-1}^1 \ldots b_2^1 b_1^1$ and $str(H_2) = b_k^2 b_{k-1}^2 \ldots b_2^2 b_1^2$ are added. Let $b_k^R b_{k-1}^R \ldots b_2^R b_1^R$ be the resulting string.

The resulting heaptrees are built in H_2 according to the result of the addition above. In particular, for each bit equal to 1 in the resulting string the corresponding heaptree is built as follows. A value equal to 1 for bit b_i^R may be generated in one out of four ways:

(a) it comes from bit b_i^1 equal to 1, only;

(b) it comes from bit b_i^2 equal to 1, only;

(c) it comes from a carry in the addition and bits b_i^1 and b_i^2 are both equal to 1;

(d) it comes from a carry in the addition, only.

Case (a):

IF $i < left(H_1)$ or $\mathcal{HT}_{left(H_1)}$ is complete
THEN heaptree $\mathcal{HT}_i$ of H_1 is copied into heaptree $\mathcal{HT}_i$ of H_2
ELSE heaptree $\mathcal{HT}_i$ of H_2 is built from scratch using items in S and in $\mathcal{HT}_i$ of H_1.

Case (b): No operation is performed.

Case (c): No operation is performed, since heaptree $\mathcal{HT}_i$ of H_2 remains in its place while heaptree $\mathcal{HT}_i$ of H_1 will be used for building some heaptree $\mathcal{HT}_j$, $j > i$, in the resulting heap (see case (d)). Note that if $i = left(H_1)$ and $\mathcal{HT}_{left(H_1)}$ is not complete, then elements from both $\mathcal{HT}_i$ and S will be used for building $\mathcal{HT}_j$.

Case (d): The building of heaptree $\mathcal{HT}_i$ from scratch is required, starting from smaller sized heaptrees in H_1 and H_2. For building the new heaptree all those heaptrees $\mathcal{HT}_j$, $j < i$, in H_1 and H_2 will be used such that bits b_j^1, b_j^2 have contributed to the generation or propagation of a carry in the chain of carries which ends in position i, except for those heaptrees of H_2 which are involved in case (c). Again, note that if $j = left(H_1)$ for some of the heaptrees $\mathcal{HT}_j$ above and $\mathcal{HT}_{left(H_1)}$ is not complete, then elements from both $\mathcal{HT}_j$ and S will be used for building $\mathcal{HT}_i$.

Note that, after all the resulting heaptrees have been built in H_2, it may happen that $\mathcal{HT}_{left(H_2)}$ does not respect Invariant 1 anymore. In particular, it may happen that

$$elem(\mathcal{HT}_{left(H_2)}) \leq size(\mathcal{HT}_{left(H_2)})/2.$$

In such a case, if $\mathcal{HT}_{left(H_2)-1}$ is empty, then $\mathcal{HT}_{left(H_2)}$ is moved to $\mathcal{HT}_{left(H_2)-1}$, otherwise a new heaptree $\mathcal{HT}_{left(H_2)}$ is built from scratch using the elements previously in $\mathcal{HT}_{left(H_2)}$ and in $\mathcal{HT}_{left(H_2)-1}$.

Finally, *Deleteheap*(H_1) is invoked.

4. Analysis

4.1. Preliminaries

We shall analyze the complexity of each operation amortized over a sequence of operations: t_a will denote the resulting amortized time complexity.

We shall use for our analysis the credit-debit technique (banker's view) [17]. Operations may either (i) directly manage the elements or (ii) consider each heaptree as a whole or (iii) be concerned with whole heaps or (iv) manipulate the overall structure. Therefore, for ease of proof, we associate credits to different parts of the structure: namely we will define *element-credits*, *heaptree-credits*, *heap-credits*, and *structure-credits*. Moreover, a pool of credits used to cope with the growing of the overall structure, denoted *pool-credits*, is defined.

We shall denote by $cred(e)$ the number of credits owned by element e (element-credits), with $cred(\mathcal{HT})$ the number of credits owned by heaptree $\mathcal{HT}$ (heaptree-credits), and with $cred(H)$ the number of credits owned by heap H (heap-credits). Given an element e of heap H, let i_e be the index of the bit corresponding to $\mathcal{HT}(e)$ in $str(H)$. Similarly, given a position p of heap H, let i_p be the index of the bit corresponding to $\mathcal{HT}(p)$ in $str(H)$, where $\mathcal{HT}(p)$ denotes the heaptree to which p belongs.

Let c_m $(1 < c_m < 2)$ be the constant for linear time heap construction [19]. The following invariants hold (recall that $k = \lceil \lg n \rceil + 1$).

Invariant 2. For each position p in a non-empty heaptree:

$$c_m \sum_{i > i_p}^{k+1} (1 - b_i) \leq cred(p).$$

From Invariant 1 this means that:

- each element e has always at least as many element-credits as the number of empty heaptrees from the one currently containing e up to the one following $\mathcal{HT}_k$, that is, enough credits to pay for its promotion up to the heaptree following the currently most significant one;
- the same holds for each empty position in the leftmost non-empty heaptree.

