Interval Heaps

J. VAN LEEUWEN* AND D. WOOD†

*Department of Computer Science, University of Utrecht, PO Box 80.089, 3508 TB Utrecht, The Netherlands

†Department of Computer Science, University of Western Ontario, London, Ontario N6A 3K7, Canada

We present new solutions to the problems of constructing efficient implicit data structures for min–max and min–max–median priority queues. The novelty in our approach is that we use the standard heap (or priority queue) structure with multiple values at the nodes. This technique yields a consistent approach to the implementation of min–max and min–max–median priority queues. The first advantage of this approach is that the updating algorithms are almost the same as those for the standard heap implementation of a priority queue. The second advantage is that we can easily generalize the heap structure to accommodate multidimensional data. The third advantage is that we immediately derive optimal query algorithms for complementary-range queries. A number of applications to computational geometry are discussed. By generalizing the approach for d-dimensional data, a (dynamic) implicit data structure is obtained for complementary-range searching in $\Theta(K)$ time per query and with $\Theta(\log n)$ update times, for fixed d, where K is the number of answers to a query. Several related ideas and applications are also discussed.

Received April 1988, revised June 1992

1. INTRODUCTION

Let X be a totally ordered domain of values. (Think of X as being some submultiset of $\Re$.) A **double-ended** or **min–max priority queue** is a data structure for finite submultisets of X that supports the following operations:

1. MIN—determine a smallest value.
2. MAX—determine a largest value.
3. INSERT(x)—add a value $x \in X$ to the submultiset.
4. DELETEMIN—remove a smallest value.
5. DELETEMAX—remove a largest value.

We consider the problem of designing an efficient implicit data structure for double-ended priority queues, i.e. a data structure for maintaining a multiset of n values from X in the positions $A[1]$ through $A[n]$ of an array A without additional pointers and supporting the listed operations in $\Theta(\log n)$ time or less per operation. We assume that the reader is familiar with the usual data structures for implementing ordinary, that is, single-ended priority queues [1, 9, 14]. In particular, we will assume familiarity with heaps. Heaps support only the operations (1), (3) and (4), or, alternatively, the operations (2), (3) and (5) from the preceding list.

An interesting implicit data structure for implementing double-ended priority queues was proposed by Atkinson *et al.* in 1984 [2, 3]. The MIN–MAX heap proposed by Atkinson *et al.* is organized like a heap, except that a different ordering relation is employed: values at even-level nodes must be less than or equal to all values at descendant nodes, and values at odd-level nodes must be greater than or equal to all values at descendant nodes. They show that MIN–MAX heaps can be constructed in $\Theta(n)$ time, that MIN–MAX heaps support MIN and MAX operations in $\Theta(1)$ time, and that MIN–MAX heaps support the remaining operations for double-ended priority queues in $\Theta(\log n)$ time. (By applying the techniques of Gonnet and Munro [7] the time-bound for INSERTs can be improved to $\Theta(\log \log n)$.) They argue that the kind of ordering relation imposed on MIN–MAX heaps can also be applied to other structures for single-ended priority queues, to obtain implementations of double-ended priority queues. Another implicit data structure for the problem has been devised by Carlsson [5]. The data structure he proposed consists of two heaps, one organized as a MIN heap and the other as a MAX heap, with the property that the value in any leaf of the MIN heap is smaller than the value in the corresponding leaf (or its parent, when the corresponding position is not a leaf) of the MAX heap. The structure is easy to maintain and, in fact, quite closely related to the ideas that we develop in this paper. A much earlier approach suggested by Williams [16] uses two heaps placed back to back, so that one heap is a MIN heap, one heap is a MAX heap and, at each position, the MIN value is no greater than the MAX value. This approach is also closely related to our ideas; however, our emphasis on intervals is new.

We propose a different implicit data structure for double-ended priority queues that generalizes heaps in a more consistent way. The data structure is called an **interval heap** and consists, essentially, of a heap in which each node carries a pair of values a, b (with $a \leqslant b$) from the current multiset, that are considered to be the endpoints of an interval of X. The usual heap property is replaced by the requirement that, for each node, the interval at the node contains the intervals at its children. (Note the similarity with the heap property for ordinary heaps.) If $[a, b]$ is the interval at the root, then the heap

property implies that a is the current MIN and b the current MAX of the multiset. A more precise definition of interval heaps is given in Section 2. In Section 2, we argue that interval heaps can be built in $\Theta(n)$ time, support MIN and MAX operations in $\Theta(1)$ time, and the remaining operations for double-ended priority queues in $\Theta(\log n)$ time or less. We also discuss a number of interesting applications of interval heaps to, for example, the theory of order statistics, in Section 3.

