INFORMATION AND COMPUTATION **86**, 69–86 (1990)

A Characterization of Heaps and Its Applications

JÖRG-RÜDIGER SACK*

School of Computer Science, Carleton University, Colonel By Drive, Ottawa, Ontario, Canada K1S 5B6

AND

THOMAS STROTHOTTE[†]

Institut für Informatik, Universität Stuttgart, Azenbergstrasse 12, D-7000 Stuttgart 1, Federal Republic of Germany

In this paper we present a *new view* of a classical data structure, the *heap*. We view a heap on n elements as an ordered collection of $\lceil \log_2(n+1) \rceil$ substructures of sizes 2^i with i in $\{0, ..., \lfloor \log_2(n) \rfloor\}$. We use the new view in the design of an *algorithm for splitting* a heap on n elements into two heaps on k and $n-k$ elements, respectively. The algorithm requires $O(\log^2(n))$ comparisons, improving the previous bound of $O(k)$ comparisons for all but small values of k, i.e., for $k > \log^2(n)$. We also present a new and conceptually simple *algorithm for merging* heaps of sizes n and k into one heap of size $n+k$ in $O(\log(n) * \log(k))$ comparisons. © 1990 Academic Press, Inc.

1. INTRODUCTION

1.1. *Background*

A (min-)heap (Williams, 1964) is a binary tree with the following properties:

(1) it is *heap-ordered*, i.e., a key contained in any node is not greater than the keys of its offspring, and

(2) it is *heap-shaped*, i.e., all leaves are on at most two adjecent levels, all leaves on the last level are as far to the left as possible, and all other levels are complete. A heap is an implicit data structure in the sense of (Munro *et al.*, 1980).

We refer to the number of elements in the heap as its *size*. The *height* of

* Research supported by the NSERC under Grant A0332.

[†] Present address: IBM Heidelberg, Scientific Center, Tiergartenstrasse 15, D-6900 Heidelberg, Federal Republic of Germany.

TABLE I

Traditional Heap-operations and Their Complexities

Operation	Description	Comparisons
Create	Construct a heap	$O(n)$
FindMin	Find element with minimum key	$O(1)$
DeleteMin	Remove element with minimum key	$O(\log(n))$
Insert	Insert a new element	$O(\log(n))$

the heap is $\lfloor \log(\text{size}) \rfloor$. As usual, log stands for $\log_2$ and, whenever integer values are expected, we will use the floor function, e.g., as in $\lfloor \log(\text{size}) \rfloor$.

A heap is an efficient implementation of the data type *priority queue*, as shown in Brown (1980) and Gonnet (1984). The operations shown in Table I are commonly performed on heaps (see Floyd, 1964; Knuth, 1973; Gonnet, 1984).

By exploiting the property that elements on a path from the root of a heap to a leaf occur in sorted order, the number of comparisons for inserting a new element into a heap can be reduced to $O(\log\log(n))$ (Gonnet and Munro, 1986).

In this paper, we consider two additional operations on priority queues and show how these can be implemented using heaps. These operations are given in Table II.

In addition to being of theoretic interest, the operations *Split*(*heap*, k) and *Merge*(*heap*1, *heap*2) are useful in the context of a multi-processor system where processors are deactivated (e.g., because of failure) and subsequently reactivated (e.g., start-up after repair). Each processor has associated with it a priority queue of jobs and the queue of any deactivated processor must be merged with the queue of some other processor. Subsequently, after reactivation of the processor its priority queue is obtained by splitting some queue into two. In this case, the maximum queue size, k, of the newly activated processor, will be given as the second parameter for the *Split*-operation; two queues of sizes k and $n-k$, respectively, are produced.

A related *Merge* operation for priority queues has previously been studied (see Aho *et al.*, 1974, pp. 152–157). The authors designed a *mergeable heap* which consists of a 2-3 tree with a relaxation of the left-to-right

TABLE II

Description of the Operations *Split* and *Merge*

Merge	Merge two heaps of sizes n and k
Split	Split a heap into two heaps of sizes k and $n-k$

ordering condition on the leaves. Only the value of the minimum key of a subtree is recorded at its root. Such heaps can be merged in $O(\log(n))$ time, but cannot be stored implicitly.

In a similar vein, Aho, *et al.* studied a *Split* operation which is related to the one we consider in this paper. Their operation *Split*(a, S) partitions a set S into two sets $S_1 = \{b \in S \mid b \leqslant a\}$ and $S_2 = \{b \in S \mid b > a\}$. They proposed *concatenable queues*, also based on 2-3 trees, to implement this operation in $O(\log(n))$ time. Two such concatenable queues S_1, S_2 can be merged in $O(\log(n))$ time, provided that all elements of S_1 are less than all elements of S_2. However, $O(n \log n)$ time is required to construct such queues and they cannot be stored implicitly.

