Programming Techniques and Data Structures

Ian Munro Editor

Min-Max Heaps and Generalized Priority Queues

M. D. ATKINSON, J.-R. SACK, N. SANTORO, and T. STROTHOTTE

ABSTRACT: A simple implementation of double-ended priority queues is presented. The proposed structure, called a min-max heap, can be built in linear time; in contrast to conventional heaps, it allows both FindMin *and* FindMax *to be performed in constant time;* Insert, DeleteMin, *and* DeleteMax *operations can be performed in logarithmic time.*

Min-max heaps can be generalized to support other similar order-statistics operations efficiently (e.g., constant time FindMedian *and logarithmic time* DeleteMedian*); furthermore, the notion of min-max ordering can be extended to other heap-ordered structures, such as leftist trees.*

1. INTRODUCTION

A (single-ended) priority queue is a data type supporting the following operations on an ordered set of values:

1) find the maximum value (FindMax);
2) delete the maximum value (DeleteMax);
3) add a new value x (Insert(x)).

Obviously, the priority queue can be redefined by substituting operations 1) and 2) with FindMin and DeleteMin, respectively.

Several structures, some implicitly stored in an array and some using more complex data structures, have been presented for implementing this data type, including max-heaps (or min-heaps) [8, 12].

This research was supported in part by the National Science and Engineering Council of Canada under Grants A2419, A2415, and A0332.

Conceptually, a *max-heap* is a binary tree having the following properties:

a) *heap-shape*: all leaves lie on at most two adjacent levels, and the leaves on the last level occupy the leftmost positions; all other levels are complete.

b) *max-ordering*: the value stored at a node is greater than or equal to the values stored at its children.

A max-heap of size n can be constructed in linear time and can be stored in an n-element array; hence it is referred to as an implicit data structure [9]. When a max-heap implements a priority queue, FindMax can be performed in constant time, while both DeleteMax and Insert(x) have logarithmic time.

We shall consider a more powerful data type, the double-ended priority queue, which allows both FindMin and FindMax, as well as DeleteMin, DeleteMax, and Insert(x) operations. An important application of this data type is in external quicksort [5, p. 162].

A traditional heap does not allow efficient implementation of all the above operations; for example, FindMin requires linear (instead of constant) time in a max-heap. One approach to overcoming this intrinsic limitation of heaps, is to place a max-heap "back-to-back" with a min-heap as suggested by Williams [8, p. 619]. This leads to constant time Find either extremum and logarithmic time to Insert an element or Delete one of the extrema, but is somewhat trickier to implement than the method following.

In this article, we present a new and different generalization of a heap. Our structure, called a min-max heap, is based on the heap structure under the notion of min-max ordering: values stored at nodes on even (odd) levels are smaller than or equal to (respectively, greater than) values stored at their descendants.

The proposed structure can be constructed in linear time and both FindMin and FindMax can be performed in constant time; all other operations require logarithmic time. Furthermore, no additional pointers are required; that is, the data structure can be stored *in situ*.

The structure can also be generalized to support the operation Find(k) (determine the kth smallest value in the structure) in constant time and the operation Delete(k) (delete the kth smallest value in the structure) in logarithmic time, for any fixed value (or set of values) of k. As an example, we describe the min-max-median structure. The notion of min-max ordering can be extended to other structures based on the max- or (min-)ordering, such as leftist trees, generating a new (and more powerful) class of data structures [7].

2. MIN-MAX HEAPS

Given a set S of values, a *min-max heap* on S is a binary tree T with the following properties:

1) T has the heap-shape

2) T is *min-max ordered*: values stored at nodes on even (odd) levels are smaller (greater) than or equal to the values stored at their descendants (if any) where the root is at level zero. Thus, the smallest value of S is stored at the root of T, whereas the largest value is stored at one of the root's children; an example of a min-max heap is shown in Figure 1 (p. 998).

A min-max heap on n elements can be stored in an array $A[1 \ldots n]$. The ith location in the array will correspond to a node located on level $\lfloor(\log_2 i)\rfloor$ in the heap. A max-min heap is defined analogously; in such a heap, the maximum value is stored at the root, and the smallest value is stored at one of the root's children.