Note that, in general, in presence of different heaps it may easily happen that $left(H) < k$ for all heaps.

Invariant 3. For each non-empty heaptree $\mathcal{HT}_i$ in a heap H

$$2 \cdot (size(\mathcal{HT}_i) - elem(\mathcal{HT}_i)) \leq cred(\mathcal{HT}_i).$$

This means that each heaptree $\mathcal{HT}_i$ has always at least as many heaptree-credits as twice its missing elements.

Invariant 4. For each heap H in $\mathbb{M}$

$$2k \leq cred(H).$$

This means that each heap H has always enough heap-credits to pay for the execution of two operations, each with cost lg n.

The amortized time complexity is now analyzed for each operation.

4.2. Analysis of Findmax(H).

The time complexity is trivially O(1).

4.3. Analysis of Deleteheap(H)

We assume that each *Insert* operation gives 3 structure-credits upon each element insertion for the purposes of *Deleteheap*. Moreover, we assume that each *Deleteheap* operation gives 7 structure-credits when it is executed.

When the threshold T is not equal to 0 the *Deleteheap* requires a constant time. Let us denote as DH_{nc} (i.e. not costly) this kind of *Deleteheap*.

When the threshold T is equal to 0 all the elements in the structure are moved. Let us denote as DH_c (i.e. costly) this kind of *Deleteheap*.

The cost of moving n elements is given by three components. The first one considers the time required for finding the heaptrees which contain elements to be moved. The second one considers the time required for executing all the movements. The last one considers the time required to calculate the value δ_i for all the heaps in the structure and it is trivially bounded by P. For what regards the first two components, note that for each heap H the cost of moving all its elements is given by two terms. The first one considers the time required for scanning the $left(H)$ less significant bits of $str(H)$ in order to find the non-empty heaptrees of H. The second one considers the time required for moving all the elements of H. It is easy to see that for each heap the sum of these two terms is bounded by two times the total number of the elements in the heap. This leads to an overall bound of $2n+P$ for moving the n elements in the structure.

After any DH_c the counter P is reset to m. Immediately before any DH_c is executed it is $m' \geq P = m + G$, where G denotes the number of deleted heaps in positions $\mathbb{M}[1], \ldots, \mathbb{M}[P]$.

Suppose we are immediately before the execution of a DH_c. Then G indicates also the number of DH_{nc} which have been done since the last DH_c. Let us denote with n_A the number of elements which were in the structure immediately after the last DH_c, and with n_B the number of elements added since the last DH_c. Considering that m is always bounded by n, and that the threshold mechanism is such that $G = \lceil n_A/2 \rceil$, we have

$$2n + P \leq 2(n_A + n_B) + (n_A + n_B) + G = 7G + 3n_B.$$

Given the initial assumptions on the number of credits given by *Insert* and *Deleteheap*, it is easily seen that the overall cost of a DH_c is always covered by previously stored credits. This results in the following lemma.

Lemma 4.1. *The amortized cost for the Deleteheap operation is* $O(1)$.

Proof. Is derived easily from the considerations above. □

4.4. Analysis of Newheap(H)

We assume that each *Insert* operation gives 2 structure-credits upon each element insertion for the purposes of *Newheap*. Moreover, we assume that each *Newheap* and each *Deleteheap* give 4 structure-credits when it is executed.

When $P < m'+1$, *Newheap* requires a constant time. Let us denote as NH_{nc} (i.e. not costly) this kind of *Newheap*.

When $P = m'+1$, all the elements in the structure are moved. Let us denote as NH_c (i.e. costly) this kind of *Newheap*. Let us call *free-positions* the space available for future heap creations by means of $(m'-P)$ NH_{nc} operations.

The cost of moving n elements in this case is given by only the first two components considered in the analysis of *Deleteheap*. The fact that all the elements are moved means that the space previously opened by the last NH_c has been completely used. This space can have been used either by the execution of NH_{nc} operations or by the execution of DH_c operations (since DH_{nc} do not create free-positions we shall not consider them, for the time being).

Suppose we are immediately before the execution of a NH_c. Let us denote with F the number of operations (either NH_{nc} or DH_c) executed since the last NH_c, with n_C the number of elements in the structure immediately before the last NH_c (which is also the number of free-positions created by the last NH_c), with n_D the number of elements still remaining in the structure among the ones which were in the structure when the last NH_c was executed (clearly $n_C \geq n_D$), and with n_E the number of elements added since the last NH_c.

The number of elements moved by this NH_c is $n = n_D + n_E$. Under the initial assumptions on the number of credits given by *Insert*, *Newheap*, and *Deleteheap*, we have only to show the $4F \geq 2n_D$, i.e. $F \geq n_D/2$. We consider two subcases, depending on the number of DH_c operations executed.

Suppose that the number of DH_c operations is 0. Then, since there were n_C free positions, there have been at least n_C NH_{nc}: therefore $F \geq n_C \geq n_D$.