In Section 4, we consider a variant of interval heaps that we call **interval *heaps**. In interval *heaps, the heap property is the reverse of the one for interval heaps, i.e. for each node, the associated interval is contained in all the intervals in its subtree. This implies that the two values at the root are medians of the given multiset of values. The structure allows us to dynamically maintain the median. We show that interval *heaps can be updated in $O(\log n)$ time, and show how they can be used to answer specific restricted-range queries in $\Theta(K)$ time, where K is the number of answers. Finally, in Section 5, we develop a generalization of interval heaps for the case of multidimensional data. The resulting implicit data structure (called a ***d*-interval heap**) enables us to solve the complementary-range searching problem in $\Theta(K)$ time per query and with $\Theta(\log n)$ update time, for fixed dimension d and for K the number of answers to the query.

We have developed an implicit data structure based on heaps, but with different kinds of objects at the nodes and with a very special ordering relation enforced between the objects in the heap structure. The interval heaps studied in this paper expose the generality of the traditional algorithms for manipulating heaps, and show that heaps are a more universal data structure than commonly assumed in the theory of sorting. At the same time our study seems to open the way for the design and analysis of implicit data structures for use in the theory of multidimensional data structures and computational geometry. The complementary-range searching problem is only the first of a number of problems that have been explored from this perspective; see also the work of Klein *et al.* [8].

2. INTERVAL HEAPS

There are two ways of looking at ordinary heap structures with n elements; see, e.g. [1, 9, 14].

1. As a binary tree of n nodes with all levels completely filled from left to right except for, perhaps, the lowest level which contains only as many nodes as are needed to bring the count to n; that is, a left-complete binary tree.
2. As an array H with its locations $H[1]$ to $H[n]$ corresponding to the consecutive nodes, taken level by level, of the binary tree.

We will use the binary-tree representation of heaps to simplify our discussion. Note that all the manipulations can be carried out by simple address calculations and accesses on the array H, as with ordinary heap structures. Also note that we have divorced the concept of heap structure from the details of the represented values and of the ordering relation that should hold between these values. As it stands, we could associate any type of objects to the nodes and impose any ordering relation between the objects at the nodes that is consistent with the idea of heaps, to obtain some kind of **object heap**. We now introduce interval heaps, which are obtained by associating (closed) intervals $[a, b]$, where $a, b \in X$, to the nodes. Let $I(v)$ denote the interval associated with a node v. Let the nth (and last) node of a heap structure be called the **left-hand node** or, simply, the ***L*-node** of the structure.

Definition 2.1. *An* interval heap *is a heap structure that is either empty or satisfies the following three conditions (the* **heap property***):*

1. *For each node v different from the L-node, $I(v)$ is an interval $[a, b]$, where $a, b \in X$.*
2. *For the L-node v, $I(v)$ is either a single value a, where $a \in X$, or an interval $[a, b]$, where $a, b \in X$.*
3. *For all nodes v and w, if v is a child of w, then $I(v) \subseteq I(w)$.*

In the definition a distinction is made between single values and intervals $[a, a]$ that contain only one distinct value from X (although it takes two values from X to specify the interval). For simplicity the $\subseteq$-relation is extended to cover the inclusion of single values in intervals as well. Observe that interval heaps are easily implemented as implicit data structures: every location of the array H holds two values except possibly for the last location which contains either one or two values, depending on whether the multiset size is odd or even.

An interval heap represents the multiset or, more precisely, the multiset consisting exactly of the values of X that are used to delimit the left or right end of an interval in the structure and, in case the L-node does not hold an interval, the value at the L-node. The main result of this section is that interval heaps are a viable implicit data structure for implementing double-ended priority queues.

Observe that an n-node interval heap represents a multiset of $2n-1$ or $2n$ elements. By inserting elements into an initially empty interval heap, it is clear that every finite multiset can be represented as an interval heap. The insertion procedure is essentially the procedure for insertion into an ordinary heap, namely the heap of all left-end values or all right-end values (together with the value at the L-node, when it does not contain an interval). The heap given by removing all right-end values is a MIN heap, whereas the heap given by removing all left-end values is a MAX heap. We call these heaps the *underlying MIN and MAX heaps*. It follows that the technique of Gonnet and Munro [7] can be applied to obtain a faster insertion algorithm. It

is a simple exercise to prove that interval heaps also support efficiently the remaining operations for double-ended priority queues.

