SIAM J. COMPUT.
Vol. 15, No. 4, November 1986

006

HEAPS ON HEAPS*

GASTON H. GONNET† AND J. IAN MUNRO†

Abstract. As part of a study of the general issue of complexity of comparison based problems, as well as interest in the specific problem, we consider the task of performing the basic priority queue operations on a heap. We show that in the worst case:

$\lg \lg n \pm O(1)$ comparisons are necessary and sufficient to insert an element into a heap. (This improves the previous upper and lower bounds of $\lg n$ and $O(1)$.)

$\lg n + \log^* n \pm O(1)$ comparisons are necessary and sufficient to replace the maximum in a heap. (This improves the previous upper and lower bounds of $2 \lg n$ and $\lg n$.)

$1.625n + O(\lg n \log^* n)$ comparisons are sufficient to create a heap. 1.37... n comparisons are necessary not only in the worst case but also on the average.

Here lg indicates the logarithm base 2 and log* denotes the iterated logarithm or number of times the logarithm base 2 may be taken before the quantity is at most 0.

Key words. heap, comparisons, lower bound

1. Introduction. One of the most elegant of storage structures is the representation of a priority queue as a heap. A heap [8], [1], [4], [2] is defined as a structure on locations 1 through n of an array with the property that the element in location i is smaller than that in location $\lfloor i/2 \rfloor$, thus inducing a complete binary tree with the property that the value of the parent is greater than that of the child. Such a pointer free representation has been called an *implicit data structure* [5]. It is well known that a heap enables us to perform the basic priority queue operations, insert an element and rebalance after extracting the maximum in $O(\lg n)$ basic operations. Furthermore, a heap can be created in about $2n$ comparisons [1], [4]. These results are very old by the standards of our field, dating back to the decade before the last. Our aim, in this paper, is to re-examine the algorithms for performing these basic operations on a heap. We are able to establish new upper and lower bounds on the number of comparisons necessary in the worst case to perform these tasks. While our algorithms may be of some interest in implementing heaps, we are using this structure primarily as a paradigm for the study of computation complexity of comparison based problems.

2. Insertion and promotion. Observe that the elements on the path from any node to the root must be in sorted order. Our idea is simply to insert the new element by performing a binary search on the path from location $n+1$ to 1. As, for $n \geqq 2$, this path contains $\lfloor \lg (n+1) \rfloor$ old elements, the algorithm will require $\lceil \lg (\lfloor 1 + \lg (n+1) \rfloor) \rceil$ comparisons in the worst case. This expression may be rewritten as $\lceil \lg \lg (n+2) \rceil$ which also indicates the number of comparisons required when $n = 0$ or 1. We note that the number of moves will be the same as those required in a carefully coded standard algorithm. It was this simple observation, also used in [3] and [6] for priority queues on a bounded domain, that sparked our interest in heap manipulation algorithms. Indeed, it is very useful as a basis for the extraction algorithm presented in the next section. However, the reader may find the fact that this bound is tight more interesting.

* Received by the editors August 7, 1984, and in revised form August 23, 1985. A preliminary report on some of the results in this paper appeared in ICALP 1982. This research has been supported by the Natural Sciences and Engineering Research Council of Canada under grants A8237 and A3353.

† Data Structuring Group, Department of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada.

THEOREM 1. *$\lceil \lg \lg (n+2) \rceil$ comparisons are sufficient and $\lceil \lg \lg (n+2) \rceil - 2$ are necessary in the worst case to insert an element into a heap of size n.*

Proof. The upper bound has been given. The lower bound comes from considering a path in the updated heap from the root through the new element and on to a leaf. Such a path is of length either $\lceil \lg (n+2) \rceil$ or $\lfloor \lg (n+2) \rfloor$ and so contains at least $\lfloor \lg (n+2) \rfloor - 1$ values from the old heap.

We develop an adversary strategy as follows. The adversary answers queries in a manner consistent with the elements in positions 2^i through $2^{i+1}-1$ of the old heap being of ranks 2^i through $2^{i+1}-1$ in the structure, although not necessarily in consecutive order among themselves. We refer to such elements, which were the same distance from the root in the old heap, as being of the same *level.* The adversary answers queries involving the new element in a manner consistent with it falling in one of the $\lceil \lg (n+1) \rceil + 1$ interlevel gaps. Indeed it will answer any comparison so as to maximize the number of gaps into which the new element could fall.