It is interesting to observe that the Hasse diagram for a min-max heap (i.e., the diagram representing the order relationships implicit within the structure) is rather complex in contrast with the one for a traditional heap (in this case, the Hasse diagram is the heap itself); Figure 2 (p. 998) shows the Hasse diagram for the example of Figure 1.

Algorithms processing min-max heaps are very similar to those corresponding to conventional heaps. Creating a min-max heap is accomplished by an adaption of Floyd's [4] linear-time heap construction algorithm. Floyd's algorithm builds a heap in a bottom-up fashion. When the algorithm examines the subtree rooted at $A[i]$ then both subtrees of $A[i]$ are max-ordered, whereas the subtree itself may not necessarily be max-ordered. The TrickleDown step of his algorithm exchanges the value at $A[i]$ with the maximum of its children. This step is then applied recursively to this maximum child to maintain the max-ordering. In min-max heaps, the required ordering must be established between an element, its children, and its grandchildren. The procedure must differentiate between min- and max-levels. The resulting description of this procedure follows:

```
procedure TrickleDown(i)
  -- i is the position in the array
  if i is on a min level then
    TrickleDownMin(i)
  else
    TrickleDownMax(i)
  endif

procedure TrickleDownMin(i)
  if A[i] has children then
    m := index of smallest of the
         children and grandchildren
         (if any) of A[i]
    if A[m] is a grandchild of A[i] then
      if A[m] < A[i] then
        swap A[i] and A[m]
        if A[m] > A[parent(m)] then
          swap A[m] and A[parent(m)]
        endif
        TrickleDownMin(m)
      endif
    else {A[m] is a child of A[i]}
      if A[m] < A[i] then
        swap A[i] and A[m]
      endif
  endif
```

The procedure TrickleDownMax is the same except that the relational operators are reversed. The operations DeleteMin and DeleteMax are analogous to deletion in conventional heaps. Specifically, the required element is extracted and the vacant position is filled with the last element of the heap. The min-max ordering is maintained after applying the TrickleDown procedure.

An element is inserted by placing it into the first available leaf position and then reestablishing the ordering on the path from this element to the root. An efficient algorithm to insert an element can be designed by examining the Hasse diagram (recall Figure 2). The leaf-positions of the heap correspond to the nodes lying on the middle row in the Hasse diagram. To reestablish the min-max ordering, the

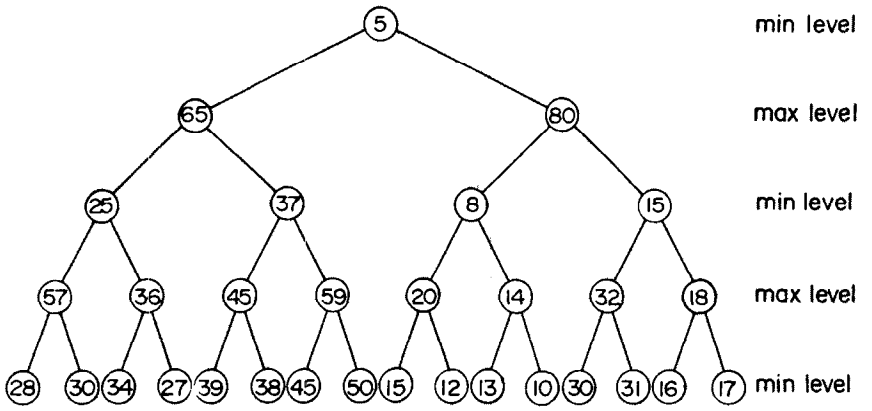

FIGURE 1. Sample of a Min-Max Heap. The heap condition alternates between minimum and maximum from level to level.