We define a *perfect heap* as a heap of size $2^i - 1$, while heaps of all other sizes are *imperfect* (Sack and Strothotte, 1985). Further, we define a *pennant* as a tree on 2^i elements, with the following properties:

(a) the smallest element is located at the root, and

(b) for $i > 0$, the root has exactly one child which is a perfect heap on the remaining $2^i - 1$ elements.

See Fig. 1 for an illustration.

The result of merging two equal-sized pennants is itself a pennant. Furthermore, a pennant can be split into two pennants of equal size. Thus, in contrast to perfect heaps, the set of pennants is closed under these two operations.

In this paper, we develop an alternative view of heaps. Our view will be based on a 1–1 correspondence between heaps and certain ordered forests of pennants. The algorithms presented for splitting and merging heaps exploit this correspondence.

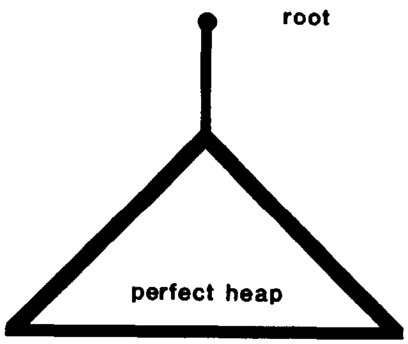

FIG. 1. The root of a pennant has one child, consisting of a perfect heap.

2. An Alternative View of Heaps

In this section we show that a heap can be viewed as a unique ordered collection of pennants. A *pennant forest* or **PF**, $P = (P_m, ..., P_0)$, is defined to be an ordered collection of pennants satisfying the following properties:

(a) each P_i, $i = 0, ..., m$, is a pennant,

(b) $\text{key}(\text{root}(P_i)) \leqslant \text{key}(\text{root}(P_{i-1}))$ for $0 < i \leqslant m$, and

(c) $\text{size}(P_i) \geqslant \text{size}(P_{i-1})$.

We say that a PF *corresponds* to a heap H if it contains the same elements as H, and furthermore, if a node p has a child s in the PF then this relationship holds also in H.

In general, there is more than one way of constructing a pennant forest corresponding to a given heap. Conversely, it may be impossible to construct a heap to which a given pennant forest corresponds without performing additional comparisons. To illustrate the latter point and to provide the reader with a more intuitive view of the pennant structure and its relation to heaps, we discuss two examples.

Example 1. Let $P = (P_3, P_2, P_1, P_0)$ be a PF in which all pennants have the same height h, where h is greater than 0. Intuitively, there are *too many equal-sized pennants* and thus the leaves of the heap will not be on adjacent levels. Any attempt to assemble a corresponding heap from this PF will result in a violation of the heap-shape.

Another reason why a PF might not correspond to any heap is illustrated by the following example.

Example 2. Let $P = (P_2, P_1, P_0)$, where the height of P_0 equals 3. From the defintion of PF it follows that both P_2 and P_1 are at least as large as P_0. Intuitively, the *small pieces* needed for the heap creation seem to be *missing* and again the heap-shape is violated.

Thus it is relevant to count the number of pennants a given PF has. We associate with every PF a *descriptor* $D = (d_{m'}, ..., d_0)$ such that the PF has d_i pennants of sizes 2^i, for all $i = 0, ..., m'$. If the number of nodes in the PF is k then

$$k = \sum_{i=0}^{m'} d_i * 2^i.$$

In Section 2.1 we show how a *unique* PF is constructed corresponding to a given heap. The class of PF's whose elements are created in this manner are characterized using descriptors. In Section 2.2, we prove that this class of PF's contains precisely all PF's for which a corresponding heap can be constructed. This provides us with a 1–1 correspondence between heaps and this well-defined class of PF's.

2.1. *Constructing a PF from a Heap*

Since a given heap may have many PF's to which it corresponds, we need to specify a "canonical" PF representation. The idea is to split the heap by removing all arcs lying on the path from the root to the last leaf in the heap. Each node on the path from the root to the last leaf in the heap is the root of some pennant (see Fig. 2). This yields exactly $\lceil \log(n+1) \rceil$ subtrees, each of which is a pennant. Since the roots of these pennants are also sorted, the collection of pennants forms a pennant forest. Algorithm *ConstructPF* states the procedure more formally; it also produces the descriptor for the PF corresponding to the given heap.

```
Algorithm ConstructPF
Input: A heap $H$ of height $m$
Output: The PF $P$ corresponding to $H$ as well as its descriptor $D$
  begin
    Initialize the descriptor $D$ to 0
    $j := m$
    for each edge (Node, Child) on the path in $H$ from the root to the
      rightmost leaf on the last level do
      $P_j :=$ the subtree rooted at Node excluding the subtree
      rooted at Child {Node now has one child less}
      $j := j - 1$
      increment $d_{\mathrm{height}(P_j)}$ by 1
    {Now collect the remaining leaf}
    $P_0 :=$ last leaft in heap $H$
    increment $d_{\mathrm{height}(P_0)}$ by 1
end
```

In Fig. 3 we illustrate the PF corresponding to the heap of Fig. 2. The correctness of Algorithm *ConstructPF* is established in the following lemma.