On the other hand, observe that if k elements from any level of the old heap occur on such a path, then at least $k-1$ comparisons between elements of that level must have been performed. Hence the algorithm outlined above appears to be optimal. We must, however, take into account that every such path could be of length $\lfloor \lg (n+1) \rfloor$ rather than $\lceil \lg (n+1) \rceil$ as in our algorithm, and that it may contain an element from the bottom (incomplete) level of the old heap. Taking these into account we can make the more modest claim that $\lceil \lg \lg (n+2) \rceil - 2$ comparisons are necessary in the worst case. □

We emphasize that the above bound is on the number of comparisons required to perform on insertion. The number of data items that are moved is also an interesting metric. Our method and the standard one use exactly the same number of moves. If the new element is larger than any currently in the structure this number is $1 + \lfloor \lg (n+1) \rfloor$. Under such circumstances, and if the old heap is arranged in the manner suggested in the proof, it follows immediately that this number of elements must be moved to perform an insertion.

3. Extraction and demotion. Based on the insertion algorithm of the previous section, we can easily extract the maximum and reorder the heap in $\lg n + \lg \lg n$ comparisons. Simply let the "empty location" filter down to the bottom level ($\lceil \lg (n+1) \rceil - 1$ comparisons) and then perform an insertion of the element previously in location n (or a new element if one is to be added) along the path from the empty spot to the root. This bound can, however, be improved as follows. For simplicity assume we are removing the maximum and simultaneously inserting a new element.

```
begin
  Remove the maximum, creating a "hole" at the top of the heap;
  Find the path of maximum children down r levels to A(i);
  if New element > A(i)
  then Perform a binary search with new element along path of length r
  else Promote each element on the path to the location of its parent and
    recursively apply the method starting at location A(i)
end
```

The number of comparisons required is, then,

$$C(n) = r + 1 + \max(\lceil \lg (r+1) \rceil, C(\lceil n/2^r \rceil)).$$

Choosing $r \simeq \lceil \lg n - \lg \lg n \rceil$, we see that $C(n)$ can be reduced to $\lceil \lg n \rceil + \log^* n$ where $\log^*(x) = 0$ for $x \leqq 1$ and $\log^* n = \log^*(\lceil \lg n \rceil) + 1$. Indeed the optimal choice

of r may differ by 1 from the bound suggested and the bound on $C(n)$ may also be reduced by 1 for certain values of n. However, we omit these awkward details.

As it happens, this algorithm, with judicious choice of r, essentially minimizes the number of comparisons necessary to perform the update. The key idea of our proof is, very informally, to give outcomes to comparisons in a manner consistent with the worst case of the algorithm outlined and to provide extra information so that any algorithm "might as well" have followed the given technique. The lower bound then follows from the optimal choice of r in the method presented. More formally:

THEOREM 2. $\lg n + \log^* n \pm O(1)$ *comparisons are necessary and sufficient to perform the operation replace maximum on a heap.*

Proof. Suppose that the heap upon which the extraction is to be performed is of the form indicated in Fig. 1.

(i) The largest j (j unknown but $\leqq \lg n$) elements are arranged along a path from the root. Call this path of elements j the *shaft.*

(ii) The smallest elements in the structure are the ($n/2^j$ or so) descendants of the bottom element of the shaft.

(iii) The other elements lie between these, satisfying the heap property, and furthermore, tend to be arranged so that the higher their closest shaft ancestor is located, the larger the element.

(iv) The new element is smaller than all shaft elements but larger than all others.

The crucial property is that the last element of the shaft must be determined in order to perform the update. This follows since on removal of the maximum, the shaft element of level i is the $(i-1)$st largest element in the heap and so must be moved to a higher level. On the other hand, the largest of the small elements has the property that it cannot be raised to a higher level as there are not enough smaller elements to support it.