Proposition 2.1. *Interval heaps support DELETEMIN and DELETEMAX operations in $\Theta(\log n)$ time.*

Proof. We sketch only the argument for DELETEMIN operations. (The argument for DELETEMAX operations is handled in a very similar way.) Let I be an interval heap. If the root node is the L-node of I, then the DELETEMIN operation is executed by simply dropping the MIN value from the structure. Assume that the root node is not the L-node of I, and that $[a, b]$ is the interval at the root. Now MIN $= a$ and the DELETEMIN operation can be executed as follows. Look at the underlying MIN heap of I. We replace a with the left-end value x, say, of the L-node, and remove the left-end value from the L-node. (If the L-node now has no values associated with it, then we delete it; otherwise, we have an L-node with one value.) We now trickle down the value x until we have an underlying MIN heap once more. It is clear that the DELETEMIN operation takes $\Theta(\log n)$ time. ■

THEOREM 2.2. *Interval heaps are an efficient, implicit data structure for implementing double-ended priority queues which support MIN and MAX operations in $\Theta(1)$ time and the remaining operations in $\Theta(\log n)$ time.*

From the given arguments it should be clear that interval heaps have essentially the same efficiency as heaps (or any variant of them), because the update operations work almost entirely on the separate underlying heaps of all left-end values and all right-end values, respectively.

A remaining question concerns the construction of an n-node interval heap from a given multiset of $2n-1$ or $2n$ values in arbitrary order. A linear-time interval-heap construction algorithm is obtained by applying the well-known construction of Floyd [6] as follows. The crucial observation is contained in the following lemma.

LEMMA 2.3. *Let I_1 and I_2 be interval heaps, $[a, b]$ an (arbitrary) interval, and I the tree with I_1 and I_2 as its left and right subtrees, respectively, and $[a, b]$ be the interval at the root. Then, the heap property for I can be restored in time proportional to the maximum of the depths of I_1 and I_2.*

Proof. (Note that I is not necessarily an interval heap, because it may not be heap structured.) We give a simple, but not very efficient procedure for it. Look at the trees T_1 and T_2 of all left-end and all right-end values, respectively, both structured like I. Remove a from the root of T_1 and follow the usual procedure for heaps to fill its place and heapify the structure. It will put some value a' at the root (the smallest from the multiset) but leave an open place at some leaf node v. Remove b from the root of T_2 and carry out the same procedure. It will put some value b' at the root (the largest from the multiset) and leave an open place at some leaf node w. Observe that $a' \leqslant b'$ and that all values in I_1 or I_2 are contained in $[a', b']$. It is easily seen that combining the values at corresponding nodes of T_1 and T_2 leads to valid intervals that satisfy the heap property. Now insert a at node w as in interval heaps and, subsequently, b at node v. This will restore the structure of I and ensure that it satisfies the heap property, as required, in time no greater than four times the maximum of the depths of I_1 and I_2 (plus one). ■

THEOREM 2.4. *Interval heaps can be created in $\Theta(n)$ time and constant extra space.*

3. APPLICATIONS OF INTERVAL HEAPS

We discuss a number of applications of interval heaps in the theory of data structures; see also Section 4. Clearly interval heaps can be used to obtain an efficient, implicit data structure for implementing **order-statistics trees** as defined by Atkinson *et al.* [3]. We mention a number of different applications to problems for which no efficient, implicit data structures were given before. The work of Klein *et al.* [8] provides an additional application.

In the usual embedding in an array H, an interval heap always grows and shrinks at the right-hand end (at the current L-node) when insertions and deletions are carried out. It may be useful to have a structure with the property that it shrinks on the left when DELETEMIN is carried out and on the right when DELETEMAX is carried out. An example of where this can be used is **double heapsort**, the obvious modification of heapsort in which an array is sorted by working both at the low and at the high end of the ordering simultaneously (i.e. alternatingly). For this example interval heaps provide a particularly cheap and easy solution, in the following manner. Assume the array to be sorted contains $2n$ values. Rearrange the values into an n-node interval heap in $\Theta(n)$ time, with the left-end and the right-end values of the interval associated to node i in positions $n+1-i$ and $n+i$ of the array (i from n to 1). Because DELETEMIN and DELETEMAX operations alternate in double heapsort, the interval heap can indeed be made to shrink at the proper ends during every sort step in this representation. It makes double heapsort a valid sorting algorithm for arrays which, in view of the efficiency of the deletion operations, will be competitive with heapsort.

A more interesting application concerns the range query problem in the theory of multidimensional data structures and computational geometry; see, e.g. Preparata and Shamos [13]. We study only the 1D version of it and, in fact, consider the complementary-range query problem first. This problem is defined as follows: *given a multiset V of n points on the real line and any interval $[x, y]$, determine the points of V that are not contained in the interval.* In the dynamic variant of the problem one asks for an algorithm (and a data structure) that is efficient even when insertions and deletions of points are allowed. Organizing the points of V in a (dynamic)

balanced tree [9] easily leads to a solution with a query time of $\Theta(\log n + K)$, where K is the number of points to be reported.