LEMMA 2.1. *For any given heap H, Algorithm ConstructPF constructs a corresponding PF and its associated descriptor.*

Proof. Let H be a heap and let m denote its height. We observe that any heap H contains a perfect subheap of height $h-1$ rooted at the left child of the root, or it contains a perfect subheap of height $h-2$ rooted at

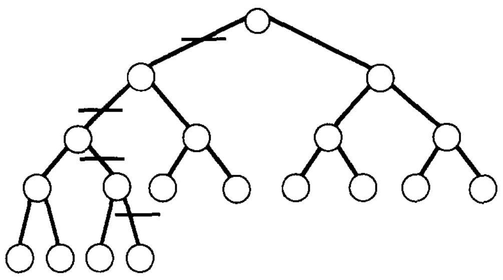

FIG. 2. A heap on 19 elements showing the substructure of pennants (sizes 8, 4, 4, 2, 1).

the right child of the root of H. In the former case, the left subtree together with the root forms a pennant; in the latter case, the right subtree together with the root forms a pennant. The algorithm extracts such a pennant and then iterates this procedure on the other subtree of H. This subtree is again a heap, since subtrees of heaps are heaps. Thus the trees $P_m, ..., P_0$ are pennants. Furthermore, all roots of these pennants lie on a path in H from the root to a leaf. It follows that $\text{key}(\text{root}P_j)) \leqslant \text{key}(\text{root}(P_{j-1}))$, and that $\text{size}(P_j) \geqslant \text{size}(P_{j-1})$, for all j, with $0 < j \leqslant m$. Q.E.D.

The PF's constructed by Algorithm *ConstructPF* have certain properties. In particular, we will show that since leaves in a heap lie on at most two levels, the heights of successively produced pennants differ by at most 2. Furthermore, the height of the pennant P_i is either i or $i-1$.

To formally characterize the PF's produced by Algorithm *ConstructPF*, we introduce the notion of a valid descriptor. A *valid descriptor* $(d_{m'}, ..., d_0)$ is defined to be a descriptor which satisfies the following conditions:

(1) For all $j \leqslant m'$, the partial sums $\sum_{i=0}^{j} d_i$ are either $j+2$ or $j+1$, and

(2) $0 \leqslant d_j \leqslant 2$.

We say that *a PF is valid* if its corresponding descriptor is valid.

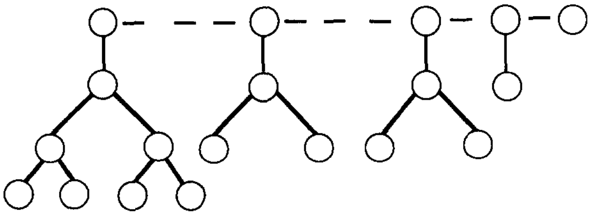

FIG. 3. The heap of Fig. 2 represented as a PF.

EXAMPLES OF DESCRIPTORS.

Descriptor	valid/invalid
(2, 1, 0, 1, 1, 2)	valid
(**4**, 1, 1, 1, 1, 1)	invalid
(2, **0**, 1, 1, 0, 2)	invalid
(1, **2**, 1, 1, 2, 1)	invalid
(1, 1, 1, 1, **0**, 1)	invalid

For invalid descriptors, in the above table, the d_i with smallest index i which makes $(d_i, ..., d_0)$ invalid is indicated in **bold**.

We will now examine the structure of valid PF's more closely. Note that for any $i \leqslant m$ the subsequence $(P_i, ..., P_0)$ of a valid pennant forest $(P_m, ..., P_0)$ is itself valid.

LEMMA 2.2. *Let $P = (P_m, ..., P_0)$ be a valid PF. Then*

(a) *for all* $0 \leqslant i \leqslant m$, height$(P_i)$ *is either* $i-1$ *or* i, *and*

(b) *for all* $0 < i \leqslant m$, height(P_i) − height$(P_{i-1}) \leqslant 2$.

Proof. (a) Since P is a valid *PF*, its corresponding descriptor $D = (d_{m'}, ..., d_0)$ is valid. There are $m+1$ pennants in P. Thus $m+1 = \sum_{i=0}^{m'} d_i$. By definition, $\sum_{i=0}^{m'} d_i$ is either $m'+1$ or $m'+2$. Since m' is the height of P_m, height(P_m) is either m or $m-1$. Applying this argument to the valid *PF* $(P_i, ..., P_0)$, $i<m$, proves part (a). Part (b) of the lemma follows directly from (a). Q.E.D.

A property of a heap is that its shape is uniquely determined by the heap size n. In fact, as will be illustrated in Lemma 2.3, just by examining the binary representation R of n, the descriptor for the PF constructed by Algorithm *ConstructPF* can be generated. As shown below, this descriptor is valid.