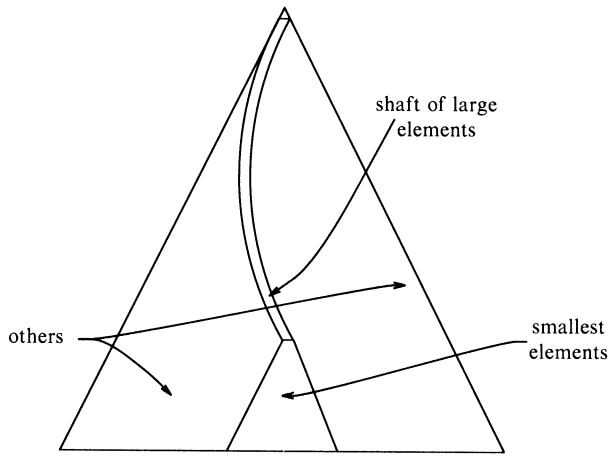


FIG. 1. *A hard case for heap update "the way it is".*

The adversary strategy is based on viewing the information which has been gathered as finding the path of maximum children to some level (see Fig. 2). Call this path the *chain.* The chain partitions the remaining elements into those which are descendants of the last chain element (the *inside*) and those which are not (the *outside*). Notice that the chain (what we have learned) and the shaft (what we are to discover) coincide for the length of the shorter. The general approach of the adversary is to respond to

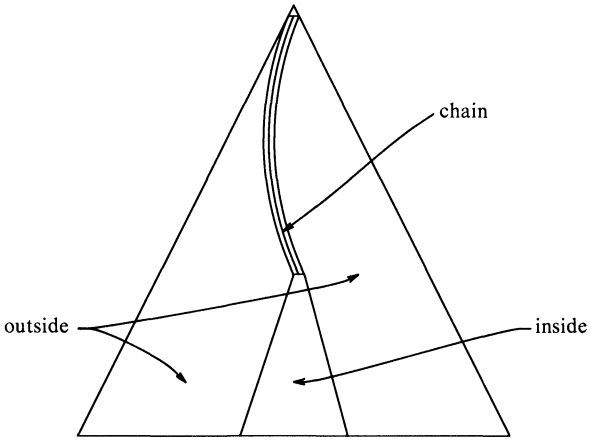


FIG. 2. *Information yielded by the adversary "what we know".*

queries so that the algorithm learns (almost) nothing about the relation of the inside elements to any but their ancestors. Furthermore, note that anything learned about the outside alone is of no help in determining the end of the shaft. The chain is, of course, permitted to be extended one level per comparison, and the algorithm can always check to see whether the shaft is shorter than the chain.

The outcome of comparisons is given below:

Chain element—New element
—Answer according to the worst case as implied by the algorithm
Chain element—Outside element
—Answer as Chain element—New element.
Inside—Outside
—Outside element is larger, supply the additional information that the chain is extended by the one element which is not an ancestor of the (hitherto) inside element considered.
Inside—New
—As inside-outside. The new element is declared to be smaller and the additional information is given that the chain can be extended by an element avoiding the path to the hitherto inside element
Outside—New
—The new element is smaller. No other information need be given.
Outside—Outside
—Answer arbitrarily but consistently.
Inside—Inside (this may be a compound step)
—If both elements are descendants of the same child of the last chain element, give an arbitrary outcome and extend the chain 1 step to avoid both.
—If both are children of the last chain member, extend the chain arbitrarily.
—If one is a child of the last chain member and the other is not, extend the chain to include the former.
—Otherwise we cannot avoid giving up some information about inside elements and so delay advancing the chain as sketched below.

If the next comparison does not involve an inside element it is answered as indicated above. If it does involve at least 1 inside element but is not of this (awkward)

subtype, then the outcome is as indicated above and the chain is extended 2 steps avoiding all hitherto inside elements that were involved in comparisons. We must be careful of a minor lacuna that this strategy may prohibit an element on the chain from being the end of the shaft. It will, however, not prevent 2 elements in a row from being the end. Hence the lower bound is weakened by at most one comparison over all.

This leaves the case in which the next comparison involves a pair of inside elements which are proper descendants of different children of the chain end. If one is a grandchild of the chain end, it wins the comparison and the chain is extended two positions as indicated above. Otherwise, we declare the higher element to be larger and defer chain extension for the last time. On the next query involving an inside element we follow the basic approach by the previous query, except that the chain may be extended 3 steps. This follows since at most 6 elements on the inside have been involved in comparisons and there are 8 subtrees three steps from the chain end, and so an "open" node may be found to which the chain may be extended. Again some chain elements may be precluded from being the chain end, but no more than 3 in a row. Hence we see the algorithm presented is within one comparison of optimal. □

A bound on performing a simple extraction follows easily.