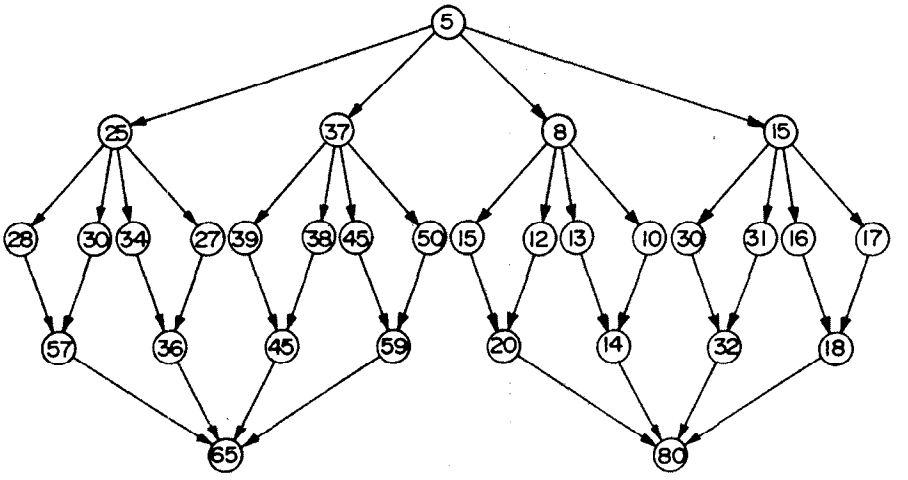

FIGURE 2. Hasse Diagram for Min-Max Heap of Figure 1.

new element is placed into the next available leaf position, and must then move up the diagram toward the top, or down toward the bottom, to ensure that all paths running from top to bottom remain sorted. Thus the algorithm must first determine whether the new element should proceed further down the Hasse diagram (i.e., up the heap on max-levels) or up the Hasse diagram (i.e., up the heap on successive min-levels). Once this has been determined, only grandparents along the path to the root of the heap need be examined—either those lying on min-levels or those lying on max-levels.

The algorithms are as follows:

```
procedure BubbleUp(i)
-- i is the position in the array
  if i is on a min-level then
    if i has a parent cand A[i] >
       A[parent(i)] then
      swap A[i] and A[parent(i)]
      BubbleUpMax(parent(i))
    else
      BubbleUpMin(i)
    endif
  else
    if i has a parent cand A[i] <
       A[parent(i)] then
      swap A[i] and A[parent(i)]
      BubbleUpMin(parent(i))
    else
      BubbleUpMax(i)
    endif
  endif

procedure BubbleUpMin(i)
  if A[i] has grandparent then
    if A[i] < A[grandparent(i)] then
      swap A[i] and A[grandparent(i)]
      BubbleUpMin(grandparent(i))
    endif
  endif
```

The **cand** (conditional and) operator in the above code evaluates its second operand only when the

first operand is true. BubbleUpMax is the same as BubbleUpMin except that the relational operators are reversed.

From the similarity with the traditional heap algorithms, it is evident that the min-max heap algorithms will exhibit the same order of complexity (in terms of comparisons and data movements). The only difference rests with the actual constants: for construction and deletion the constant is slightly higher, and for insertion the constant is slightly lower. The value of the constant for each operation is summarized in Table I; the reader is referred to [2] for a detailed derivation of these values. All logarithms are base 2.

TABLE I. Worst-Case Complexities for Min-Heaps and Min-Max Heaps

	min-heap	min-max heap
Create	$2n$	$7n/3$
Insert	$\log(n+1)$	$0.5\log(n+1)$
DeleteMin	$2\log(n)$	$2.5\log(n)$
DeleteMax	$0.5\,n + \log(n)$	$2.5\log(n)$

Slight improvements in the constants can be obtained by employing a technique similar to the one used by Gonnet and Munro [6] for traditional heaps. The resulting new values are shown in Table II; again, details of the derivation can be found in [2]. In Table II the function $g(x)$ is defined as follows: $g(x) = 0$ for $x \leq 1$ and $g(n) = g(\lceil \log(n) \rceil) + 1$.

3. ORDER-STATISTICS TREES

The min-max heap structure can also be generalized to perform efficiently the operations Find(k) (locate the kth smallest value in the structure) and Delete(k) (delete that value) for any predetermined set $K = \{k_1, \ldots, k_m\}$ of values of k. This new structure, called an *order-statistics tree*, also allows optimal processing of range queries of the form List$[k_i \ldots k_j]$: find all elements in the structure whose rank is between k_i and k_j (for $k_i < k_j$).