LEMMA 2.3. *Let H be a heap and R the binary representation of its size n, where r_j denotes the jth bit for $j = 0, ..., \lfloor \log n \rfloor$. The descriptor D constructed by Algorithm ConstructPF on input H satisfies the relationship*

$$d_0 = 2 - r_0$$

$$d_j = r_{j-1} + (1 - r_j),$$

for $j = 1, ..., \lfloor \log n \rfloor$.

Proof. Algorithm *ConstructPF* constructs the descriptor D by examining the path from the root of the heap to its nth node. We will use the fact that this path from the root of a heap to its nth node is described by the binary representation R of n, where the high order bit is ignored, "0" stands for "go to left son," and "1" stands for "go to right son." The least significant bit of R is assumed to be r_0. Combining this with Lemma 2.1, we observe that for $j > 0$

$$\text{height}(P_j) = \begin{cases} j-1 & \text{if } r_{j-1} = 0 \\ j & \text{if } r_{j-1} = 1. \end{cases}$$

Hence d_j can be computed from r_j and r_{j-1}, for $j > 0$ as

r_j	r_{j-1}	d_j
0	0	1
0	1	2
1	0	0
1	1	1

and for $j = 0$ as

r_1	r_0	d_0
0	0	2
0	1	1
1	0	2
1	1	1

These values of d_j coincide exactly with the ones obtained by the relationship stated in the lemma. Q.E.D.

From Lemma 2.3 follows

COROLLARY 2.1. *Let H be a heap, $D = (d_m, ..., d_0)$ its descriptor as produced by Algorithm ConstructPF, and $R = r_m \cdots r_0$ the binary representation of its size n. Then for all $0 \leqslant k \leqslant m$*

(a) $\sum_{j=0}^{k} d_j = (k+2) - r_k$,

(b) $\sum_{j=0}^{k} d_j 2^j = 2^{k+1} - r_k 2^k + \sum_{j=0}^{k-1} \overline{r_j} 2^j$.

LEMMA 2.4. *Let H be a heap. The PF constructed by Algorithm ConstructPF on input H is valid.*

Proof. Follows from Corollary 2.1(a). Q.E.D.

EXAMPLES.

Number	Binary representation	Valid descriptor
22	(1, 0, 1, 1, 0)	(2, 1, 0, 2)
19	(1, 0, 0, 1, 1)	(1, 2, 1, 1)

2.2. *Constructing a Heap from a Valid PF*

So far we have shown how to construct a valid PF corresponding to a given heap. We shall now turn to the inverse operation: given a valid PF, construct a heap. This establishes a 1–1 correspondence between heaps and valid pennant forests.

We will give a general algorithm for converting any PF into a binary tree, and then show, by using Lemma 2.3, that in the special case of valid PF's as input, the binary tree generated by the algorithm is a heap.

```
Algorithm ConstructBinaryTree
Input: PF = $(P_m, ..., P_0)$
Output: A binary tree $T$.
  begin
    $T := P_0$
    for $i := 1$ to $m$ do
      begin
        if height($T$) $\neq$ height($P_i$) then  Let the child of the root of $P_i$ be
            the left child and let $T$ be the right child of $P_i$
        else
          Let the child of the root of $P_i$ be the right child (if any) and
            let $T$ be the left child of $P_i$
        $T := P_i$
      end
  end
```

LEMMA 2.5. *If $(P_m, ..., P_0)$ is a valid PF, then Algorithm ConstructBinaryTree constructs a heap of height m.*

Proof. Let $P = (P_m, ..., P_0)$ be a valid PF. We prove the lemma by induction on m:
$m = 0$: Since $d_0 \geqslant 1$, P_0 contains exactly one element, and is thus a heap of height 0.
$m = i > 0$: Assume that the result is true for each valid subforest $(P_{i-1}, ..., P_0)$ and let H_{i-1} be the heap constructed from this subforest. By the induction hypothesis, height$(H_{i-1}) = i - 1$. We will show that the

algorithm maintains the heap properties when H_{i-1} is merged with P_i to give H_i. We will show that

(i) the heap shape is maintained,

(ii) the heap ordering is maintained, and

(iii) $\text{height}(H_i) = \text{height}(H_{i-1}) + 1$.

(i) P_i has a height of either i or $i-1$ (Lemma 2.2(a)). Hence its perfect child C has a height of either $i-1$ or $i-2$. In the case where the height of C is $i-1$, the algorithm makes C the left child of the new heap, whereas H_{i-1} becomes the right child. Alternatively, if the height of C is $i-2$, C becomes the right child of the new root and H_{i-1} becomes the left child. In both cases the heap shape is maintained.

(ii) From the definition of PFs, $\text{key}(\text{root}(P_i)) \leqslant \text{key}(\text{root}(P_{i-1}))$, for $i = 1, ..., m$. Since $\text{key}(\text{root}(H_{i-1})) = \text{key}(\text{root}(P_{i-1}))$, we see that $\text{key}(\text{root}(P_i)) \leqslant \text{key}(\text{root}(H_{i-1}))$, hence the heap ordering is maintained.