THEOREM 3.1. *There is an implicit data structure for the (dynamic) 1D complementary-range query problem that can be built in $\Theta(n)$ time and supports queries in $\Theta(K)$ time, where K is the number of reported points.*

Proof. Organize the points of V into an interval heap I. We show that complementary-range queries can be answered in $\Theta(K)$ time. Let $[x, y]$ be the query interval. The query algorithm starts at the root and proceeds recursively in the following manner whenever a node v is visited. We distinguish the two cases that can arise when we compare $I(v)$ and $[x, y]$.

Case (i) $I(v) \subseteq [x, y]$. In this case no point of V at v or in its subtree must be reported. The algorithm ignores the entire subtree and progresses to the next node.

Case (ii) $I(v) \cap [x, y] \subset I(v)$. In this case $I(v)$ contains at least one end-point that is not contained in $[x, y]$ and this value must be reported. If the other endpoint is not contained in $[x, y]$ either, it must be reported as well. If v is not a leaf, the algorithm will visit the children of v next.

The query algorithm is easily implemented as a preorder traversal of I, with pruning of the subtrees that are recognized as not containing any answers (Case (i)).

The complexity of the query algorithm is $\Theta(K)$ by the following argument. The algorithm is linear in the number of nodes that are visited. If Case (ii) arises in a node, the cost for visiting the node can be charged to the one or two points that are reported at this node. If Case (i) arises in a node v, then observe that we can have reached v only if Case (ii) arose in the parent of v. (The argument is trivially modified when v is the root.) Thus, the cost for visiting v can be charged to the point(s) reported at the parent of v. As no node has more than two children, no point can obtain more than $\Theta(1)$ extra charge. The time bound of $\Theta(K)$ follows. ■

Note that the implicit data structure in the proof of Theorem 3.1 is dynamic in the following sense. It supports insertions in $\Theta(\log n)$ time, but deletions (in the same time bound) only if the location of the element to be deleted is known. The fact that interior points of a heap can be deleted in $\Theta(\log n)$ time is well known, and the algorithm for heaps easily extends to interval heaps.

The 1D range query problem is defined as follows: *given a multiset V of n points and any interval $[x, y]$, determine the points of V that are contained in the interval.* It is much harder to design a dynamic, implicit data structure for the 1D range query problem, which can be built in linear time, than it was for the complementary-range query problem. We mention only the following result. The 1D halfspace query problem is the version of the 1D range query problem in which either $x = -\infty$ or $y = +\infty$.

THEOREM 3.2.

1. *There is an implicit data structure for the (static) 1D range query problem that can be built in $\Theta(n \log n)$ time and supports queries in $\Theta(\log n + K)$ time.*
2. *There is an implicit data structure for the (dynamic) 1D halfspace query problem that can be built in $\Theta(n)$ time and supports queries in $\Theta(K)$ time.*

Proof.

1. Sort V in $\Theta(n \log n)$ time, and assign the values of V in this order to the consecutive nodes visited in an inorder traversal of a heap structure of n nodes. Let T be the resulting tree, and $[x, y]$ an arbitrary query interval. (Observe that T is a binary search tree [9].) Let $T(v)$ be the value at node v of T. Consider the following algorithm for answering the query. The algorithm consists of two phases. Phase I starts at the root, and proceeds as follows in any node v that is reached. If $v \in [x, y]$ then stop Phase I and start Phase II (at v). Otherwise, proceed to the left child of v if $y < T(v)$, and to the right child of v if $T(v) < x$. If v has no children, then stop Phase I and report that there are no answers. Clearly Phase I must finish within $\Theta(\log n)$ time. Next consider Phase II, and suppose it starts at node v. Report $T(v)$, and proceed along two lines: the x-line and the y-line. (We sketch only what happens along the x-line.) The x-line starts at the left child of v, and the algorithm proceeds as follows in any node w that is reached. If $T(w) \in [x, y]$ (or, equivalently, $x \leqslant T(w)$), then report $T(w)$ and all values in the right subtree of w and proceed to the left child of w. If $T(w) \notin [x, y]$ (or, equivalently, $T(w) < x$), then simply proceed to the right child of w. If w has no left or right child, respectively, then the x-line stops. Phase II ends when both the x-line and the y-line have ended. Clearly Phase II reports all answers to the query, and takes only $\Theta(\log n + K)$ time. (The query algorithm is very similar to that for range trees [13].)
2. Observe that 1D halfspace queries can be viewed as 1D complementary-range queries, because the complement of a 1D halfspace is a 1D halfspace (i.e. a range). The technique of Theorem 3.1 applies without any change. ■

4. INTERVAL *HEAPS

The idea of interval heaps suggests another, related implicit data structure which we will call an **interval *heap**. Essentially the interval *heap is an interval heap with the ordering relation between the associated intervals reversed. We use the same notation that we introduced for interval heaps.

