A note on meldable heaps relying on data-structural bootstrapping*

Claus Jensen

The Royal Library, Postbox 2149, 1016 Copenhagen K, Denmark

Abstract. We introduce a meldable heap which guarantees the worst-case cost of $O(1)$ for *find-min*, *insert*, and *meld* with at most 0, 3, and 3 element comparisons for the respective operations; and the worst-case cost of $O(\lg n)$ with at most $3 \lg n + O(1)$ element comparisons for *delete*. Our data structure is asymptotically optimal and nearly constant-factor optimal with respect to the comparison complexity of all the meldable-heap operations. Furthermore, the data structure is also simple and elegant.

1. Introduction

A (min-)heap is a data structure which stores a collection of elements and supports the following operations:

find-min(Q): Return the reference of the node containing a minimum element held in heap Q.

insert(Q, p): Insert the node (storing an element) pointed to by p into heap Q.

delete(p): Remove the node pointed to by p from the heap in which it resides.

meld(Q_1, Q_2): Move the nodes from one heap into the other and return a reference to that heap.

It is easy to see that the deletion of the minimum element can be accomplished by invoking *find-min*, followed by *delete* given the reference returned by *find-min*. Throughout this paper we use m and n to denote the number of elements stored in a data structure prior to an operation and $\lg n$ as a shorthand for $\log_2(\max\{2, n\})$.

As our model of computation, we use the *word RAM*. It would be possible, with some modifications, to realize our data structure on a pointer machine. However, when using a word-RAM model the data structure and the operations are simpler and more elegant. The term *cost* is used to denote the sum of instructions and element comparisons performed.

We want to obtain a data structure where the operations *find-min*, *insert*, and *meld* have worst-case constant cost, and using this as a base we want to minimize the comparison complexity of *delete*. In the amortized sense a cost

Table 1. The worst-case comparison complexity of heap operations for selected data structures. For all structures *find-min*, *insert*, and *meld* have $O(1)$ worst-case cost. Observe that the constant factors are not proved in some of the original sources, but are derived by us.

Source	delete-min	delete
[1]	$7 \lg n + O(1)$	$7 \lg n + O(1)$
[2]	$4 \lg n + O(1)$	not supported
this paper	$3 \lg n + O(1)$	$3 \lg n + O(1)$
[9]	$2.5 \lg n + O(1)$	$2.5 \lg n + O(1)$
[8]	$2 \lg n + O(1)$	$2 \lg n + O(1)$

of $O(1)$ for *find-min*, *insert*, and *meld*; and a cost of $O(\lg n)$ for *delete* is achieved by binomial heaps[1], Fibonacci heaps [10], thin heaps [12], and thick heaps (one-step heaps) [12, 11]. The same worst-case bounds are achieved by the meldable heaps described in [1], [2], and [8], see Table 1.

Given the comparison-based lower bound for sorting, it follows that *delete* has to perform at least $\lg n - O(1)$ element comparisons, if *find-min* and *insert* only perform $O(1)$ element comparisons. Furthermore, in [1] it is shown that if *meld* can be performed at the worst-case cost of $o(n)$ then *delete* cannot be performed at the worst-case cost of $o(\lg n)$.

In this paper we present a meldable heap which is a binomial heap transformed using data-structural bootstrapping. We obtain an good bound for the worst-case comparison complexity of *delete* given that the heap operations *find-min*, *insert*, and *meld* have a worst-case constant cost. Observe that there exist other data structures [8, 9] that obtain slightly better worst-case comparison complexity for *delete*. However, these data structures are advanced and mainly of theoretical interest. We reuse some of the techniques described in [2], [14], and [5], but our focus is on good constant factors and a simple data structure. In Section 2, we describe a modified binomial heap that supports *insert* at the worst-case cost of $O(1)$ and *meld* at the worst-case cost of $O(\lg n)$. In Section 3, we describe how to transform a binomial heap supporting *meld* at the worst-case cost of $O(\lg n)$ into a heap that supports melding at the worst-case cost of $O(1)$ and we prove the worst-case cost of $O(1)$ for *find-min*, *insert*, and *meld*; and the worst-case cost of $O(\lg n)$ with at most $3 \lg n + O(1)$ element comparisons for *delete*. In Section 4, we conclude the paper with some final remarks.