(iii) Since H_{i-1} becomes a new child of the root of H_i, we obtain $\text{height}(H_i) \geqslant \text{height}(H_{i-1}) + 1$. Using the induction hypothesis, we see that $\text{height}(H_i) \geqslant i$. Furthermore, $\text{height}(P_i) \geqslant \text{height}(H_{i-1})$. As shown in Lemma 2.2(a), $\text{height}(P_i)$ is either $i-1$ or i. Thus, we obtain that $\text{height}(H_i) \leqslant i$. Therefore, $\text{height}(H_i) = i$ as was to be shown. Q.E.D.

By Lemma 2.5, a heap can be constructed from a given valid pennant forest. By Lemmas 2.1 and 2.3 and Corollary 2.1, a valid pennant forest can be constructed from a given heap. Thus, summarizing this section, we see that heaps and valid pennant forests are in 1–1 correspondence. Furthermore, since the heap-shape is uniquely determined by the heap-size we obtain:

LEMMA 2.6. *For any integer n there exists a unique valid descriptor*

$$D = (d_m, ..., d_0) \qquad \textit{with} \quad \sum_{i=0}^{m} d_i 2^i = n.$$

We say that D is the *descriptor for integer n.*

Although we have given algorithms to construct a valid PF from a given heap and vice versa, no key comparisons must be done to "see" a heap as a pennant forest or conversely to "see" a valid PF as a heap. This view is crucial for the design of the algorithms to follow.

3. Splitting Heaps

In this section we will show how the pennant structure can be utilized for solving the problem of splitting a given heap into two heaps. We define the operation *Split* as:

input: a heap on n elements and an integer k,

output: two heaps of sizes k and $n-k$, respectively.

The algorithm may choose which elements are to be placed into each output heap. This problem can easily be solved by removing the last k item from the heap and building a new heap from these items. The total cost of this solution is $O(k)$, which is good for small values of k. Here we describe an algorithm for the operation *Split* which makes use of the pennant view of heaps. The algorithm is more efficient for all but small values of k, i.e., $k > \log^2(n)$.

To write the *Split* algorithm, we first define a primitive operation, *Split-Pennant*, which takes a pennant of size s and produces two pennants each of size $s/2$. The procedure is as follows:

```
**Algorithm** *SplitPennant*
**Input**: A pennant $P$ of size $s$
**Output**: Pennants $Q$ and $R$ each of size $s/2$
    $r :=$ node at root of $P$
    $Q :=$ left child of $P$ attached to $r$
    $R :=$ child of $P$ with its right subheap
```

Remark 3.1. A pennant of size s can be split into two pennants each of size $s/2$ without using any key comparisons.

One further primitive operation is required for the algorithm *Split*. We use the procedure *TrickleDown*(Root) which is analogous to the procedure *Sift-Up* described in Knuth (1973). In our algorithm, *TrickleDown* is applied to the root of a data structure P^* which is a pennant forest except that the heap-ordering may be violated at root. The procedure restores the heap-ordering in P^* while maintaining its shape.

We now have the tools necessary to present the algorithm *Split* for splitting a heap. The input for the algorithm is a heap, called H, on n elements, and an integer k.

Let $P = (P_m, ..., P_0)$ be the valid PF for the heap H, $P' = (P'_{m'}, ..., P'_0)$ for the heap H' containing k elements, and $P'' = (P''_{m''}, ..., P''_0)$ for the heap H'' on the remaining $n-k$ elements.

We formulate the *Split* algorithm by using the valid pennant forests representation of the heaps. The algorithm is as follows:

```
Algorithm Split
Input: A valid pennant forest $P$ of size $n$ and an integer $k$.
Output: Two valid pennant forests $P'$ and $P''$ of size $k$ and $n-k$, respectively.
Compute the descriptor representations $D=(d_m, ..., d_0)$, $D'=(d'_{m'}, ..., d'_0)$,
    and $D''=(d''_{m''}, ..., d''_0)$ for $n$, $k$, and $n-k$, respectively.
{For convenience, we define $d'_i=0$ for all $i>m'$ and $d''_i=0$ for all $i>m''$.}
  for $i:=0$ to $\max(m', m'')$ do
    {If $P$ does not have enough pennants of height $i$, we have to split
    larger pennants.}
    while $d'_i+d''_i>d_i$ do
      Let $S$ be the smallest indexed pennant in $P$ such that
      $h=\text{height}(S)>i$.
      Using SplitPennant split $S$ into two pennants $Q$ and $R$ of height
      $h-1$.
      Replace $S$ by ($R$ and $Q$) in $P$.
      TrickleDown(root($R$)) in PF $P$
      $d_h:=d_h-1$
      $d_{h-1}:=d_{h-1}+2$
$P'$ is constructed by removing a subsequence of $P$ consisting of $d'_i$ pennants
of height $i$, for $i:=m', ..., 0$.
The remaining subsequence, consisting of $d''_i$ pennants of height $i$, for
$i:=m'', ..., 0$, is $P''$.
```