Suppose now that at least one DH_c has been executed. Since its effect is to halve the number of free-positions, the worst case for this analysis is when it is executed immediately after the last NH_c. This means that after the execution of this DH_c only $n_C/2$ free-positions remain, which require at least $n_C/2$ NH_{nc} to be filled: therefore $F \geq n_C/2+1 \geq n_D/2$.

To complete the analysis let us now take into account the case when DH_{nc} operations have also been executed. Let us call x_{nc} and x_c, respectively, the number of DH_{nc} and DH_c operations which have been executed since the last NH_c. Then the total number F of operations executed since the last NH_c cannot be less than $x_{nc}+x_c+n_C/2$. Therefore $F \geq x_{nc}+x_c+n_D/2 \geq n_D/2$. This results in the following lemma.

Lemma 4.2. *The amortized cost for the Newheap operation is* $O(1)$.

Proof. Is derived easily from the considerations above. □

4.5. Analysis of Insert(e, H)

Let us assign, upon the insertion of a new element e in heap H, $c_m(k+1)+k+2$ element-credits to e, 2 heap-credits to H, and 8 pool-credits.

The first step requires possibly the execution of a *Newheap*, which results in a O(1) amortized complexity by Lemma 4.2.

The second step requires at most k substeps, whose execution is paid by k element-credits assigned to e upon its insertion.

The time complexity of the third step requires a more detailed analysis:

(a) if the new element is inserted in $\mathcal{HT}_{left(H)}$ then $\lg(elem(\mathcal{HT}_{left(H)})) \leq left(H)$ time is spent to insert e in its proper position in $\mathcal{HT}_{left(H)}$. The insertion is paid by using $left(H) \leq k$ credits assigned to e upon its insertion. Therefore an amortized complexity of O(1) results.

(b) if a new heaptree $\mathcal{HT}_i$ is built from scratch, the time complexity is $c_m \cdot size(\mathcal{HT}_i)$. The cost of the operation is covered by letting each element involved in heaptree building pay c_m credits. Again, an amortized complexity of O(1) results.

The fourth step requires only 1 comparison.

Let us then state the following lemma.

Lemma 4.3. *If invariants* 1, 2, 3, *and* 4 *hold just before the execution of an Insert operation, then*

(a) *the Insert operation has amortized time complexity* O(lg n),

(b) *the invariants hold also after such an operation.*

Proof. If all invariants hold, it is easy to verify that there are enough credits to perform the *Insert* operation. Since $c_m(k+1)+k+2$ element-credits are assigned to e (plus a constant number of credits for other operations), this results in a O(lg n) amortized complexity.

For what concerns invariants maintenance, let us consider them case by case:

Invariant 1: If $\mathcal{HT}_1$ is empty the invariant is maintained, since element e is inserted in $\mathcal{HT}_1$, which is then complete. If $\mathcal{HT}_1$ is not empty, then either e is added to $\mathcal{HT}_{left(H)}$ or some complete heaptree $\mathcal{HT}_j$ is built: in both cases it is easy to see that the invariant is still maintained.

Invariant 2: The number of element-credits assigned to e which are spent during the first, second and fourth step is bounded by $k+2$.

For what the third step concerns, if $\mathcal{HT}_1$ is empty the invariant is maintained, since the insertion of element e in $\mathcal{HT}_1$ requires one credit and thus maintains a number of element-credits $cred(e) \geq c_m(k+1)-1 \geq c_m k \geq$ (c_m times the number of empty heaptrees from $\mathcal{HT}_2$ to $\mathcal{HT}_{k+1}$), while other elements do not spend their element-credits.

If $\mathcal{HT}_1$ is not empty, two cases are possible:

(a) e is added to $\mathcal{HT}_{left(H)}$: the invariant is maintained since $left(H) \leq k$ element-credits are spent for the insertion of e in $\mathcal{HT}_{left(H)}$, resulting in a number of remaining element-credits $cred(e) \geq c_m(k+1) - left(H) \geq c_m(k+1-left(H)) \geq$ (c_m times the number of empty heaptrees from $\mathcal{HT}_{left(H)}$ to $\mathcal{HT}_{k+1}$), while other elements do not spend their element-credits.

(b) some complete heaptree $\mathcal{HT}_j$ is built from elements in $\mathcal{HT}_{j-1}, \ldots, \mathcal{HT}_1$ plus element e: in such a case, the cost $c_m \cdot size(\mathcal{HT}_j)$ for building $\mathcal{HT}_j$ is covered by

letting each involved element pay c_m element-credits. For what e concerns, this results in a number of remaining element-credits $cred(e) \geq c_m k \geq c_m(k+1-j) \geq (c_m$ times the number of empty heaptrees from $\mathcal{HT}_j$ to $\mathcal{HT}_{k+1})$; for the other elements in $\mathcal{HT}_j$, note that, while each element spends c_m credits, the corresponding term $c_m \sum_{i>i_e}^{k+1} (1-b_i)$ is decreased by at least one, thus the invariant will still hold.

