Acta Informatica 22, 171–186 (1985)

An Algorithm for Merging Heaps*

Jörg-R. Sack [1] and Thomas Strothotte [2]

[1] School of Computer Science, Carleton University, Ottawa, Ont. K1S 5B6, Canada, USA
[2] INRIA Rocquencourt, B.P. 105, F-78153 Le Chesnay, France

Summary. We present an algorithm to merge priority queues organized as heaps. The worst case number of comparisons required to merge two heaps of sizes k and n is $O(\log(n)*\log(k))$. The algorithm requires $O(k+\log(n)*\log(k))$ data movements if heaps are implemented using arrays and $O(\log(n)*\log(k))$ for a pointer-based implementation. Previous algorithms require either $O(n+k)$ data movements and comparisons, or $O(k*\log(\log(n+k)))$ comparisons and $O(k*\log(n+k))$ data movements. The algorithm presented in this paper improves on the previous algorithms for the case when $k>\log(n)$.

1. Introduction

1.1. Priority Queues

One of the fundamental data-types in computer science is the priority queue. A *priority queue* is a set Q of keys and an operation defined on Q; each key k in Q has an associated priority $p(k)$ in the set of integers. The operations defined on priority queues are:

min: returns the element or the address of the element with smallest priority.
insert(k): adds the item containing the key k to the queue Q.
delete: removes the item containing the smallest key from the queue.
merge(Q, Q'): all elements of Q' are added to Q while Q' is destroyed.

Priority queues are called *non-mergeable* if merging is inefficient, i.e. merging should not require examining a positive fraction of the items in the queues [3]. Several organizations for mergeable and non-mergeable priority queues have been proposed, see [2, 3, 5, 8].

* This work was done while the authors were at McGill University, Montrêal, Canada

Brown's results [3] show that:

(1) Sorted linear lists are the best implementation of small priority queues, say less than 20 items.

(2) Heaps [4, 8, 11] are the best implementation of non-mergeable priority queues.

(3) Binomial queues [10] are an efficient implementation of mergeable priority queues.

(4) Pagodas [5] are efficient implementations of priority queues if the measure of efficiency considered is average case time-complexity.

A *heap* [11] is a tree which has the following properties: a) it is *heap-ordered*, that is, a key contained in any node is not greater than the keys of its offspring, and b) all leaves are on at most two adjacent levels, and all leaves on the last level are as far to the left as possible. A concise survey of known complexity results is given in [6]. We will call a heap with n elements an $n$heap, and restrict ourselves to heaps where each node has at most two siblings.

A remarkable feature of heaps is that they can be built in linear time using an algorithm due to [4]. The complexity analysis of Floyd's algorithm can be found in [1] and [7]. A worst-case performance and average case run-time study given by [3] shows that heaps are the most efficient implementation of priority queues when insertions and deletions are considered. When implemented using arrays, heaps are non-mergeable. Tarjan [9] also observes that "heaps are not easy to meld [merge]".

1.2. Merging Heaps

Two simple approaches to merging a heap of k elements, $k$heap, into a heap of n elements, $n$heap, are:

Algorithm 1. Insert the k elements into the heap of n elements one by one. Using the insertion algorithm of [7], this requires $O(k*\log(\log(n+k)))$ comparisons and $O(k*\log(n+k))$ data movements in the worst case. [Throughout this paper, "log" will stand for the logarithm to the Base 2.]

Algorithm 2. Neglect the structure in the $n$heap and $k$heap and reconstruct a new heap, requiring $2(n+k)$ comparisons and data movements.

Considering the total complexity, *Algorithm 2* is better when

$$k > \text{approximately } \frac{2n}{\log(\log(n))}.$$

If k approaches n in size, is advantageous to ignore the initial heaps and to construct a new heap of the $n+k$ elements from scratch. It is puzzling that a method which ignores the structure already being created should be faster than methods incorporating the initial heap. We will show that it is indeed possible to merge two heaps, making use of their structure.

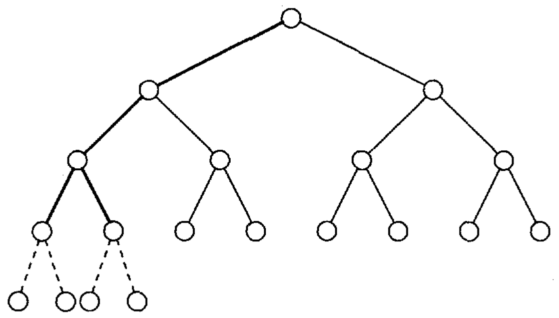
Fig. 1. Individually inserting 4 new elements into a heap with 15 elements. The positions to be filled are joined to the existing heap by dashed lines. The paths taken by the bubble-up operations is given in bold lines

1.3. *Overview of Algorithm*

Consider the task of sequentially inserting a small number of new elements into a large heap stored in an array. In the worst case, each of these insertions requires "bubbling up" the new element from the leaf-position to the root of the heap. The situation is illustrated in Fig. 1 for inserting 4 elements into a heap of 15 elements. Notice however, that the paths taken by bubbling up processes have some arcs in common. In the first part of our algorithm, we will ensure that when inserting k elements into the heap, this path will be traversed only once.