THEOREM 3.1. *Let P be a valid pennant forest of size n and $k \leqslant n$ be a positive integer. Algorithm Split constructs two valid pennant forests on k and $n-k$ elements, respectively, using $O(\log^2 n)$ comparisons.*

Proof. Throughout this algorithm, P is a (not necessarily valid) pennant forest of size n described by the descriptor D. Since D, D' and D'' are descriptors for n, k, and $n-k$, it follows that

$$\sum_{i=0}^{m} d_i 2^i = n, \qquad \sum_{i=0}^{m'} d'_i 2^i = k, \qquad \sum_{i=0}^{m''} d''_i 2^i = n-k.$$

We show that, for $0 \leqslant t \leqslant m$,

$$\sum_{j=0}^{t} d_j 2^j \leqslant \sum_{j=0}^{t} (d'_j + d''_j) 2^j \tag{1}$$

is always true and, after the iteration of the **for**-loop in which the loop index has value i, $d_j = d'_j + d''_j$ for all $0 \leqslant j \leqslant i$. In particular, this proves the correctness of the algorithm.

Initially, D, D', and D'' are all valid PFs. Thus, using Corollary 2.2 it is easy to show that

$$\sum_{j=0}^{t} d_j 2^j \leqslant \sum_{j=0}^{t} (d_j' + d_j'')2^j \tag{2}$$

for $0 \leqslant t \leqslant m$. Also, notice that the value of d_j does not change during the ith iteration of the loop, for $j < i$.

The proof proceeds by induction. Assume that the claim is true before some iteration of the **while**-loop during the ith iteration of the **for**-loop. Suppose that during that iteration, a pennant of height h is replaced by two pennants of height $h-1$. Since $d_{h-1}2^{h-1} + d_h 2^h = (d_{h-1}+2)2^{h-1} + (d_h - 1)2^h$, the value of $\sum_{j=0,\ldots,t} d_j 2^j$ remains unchanged for $k \neq h-1$. However, we must show that

$$\sum_{j=0}^{h-2} d_j 2^j + (d_{j-1}+2)2^{h-1} \leqslant \sum_{j=0}^{h-1} (d_j' + d_j'')2^j. \tag{3}$$

For $0 \leqslant j < i$, Eq. (3) is equivalent to

$$\sum_{j=i}^{h-1} d_j 2^j + 2^h \leqslant \sum_{j=i}^{h-1} (d_j' + d_j'')2^j, \qquad \text{because } d_j = d_j' + d_j''. \tag{4}$$

We assume that this is false and obtain a contradiction. Now we know that $d_i < d_i' + d_i''$ and $d_j = 0$, for $i < j < h$, by definition of h. Thus

$$d_i 2^i < (d_i' + d_i'')2^i \leqslant \sum_{j=i}^{h-1} (d_j' + d_j'')2^j < d_i 2^i + 2^i + 2^h \tag{5}$$

and $\sum_{j=i,\ldots,h-1} (d_j' + d_j'')2^j = d_i 2^i + \alpha$, where $0 < \alpha < 2^h$. Finally,

$$\begin{aligned}\sum_{j=0}^{i-1} d_j 2^j + d_i 2^i + \sum_{j=h}^{m} d_j 2^j &= \sum_{j=0}^{m} d_j 2^j = \sum_{j=0}^{m} (d_j' + d_j'')2^j \\ &= \sum_{j=0}^{i-1} (d_j' + d_j'')2^j + \sum_{j=i}^{h-1} (d_j' + d_j'')2^j + \sum_{j=h}^{m} (d_j' + d_j'')2^j \\ &= \sum_{j=0}^{i-1} d_j 2^j + d_i 2^i + \alpha + \sum_{j=h}^{m} (d_j' + d_j'')2^j\end{aligned}$$

implies that

$$\sum_{j=h}^{m} d_j 2^j = \alpha + \sum_{j=h}^{m} (d_j' + d_j'')2^j,$$

which is impossible, since α is not divisible by 2^h. Hence the correctness of (1) is established.

By construction, after the ith iteration of the **for**-loop, $d_i \geqslant d'_i + d''_i$, but

$$\sum_{j=0}^{i} d_j 2^j \leqslant \sum_{j=0}^{i} (d'_j + d''_j) 2^j = (d'_i + d''_i) 2^i + \sum_{j=0}^{i-1} d_j 2^j,$$

so $d_i \leqslant d'_i + d''_i$. Thus $d_i = d'_i + d''_i$.

The $O(\log^2 n)$ time complexity of the algorithm follows by observing that at most one *SplitPennant* operation need be performed for each pennant in P since initially $d_i + 2 \leqslant d'_i + d''_i$ for all i. For each such *SplitPennant* operation, the algorithm carries out a *TrickleDown* operation at a cost of $O(\log n)$. Since P contains $O(\log n)$ pennants, the overall complexity is $O(\log^2 n)$ in the worst case. Q.E.D.