Definition 4.1. *An* interval *heap *is a heap structure that is either empty or satisfies the following conditions (the* ***heap property**):*

1. *For each node v different from the L-node, I(v) is an interval [a, b] (a, b ∈ X).*
2. *For the L-node v, I(v) is either a single value a (a ∈ X), considered as the 'interval' [a, +∞], or an interval [a, b] (a, b ∈ X).*
3. *For all nodes v and w, if v is a child of w, then I(w) ⊆ I(v).*

Observe again that interval *heaps are easily implemented as implicit data structures.

LEMMA 4.1. *Let I be an interval *heap representing a multiset of values V, and [a, b] be the interval at the root. Then, a and b are medians of V.*

Proof. Let I have n nodes ($n \geqslant 1$) and let $|V| \geqslant 2$. (Observe that $|V|$ is either $2n-1$ or $2n$.) By the *heap property each interval in I has a left-end value no greater than a and a right-end value no less than b. When $|V| = 2n$, it follows that a and b are the nth and $(n+1)$st value in the ordering of V. When $|V| = 2n-1$, the position of a and b depends on the value c at the L-node of I. If $c \leqslant a$, then a and b are the nth and $(n+1)$st value of V as before, otherwise (that is, when $b \leqslant c$) they are the $(n-1)$st and nth value in the sorted order of V. ■

It follows that the median of a multiset can always be read off at the root of the interval *heap. Hence, interval *heaps can be used to dynamically maintain the median of a multiset. Although there are implicit data structures for this already, for example, the **mmm-heap** of Atkinson *et al.* [3], the interval *heap is a simpler structure with the same performance. However, interval *heaps also seem to be useful for a particular type of range query in the multidimensional case, although as implicit data structures they seem to be more complex than (say) a pair of interval heaps back-to-back for the equal halves of a multiset. We will pursue the analysis here to see the intricacies of this type of structure.

It is easily seen that interval *heaps can be built from scratch in $\Theta(n)$ time using $O(1)$ additional space. One algorithm for it could be as follows. (Without loss of generality we assume that $|V| = 2n$.) Determine a and b as the nth and $(n+1)$st elements in the sorted order of V using the $\Theta(n)$-time median finder of Blum *et al.* [4] as modified by Lai and Wood [10, 11] to work implicitly. Let $V_a \subseteq \{x \in V | x \leqslant a\}$ and $V_b = \{x \in V | b \leqslant x\}$ be such that $V_a \cup V_b = V$ and $|V_a| = |V_b| = n$. (V_a and V_b are easily found implicitly in $\Theta(n)$ time as well.) Now organize the elements of V_a as a heap under the $\geqslant$-ordering and the elements of V_b as a heap under the $\leqslant$-ordering. (Both heaps have the same underlying structure.) Overlapping the two heaps and taking the two values at every node as the defining endpoints of an interval results in a valid interval *heap. In the case that $|V| = 2n+1$, we let a and b be the $(n+1)$st and $(n+2)$nd values in the sorted order of V. Then, V_a contains n values and V_b contains $n-1$ values. This ensures that the L-node satisfies the *heap property.

We now demonstrate that by considering the two underlying heaps separately, we can obtain $O(\log n)$ update time.

LEMMA 4.2. *Interval *heaps support INSERT operations in $\Theta(\log n)$ time.*

Proof. Let $[a, b]$ be the interval of the root of an interval *heap of $2n$ values. The case of $2n+1$ values is similar and is, therefore, left to the reader. Let x be the new value to be inserted. There are three cases to consider:

Case 1: $x < a$. The new *heap has $2n+1$ values; thus, the $(n+1)$st and $(n+2)$nd values must appear at the root. As a and b are, before insertion, the nth and $(n+1)$st values, this implies that they remain in the root. All that remains is to insert x into the underlying MAX heap. This is done with the standard algorithm that creates a new L-node with a single value.

Case 2: $a \leqslant x \leqslant b$. In this case, $[x, b]$ becomes the new interval at the root and a is inserted into the underlying MAX heap.

Case 3: $b < x$. This is the more difficult case. We need to move a value into the underlying MAX heap and this should be the smallest value in the underlying MIN heap, that is, b itself. So, we first remove b from the root, replacing it with x. In the underlying MIN heap we trickle x down until it finds its place. This is the standard heap deletion operation. Second, we insert b, in the standard manner, into the underlying MAX heap. ■

In a similar manner, with the same time bounds, we can implement DELETE, if the position of the deleted element is given; in particular, we can delete either of the medians in $O(\log n)$ time.

LEMMA 4.3. *Interval *heaps support DELETE in $\Theta(\log n)$ time.*

Proof. (We assume here that the location of the element to be deleted from the *heap is known.) Suppose x must be deleted from the multiset represented by the interval *heap, and let v be the node where x appears. If v is the L-node, then we can just delete x and are done. Otherwise, assume, without loss of generality, that $I(v) = [x, a]$. Delete x and use the ordinary algorithm to fill the hole in the underlying MIN heap of left-end values below v. It recreates the *heap property, except that a hole may appear in some leaf node u different from the L-node. Remove an element y from the current L-node and add it to u with the algorithm of Lemma 4.3. ■

As an application of interval *heaps we consider a restricted range searching problem. We say that an interval $[x, y]$ is a **median range** with respect to a multiset V, if the median of V is contained in the query interval. We have the associated query:

MED—determine the median value.