We will define a *perfect heap* as a heap with $2^i - 1$ elements, in which all leaves are on the same level, otherwise the heap is *non-perfect*. For the sake of clarity we will develop the algorithm by first showing how to merge two perfect heaps of equal sizes, then of unequal sizes. Next we will merge a perfect heap into a non-perfect heap, and finally we will merge two non-perfect heaps.

The algorithms will be based on an array implementation of heaps. The notation used is to make an element, p, in a heap synonymous with the subheap rooted at p. In the discussion, we will analyze the complexity of the algorithm for a pointer-based implementation.

2. Merging Perfect Heaps

2.1. *Merging two Perfect Heaps of Equal Sizes*

The algorithm merge_equal_perfect_heaps takes two heaps, $heap_1$ and $heap_2$, each of size k, and produces a new heap $heap_1$ with $2k$ elements.

```
procedure merge_equal_perfect_heaps($heap_1$,$heap_2$)
    invoke simple_merge on $heap_1$ and $heap_2$, taking
        the last element of $heap_2$ as the new root.
end merge_equal_perfect_heaps
```

```
**procedure** simple_merge($heap_1$,$heap_2$,newroot)
    copy $heap_1$ ro temporary location $t$
    place newroot at root of $heap_1$
    copy $t$ to left son of $heap_1$
    copy $heap_2$ to right son of $heap_1$
    tricke_down $heap_1$
**end** simple_merge
```

The *trickle_down* function restores the heap structure of its argument by allowing the root, which may be larger than either of its sons, to "trickle" down. Knuth [8, pp. 146–147] refers to this function as "sift_up".

Lemma 2.1. *Two perfect heaps of equal size k can be merged with $O(\log(k))$ comparisons and $O(k)$ data movements.*

Proof. The number of comparisons in the algorithm to merge two perfect heaps of equal size is dominated by the trickle_down operation, which requires $O(\log(k))$ comparisons [8]. The number of data movements required is dominated by the operation of copying the two small heaps and the resulting heap ($heap_1$), requiring $O(k)$ data movements. The correctness follows immediately. □

2.2. Merging Perfect Heaps of Different Sizes

We will consider the simple case of inserting a perfect heap of k elements, $k$heap, into a perfect heap with n elements, $n$heap. Without loss of generality, assume that $k \leq n$.

Referring to Fig. 1, recall that the bubble_up operations of individual insertions would share a common path. We proceed as follows: Begin with the root of $n$heap, and compare it to the root of the $k$heap. If the root of $k$heap is smaller than the root of $n$heap, exchange the two roots and perform the trickle_down operation on the $k$heap. This step is then repeated for successive elements on the path which all individual bubble_ups would have in common. This procedure is referred to as walk_down and the path is referred to as the walk_down path. Although $n$heap and $k$heap are altered throughout the execution of procedure walk_down, we will continue referring to these modified heaps as $n$heap and $k$heap, respectively.

```
**procedure** walk_down($n$heap,from,to,$k$heap)
    **if** $n$heap(from) < $k$heap(root)
        **then**
            exchange($n$heap(from),$k$heap(root))
            trickle_down ($k$heap)
    **if** from = to **then** return
    **else**
        next: = {next node on path from *from* to *to*}
        walk_down($n$heap,next,to,$k$heap)
**end** walk_down
```

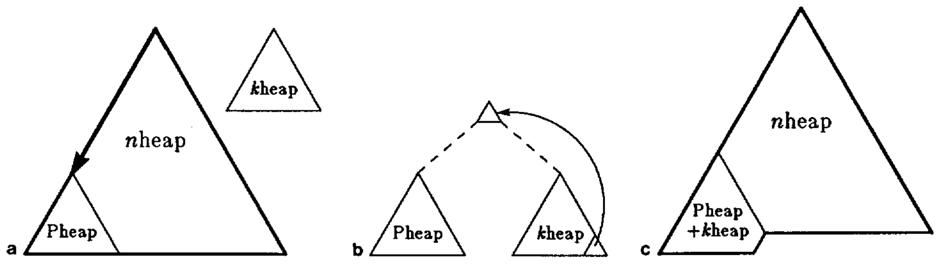

Fig. 2a–c. Algorithm for merging two perfect heaps. **a** The leftmost subheap of size k in n heap is identified. It is referred to as Pheap. **b** Pheap is merged with k heap to make a new heap of size $2k$. **c** The heap resulting from (b) is placed back into n heap

Notice that given the indices of the starting and finishing positions of the path, the indices of the intermediate nodes (successive *nexts* on the path) can be computed by examining the bits in the binary representation of *to* and *from* (see for example [8, p. 154]).

Since $k$heap and $n$heap are perfect, there are exactly k elements in the subheap rooted at p, which we will refer to as Pheap (see Fig. 2(a)).