2. Binomial heaps

In this section, we give a description of binomial heaps. However, be aware that the binomial heaps described here are different from the standard binomial heaps described in most textbooks.

A *binomial tree* [15] is an ordered tree defined recursively as follows: A binomial tree of rank 0 is a single node; for ranks higher than 0, a binomial

---

[1] This result is folklore; it can be achieved by applying lazy melding to binomial heaps [15] as in Fibonacci heaps [10].

tree of rank r consists of the root and its r subtrees of rank 0, 1, ..., $r - 1$ connected to the root in that order. Given a root of a tree, we denote the root of the subtree of rank 0 the *smallest child* and the root of the subtree of rank $r - 1$ the *largest child*. The size of a binomial tree is always a power of two, and the rank of a tree of size 2^r is r.

A node in a binomial tree contains an element, a rank, a parent pointer, a child pointer, and two sibling pointers. The child pointer of each node points to its largest child if it exists. The children of a node are kept in a *child list*, which is implemented as a doubly-linked list using the sibling pointers. If a node is a root, the parent pointer points to the heap in which the node resides. Furthermore, the right-sibling pointer of a root points to the root itself.

The binomial trees used by us are *heap ordered* which means that the element stored at a node is no greater than the elements stored at the children of that node. If two heap-ordered binomial trees have the same rank, they can be linked together by making the root that stores the non-smaller element the largest child of the other root. We call this linking of trees a *join*. A join involves a single element comparison and has the worst-case cost of $O(1)$.

The *binomial heap* is a collection of a logarithmic number of binomial trees storing n elements. To obtain a strict bound on the number of trees τ, we maintain the invariant $\tau \leq \lfloor \lg n \rfloor + 1$. A similar approach is used for run-relaxed heaps [5].

Lemma 1. *In a binomial heap of size n, the rank of a tree can never be higher than $\lfloor \lg n \rfloor$.*

Proof. Let the highest rank be k. None of the trees can be larger than the size of the whole heap, i.e. $2^k \leq n$. Since k is an integer, $k \leq \lfloor \lg n \rfloor$.   □   □

The roots of the trees in the binomial heap are maintained using a data structure, adopted from [5], which we call a *root store*. The central structure is a resizable array which supports growing and shrinking at the tail at worst-case constant cost; this can be obtained, for example, by doubling, halving, and incremental copying. Each entry in this resizable array corresponds to a rank. For each entry (rank) there exists a *rank list* (doubly-linked) containing pointers to the roots of trees having this rank. Each root uses its left-sibling pointer as a *back pointer* to its corresponding item in the rank list. For each entry of the resizable array that has more than one root, a pointer to this entry is kept in a *pair list* (doubly-linked). An entry of the resizable array contains a pointer to the beginning of a *rank list* and a pointer to an item in the pair list if one exist.

The root of a tree of arbitrary rank is inserted into the root store using *add*. This operation inserts an item pointing to the given root into the corresponding rank list, sets the back pointer of the root to point to that item, and updates the pair list if necessary. If $\tau > \lfloor \lg n \rfloor + 1$, *add* uses the pair list to find two trees of the same rank which it removes from the rank list. It then performs a join of the two trees followed by an insert of the

new tree into the corresponding rank list, after which it updates the pair list if necessary. A tree is removed from the root store using *subtract*, which removes the item in the rank list referred to by the back pointer of the given root and updates the pair list if necessary. Both *add* and *subtract* have the worst-case cost of $O(1)$. In connection with each *add* it may be necessary to perform a single element comparison.

In the following, the binomial-heap operations are described and their worst-case bounds are analysed.

The candidates for a minimum element are stored at the roots of the binomial trees. A fast *find-min* is obtained by maintaining a *minimum pointer* to the node containing a minimum element. Thus *find-min* can be accomplished at the worst-case cost of $O(1)$ using no element comparisons.

When inserting an element, the node containing the element is treated as a tree with rank 0 and added to the root store using *add*. The element of the inserted node and the element of the node pointed to by the minimum pointer are compared and the minimum pointer is updated if necessary. From our earlier analysis, *add* has the worst-case cost of $O(1)$ and at most one element comparison is performed. Checking the minimum pointer has the worst-case cost of $O(1)$ and at most one element comparison is performed. Therefore *insert* has the the worst-case cost of $O(1)$ and at most two element comparisons are performed.