We can define **minimum range** and **maximum range** in a similar manner. It is straightforward to prove the following result:

THEOREM 4.4. *There is an implicit data structure for the (dynamic) median range query problem that requires $\Theta(n)$ time to build and supports median range queries in $\Theta(K)$ time, where K is the number of answers.*

Just as we designed an interval heap to solve both MIN and MAX queries, we can combine the interval heap and the interval *heap to give the **mmm-interval heap** for MIN, MED and MAX queries. It has four values at each node except, of course, for the L-node, which has between 1 and 4 values. At a node with values $a \leqslant b \leqslant c \leqslant d$, we require that the **outer interval** $[a, d]$ contains all outer intervals at the nodes in its subtree and that the **inner interval** $[b, c]$ is contained in all inner intervals at the nodes in its subtree.

The mmm-interval heap can be updated in $O(\log n)$ time and it can answer complementary, median, minimum, and maximum range queries in $\Theta(K)$ time, where K is the number of answers. Also, the mmm-interval heap can be constructed in $\Theta(n)$ time using constant additional space.

5. GENERALIZATIONS OF INTERVAL HEAPS

In this section we will consider the d-dimensional analog of interval heaps. As a main problem we consider the design of an implicit $2d$-**ended priority queue** for representing multisets $V \subseteq X^d$ which supports the following operations:

1. MIN_i—determine the element with the smallest ith coordinate.
2. MAX_i—determine the element with the largest ith coordinate.
3. INSERT—add a value $x \in X^d$ to the submultiset.
4. DELETEMIN_i—remove the element with the smallest ith coordinate.
5. DELETEMAX_i—remove the element with the largest ith coordinate.

We will see that the resulting, implicit data structure can be used to obtain an efficient solution to the d-dimensional complementary-range query problem as well. We assume $d \geqslant 2$.

The implicit data structure we propose for implementing $2d$-ended priority queues is a natural generalization of the interval heap. What we need is the proper generalization of an interval containing some multiset of points $W \subseteq X^d$. Assume first that $|W| \geqslant 2d$.

Definition 5.1. *The **d-interval** defined by a multiset of points $W \subseteq X^d$ is the* **box** *enclosed by (i.e. the intersection of) the hyperplanes H_1 through H_{2d} that are defined as follows:*

1. *H_1 is the 1-hyperplane defined by a point $h_1 \in W$ with least 1st coordinate, H_2 is the 1-hyperplane defined by a point $h_2 \in W - \{h_1\}$ with largest 1st coordinate.*
2. *For $1 < i \leqslant d$, H_{2i-1} is the i-hyperplane defined by a point $h_{2i-1} \in W - \{h_1, \ldots, h_{2i-2}\}$ with least ith coordinate, and H_{2i} is the i-hyperplane defined by a point $h_{2i} \in W - \{h_1, \ldots, h_{2i-1}\}$ with largest ith coordinate.*

(For simplicity we use the term ***i*-hyperplane** to denote any hyperplane of points with fixed ith coordinate.) If $|W| = j < 2d$, then we could still use the same definition, but only j hyperplanes would be defined and the box would not be entirely bounded. For finite multisets $W \subseteq X^d$ the defining d-interval I_W (that is, the hyperplanes H_i and the boundary points h_i) is easily determined in $\Theta(d^2)$ time. Abusing notation, we will say that W is **contained in** I_W although, strictly speaking, only $W - \{h_1, \ldots, h_{2d}\} \subseteq I_W$. A d-interval is uniquely represented by the (ordered) sequence of boundary points $h_1, \ldots, h_{2d}$ or by $h_1, \ldots, h_j$ in the case of an incomplete d-interval and, therefore, denoted by $[h_1, \ldots, h_{2d}]$ or $[h_1, \ldots, h_j]$, respectively.

Definition 5.2. *A **d-interval heap** is a heap structure that is either empty or satisfies the following conditions (the **d-heap property**):*

1. *For each node v different from the L-node, $I(v)$ is a d-interval $[a_1, \ldots, a_{2d}]$ ($a_i \in X^d$, for $1 \leqslant i \leqslant 2d$).*
2. *For the L-node v, $I(v)$ is either a multiset of values from X^d with $1 \leqslant |I(v)| < 2d$ or a d-interval $[a_1, \ldots, a_{2d}]$ ($a_i \in X^d$ for $1 \leqslant i \leqslant 2d$).*
3. *For all nodes v and w, if v is a child of w then $I(v) \subseteq I(w)$.*

The definition is to be understood as for (1-)interval heaps, *cf.* Section 2. A d-interval heap represents the multiset V consisting exactly of the values of X^d that are used to delimit the boundaries of the d-intervals and, in the case that the L-node does not hold a d-interval, the elements of the stored multiset. Observe again that d-interval heaps are easily implemented as implicit data structures. The following proposition is immediate.