COROLLARY 3. $\lg n + \log^* n \pm O(1)$ *comparisons are necessary and sufficient to remove the maximum element from a heap and reconstitute the heap structure.*

4. Creating a heap. The usual algorithm for creating a heap [1] requires $2n - O(\lg n)$ comparisons. It is most easily described by a call to Create $(A, 1, n)$ which creates a heap in place on elements in locations 1 to n of the array A.

```
Create (A, i, n)
  Do case 2 * i: n
    =if A(i) < A(n) then Swap (A(i), A(n))
    >do nothing
    <Begin
        Create (A, 2 * i, n)
        Create (A, 2 * i + 1, n);
        Perform replace maximum operation as if a large element in
        A(i) has been replaced by the actual value of A(i)
      end
```

Using the "standard" replace maximum technique this leads to the recurrence

$$T(n) = 2T(n/2) + 2 \lg n, \qquad T(2^k - 1) = 2(2^k - k - 1)$$

and hence the stated bound on the number of element to element comparisons. Clearly the method of the previous sections can be employed to reduce this bound. This is an improvement, but disappointing, as about 1.77... n comparisons are required. By way of contrast the following lower bound is easily derived.

THEOREM 4. 1.3644... $n + O(\lg n)$ *comparisons are necessary, not only in the worst case, but also on the average to create a heap on n elements.*

Proof. A reasonably straightforward enumeration shows that there are $H(n) = n!/\Pi t_i$ valid heaps on a set on n numbers where t_i is the size of the heap rooted at node i. A lower bound on the average number of comparisons required to permit one of $n!$ possible input sequences to one of these orders is

$$\lg (n!/H(n)) = \sum \lg t_i. \qquad \square$$

We are unable to give an algorithm which achieves this bound. Indeed we conjecture that it is not achievable and that the algorithm below minimizes the number of comparisons to create a heap on n elements in the worst case.

THEOREM 5. $\frac{13}{8}n + O(\log^* n \lg n)$ *comparisons are sufficient to construct a heap on* n *elements.*

Proof. We will outline a method of constructing a heap on 2^k elements using $\frac{13}{8}2^k - k - 2$ comparisons ($k \geqq 3$). A heap on n nodes can be viewed as a maximum element, a heap on $2^k - 1$ elements ($k = \lceil \lg(n-1) - 1 \rceil$ or $\lfloor \lg(n-1) - 1 \rfloor$) and a heap on the remaining nodes. Using the technique below to form structures of size $2^k - 1$ for appropriate values of $k < \lg n$, a set of at most $\lg n$ such heaps are grafted in about $(\lg n)(\lg n + \log^* n)/2$ comparisons using the technique of the preceding section. The $(\lg n)^2/2$ term in this expression balances the "$-k$" term in the following construction for $n = 2^k$, leaving the bound claimed.

As seen in the discussion above, what we require is a method of constructing a heap of size $2^k - 1$. We find it easier to express our method for size 2^k. Since these structures are created and used serially, the task is easily completed by "throwing away" the single element on the "bottom level". The basis of the method is the formation of binomial trees of size 2^i (see [7]). As illustrated in Fig. 3, this is simply a tree structure on 2^i elements such that

(i) a single element is a binomial tree of order 2^0;

(ii) a binomial tree of order 2^i is constructed from two of order 2^{i-1} by making the smaller of the maximum values in these trees a child of the larger.

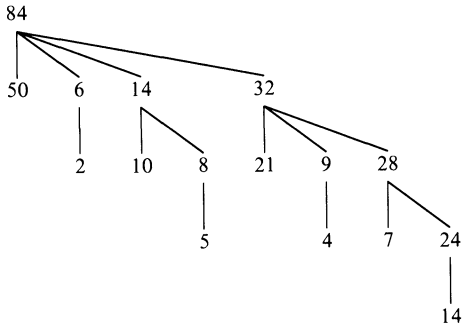


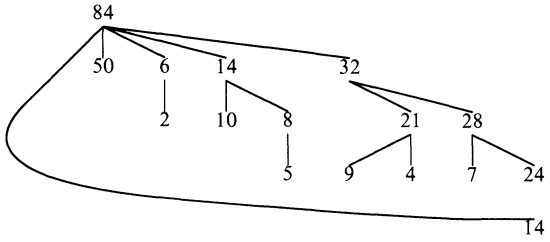
FIG. 3. *A binomial tree of order* 16.

Our method proceeds as follows; the procedure Convert (see also Fig. 4) converts a binomial tree to a heap.