Remark 3.2. The total number of comparisons required by Algorithm *Split* is $O(\log^2(\max(k, n-k))) = O(\log^2(n))$.

An Example. We illustrate the algorithm presented in this section by showing how a heap H on 22 elements is split into two heaps H' and H'' on 3 and 19 elements, respectively. We refer to the corresponding PFs as P, P', and P'' and to their descriptors as D, D', and D'', respectively. Their descriptors are

$$D = (2, 1, 0, 2)$$
$$D' = (0, 0, 1, 1)$$
$$D'' = (1, 2, 1, 1)$$

The algorithm first splits pennants in P such that for each pennant in P' or P'' there is exactly one pennant of the same size in P. This leaves P with a descriptor of $(1, 2, 2, 2)$. Now the algorithm extracts from P the subsequence of pennants required for P', leaving P with exactly the sequence required for P''. This completes the execution of the algorithm.

4. Merging Heaps

In Section 2 we presented a new way of seeing a heap as a forest of pennants. This enabled us to design an efficient algorithm for solving the problem of splitting a heap on k elements into two heaps of size k and $n-k$, respectively, where $k \leqslant n$. Here we show how to employ this view in the design of an algorithm for merging heaps. This operation, called *Merge*, merges two heaps, i.e., it creates a heap H containing all $n+k$ elements given two heaps H', H'', with n and k elements, respectively. An alternate solution exhibiting the same time bound has been presented in Sack and Strothotte (1985). The algorithm described here achieves a higher degree of clarity by making use of the pennant structure of heaps.

The algorithm first "sees" the two input heaps as valid PFs, referred to as P' and P''; it then builds the "union" of both PFs. Notice, however, that by the union process a PF may be created which is no longer valid (e.g., the union may produce more than two pennants of equal height). Even worse, since the roots of pennants originating from different heaps are not necessarily sorted, the resulting collection of pennants may not even be a pennant forest. Thus, some care must be taken to ensure that the output of the *Merge* algorithm is indeed a *valid* pennant forest.

We require a primitive operation *Merge Pennants*, which merges two equal-sized pennants into one larger pennant:

Algorithm *MergePennants*
Input: Two pennants, P_0, P_1, both of size s
Output: One pennant, R, of size $2s$, containing the elements from P_0 and P_1

```
    $i :=$ index of the pennant with smaller root
    $j :=$ index of the other pennant
    $R :=$ root of $P_i$
    right child of root of $P_j :=$ child of $P_j$
    left child of root of $P_j :=$ child of $P_i$
    child of $R := P_j$
    TrickleDown(child of $R$)
```

Remark 4.1. Two pennants, each of size s, can be merged in $O(\log(s))$ time.

We formulate the *Merge* algorithm by using the valid pennant forests representation of the heaps. The following algorithm constructs a descriptor D for the valid pennant forest P. We use $\underline{D}$ to represent a temporary descriptor used during the execution of the algorithm. The algorithm terminates when $\underline{D}$ is equal to D.

Algorithm *Merge*
Input: Two valid pennant forests P' and P'' of sizes n and k, respectively $(n \geqslant k)$
Output: A valid pennant forest P of size $n+k$

```
[Initialization]
compute the descriptor representations $D=(d_m, ..., d_0)$, $D'=(d'_{m'}, ..., d'_0)$
  and $D''=(d''_{m''}, ..., d''_0)$ for $n+k$, $n$ and $k$, respectively
initialize $P$ to empty
set descriptor $\underline{D}=(\underline{d}_m, ..., \underline{d}_0)$ as follows:
  $\underline{d}=\begin{cases} d'_i + d''_i \text{ for all } 0 \leqslant i \leqslant m'' \\ d'_i \text{ for all } m'' < i \leqslant m' \end{cases}$
```

```
Step [A]
for every pennant $p$ in $P'$ whose size is greater than
  the size of the largest pennant in $P''$ (in order of decreasing size) do
  if root($p$) > root($P''$) then
      swap root($p$) and root($P''$)
      TrickleDown(root($P''$))
  append $p$ to $P$

Step [B]
(1) append the pennants remaining in $P'$ and in $P''$ to $P$ such that the
    pennants appear in decreasing order of size and among pennants of
    like size, they
    appear in non-decreasing order of the keys stored and the roots
(2) {now establish the heap-ordering among the keys of roots of pennants
    in $P$}

  let $l$ equal to the number of pennants appended to $P$ in step [B(1)]
  for $i := 0$ to $l - 1$ do
      TrickleDown(root($P_i$)) in pennant forest $P$

Step [C]
  {sweep through $P$ once more to make it valid}
  for $i := 0$ to $m$ do
      while $d_i < \underline{d}_i$ do
          merge the two pennants of size $i$ in $P$ with the smallest roots
              using MergePennants
          $\underline{d}_i := \underline{d} - 2$
          $\underline{d}_{i+1} := \underline{d}_{i+1} + 1$
```

THEOREM 4.1. *Let P' and P'' be valid pennant forest of sizes n and k, respectively, for $n \geqslant k$. Algorithm Merge constructs a pennant forest P on these $n+k$ elements in $O(\log(n) * \log(k))$ comparisons in the worst case.*

Proof. We shall show the following:

(1) the output P has the heap-shape,

(2) the output P is heap-ordered, and

(3) the algorithm terminates after at most $O(\log(n) * \log(k))$ comparisons. In particular, this will prove the stated result.