In case the insertion of the new element makes k increase by one, let us prove that each element in H may receive the one additional element-credit to pay for its (possible) promotion to a position one higher than before.

In order to do that, we will prove, assuming all elements in H have enough credits to pay for their promotion up to heaptree $\mathcal{HT}_{k+1}$ that if k is increased by one, then they may obtain the additional credits from the credit pool. Note that in order to increase k by one, *Insert* has been performed at least $n/2$ times after the last time k was increased: hence, at least $4n$ pool-credits are available and an amount n of them can be distributed among elements. Note that at least $3n$ credits are still available: $2n$ of them will be used to maintain Invariant 4 under the *Insert* operation, while n more are associated to the

$$\sum_{\text{all heaps } H} (size(\mathcal{HT}_{left(H)}) - elem(\mathcal{HT}_{left(H)})) < n$$

empty positions in the most significant heaptrees of all heaps: this is done, as we will show, to maintain Invariant 2 under the *Merge* operation.

Invariant 3: If a new complete heaptree $\mathcal{HT}_j (j>1)$ is built as a result of the operation, by merging heaptrees $\mathcal{HT}_1, \ldots, \mathcal{HT}_{j-1}$ and element e, then the condition in this invariant is true for $\mathcal{HT}_j$(both terms in the expression are = 0), but does not apply anymore to $\mathcal{HT}_1, \ldots, \mathcal{HT}_{j-1}$ (since they are now empty). The condition remains true for all other heaptrees (they are not modified).

If e is added to $\mathcal{HT}_{left(H)}$, then the condition in this invariant is true for $\mathcal{HT}_{left(H)}$ (the left term is decreased by two, while the right term is not modified) and remains true for all other heaptrees (they are not modified).

Invariant 4: The invariant is maintained, since the operation adds 2 heap-credits to $cred(H)$, without spending any heap-credit. In case the insertion of the new element makes k increase by one, let us now prove that each element in H may receive the two additional heap-credits required to maintain this invariant.

In order to do that, we will prove that, if all heaps in $\mathbb{M}$ have enough credits to make the condition in Invariant 4 true, then, if k is increased by one, they may obtain the additional credits from the credit pool. Note, again, that in order to increase k by one, *Insert* has been performed at least $n/2 \geq m/2$ times after the last time k was increased: hence, at least $2n$ pool-credits are still available, after the ones to maintain Invariant 2 are used and a number $2m \leq 2n$ of them can be distributed among the heaps. □

4.6. Analysis of Deletemax(H)

We assume that each *Deletemax* operation assigns $2c_m$ heaptree-credits to heaptree $\mathcal{HT}_{left(H)}$.

The THEN branch requires possibly the execution of a *Deleteheap*, which results in a O(1) amortized complexity by Lemma 4.1.

The first step of the ELSE branch has a time complexity O(1) for identifying $HT_{max(H)}$ and for moving the element corresponding to $hleaf(H)$ from $\mathcal{HT}_{left(H)}$ to $\mathcal{HT}_{max(H)}$. Finding a new maximum of $\mathcal{HT}_{max(H)}$ requires at most k comparisons.

The second step of the ELSE branch requires a more detailed analysis. For the sake of shortness during the analysis of this step let $j = left(H)$. The operation requires either (i) the moving of 2^{j-2} elements into $\mathcal{HT}_{j-1}$ or (ii) the building from scratch of a heaptree of 2^{j-1} elements.

(i) Moving all the elements requires 2^{j-2} steps and moreover, since after completion of the operation $\mathcal{HT}_j$ will be completely empty, each element moved to $\mathcal{HT}_{j-1}$ requires, in order to maintain Invariant 2, c_m additional credits. That results in $c_m \cdot 2^{j-2}$ overall credits. Note that only elements in $\mathcal{HT}_{j-1}$ require such additional credits, since from the point of view of elements in $\mathcal{HT}_{j-2} \ldots \mathcal{HT}_1$ there is still the same number of empty heaptrees. A total of $(c_m+1) \cdot 2^{j-2}$ credits is therefore needed and these are provided by the 2^{j-2} *Deletemax* operations that have half emptied $\mathcal{HT}_j$ and have increased the total amount of heaptree-credits of $\mathcal{HT}_j$ by exactly $2c_m \cdot 2^{j-2} > (c_m+1) \cdot 2^{j-2}$ credits. Therefore, an amortized complexity $t_a = O(1)$ results.