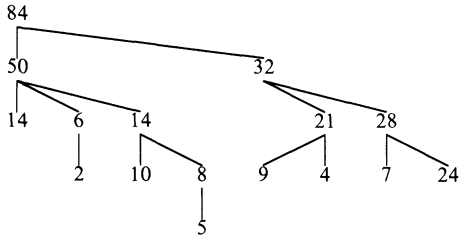
```
Procedure Convert (T, 2^r)
Begin
  Convert (subtree of root of order 2^(r-1), 2^(r-1));
  This leaves an "extra element" on the bottom level
  of this heap, make it a child of the root of T;
  We now have 1 binomial tree of order 2^i
    (i = 1, ···, r-2) and 2 singleton nodes
    all hanging from the root;
  Construct a binomial tree, S, of order 2^(r-1) from these;
  Convert (S, 2^(r-1))
end
```

The number of comparisons required by this method, including binomial tree creation,

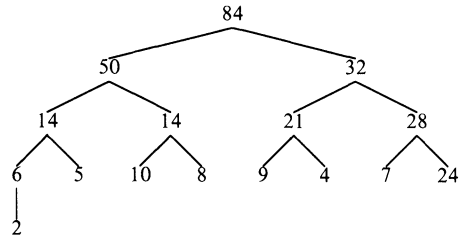
Procedure Convert (T, 2^r)
begin
  Convert (subtree of root of order 2^{r-1}, 2^{r-1});
  This leaves an "extra element" on the bottom level
  of this heap, make it a child of the root of T;



  We now have 1 binomial tree of order 2^i ($i = 1,...,r-2$) and 2
  singleton nodes all hanging from the root;
  Construct a binomial tree, S, of order 2^{r-1} from these;



  Convert(S, 2^{r-1})



end.

FIG. 4. *Illustration of heap the construction algorithm with the approach construct a binomial tree of 2^r nodes and convert it to a heap.*

can be shown to be

$$T(2^k) = 2T(2^{k-1}) + k.$$

A binomial tree of order 2 or 4 is a heap; hence the recursive call on these small binomial trees can be omitted. A more important observation is that a binomial tree on 8 nodes can be converted to a heap in 1 comparison (rather than 2). Jumping out of the recursion in the above algorithm on trees of size 8 and using $T(8) = 8$ as a basis yields the solution.

$$T(2) = 1, \quad T(4) = 3 \quad \text{and}$$
$$T(2^k) = \tfrac{13}{8}2^k - k - 2 \quad (\text{for } k \geqq 3). \qquad \square$$

We are inclined to believe our technique is optimal in the worst case when n is of the form 2^k or $2^k - 1$ and within a lower order term otherwise. By a careful

examination of structures on 3 and 4 elements we can show that the method is optimal for 7-heaps.

THEOREM 6. 8 *comparisons are necessary and sufficient, in the worst case, to form a 7-heap.*

However, a 7-heap can be constructed using $7\frac{2}{7}$ comparisons on the average based on the binomial tree "building blocks". This yields an improvement in the average behavior of our algorithm:

THEOREM 7. $1\frac{15}{28}n = 1.5357...$ *n comparisons are sufficient, on the average, to construct a heap on n nodes.*

REFERENCES

[1] R. W. FLOYD, *Algorithm* 245, *Treesort* 3, Comm. ACM, 7 (1964), p. 701.
[2] G. H. GONNET, *A Handbook of Algorithms and Data Structures*, Addison-Wesley, Reading, MA, 1984.
[3] D. B. JOHNSON, *A priority queue in which initialization and queue operations take* $O(\log \log D)$ *Time*, Math. Systems Theory, 15 (1982), pp. 295–309.
[4] D. E. KNUTH, *The Art of Computer Programming, Vol.* 3: *Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.
[5] J. I. MUNRO AND H. SUWANDA, *Implicit data structures for fast search and update*, J. Comput. System Sci., 21 (1980), pp. 236–250.
[6] P. VAN EMDE BOAS, R. KAAS AND E. ZILJLSTRA, *Design and implementation of an efficient priority queue*, Math. Systems Theory, 10 (1977), pp. 99–127.
[7] J. VUILLEMIN, *A data structure for manipulating priority queues*, Comm. ACM, 21, 4 (1978), pp. 309–314.
[8] J. W. J. WILLIAMS, *Algorithm* 232, *Heapsort*, Comm. ACM, 7 (1964), pp. 347–348.