(1) *Heap-shape.* By the same argument used for Eq. (2) (from Section 3), after Step [B] of the algorithm, the following holds:

$$\sum_{j=0}^{i} d_j 2^j \leqslant \sum_{j=0}^{i} \underline{d}_j 2^j, \qquad \text{for all } i, \quad \text{where} \quad 0 \leqslant i \leqslant m.$$

Indeed, by induction, we can show that this property always holds throughout Step [C] of the algorithm and in particular, at the end of Step [C], $d_j = \underline{d}_j$, for all j, where $0 \leqslant j \leqslant m$. The proof is analogous to that of the correctness proof of the *Split* algorithm (Theorem 3.1) and is omitted here.

(2) *Heap-ordering.* It is straightforward to show by induction that the following conditions hold after the ith iteration of the loop of Step [A]:

(a) the keys of the roots of the pennants in P are in non-decreasing order, and

(b) all elements remaining in P' and P'' have keys greater than or equal to the keys of the roots of the smallest pennants in P.

Similarly, it follows by induction that P is a (not necessarily valid) pennant forest after Step [B]. Finally, an inductive argument can be used to show that after every iteration of the **while**-loop of [C], P remains a pennant forest, since

(a) the subtrees created within the **while**-loop are pennants, and

(b) the relative order of the roots of the pennants in P remaining after Step [C] is the same as before Step [C].

(3) *Complexity analysis.* We shall analyze each step individually:

Step [A]: Observe that the loop is executed $O(\log(n))$ times, each iteration performing a *TrickleDown* on a heap of k elements. Thus this step costs $O(\log(n) * \log(k))$ comparisons.

Step [B]: Step B(1) takes $O(\log n)$ comparisons. In Step B(2), the loop is executed $O(\log(k))$ times. Since there are only a constant number of equal-sized pennants in P, each iteration costs $O(\log(k))$ comparisons. Hence this step costs $O(\log^2(k) + \log(n))$ comparisons.

Step [C]: Since the **for**-loop is executed $O(\log(n))$ times, the number of comparisons is bounded by $O(\log^2(n))$. However, a more careful analysis reveals a bound of $O(\log(n) + \log^2(k))$. Observe that in the worst case, the first $O(\log(k))$ iterations of the loop may each require a *TrickleDown* at a cost of $O(\log(k))$ each. However, by examining the binary representation of n and $n+k$, we observe that $d_i \neq d'_i$ for at most two values of i greater than $\log(k)$. Hence the body of the **while**-loop of the algorithm is entered at most twice during the last $\lfloor \log(n) \rfloor - \lfloor \log(k) \rfloor$ iterations of the **for**-loop, at a cost of at most $O(\log(n))$ comparisons each. Thus the overall cost of this step is $O(\log(n) + \log^2(k))$.

Thus the cost of the merge algorithm is $O(\log(n) * \log(k))$ comparisons.

Q.E.D.

ACKNOWLEDGMENT

The authors thank an anonymous referee for very detailed comments and helpful suggestions for improving the presentation of this paper.

RECEIVED May 13, 1985; ACCEPTED February 17, 1989

REFERENCES

AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. (1974), "The Design and Analysis of Computer Algorithms," Addison-Wesley, Reading, MA.

BROWN, M. R. (1980), "The Analysis of a Practical and Nearly Optimal Priority Queue," Garland, New York.

FLOYD, R. W. (1964), Algorithm 245: treesort3, *Comm. ACM* **7**, No. 12, 701.

GONNET, G. H. (1984), "Handbook of Algorithms and Data Structures," Addison-Wesley, Reading, MA.

GONNET, G. H., AND MUNRO, J. I. (1986), Heaps on heaps, *SIAM J. Comput.* **15**, No. 4, 964–971.

KNUTH, D. E. (1973), "The Art of Computer Programming," Vol. III, "Sorting and Searching," Addison-Wesley, Reading, MA.

MUNRO, J. I., AND SUWANDA, H. (1980), Implicit data structures for fast search and update, *J. Comput. System Sci.* **21**, No. 2, 236–250.

SACK, J.-R., AND STROTHOTTE, TH. (1985), An algorithm for merging heaps, *Acta Inform.* **22**, 171–186.

WILLIAMS, J. W. J. (1964), Algorithm 232: heapsort, *Comm. ACM* **7**, No. 6, 347–348.