(ii) Building the heaptree $\mathcal{HT}_j$ from scratch requires $c_m \cdot 2^{j-1}$ steps and moreover, since after completion of *Deletemax* $\mathcal{HT}_{j-1}$ will be completely empty, elements in $\mathcal{HT}_{j-2} \ldots \mathcal{HT}_1$ require in order to maintain Invariant 2, c_m additional credits each. That is, a number of credits bounded by

$$c_m \sum_{i=1}^{j-2} size(\mathcal{HT}_i) = c_m \sum_{i=1}^{j-2} 2^{i-1} = c_m(2^{j-2}-1)$$

which gives a total of $c_m \cdot (2^{j-1}+2^{j-2}-1)$ credits needed. The 2^{j-2} *Deletemax* operations provided $2c_m \cdot 2^{j-2}$ credits, which balance the first term in the total above, while $c_m \cdot 2^{j-2}$ credits are provided by the elements of $\mathcal{HT}_{j-1}$, which have been moved one position towards the more significant end of the heap. Therefore we have $t_a = O(1)$.

The third step of the ELSE branch requires at most k comparisons.

Let us now state the following lemma.

Lemma 4.4. *If invariants* 1, 2, 3, *and* 4 *hold just before the execution of a Deletemax operation, then*:

(a) *the Deletemax operation has amortized time complexity* O(lg n),

(b) *the invariants hold also after such an operation.*

Proof. If all invariants hold, it is easy to verify that there are enough credits to perform the *Deletemax* operation. Since $2c_m \cdot k$ credits are assigned to some heaptree and at most $2k$ comparisons are performed during the first and third steps of the ELSE branch, this results in a O(lg n) amortized complexity.

For what invariants maintenance concerns, let us consider them case by case:

Invariant 1: If, just before the execution of the operation,

$$elem(\mathcal{HT}_{left(H)}) > 1 + \tfrac{1}{2} size(\mathcal{HT}_{left(H)}),$$

then the invariant remains true after the operations, since no heaptree is affected except $\mathcal{HT}_{left(H)}$. Otherwise, in both cases, a new complete most significant heaptree $\mathcal{HT}_{left(H)}$ is built, thus maintaining the invariant.

Invariant 2: Derives from the considerations above, regarding the analysis of time complexity of the second step of the ELSE branch.

Invariant 3: If, just before the execution of the operation,

$$elem(\mathcal{HT}_{left(H)}) > 1 + \tfrac{1}{2} size(\mathcal{HT}_{left(H)}),$$

then *Deletemax* modifies both sides of the invariant by the same amount, thus maintaining the invariant true. Otherwise, in both cases a new complete most significant heaptree $\mathcal{HT}_{left(H)}$ is built, and this maintains the invariant.

Invariant 4: The invariant terms are not affected by the operation. □

4.7. *Analysis of Merge*(H_1, H_2)

Let us assign, upon execution of a *Merge* operation, 2 structure-credits to $\mathbb{M}$.

Time complexity for scanning $str(H_1)$ and $str(H_2)$ is $2k$ and is paid by $2k$ heap-credits, taken from the smaller heap.

Analysis of cases (b) and (c) is trivial since no operation is performed.

Case (d) requires the building of a heaptree $\mathcal{HT}_j$, starting from the elements of smaller sized heaptrees in H_1 and H_2. Each one of these elements pays one element-credit and it is moved towards the more significant end of the heap by at least one position: this maintains Invariant 2 true, and gives an amortized time complexity of O(1).

Case (a) is slightly more complicated and involves some technicalities. In the analysis of this case, whenever we refer to credits we shall mean element-credits.

We shall now introduce some notation. Let us denote with $str(H_R)$ the string resulting from the addition between $str(H_1)$ and $str(H_2)$ and with $str(C)$ the string of carries generated during the addition itself. Hence, we denote with b_i^R the bits of $str(H_R)$ and with b_i^C the bits of $str(C)$. Given a sequence S of bits, we will denote as $sum(S)$ the number of bits equal to 1 in S. Let us indicate with q the index of the most significant bit involved in case (a). Then $b_q^1 = 1$, $b_q^2 = 0$, $b_q^R = 1$, $b_q^C = 0$. Since $str(H_1) < str(H_2)$, there exists an index p $(p > q)$ such that $b_i^1 = b_i^2$ for $i > p$, and $b_i^1 < b_i^2$ for $i = p$. Let us denote with α the sequence of indexes $k, k-1, \ldots, p+1$, and with β the sequence $p, p-1, \ldots, q+1$. Let us denote S_α^1, S_α^2, and S_α^R the subsequences of bits of $str(H_1)$, $str(H_2)$ and $str(H_R)$, respectively, whose indices belong to α.

Suppose now, for the sake of simplicity, that other patterns like that involved in case (a) either do not exist or exist at each one of the positions $q-1, q-2, \ldots, 1$. It is clear that either $sum(S_\alpha^R) = sum(S_\alpha^1)$ (when $b_p^C = 0$) or $sum(S_\alpha^R) = sum(S_\alpha^1) + 1$

(when $b_p^C = 1$). In any case, the number of zeros in $str(S_\alpha^R)$ does not increase with respect to $str(S_\alpha^1)$, thus maintaining Invariant 2.