Proposition 5.1. *d-Interval heaps support MIN_i and MAX_i operations in $\Theta(d)$ time.*

LEMMA 5.2. *d-Interval heaps support INSERT operations in $\Theta(d \log n)$ time.*

Proof. Let I be a d-interval heap, $x \in X^d$ a value to be added to I. If I is empty or the L-node does not contain a full d-interval but coincides with the root, then the insertion is trivially processed at the root. Consider the non-trivial case that the L-node v does not contain a full d-interval and is different from the root. Let w be the parent of v, and $I(w) = [a_1, \ldots, a_{2d}]$. If $x \in I(w)$, i.e. x is contained in the box at w, then add x to $I(v)$ and turn $I(v)$ into a d-interval if it now contains $2d$ values. This preserves the d-heap property. If $x \notin I(w)$, then climb towards the root and check at every node w that is visited whether $x \in I(s)$ for s the parent of w. (For w

equal to the root, we assume that $I(s)$ is the infinite box X^d.) Let w_0 be the first node encountered on the path towards the root for which the test returns true. Note that w_0 is well defined (the test will always be true at the root) and that in every node the test $x \in I(s)$ can be evaluated in $\Theta(d)$ time. Let $I(w_0) = [a_1, ..., a_{2d}]$, where clearly $x \in I(w_0)$. Now replace $I(w_0)$ by the d-interval defined by the multiset $\{a_1, ..., a_{2d}, x\}$. The defining boundary points will consist of x and all of the points $a_1, ..., a_{2d}$ apart from one, say a_t. Observe that $a_t \in I(w_0)$. We now use the fact that in a heap structure one can always determine in $\Theta(1)$ time in which subtree a given node is located. Thus, we can trace the path from w_0 back towards v (the L-node) and find the first node w_1 with $a_t \notin I(w_1)$. If no such node w_1 exists, then the journey ends at v and we can simply add a_t to the multiset at v. This will restore the d-heap property and we can stop. If w_1 exists, then we proceed with a_t at w_1 as we did with x at w_0. This gives a new d-interval at w_1, and an extra point $a_{t'} \in I(w_1)$. Proceed downwards and repeat the same procedure with $a_{t'}$, etc., until eventually v is reached. Whatever extra point we have, it can be added to v and we are done.

It remains to consider the case that the L-node v contains a full d-interval. Allocate the next node u of the heap structure, and initialize $I(u)$ to be the empty multiset. Now start the insertion procedure for x at u, and proceed exactly as before. This will restore the d-heap property, and leave a valid d-interval heap with x inserted. Clearly the whole insertion process takes only $\Theta(\log n)$ steps of cost $\Theta(d)$ each, hence, $\Theta(d \log n)$ total time. ■

LEMMA 5.3. *d-Interval heaps support $DELETEMIN_i$ and $DELETEMAX_i$ operations in $\Theta(d^2 \log n)$ time.*

Proof. We sketch the argument for only DELETEMIN_i operations. (The argument for DELETEMAX_i operations is handled analogously.) Let I be a d-interval heap. If the root node happens to be the L-node of I, then the DELETEMIN_i operation is executed by simply dropping the MIN_i-value from the structure (*cf.* the proof of Proposition 2.3). Let us assume that the root is not the L-node of I and that $[a_1, ..., a_{2d}]$ is the d-interval at the root.

Clearly MIN_i is determined as one of the points a_j $(1 \leqslant j \leqslant 2d)$ with least ith coordinate. (Note that it is not necessarily the point a_{2i-1}, because of the way d-intervals are determined.) Let $a_j = \text{MIN}_i$. Essentially, we now wish to delete a_j and fill its place, but this is slightly more tedious than it was in the 1D case. Let the children of the root be u and v. (If the root has only one child, then the argument is completely similar.) We now show that the d-interval at the root (minus a_j) can be restored to be a box enclosing all other points by borrowing precisely one point from either $I(u)$ or $I(v)$.