A *delete* is performed in the following way. A root is deleted by subtracting it from the root store, unlinking its subroots and adding them to the root store. If the node to be deleted is not a root, the node is swapped with its parent until it becomes a root, the old root is subtracted from the root store, and the subroots of the new root are unlinked and added to the root store. Furthermore, a scan of all roots is performed and the minimum pointer is updated if necessary. The operation of swapping a node until it becomes a root has the worst-case cost of $O(\lg n)$. A *subtract* has the worst-case cost of $O(1)$. Both the unlinking and addition of a subroot have the worst-case cost of $O(1)$ and at most one element comparison is performed in connection with each addition. By Lemma 1, a root can have at most $\lg n$ subroots which means that at most $\lg n$ element comparisons are performed when adding the roots to the root store. A scan of all roots has the worst-case cost of $O(\lg n)$ and requires at most $\lg n$ element comparisons. To sum up, each *delete* has the worst-case cost of $O(\lg n)$ and requires at most $2 \lg n + O(1)$ element comparisons.

In *meld*, two binomial heaps Q_1 and Q_2 are to be melded. Without loss of generality, we assume that the number of trees in Q_1 is smaller than or equal to that in Q_2. The root store of Q_1 is emptied using *subtract*; after each *subtract* the removed tree is added to the root store of Q_2 using *add*. The elements of the nodes pointed to by the minimum pointer of Q_1 and Q_2 are compared and the minimum pointer of Q_2 is updated if necessary. When all trees have been removed from Q_1, the root store of Q_1 is released. When the respective sizes of Q_1 and Q_2 are m and n, the number of trees moved is at most $\min \{\lg m, \lg n\} + 1$. In connection with every *add* at most one element
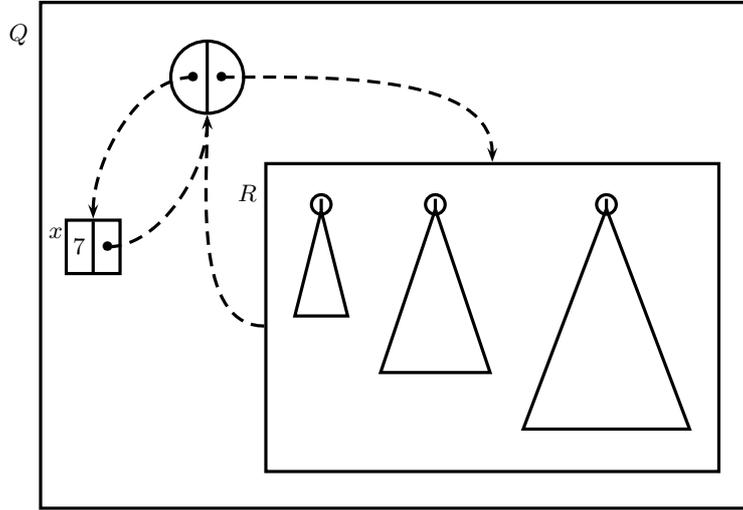
Figure 1. A simplified view of the transformed heap.

comparison is performed, and both *add* and *subtract* have the worst-case cost of $O(1)$. Finding the minimum pointer has the worst-case cost of $O(1)$ and at most one element comparison is performed. Therefore, the worst-case cost of *meld* is $O(\min\{\lg m, \lg n\})$ and at most $\min\{\lg m, \lg n\} + O(1)$ element comparisons are performed.

3. Transformation of heaps

We now describe how the binomial heap can be modified to support the *meld* operation at a constant cost using a transformation called *data-structural bootstrapping* [3, 4, 2, 14]. We use the structural abstraction technique of data-structural bootstrapping which bootstrap efficient data structures from less efficient data structures. Observe that in the heap context bootstrapping results in a structure where heaps contain heaps.

The transformed heap is represented by a binomial heap (as described in Section 2) containing one node. This node stores a pair which contains a reference to a node storing an element and a reference to a heap storing pairs. There is a back reference from the element node to the pair and from the heap to the pair. The back reference indicates that the pair is the *owner* of the element node and the heap. The element of a pair is no greater than any element stored within the heap of that pair. Furthermore, the order of the pairs within a heap is based on the elements associated with the pairs.