Consider now S_β^1 and S_β^R. Invariant 2 is maintained if it happens that either $sum(S_\beta^R) > sum(S_\beta^1)$, which means that the number of zeros in $str(S_\beta^R)$ decreases, or $cred(b_j^R) > cred(b_j^1)$ $(1 \leq j \leq q)$, which means that the number of credits of $\mathscr{HT}_q, \mathscr{HT}_{q-1}, \ldots, \mathscr{HT}_1$ in the resulting heap increases for balancing the increase of zeros in $str(S_\beta^R)$.

Table 1 shows which are the feasible patterns of bits in the addition between $str(H_1)$ and $str(H_2)$ for positions $p-1, p-2, \ldots, q+1$.

Table 1

	Operands			Result		
	b_j^1	b_j^2	b_j^C	b_j^R	b_{j+1}^C	$b_j^R - b_j^1$
A	0	1	1	0	1	0
B	0	1	0	1	0	+1
C	0	0	1	1	0	+1
D	0	0	0	0	0	0
E	1	1	1	1	1	0
F	1	1	0	0	1	−1
G	1	0	1	0	1	−1

Rows A, B, C, D, E make the number of ones in $str(H_R)$ either greater than or equal to the number of ones in $str(H_1)$. Therefore they maintain Invariant 2 by decreasing zeros.

Rows F and G cause an increase of zeros in $str(H_R)$ with respect to $str(H_1)$ and therefore we have to show that, for each occurrence of such patterns, there is a corresponding increase of credits for the heaptrees of the resulting heap in positions $q, q-1, \ldots, 1$.

This can be shown as follows: Since in position q no carry is generated, the right-most pattern between F and G (let us remember that we are interested only in patterns which induce an increase of zeros in $str(H_R)$) at the left of q may only be pattern F. The reason is the following:

Consider the indices among $p-1, p-2, \ldots, q+1$ where patterns corresponding either to row F or to row G occur. Let us denote with s the smallest of such indices. We claim that at index s only pattern F can be found. In fact, assume the pattern at index s corresponds to row G: this means that at index $s-1$ either pattern E or pattern A is found, since these are the only rows different from F and G that may generate a carry. Both rows A and E correspond to situations where a carry is received from the right. This means that, again, only patterns A and E can be found in positions $s-2, s-3, \ldots, q+1$. This fact leads to an inconsistency, for neither pattern A nor pattern E may be in position $q+1$, since no carry is generated in position q. Then $b_s^1 = b_s^2 = 1$, $b_s^R = 0$, $b_s^C = 0$, $b_{s+1}^C = 1$.

Let us now focus on the generated carry b^C_{s+1}. Either it propagates without interruption up to b^C_p (hence to b^C_{p+1}, since $b^1_p = 0$ and $b^2_p = 1$), that is $b^C_p = b^C_{p-1} = \cdots = b^C_{s+2} = b^C_{s+1} = 1$, or it stops somewhere between p (included) and $s+2$ (included): that is, there exists an index t ($1 < t \leq p-s-1$) such that $b^C_{s+t+1} = 0$, and $b^C_{s+t} = \cdots = b^C_{s+1} = 1$. Note that $t = 1$ is a trivial case, since no increase of zeros occurs.

In the former case, possible patterns for position $p-1, p-2, \ldots, s+1$ are only those which both involve a carry and generate a carry (rows A, E, and G). Among them, the only one which causes an increase of zeros is G. Suppose therefore that there are L such patterns. This means an increase of $L+1$ zeros (L for patterns G and 1 for the pattern at index s) in positions between $p-1$ (included) and s (included). Each one of the patterns in such positions contains bits which correspond either to heaptrees which are moved to position $p+1$ (the heaptrees relative to patterns A and G, and the heaptrees of H_1 relative to pattern E) or to heaptrees which remain in their position (the heaptrees of H_2 relative to pattern E). Therefore, there are at least $L+2$ heaptrees moved (L heaptrees relative to pattern G and 2 heaptrees at index s), and all of them are moved at least 2 positions upward. Therefore, each of them may release a number of credits equal to its size, which is at least twice the size of $\mathcal{HT}_q$, for a total amount of $2(L+2) \cdot size(\mathcal{HT}_q)$ released credits. Note that, in order to cover the additional request of credits for all heaptrees in positions $q, \ldots, 1$, due to the $L+1$ additional zeros, at most $2(L+1) \cdot size(\mathcal{HT}_q)$ are required.

In the latter case, since we have $b^C_{s+t} = 1$ and $b^C_{s+t+1} = 0$, this implies $b^1_{s+t} = b^2_{s+t} = 0$ and $b^R_{s+t} = 1$. Possible patterns for position $s+t-1, \ldots, s+1$ are only those which both involve and generate a carry (rows A, E, and G). Among them, the only one which induces an increase of zeros is G. Suppose therefore that there are M such patterns. This means an increase of $M+1$ zeros (M for patterns G and 1 for the pattern with index s) in positions between $s+t-1$ (included) and s (included) and a decrease of one zero in position $s+t$. Therefore, it is necessary to provide for M additional zeros.