Using procedure merge_equal_perfect_heaps, we can now merge $k$heap and Pheap to form a new subheap of $n$heap. The complete algorithm for merging perfect heaps is given below and is illustrated in Fig. 2.

```
procedure merge_perfect_heaps(nheap,kheap)
    p:={common parent of positions in nheap to be filled}
    walk_down(nheap,nheap(root),p,kheap)
    (* now merge the subheap rooted at p with kheap *)
    merge_perfect_equal_heaps(nheap(p),kheap)
end merge_perfect_heaps
```

Lemma 2.2. *After execution of procedure walk_down:* (a) *all ancestors of p are less than or equal to all elements in kheap,* (b) *the heap structure of nheap and kheap are maintained.*

Proof. By induction, we will show that after the i'th element on the walk_down path has been considered, all ancestors of i are less than or equal to all elements in $k$heap and the heap structures of $n$heap and $k$heap are maintained.

For $i=1$, the root of $n$heap, the result follows trivially. Let $i>1$. By induction the elements 1 to $i-1$ on the path are less than or equal to the i'th element and less than or equal to all elements in $k$heap. In processing the i'th element two cases arise:

(1) i'th element$>$root($k$heap): The walk_down procedure exchanges the i'th and the root of $k$heap thereby replacing i by a smaller element, which by induction is nonetheless $\geq$ to the $i-1$'st element. This maintains the heap structure of $n$heap. Using procedure trickle_down the heap structure of $k$heap is subsequently restored.

(2) i'th element$\leqq$root($k$heap): No exchange is performed and thus the result follows trivially.

The proof is completed by letting i equal to p. □

Lemma 2.3. *Two perfect heaps of sizes n and k can be merged with* $O(\log(n)*\log(k))$ *comparisons and* $O(k+\log(n)*\log(k))$ *data movements.*

Proof. The correctness of the algorithm follows directly from Lemmas 2.1 and 2.2. The length of the path traversed by the walk_down procedure is at most $\log(n)$. For each node on the path, restoring of $k$heap at a cost of $\log(k)$ comparisons may be required (trickle_down). Thus the number of comparisons as well as data movements for walk_down is $O(\log(n)*\log(k))$. From Lemma 2.1 follows that the number of comparisons for the algorithm to merge two perfect heaps of sizes n and k is dominated by the time it takes to execute the walk_down procedure. Thus the total number of comparisons is $O(\log(n)*\log(k))$. The number of data movements is $O(\log(n)*\log(k))$ for the walk_down procedure and by Lemma 2.1, $O(k)$ for merging equal sized heaps. The total number of data movements is thus $O(k+\log(n)*\log(k))$. □

In this proof of Lemma 2.3, the length of the walk_down path was approximated by $O(\log(n))$. However, a better approximation is $O(\log(n) -\log(k))$ which reduces to $O\left(\log\frac{n}{k}\right)$. Analogously to the proof it follows that two perfect heaps of sizes n and k can be merged with $O\left(\left(\log\left(\frac{n}{k}\right)+1\right)*\log(k)\right)$ comparisons and $O\left(k+\left(\log\left(\frac{n}{k}\right)+1\right)*\log(k)\right)$ data movements. Thus for $k=O(n)$, two perfect heaps of sizes n and k can be merged in $O(\log(n))$ time (see Lemma 2.1).

Up to now, we have seen how to merge two perfect heaps. The ideas remain the same when the heaps are not perfect, although the details are somewhat more complex.

3. Merging a Perfect Into a Non-Perfect Heap

We will now discuss how to merge a perfect $k$heap into a non-perfect $n$heap. Let the *size* of a heap (or subheap) refer to the number of elements it contains, and the *height* be defined as $\lfloor\log(size(heap))\rfloor$. We will use a function h(heap) to return the height of *heap*. We will define *slots* to be those leaf positions in $n$heap which are to be filled by the merging process. We will say that a node p *covers* a group of *slots* if all slots are descendents of p.

A problematic situation is illustrated in Fig. 3, where $k=3$ and $n=22$. Here the lowest common ancestor of all k slots is the root of $n$heap. In some cases, this common ancestor may be somewhere in the middle of the tree.

We first apply the walk_down procedure to the lowest common ancestor, p, of all k slots. If the subheap, Pheap, rooted at p contains k or $k+1$ elements, then the merge is analogous to that for merge_perfect_heaps. The last element of Pheap is taken as the root and then trickle_down operation applied. The procedure is completed in this case.

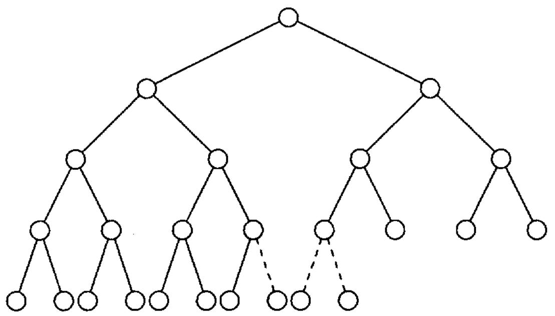
Fig. 3. Inserting 3 elements into a heap of size 22. The positions to be filled are joined to the existing heap by dashed lines. In this case, the only common ancestor of the slots to be filled is the root

If the subtree rooted at p is deeper than $k$heap, we perform the procedure find_2_nodes as described below to find nodes p_l and p_r. The existence of p_l and p_r will be established later in Lemma 3.1. They will be such that

(1) the subtree rooted at p_l has height $h+1$,
(2) the subtree rooted at p_r has height h, and
(3) p_l and p_r together cover all slots.