In the following the heap operations are described; Figure 2 describes the

```
find-min(Q):                                delete(x):
   if Q = ∅                                     (x, R) ← p ← owner[x]
      return nil                                Q ← parent[root(p)]
   (x, R) ← minimum[Q]                          if owner[Q] = nil and R = ∅
   return x                                        delete(p)
                                                   owner[x] ← nil
insert(Q, x):                                   elseif owner[Q] = nil and R ≠ ∅
   p ← construct pair                              (z, S) ← q ← find-min(R)
   if Q = ∅                                        delete(q)
      p ← (x, nil)                                 T ← meld(R, S)
      owner[x] ← p                                 p ← (z, T)
      insert(Q, p)                                 owner[z] ← p
      minimum[Q] ← (x, nil)                        owner[T] ← p
      return                                    else
   (y, R) ← q ← minimum[Q]                         delete(p)
   if R = nil                                      (z, Q) ← q ← owner[Q]
      R ← construct heap                           T ← meld(Q, R)
      owner[R] ← (y, R)                            q ← (z, T)
   if element[x] < element[y]                      owner[T] ← q
      p ← (y, nil)
      owner[y] ← p
      insert(R, p)                             meld(Q, R):
      owner[x] ← q                                if Q = ∅
      minimum[Q] ← (x, R)                            return R
   else                                            if R = ∅
      p ← (x, nil)                                    return Q
      owner[x] ← p                                 (x, S) ← minimum[Q]
      insert(R, p)                                 (y, T) ← minimum[R]
      minimum[Q] ← (y, R)                          if element[x] < element[y]
                                                      p ← (y, T)
                                                      insert(S, p)
                                                      minimum[Q] ← (x, S)
                                                   else
                                                      p ← (x, S)
                                                      insert(T, p)
                                                      minimum[Q] ← (y, T)
                                                   return Q
```

Figure 2. This pseudo code implements our heap operations. We use property maps in our interaction with objects, so an attribute is accessed using the attribute name followed by the name of the object in square brackets. The *construct* operation creates a new object. Given a node, the *root* operation returns the root of the tree in which the given node resides. Observe that we rely on automatic garbage collection.

same operations using pseudo code.

Let Q refer to the transformed heap given to the user. A minimum element is found within the pair (x, R) of Q where the element referenced by x is no greater than any element within R and thereby within the whole transformed heap. To reach x, a constant number of pointers has to be followed and therefore *find-min(Q)* can be accomplished at the worst-case cost of $O(1)$ using no element comparisons.

In *insert(Q, x)*, if the element referred to by x is smaller than the current minimum element referred to by y in the pair (y, R) of Q, x replaces y in this

pair and a new pair containing y and a reference to an empty heap is inserted into R. Otherwise, a pair containing x and an empty heap is inserted into R. On the basis of the above description and our earlier analysis, $insert(Q, x)$ has the worst-case cost of $O(1)$ and at most three element comparisons are performed, one when checking for a possible new minimum and two when inserting a pair into R.

In $delete(x)$, let x be a reference to the element to be deleted, (x, R) the pair containing x, and Q the heap containing this pair. Let us next consider the three cases of $delete$.

Case 1: Q is not owned by a pair and R is empty. The pair (x, R) is destroyed.

Case 2: Q is not owned by a pair and R is non-empty. A $find\text{-}min(R)$ is performed. Let (z, S) be the pair returned by $find\text{-}min(R)$. Now this pair is deleted from the heap R, the heaps R and S are melded using $meld(R, S)$, after which z and a reference to the melded heap forms the new pair replacing the pair (x, R).

Case 3: Q is owned by a pair. Let the pair owning Q be (z, Q). The pair (x, R) is deleted from Q, the heaps Q and R are melded using $meld(Q, R)$, after which z and a reference to the melded heap forms the new pair replacing the pair (z, Q).

The cost of $delete(x)$ is dominated by the cost of the binomial-heap operations $find\text{-}min$, $delete$, and $meld$. Therefore, $delete(x)$ has the worst-case cost of $O(\lg n)$ and the number of element comparisons performed is $3\lg n + O(1)$, of which $O(1)$ are performed by $find\text{-}min$, $2\lg n + O(1)$ by $delete$, and $\lg n + O(1)$ by $meld$.

In $meld(Q, R)$, the new minimum element is found by comparing the (minimum) element of the two pairs representing the two bootstrapped heaps. The pair referring to the non-smaller element is inserted into the heap of the other pair. The $meld(Q, R)$ operation has the worst-case cost of $O(1)$ and at most three element comparisons are performed, one when determining the new minimum and two during $insert$.