Each one of the patterns in positions $s+t-1, \ldots, s$ contains bits which correspond either to heaptrees which are moved to position $s+t$ (the heaptrees relative to patterns A and G and the heaptrees of H_1 relative to pattern E) or to heaptrees which remain in their position (the heaptrees of H_2 relative to pattern E). Thus, there are at least $M+2$ heaptrees moved (M heaptrees relative to pattern G, and 2 heaptrees relative to patterns with index s) and all of them, except the leftmost one in H_1, are moved at least two positions upward.

Therefore, each of them may release a number of credits equal to its size. There are $M+1$ such heaptrees (M in H_1 and one in H_2) and each of them has size at least twice the size of $\mathcal{HT}_q$, thus releasing a total of $2(M+1) \cdot size(\mathcal{HT}_q)$ credits. Note that, in order to cover the additional request of credits for all heaptrees in positions $q, \ldots, 1$, due to the M additional zeros, at most $2M \cdot size(\mathcal{HT}_q)$ credits are required.

Let us now state the following lemma.

Lemma 4.5. *If invariants* 1, 2, 3, *and* 4 *hold just before the execution of a Merge operation, then*

(a) *the Merge operation has amortized time complexity* O(1),

(b) *the invariants hold also after such an operation.*

Proof. If all invariants hold, it is easy to verify, from the analysis of the four cases above, that there are enough credits to perform the *Merge* operation, thus resulting in a O(1) amortized complexity.

For what invariants maintenance concerns, let us consider it case by case:

Invariant 1: To prove the invariant maintenance, note that the *Merge* operation, when moving the most significant heaptree from the smaller heap to the greater one, always completes it. All other moving or merging of heaptrees starts from complete heaptrees and generates complete heaptrees.

Invariant 2: If $\mathcal{HT}_{left(H_1)}$ is complete, the maintenance of the invariant follows from the considerations above, where it is shown that enough element-credits are released to maintain the invariant itself. In case $\mathcal{HT}_{left(H_1)}$ is not complete, the same considerations apply: note moreover that elements in S, which are moved to a less significant position, have to leave in $\mathcal{HT}_{left(H_2)}$ their element-credits (since the empty positions they leave must be equivalent to empty positions left by moved or deleted elements in $\mathcal{HT}_{left(H_2)}$) but can take from $\mathcal{HT}_{left(H_1)}$ the element-credits left by the $|S|$ elements which were moved or deleted from $\mathcal{HT}_{left(H_1)}$ itself.

Invariant 3: The invariant is easily maintained for all heaptrees except, possibly, $\mathcal{HT}_{left(H_1)}$ and $\mathcal{HT}_{left(H_2)}$, since neither the number of elements nor the number of heaptree-credits is modified by the *Merge* operation. For what concerns $\mathcal{HT}_{left(H_1)}$ and $\mathcal{HT}_{left(H_2)}$, note that, in case $\mathcal{HT}_{left(H_1)}$ is not complete, a number $|S|$ of elements is moved from $\mathcal{HT}_{left(H_2)}$ to $\mathcal{HT}_{left(H_1)}$: the invariant is maintained for both heaptrees by moving $2|S|$ heaptree-credits from $\mathcal{HT}_{left(H_1)}$ to $\mathcal{HT}_{left(H_2)}$.

Invariant 4: Is derived easily, since k is not increased by the operation, and the heap-credits of the larger heap—which is the one remaining after the operation—are not used. □

4.8. Main results

It is now possible to state the main theorem:

Theorem 4.6. *The structure described above makes it possible to represent a set of mergeable heaps without using pointers and with the following amortized time bounds*:

- *Insert, Deletemax*: O(lg n);
- *Newheap, Deleteheap, Findmax, Merge*: O(1).

Proof. The proof follows from Lemmas 4.1 and 4.2, for what *Deleteheap* and *Newheap* concerns. For the remaining operations the Theorem is proved by inductively showing that there are always enough credits in the whole structure to perform all the operations with the time bounds above. This can be done by proving that the invariants are always true.

It is trivial to show that all invariants hold at the beginning, when the structure is empty, while the inductive steps have been separately proved for each operation in Lemmas 4.3, 4.4, and 4.5. □

Note that the amortized complexity of *Deletemax* can be easily reduced to O(1) by assigning $2k$ more element-credits to each element, in correspondence to its insertion in the structure. Such element-credits are not affected by any operation but the *Deletemax*, where they can be spent during the first and third step of the ELSE branch—at most k element-credits are required for each step, since at most k comparisons are made in each step, see also Section 4.6—thus resulting in a constant amortized complexity. This result is not surprising, since the number of *Deletemax* operations is upper-bounded by the number of *Insert* operations, and therefore complexity of *Deletemax* can be amortized over *Insert*.

Therefore, the following theorem can be stated.

Theorem 4.7. *The structure described above makes it possible to represent a set of mergeable heaps without using pointers and with the following amortized time bounds:*

- *Insert*: O(lg n);
- *Deletemax, Newheap, Deleteheap, Findmax, Merge*: O(1).