Note that in an array implementation p_l and p_r lie in adjacent positions if $k = 2^i - 1$. We will discuss only the case where $n$heap and merge($n$heap, $k$heap) have the same height. Otherwise a simple modification of exchanging the roles of p_r and p_l can be made to the algorithm.

```
**function** find_2_nodes(nheap,p,h)
    (* function returns two subheaps of nheap *)
    (* each of size $O(k)$ and which *)
    (* together cover all slots *)
    **if** h(nheap(p)) $\leq h+1$ **then** return(p,p)
    left := left son of p
    right := right son of p
    **while** ($h$(left) $> h+1$) **do**
        left := right son of left
        right := left son of right
    **repeat**
    return(left,right)
**end** find_2_nodes
```

An example is illustrated in Fig. 4. We will refer to the subheaps rooted at p_l and p_r as $p_l$heap and $p_r$heap, respectively. We now apply a double_walk_down on the paths from p to p_l and p to p_r. This consists of using the procedure walk_down described earlier to traverse the paths from

(1) p to p_l, at each node on the path updating the $k$heap as necessary,
(2) p to p_l at each node on the path updating the $p_r$heap as necessary,
(3) p to p_r at each node on the path updating the $k$heap as necessary,
(4) p to p_r at each node on the path updating the $p_l$heap as necessary.

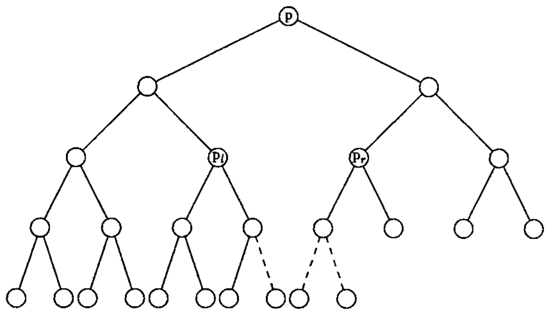

Fig. 4. For the case of $k=3$ and $n=22$, two subheaps p_l and p_r, both with $O(k)$ elements, are identified for subsequent insertions. Only the elements lying below these subheaps are affected by the insertion of the k new elements

These four steps are necessary so that later in the algorithm we can exchange the subheaps $p_l$heap, $p_r$heap and $k$heap without violating the heap condition.

```
procedure double_walk_down($n$heap,$from_1$,$to_1$,$from_2$,$to_2$,$k$heap)
    walk_down($n$heap,$from_1$,$to_1$,$k$heap)
    walk_down($n$heap,$from_1$,$to_1$,$to_2$)
    walk_down($n$heap,$from_2$,$to_2$,$k$heap)
    walk_down($n$heap,$from_2$,$to_2$,$to_1$)
end
```

This procedure is invoked as

double_walk_down($n$heap,p,p_l,p,p_r,$k$heap).

We now build two new heaps out of $p_l$heap, $p_r$heap and $k$heap. As by assumption of this section, $k$heap is perfect, it can be inserted into $n$heap directly using the following procedure. Apply simple_merge to $p_r$heap and $k$heap, this time taking the last element of $p_l$heap as the extra node for the new root. Recall that $p_r$heap and $k$heap both are of height h. The result is a perfect heap rooted at p_r of height $h+1$. We will refer to as kPheap. Now exchange the heaps rooted at p_l and p_r. This fills up the leaves at the bottom level under p_l, and maintains the heap-structure under p_r, with the empty positions coming on the right side. The description of the procedure to merge a perfect $k$heap into a non-perfect $n$heap is now complete.

```
procedure merge_perfect_into_non_perfect($n$heap,$k$heap)
    $p$:= {lowest common parent of slots in $n$heap to be filled}
    walk_down($n$heap,$n$heap(root),$p$,$k$heap)
  if (h($p$) = h($k$))
      then
          simple_merge($p$,$k$heap,{last element of $k$heap})
      else if h($p$) = h($k$heap) + 1
          then
              simple_merge($p$,$k$heap,{last element of $p$})
```

```
        else
            $(p_l,p_r)$:=find_2_nodes($n$heap($p$),$p$,h($k$heap))
            simple_exchange_merge($n$heap,$p_l$,$p_r$,$k$heap)
    end merge_perfect_into_non_perfect
    procedure simple_exchange_merge($n$heap,$p_l$,$p_r$,$k$heap)
        double_walk_down($n$heap,$p$,$p_l$,$p$,$p_r$,$k$heap)
        simple_merge($n$heap($p_r$),$k$heap,{last element of $p_l$})
        exchange($p_l$,$p_r$)
    end simple_exchange_merge
```

Lemma 3.1. *Let $k \leqq n$ denote the size of a perfect heap and h its height. Then a non-perfect heap of size n contains a subheap of height $h+1$ and a perfect subheap of height h, together covering all k* slots.