The following theorem summarizes the main result of the paper.

Theorem 1. *Let n be the number of elements in the heap prior to each operation. Our heap guarantees the worst-case cost of $O(1)$ for find-min, insert, and meld with at most 0, 3, and 3 element comparisons for the respective operations; and the worst-case cost of $O(\lg n)$ with at most $3\lg n + O(1)$ element comparisons for delete.*

4. Concluding remarks

We have described how data-structural bootstrapping can be used to transform a binomial heap supporting the *meld* operation at the worst-case cost of $O(\lg n)$ into a heap that supports melding at the worst-case cost of $O(1)$, and with a *delete* operation having the worst-case cost of $O(\lg n)$ and performing at most $3\lg n + O(1)$ element comparisons. Binomial heaps have

only been used as an example and other data structures like weak queues [6] and navigation piles [13] could, with some modifications, have been used instead. Other implementations of binomial heaps could also have been used; for example, the binomial heap described in [7]. To obtain the same comparison bounds as our heap implementation using other data structures as the basic component, it is important that the data structure used can perform the three operations *find-min*, *delete*, and *meld* using at most $3 \lg n + O(1)$ element comparisons in total, as these operations are the essential part of the *delete* operation in the bootstrapped heap.

Looking at the comparison complexity of heap operations, the following open questions still remain.

1. Is it possible to achieve a bound of $\lg n + O(1)$ element comparisons per *delete* when *meld* is required to have a constant cost? Note that, if *meld* is allowed to have a logarithmic cost, the worst-case bound of $\lg n + O(1)$ is achievable using the approach described in [7].

2. Given that *decrease* is added to the collection of operations and defined as follows:

   decrease(p, v): Replace the element stored at the node pointed to by p with element v, given that the new element is no greater than the old element.

   What would be the lowest possible number of element comparisons performed by *delete* Assuming that all other operations were required to have the worst-case cost of $O(1)$?

References

[1] G. S. Brodal, Fast meldable priority queues, *Proceedings of the 4th International Workshop on Algorithms and Data Structures*, Springer-Verlag (1995), 282–290.

[2] G. S. Brodal and C. Okasaki, Optimal purely functional priority queues, *Journal of Functional Programming* **6**, 6 (1996), 839–857.

[3] A. L. Buchsbaum, Data-structural bootstrapping and catenable deques, Ph. D. Thesis, Department of Computer Science, Princeton University, Princeton (1993).

[4] A. L. Buchsbaum, R. Sundar, and R. E. Tarjan, Data structural bootstrapping, linear path compression, and catenable heap ordered double ended queues, *SIAM Journal on Computing* **24**, 6 (1995), 1190–1206.

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

[6] A. Elmasry, C. Jensen, and J. Katajainen, Relaxed weak queues: An alternative to run-relaxed heaps, CPH STL Report **2005-2**, Department of Computer Science, University of Copenhagen (2005). Available at `http://cphstl.dk`

[7] A. Elmasry, C. Jensen, and J. Katajainen, Multipartite priority queues, *ACM Transactions on Algorithms* **5**, 1 (2008), Acticle 14.

[8] A. Elmasry, C. Jensen, and J. Katajainen, Strictly-regular number system and data structures, *Proceedings of the 12th Scandinavian Workshop on Algorithm Theory*, *Lecture Notes in Computer Science* **6139**, Springer-Verlag (2010), 26–37.

[9] A. Elmasry, C. Jensen, and J. Katajainen, Fast meldable heaps via data-structural transformations, Submitted for publication (2011).

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

[11] P. Høyer, A general technique for implemention of efficient priority queues, *Proceedings of the 3rd Israel Symposium on the Theory of Computing Systems*, IEEE (1995), 57–66.

[12] H. Kaplan and R. E. Tarjan, Thin heaps, thick heaps, *ACM Transactions on Algorithms* **4**, 1 (2008), Acticle 3.

[13] J. Katajainen and F. Vitale, Navigation piles with applications to sorting, priority queues, and priority deques, *Nordic Journal of Computing* **10** (2003), 238–262.

[14] C. Okasaki, *Purely Functional Data Structures*, Cambridge University Press (1998).

[15] J. Vuillemin, A data structure for manipulating priority queues, *Communications of the ACM* **21**, 4 (1978), 309–315.