An order-statistics tree on a set S of values and for a specified set $K = \{k_1, \ldots, k_m\}$ of rank values, where $k_i < k_{i+1}$, is a structure comprised of a one-dimensional array $V[0..m+1]$ and $m+1$ min-max heaps $T_0, \ldots, T_m$; element $V[i]$ contains the k_ith smallest element in S (if any), and T_i contains the elements whose rank is greater than $V[i]$ and smaller than $V[i+1]$ (where $V[0] = -\infty$ and $V[m+1] = +\infty$).

Using the properties of min-max heaps, an order-statistics tree can be constructed in linear time, any Find operation (on the specified set of ranks) can be performed in constant time, the Insert operation and any Delete operation (on the specified set of ranks) can be performed in logarithmic time, and any range query (where the range is from elements of the rank set) can be answered in time proportional to the query result. Since $k_1, \ldots, k_m$ are fixed in value, an order-statistics tree can be stored implicitly in an array. The interested reader is referred to [3] for a detailed description and analysis of this structure as well as a practical application.

To illustrate this structure (as well as the versatility of min-max heaps) we shall consider the problem of designing a structure which, in addition to the operations of a double-ended priority queue, supports the operations FindMedian (in constant time) and DeleteMedian (in logarithmic time).

Observe that, if we are interested only in the operations Insert, FindMedian, and DeleteMedian, then the problem is easily solved using two separate traditional heaps (a min-heap on the values greater than the median and a max-heap on the values smaller than the median). However, it is not difficult to see that, as soon as we require just the additional operations FindMin and DeleteMin, solutions employing traditional heaps do not yield efficient performance. On the other hand, data structures such as balanced binary search trees, 2–3 trees, AVL trees (which do yield the desired performance) require $O(n \log n)$ creation time.

Our solution, the *min-max-median* (or *mmm*) heap, is simple and efficient. An mmm-heap is a binary tree with the following properties:

a) The median of all elements is located at the root.

b) The left subtree of the root is a min-max heap H_l of size $\lceil ((n-1)/2) \rceil$, containing elements less than or equal to the median. The right subtree is a

TABLE II. Running Times of Improved Algorithms for Min-Heaps and Min-Max Heaps

	Data movements		Comparisons	
	Min-heap	Min-max heap	Min-heap	Min-max heap
Create	n	n	$1.625\,n$	$2.15\ldots n$
Insert	$\log(n+1)$	$0.5\log(n+1)$	$\log(\log(n+1))$	$\log(\log(n+1))$
DeleteMin	$\log(n)$	$\log(n)$	$\log(n) + g(n)$	$1.5\log(n) + \log(\log(n))$
DeleteMax	$\log(n)$	$\log(n)$	$0.5\,n + \log(\log(n))$	$1.5\log(n) + \log(\log(n))$

max-min heap H_r of size $\lfloor((n-1)/2)\rfloor$ containing only elements greater than or equal to the median.

An mmm-heap can be stored in an array A, such that the median is at location $A[1]$, the minimum at location $A[2]$, and the maximum at location $A[3]$. The algorithm to construct an mmm-heap on n element is as follows:

1. Find the median of all n elements using any one of the known linear-time algorithms (e.g., [1, p. 98]). Partition the elements into two sets, S_l, S_r of sizes $\lceil((n-1)/2)\rceil$, $\lfloor((n-1)/2)\rfloor$, respectively, such that S_l contains only elements less than or equal to the median and S_r contains only elements greater than or equal to the median.
2. Construct a min-max heap H_l of the elements in S_l and a max-min heap H_r of the elements in S_r.

To find successive medians efficiently we maintain the balance between the sizes of H_l and H_r, denoted by $|H_l|$ and $|H_r|$, respectively. Since we are considering the upper median, we must keep either $|H_l| = |H_r|$ or $|H_l| = |H_r| + 1$. The operation DeleteMedian consists of extracting either the minimum of H_r or the maximum of H_l, depending on which maintains this balance. DeleteMin is performed by extracting the minimum of H_l, followed by a rebalancing if necessary: if $|H_l|$ has become too small, the median is inserted into H_l and the minimum of H_r is extracted to become the new median. Similarly, a DeleteMax is performed by extracting the maximum of H_r, and performing a rebalancing if necessary.