Proof. As $k \leqq n$, all slots are on at most two adjacent levels in $n$heap. We assume all slots to be on the same level, as the argument is analogous in the other case. Slots are consecutive, thus any single subheap Pheap of $n$heap covering all k slots has height θ at least as large h. As $k$heap is perfect, $h = \log(k+1)-1$. We distinguish between three cases: (a) $\theta = h$, (b) $\theta = h+1$ and (c) $\theta \geqq h+2$.

(a) $\theta = h$. In this case Pheap is perfect (no leaves of Pheap are on the same level as the slots).

(b) $\theta = h+1$. Let $l > 0$ denote the number of nodes of Pheap that are located on the last level of $n$heap. As $l + k \leqq 2^{h+1}$, it follows that $l = 1$. In this case there is one leaf on the last level in Pheap and all remaining nodes on that level are slots. Thus Pheap is non-perfect and of height $h+1$, while the right son of Pheap is perfect and of height h.

(c) $\theta \geqq h+2$. As Pheap covers all slots, the left and right sons of p together also cover all slots. The subheaps are of height $\theta - 1$ and $\theta - 2$, respectively and the right subheap is perfect. Let k_l, k_r be the number of slots covered by the left subheap and the right subheap of p, respectively. As k_r is less than k, there exists a node p_r in the right subheap of Pheap such that all k_r slots are covered by a perfect subheap of height h. Similarly, there exists some subheap p_l of height $h+1$ in the left subheap of Pheap such that all slots in k_l are covered by p_l. Thus, there exist subheaps in $n$heap, one non-perfect of height $h+1$, the other perfect and of height h, which together cover all slots. □

Lemma 3.2. *Let p be the root of a subheap Pheap of $n$heap covering all slots and h be the height of $k$heap. The algorithm find_2_nodes finds 2 subheaps, p_l and p_r of Pheap, such that*

(1) both have height h or $h+1$, and

(2) every slot is covered by either p_l or p_r.

The algorithm runs in $O(\log(n))$ steps.

Proof. If the height of p is $h+1$, the result follows trivially.

Therefore assume $\mathrm{h}(p) > h+1$. As p covers all slots, so do the left son l and the right son r of p together. As p was the lowest common ancestor of all slots, the right-most descendant $plson_r$ of l, and the leftmost descendant $prson_l$ of r, are both slots. They are also adjacent. Because slots are in consecutive locations, there are at most $k-1$ slots to the left of $plson_r$, which are covered by the h'th ancestor of $plson_r$. This subheap, which has height $h+1$, will be referred to this as p_l. A similar argument holds for $prson_l$, showing that there exists a subheap of height h under $prson_l$ covering those slots not covered by $plson_r$.

The locations p_l and p_r can be found without any key comparisons. The length of the path from p to p_l or p_r is at most $\log(n)$ and thus the algorithm runs in $O(\log(n))$ steps. □

Lemma 3.3. *After procedure double_walk_down in simple_exchange_merge is executed* (a) *all ancestors of p_l and those of p_r are less than or equal to all elements in kheap, $p_l$heap and $p_r$heap,* (b) *the heap structures of nheap and kheap are maintained.*

Proof. After the first step in the algorithm, all ancestors of $p_l$heap are less than or equal to all elements in $k$heap (Lemma 2.2). In the second step, all ancestors in $p_r$heap become less than or equal to the elements of $p_l$heap. By analogy, the third step makes all ancestors of $p_r$heap less than or equal to elements in $k$heap. The final step then makes all ancestors of $p_r$heap less that or equal to all elements in $p_l$heap. □

Lemma 3.4. *The double_walk_down procedure requires $O(\log(n)*\log(k))$ comparisons and data movements.*

Proof. The procedure uses at most 3 times as many comparisons as the walk_down procedure. The result follows directly from Lemma 2.3. □

Lemma 3.5. *A perfect heap of size k can be merged into a non-perfect heap with $O(\log(n)*\log(k))$ comparisons and $O(k+\log(n)*\log(k))$ data movements.*

Proof. By Lemma 3.3, all ancestors of $p_l$heap and $p_r$heap are less than or equal to the roots of each of $p_l$heap, $p_r$heap and $k$heap. Merging $p_r$heap and $k$heap as described produces a perfect heap of height $h+1$, which is also the height of the non-perfect heap $p_l$heap. The correctness of this Step follows from Lemma 2.1. As a result of Lemma 3.3, kPheap and $p_l$heap may be exchanged while still maintaining the heap structure of $n$heap. The exchange operation fills all slots on the left, restoring the heap structure of $n$heap.

The positions p_l and p_r can be found in $O(\log(n))$ time (Lemma 3.2). By Lemma 3.4, the double_ walk_ down procedure costs $O(\log(n)*\log(k))$ comparisons and data movements (Lemma 2.1). Merging two perfect heaps of sizes k ($k$heap and $p_r$heap) requires $O(\log(k))$ comparisons and $O(k)$ data movements, and exchanging two heaps, p_l and p_r, each of size $O(k)$, requires $O(k)$ data movements (Lemma 2.1). Thus the total number of comparison is $O(\log(n)*\log(k))$ and the total number of data movements is $O(k+\log(n)*\log(k))$. □

Note that once p_l and p_r are found, we could ignore the structures of $k$heap and the subheaps rooted at p_l and p_r. Using Floyd's heap construction algorithm, we would build two new heaps, one perfect to fit under p_l and another possibly non-perfect to fit under p_r. This would require $O(k)$ data movements and $O(k)$ comparisons. Using this algorithm, the total complexity would not be altered, but the number of comparisons would increase from $O(\log(n) * \log(k))$ to $O(k)$.