Proof. From Theorem 4.6 and the considerations above. □

5. Extensions

The presented data structure may be used for managing min-max heaps with the same amortized time complexity. In fact, since min-max heaps are essentially standard heaps and have the same dynamics, they can be represented by our structure by simply treating heaptrees as min-max heaptrees. One additional index, $min(H)$, is only necessary for each heap to record the index of the heaptree which contains the element of minimum value in H. Operations *Insert, Findmax, Deletemax*, and *Merge* are exactly the same, while two new operations *Findmin* and *Deletemin* are defined, whose execution is symmetric to *Findmax* and *Deletemax*.

It is then possible to state the following theorem:

Theorem 5.1. *The structure above described makes it possible to represent a set of mergeable min-max heaps without using pointers and with the following amortized time bounds:*

- *Insert, Deletemax, Deletemin*: O(lg n);
- *Newheap, Deleteheap, Findmax, Findmin, Merge*: O(1).

Proof. Is derived easily from Theorem 4.6 and the considerations above. □

The considerations given above on the *Deletemax* operation can be applied also to min-max heaps, thus resulting in the following theorem.

Theorem 5.2. *The structure above described makes it possible to represent a set of mergeable min-max heaps without using pointers and with the following amortized time bounds:*

- *Insert*: $O(\lg n)$;
- *Deletemax, Deletemin, Newheap, Deleteheap, Findmax, Findmin, Merge*: $O(1)$.

Proof. Is derived easily from Theorem 5.1 and the considerations above. □

The data structure introduced makes it possible to manage also other operations such as *Delete*(e, H), *Decreasekey*(e, Δ), *Increasekey*(e, Δ). It can easily be seen that these operations can be performed on the structure above in $O(\lg n)$ time. It seems interesting to investigate whether the application of a lazy deletion technique [8, 11] can lead to improvements of such bounds.

Acknowledgment

We thank Jörg-R. Sack for useful discussions on these topics. Comments from the referees greatly helped in improving the presentation. We thank also Marinella Gargano and Alberto Postiglione for a careful and helpful reading of the final version of the paper.

References

[1] H. Alt, K. Mehlhorn and J.I. Munro, Partial match retrieval in implicit data structures, *Inform. Process. Lett.* **19** (1984) 2.

[2] A.V. Aho, J.E. Hopcroft and J.D. Ullman, *The design and analysis of computer algorithms* (Addison-Wesley, Reading, Ma., 1974).

[3] M.D. Atkinson, J.R. Sack, N. Santoro and T. Strothotte, Min-Max heaps and generalized priority queues, *Comm. ACM* **29** (1986) 10.

[4] A. Borodin, L.J. Guibas, N.A. Lynch and A.C. Yao, Efficient searching using partial ordering, *Inform. Process. Lett.* **12** (1981) 2.

[5] A. Borodin, F.E. Fich, F. Meyer auf der Heide, E. Upfal and A. Wigderson, A tradeoff between search and update time for the implicit dictionary problem, 13th ICALP, Rennes, France (1986).

[6] J.L. Bentley and J.B. Saxe, Decomposable searching problems I. Static to dynamic transformation, *J. Algorithms* **1** (1980).

[7] S. Carlsson, J.I. Munro and P.V. Poblete, An implicit binomial queue with constant insertion time, *Proc. SWAT* 88, *LNCS* 318 (Springer-Verlag, Berlin, 1988).

[8] D. Cheriton and R.E. Tarjan, Finding minimum spanning trees, *SIAM J. Comput.* **5** (1976) 4.

[9] G.N. Frederickson, Implicit data structures for the dictionary problem, *J. Assoc. Comput. Mach.* **30** (1983) 1.

[10] G.N. Frederickson, Recursively rotated orders and implicit data structures: a lower bound, *Theoret. Comput. Sci.*, **29** (1984) 1.

[11] M.L. Fredman and R.E. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, *J. Assoc. Comput. Mach.* **34** (1987) 3.

[12] D.E. Knuth, *The art of computer programming*, Vol. 3: *sorting and searching* (Addison-Wesley, Reading, Ma. 1973).

[13] J.I. Munro and H. Suwanda, Implicit data structures for fast search and update, *J. Comput. System Sci.* **21** (1980) 2.
[14] J.I. Munro, An implicit data structure supporting insertion, deletion and search in $O(\lg^2 n)$ time, *J. Comput. System Sci.* **33** (1986) 1.
[15] J.R. Sack and T. Strothotte, An algorithm for merging heaps, Acta Inform. **22** (1985).
[16] R.E. Tarjan, Data Structures and Network Algorithms, *CBMS–NSF Regional Conf. Ser. in Appl. Math.* **44** (1983).
[17] R.E. Tarjan, Amortized Computational Complexity, *SIAM J. Algebraic Discrete Methods* **6** (1985) 2.
[18] J. Vuillemin, A data structure for manipulating priority queues, Comm. ACM **21** (1978).
[19] J.W.J. Williams, Algorithm 232: Heapsort, Comm. ACM **7** (1964).