With the exception of the creation, the operations on mmm-heaps reduce to operations on the min-max heap H_l or the max-min heap H_r. Thus insertions and deletions can be performed in logarithmic time. The creation uses an initial median-finding process, which requires linear time, so the overall creation time is also linear. The exact complexities can be calculated from Table I.

4. CONCLUDING REMARKS

The min-max heap structure is based on the idea of alternating the relations "greater than or equal to all descendants" and "smaller than or equal to all descendants" between consecutive tree levels; the order relation implied is herein referred to as min-max ordering and can be applied to a number of structures implementing priority queues, such as P-trees, leftist-trees (see [7] for details).

We have presented efficient algorithms for implementating an implicit double-ended priority queue using min-max heaps. The cost of maintaining a min-max heap is comparable to that of maintaining a conventional heap. The algorithms presented in this article open up a number of interesting problems for further research. We are currently investigating whether the techniques for merging heaps presented in [10, 11] can be extended to merge min-max heaps. This would require the data structures to explicitly store pointers and to no longer be implicit.

Acknowledgment. We would like to thank G. J. E. Rawlins and J. M. Robson for their comments on an earlier draft of this article. We also thank the referees for their helpful suggestions.

REFERENCES

1. Aho, A.V., Hopcroft, J.E., and Ullman, J.D. *The Design and Analysis of Computer Algorithms.* Addison Wesley, Reading, Mass., 1974.
2. Atkinson, M.D., Sack, J.-R., Santoro, N., and Strothotte, Th. An efficient, implicit double-ended priority queue. Tech. Rep. SCS-TR-55, School of Computer Science, Carleton University, Ottawa, Ont., Canada, July 1984.
3. Atkinson, M.D., Sack, J.-R., Santoro, N., and Strothotte, Th. Min-max heaps, order-statistic trees and their application to the Course-Marks problem. In *Proceedings of Nineteenth Conference on Information and System Sciences*, Baltimore, Md., (Mar. 1985), 160–165.
4. Floyd, R.W. Algorithm 245, Treesort3. *Commun. ACM 7*, 12 (Dec. 1964), 701.
5. Gonnet, G.H. *Handbook of Algorithms and Data Structures.* Addison-Wesley, Reading, Mass., 1984.
6. Gonnet, G. H., and Munro, J.I. Heaps on heaps. In *Proceedings of the ICALP*, Aarhus, 9 (July 1982), 282–291.
7. Hasham, A. A new class of priority queue organizations, Master's thesis, School of Comput. Sci., Carleton University, Ottawa, Ont., Canada, 1986.
8. Knuth, D.E. *The Art of Computer Programming, Vol. III Sorting and Searching.* Addison-Wesley, Reading, Mass., 1973.
9. Munro, J.I., and Suwanda, H. Implicit data structures for fast search and update. In *Proceedings SIGACT*, Atlanta Ga., 11 (Apr. 1979), 108–117; also *JCSS* 21, 2 (Oct. 1980), 236–250.
10. Sack, J.-R., and Strothotte, Th. An algorithm for merging heaps. *Acta Inform.* 22, (1985), 171–186.
11. Strothotte, Th., and Sack, J.-R. Heaps in heaps. *Congressus Numerantium*, 49 (1985), 223–235.
12. Williams, J.W.J. Algorithm 232. *Commun. ACM 7*, 6 (June 1964), 347–348.

CR Categories and Subject Descriptors: E.1 **[Data]**: Data Structures—*trees*; F.2.2 **[Analysis of Algorithms and Problem Complexity]**: Nonnumerical Algorithms and Problems

General Terms: Algorithms

Additional Key Words and Phrases: heaps, min-max ordering, order-statistics trees, priority queues

Received 7/84; revised 5/86; accepted 6/86

Authors' Present Addresses: M. D. Atkinson, J.-R. Sack, and N. Santoro, School of Computer Science, Carleton University, Ottawa, Ont., Canada K1S 5B6. T. Strothotte, Abteilung für Dialogsysteme, Institut für Informatik, Universität Stuttgart, Stuttgart, Federal Republic of Germany.