4. Merging two Non-Perfect Heaps

Initially we find p, the lowest common ancestor of all slots, as before and perform walk_down from the root of $n$heap to p. We say that a heap $k$heap *fits* under a subheap Pheap if Pheap has enough slots for each element in $k$heap.

(1) Find the locations p_l and p_r as before. The corresponding heaps are of size $O(k)$ and cover all slots.

(2) Split $k$heap into a perfect heap, Δ, and a possibly non-perfect heap, Δ^+. This is done by removing the root r and placing it into the next available leaf position, and restoring the heap (see Fig. 5). Alternatively, with no key comparisons, we can move all elements on the path from the available leaf position to the root of the corresponding heap down one by one. Then we insert the element r at the root position of the heap. The heap Δ^+ is of height one or two less than the $k$heap.

(3) Merge Δ into p_l or p_r, depending on whether p_l has enough slots for perfect_heap. The procedure is analogous to merging a perfect heap into a non-perfect one filling up slots, beginning with the leftmost.

(4) Recursively merge Δ^+ using p_l and p_r.

More formally, the algorithm is as follows:

```
**Preprocessing:**
    find $p$, the lowest common ancestor of all slots
    walk_down($n$heap,root($n$heap),$p$,$k$heap)
**procedure** merge_non_perfect($n$heap,$p$,$k$heap)
    $(p_l, p_r)$:=find_2_nodes($n$heap,$p$,h($k$heap))
    **if** $k$heap is perfect **then**
        simple_exchange_merge($n$heap,$p_l$,$p_r$,$k$heap)
        **return**
    split_and_insert($p_l$,$p_r$,$k$heap)
    **procedure** split_and_insert($p_l$,$p_r$,$k$heap)
        double_walk_down to $(p_l, p_r)$
        split $k$heap into $\Delta$ and $\Delta^+$
        (* $\Delta$ denotes the perfect heap, and $\Delta^+$ the possibly non_perfect one *)
        (* if both parts are perfect, let $\Delta$ be the smaller *)
```

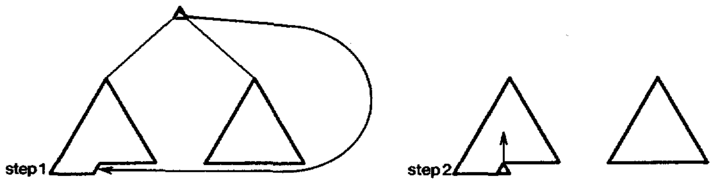

Fig. 5. Algorithm for splitting a heap. Step 1: The root is removed and placed into the next available position. Step 2: The left heap is restored by bubbling up the former root

```
        if $\Delta$ fits under $p_l$
            then
                merge_perfect_into_non_perfect($n$heap,$p_l$,$k$heap)
                if $p_l$ is now perfect
                    then
                        merge_non_perfect($n$heap, $p_r$, $\Delta^+$)
                else
                    if $p_l \neq p_r$
                        then
                            (* we can lower $p_l$ and $p_r$ because *)
                            (* h($\Delta^+$) is exactly 1 smaller than h($k$heap) was *)
                            double_walk_down($n$heap,$p_l$,$p_l$.right,$p_r$,$p_r$.left,$\Delta^+$)
                            $p_l$:=$p_l$.right
                            $p_r$:=$p_r$.left
                            split_and_insert($p_l$,$p_r$,$\Delta^+$)
                    else
                        if left son of $p_l$ is perfect
                            then
                                walk_down($n$heap,$p_r$,$p_r$.right,$\Delta^+$)
                                merge_non_perfect($n$heap,$p_r$.right,$\Delta^+$)
                        else
                            merge_non_perfect($n$heap,$p_r$,$\Delta^+$)
        else
            (* here $p_l \neq p_r$ otherwise $\Delta$ would have fit under $p_l$.*)
            (* now lower $p_l$ and $p_r$ *)
            insertion$p_l$:=$p_l$.right;
            insertion$p_r$:=$p_r$.left;
            if h($\Delta$)=h($k$heap)$-2$
                then
                    insertion$p_l$:=insertion$p_l$.right
                    insertion$p_r$:=insertion$p_r$.left
            simple_exchange_merge($n$heap,insertion$p_l$,insertion$p_l$,$\Delta$);
            merge_non_perfect($n$heap,$p_r$,$\Delta^+$)
    end split_and_insert
end merge_non_perfect
```

Before proving the correctness of the algorithm, we will first establish a number of properties of the heaps to be merged. Recall that we say that if a heap *fits* under another, then the second heap contains at least as many slots as there are elements in the first heap.

Lemma 4.1. *If the perfect heap, Δ, does not fit under p_l, then the left subheap of p_l is perfect.*

Proof. Let h denote the height of $k$heap. The height h(Δ) is either $h-1$ or $h-2$ and thus the size of Δ is 2^h-1 or $2^{h-1}-1$, respectively. Assume that h(Δ)$=h-1$. The subheap rooted at p_l is of height $h+1$ and has at most 2^{h+1} leaves. If Δ does not fit under p_l then the number of slots is less than 2^h-1 and thus the left subheap of p_l is perfect. A similar argument holds for the case that $h(\Delta)=h-2$. □

Corollary 4.1. *If the perfect heap, Δ, does not fit under p_l, the right subheap of p_l contains at least one element.*

Corollary 4.2. *If the perfect heap, Δ, does not fit under p_l and $h(\Delta)=h(k\text{heap})-2$, the left subheap of p_l as well as the left subheap of the right subheap of p_l are perfect.*

Corollary 4.3. *If the perfect heap, Δ, does not fit under p_l and $h(\Delta)=h(k\text{heap})-2$, the right subheap of the right subheap of p_l contains at least one element.*

Lemma 4.2. *The algorithm merge_non_perfect correctly merges two non-perfect heaps.*

Proof. On entry to procedure split_and_insert, p_l and p_r cover all slots. We will show that for each recursive call to split_and_insert, this assumption remains true, and that with each successive call, the height of the heap remaining to be inserted decreases by at least one. Two cases arise:

(a) Δ fits under p_l, and

(b) Δ does not fit under p_l.

(a) Δ fits under p_l: By Lemma 3.5, Δ is correctly merged under p_l. Two cases can now arise:

(1) p_l is perfect. All the remaining slots are now under p_r, and the algorithm recurses correctly.

(2) p_l is still not perfect. Now $h(\Delta^+)=h(k\text{heap})-1$, so that p_l and p_r can be lowered, so that if $p_l \neq p_r$, the recursive step can be applied correctly. If $p_l=p_r$ then if the left son of p_l covers all slots, then p_l and p_r may be lowered, otherwise we require that p_r is still the lowest node covering all slots.

(b) Δ does not fit under p_l: From Lemma 4.1 and Corollaries 4.1–4.3, we can conclude that there exist two subheaps insertionp_l and insertionp_r with the following properties:

(1) insertionp_l and insertionp_r together cover all the leftmost size (Δ) slots, and

(2) the subheap insertionp_r is perfect and of height $h(\Delta)$ while the subheap insertionp_l is non-perfect and of height $h(\Delta)+1$.

The algorithm identifies these subheaps insertionp_l and insertionp_r.

Following this, the algorithm merges Δ using insertionp_l and insertionp_r with the procedure simple_exchange_merge, whose correctness was established by Lemma 3.5. The result is that p_l is perfect. All remaining slots thus lie under p_r and are filled by recursively invoking the procedure merge_non_perfect. □

Lemma 4.3. *The algorithm merge_non_perfect makes $O(k+\log(n)*\log(k))$ data movements and $O(\log(n)*\log(k))$ comparisons.*

Proof. Walking down from the root of $n$heap to p_l and p_r requires $O(\log(n)*\log(k))$ comparisons and data movements, as in merging a perfect heap into a non-perfect heap. The remaining Steps are now independent of n.

(1) Splitting the $k$heap into 2 heaps is done with zero key comparisons and $O(\log(k))$ data movements.

(2) Let h denote the height of $k$heap. The algorithm merges the perfect heap, of height $h-1$ or $h-2$, with either $p_l$heap or $p_r$heap, of heights $h+1$ and h, respectively. Thus the path length for the walk_down from p_l or p_r is constant (at most 2), so that this walk_down costs $O(\log(k))$ comparisons and data movements. Then the actual merge requires a further $O(\log(k))$ comparisons and $O(k)$ data movements.

(3) As the height of the non-perfect heap to be inserted is decremented by one with each level of recursion, Steps 1-3 are repeated $\log(k)$ times. Note that the complexity is dominated by Step 3. The maximum number of elements in the non-perfect heap to be inserted decreases by a factor of between 2 and 4 with every level of recursion. Therefore the worst case number of comparisons is:

$$O\left(\sum_{i=0}^{\lfloor\log(k)\rfloor} \log\left(\frac{k}{2^i}\right)\right)=O(\log^2(k)),$$

and the number of data movements is:

$$O\left(\sum_{i=0}^{\lfloor\log(k)\rfloor} \frac{k}{2^i}\right)=O(k).$$

Thus the total number of comparisons to merge two non-perfect heaps is $O\left(\log\left(\frac{n}{k}\right)*\log(k)+\log^2(k)\right)$, which reduces to $O(\log(n)*\log(k))$. The total number of data movements is $O(k+\log(n)*\log(k))$. This result applies to the array implementations of heaps. □

The description of the algorithm to merge two heaps is now complete.

5. Results

Theorem 5.1. *Two heaps with n and k elements, respectively, can be merged in $O(k+\log(n)*\log(k))$ data movements and $O(\log(n)*\log(k))$ comparisons if implemented using arrays.*

Proof. The proof follows directly from Lemmas 4.2 and 4.3. □

Theorem 5.2. *Two heaps with n and k elements, respectively, can be merged with $O(\log(n)*\log(k))$ data movements and comparisons if implemented using pointers.*

Proof. From Theorem 4.1, the number of comparisons for merging two heaps is $O(\log(n)*\log(k))$. It can be verified from the previous Lemmas that the number of data movements is equal to the number of comparisons for all operations performed by the algorithms for all operations except for exchanging heaps. In an array implementation exchange operations require time linear in the number of elements to be moved. In a pointer-based implementation however, two heaps can be exchanged by reassigning the respective pointer values which takes constant time. □

6. Comparison With Other Algorithms

The worst-case performance for previous algorithms for merging heaps of sizes n and k is

(1) $O(n+k)$ comparisons and data movements for rebuilding the heaps from scratch, and

(2) $O(k*\log(\log(n+k)))$ comparisons and $O(k*\log(n+k))$ data movements for individual insertions using Gonnet's algorithm.

The new algorithm presented here, using $O(\log(n)*\log(k))$ comparisons and $O(k+\log(n)*\log(k))$ data movements, clearly performs better than Algorithm (1) in all cases. Our algorithm also performs better than Algorithm (2) when

$$\frac{k}{\log(k)} > \frac{\log(n)}{\log(\log(n+k))},$$

i.e., when

$$k > \log(n),$$

roughly speaking.

Our algorithm for merging heaps is nonetheless not as good as algorithms for merging other implementations of priority queues. For example, leftist-trees, binomial queues and 2–3 trees can all be merged in $O(\log(n))$ time [3, p. 23].

7. Inserting k Elements to a Heap of n Elements

An interesting by-product of our algorithm is an algorithm to insert a batch of k elements into a heap of n elements:

(1) Create a heap $k$heap of the k new elements; this costs $O(k)$ comparisons and data movements.

(2) Merge the $k$heap and the $n$heap; using our algorithm this costs $O(\log(n)*\log(k))$ comparisons.

The total number of comparisons and data movements for this algorithm is $O(k+\log(n)*\log(k))$. This compares to a worst-case complexity of $O(k+\log(n+k))$ for other priority-queue organizations like leftist-trees, binomial queues and 2–3 trees.

8. Discussion

Heaps can be efficiently stored in arrays, whereas a pointer-based implementation requires 2 pointers per node (left son and sight son). Thus more storage is required in a pointer-based implementation of heaps. The path from the root to an element is described by its binary representation. Associated with every heap is its size, and thus the n'th or $(n+1)$'st elements, needed for insertions and deletions, can be found in $O(\log(n))$ time. If only 2 pointers per node are used, a stack of depth $O(\log(n))$ is required for these operations. If 3 pointers per node are used, then no extra stack is required. In any case, the time-complexity for insertion and deletion remains the same for a pointer-based implementation as for an array-based implementation.

From the theoretical point of view, it is important to observe that the number of data movements is implementation dependent. A heap of size k can be re-linked to another location in constant number of data movements if the heaps are implemented using pointers, compared to $O(k)$ movements for an array implementation. Therefore the number of data movements required to merge two heaps of sizes n and k is $O(\log(n)*\log(k))$ for a pointer-based implementation as opposed to $O(k+\log(n)*\log(k))$ for an array implementation. The number of comparisons is implementation independent.

Using a pointer-based implementation, heaps are mergeable.

References

1. Aho, A.V., Hopcroft, J.E., Ullman, J.D.: The design and analysis of computer algorithms, p. 99. Reading, MA: Addison-Wesley 1974
2. Aho, A.V., Hopcroft, J.E., Ullman, J.D.: Data structures and algorithms. Reading, MA: Addison-Wesley 1983
3. Brown, M.R.: The analysis of a practical and nearly optimal priority queue. New York: Garland Publishing 1980
4. Floyd, R.W.: Algorithm 245, Treesort 3. CACM **7**, 701 (1964)
5. Francon, J., Viennot, G., Vuillemin, J.: Description and analysis of an efficient priority queue representation. Proc. 19th Ann. Symp. Found. Comput Sci. MI: Ann Arbor, pp. 1-7 (1978)
6. Gonnet, G.H.: A handbook of algorithms and data structures. Reading, MA: Addison-Wesley 1984
7. Gonnet, G.H., Munro, I.J.: Heaps on Heaps. Proc. ICALP, Aarhus 9, pp. 282-291 (July 1982)
8. Knuth, D.E.: The art of computer programming. Vol. 3: Sorting and searching. Reading, MA: Addison-Wesley 1973
9. Tarjan, R.E.: Data Structures and Network Algorithms. Philadelphia, PA: Soc. Ind. Appl. Math. 1983
10. Vuillemin, J.: A data structure for manipulating priority queues. CACM **21**, 309-315 (1978)
11. Williams, J.W.J.: Algorithm 232, heapsort. CACM **7**, 347-348 (1964)

Received August 1, 1983/March 8, 1984