Consider the way the d-interval at the root is formed. Clearly $a_1, ..., a_{j-1}$ will be determined as before, but a_j leaves a hole. The point with extreme jth coordinate-value that replaces it could be a point of $I(u)$ or $I(v)$, but it could also be one of the remaining a_k $(j < k \leqslant 2d)$. In the former case, we have a suitable replacement and can complete the d-interval with $a_{j+1}, ..., a_{2d}$ as we had. In the latter case, a_j is replaced by some a_k $(j < k)$ and we have created a hole further down the list. Thus, replace a_j by a_k and continue the construction, which forces us to take $a_{j+1}, ..., a_{k-1}$. The replacement for a_k is determined in a similar way. Eventually the hole is filled with an element from $I(u)$ or $I(v)$ and the construction is complete. Note that it requires $\Theta(d)$ steps of $\Theta(d)$ work to complete the construction. The resulting d-interval indeed encloses both $I(u)$ and $I(v)$, i.e. the original versions of it, and, thus, is a proper d-interval at the root.

It is clear that we must now iterate the process. Say the one point needed to restore the d-interval at the root came from $I(u)$. Then, remove this point from $I(u)$ and proceed as before to fill its place. Eventually, the iteration ends at a node u belonging to the frontier of I, where we just delete a point. If u is the L-node of I, then we can stop altogether, because I has been restored to be a valid d-interval heap. Otherwise, remove an arbitrary point x from the L-node and (subsequently) insert it at u by following the procedure of Lemma 4.2. This procedure will restore I to be a valid d-interval heap, in $\Theta(d^2 \log n)$ total time. ■

We conclude:

THEOREM 5.4. *d-Interval heaps are an efficient, implicit data structure for implementing $2d$-ended priority queues which support MIN_i and MAX_i operations in $\Theta(d)$ time and the remaining operations in $\Theta(d^2 \log n)$ time.*

A remaining question concerns the construction of a d-interval heap from a multiset of n points in X^d. A linear time construction can be given in the same way as in Theorem 2.4.

THEOREM 5.5. *d-Interval heaps can be created in $\Theta(d^3 n)$ time.*

Proof. (We give only a brief outline of the construction.) Essentially, we follow the same algorithm as Floyd's [6] for constructing ordinary heaps. Begin by assigning the elements in groups of $2d$ to the consecutive nodes of a heap structure, level by level. Starting at the lowest level and working upwards, again level by level, we heapify the subtrees until in the end the entire structure has been turned into a proper d-interval heap. In a typical step we visit a node w, whose children u and v are roots of subtrees that were turned into proper d-interval heaps in the previous stage. Ignore the multiset S of $2d$ points at w for the moment, and consider the points currently at u and v. Compute the d-interval determined by these points (it will be a box that encloses all other points in the two subtrees), and assign it to w. Say the interval took i points from $I(u)$ and j points from $I(v)$. Delete these points from the d-interval heaps

at u and v, respectively, by the technique of Lemma 4.3. It will take $\Theta(d^3 \cdot \text{height}(w))$ time, because $i+j=2d$. Finally, insert i elements of S (one after the other) in the open places in the subtree at u and likewise the remaining j elements of S in the open places in the subtree at v, by the technique of Lemma 4.2 (in the d-interval heap rooted at w). In a further $\Theta(d^3 \cdot \text{height}(w))$ steps this will heapify the entire structure at w. By the same analysis as for Floyd's algorithm, it follows that the entire construction of the d-interval heap takes $\Theta(d^3 n)$ time. ■

The most interesting application of d-interval heaps again seems to be in the theory of multidimensional data structures. In particular, consider the d-dimensional variant of the complementary-range query problem: *given a multiset V of n points in d-dimensional space and a d-interval B (an isooriented box), determine the points of V that are not contained in B.* (In the ordinary range query problem one wants to determine the points inside B.)

THEOREM 5.6. *There is an implicit data structure for the (dynamic) d-dimensional complementary-range query problem that requires $\Theta(d^2 K)$ time to build and supports queries in $\Theta(d^3 n)$ time.*

Proof. Organize the points of V into a d-interval heap I. Complementary-range queries can now be answered as follows, by very much the same technique as in Theorem 3.1. Let B be the query box. The query algorithm starts at the root and proceeds recursively in the following manner, whenever a node v is visited. Two cases can be distinguished.

Case (i): $I(v) \subseteq B$. In this case no point of V appearing inside $I(v)$ and, hence, in v's subtrees, is reported. Thus, each of the $2d$ boundary points of $I(v)$ must be tested and possibly reported (when not contained in B).

Case (ii): $I(v) \cap B \subset I(v)$. In this case at least one side of $I(v)$, that is, one of the $2d$ bounding hyperplanes, must lie strictly outside of B. Thus, at least one of the boundary points must be reported (namely, the point that determines this hyperplane), but clearly the other boundary points must be tested as well and reported if not inside B. If v is not a leaf, the algorithm will visit the children of v next.

The query algorithm is easily implemented as a preorder traversal of I, that visits only the nodes where answers must be reported. Once the algorithm comes to a node where it is no longer required to proceed (Case (i)) it backs up, charging the cost for the visit to the reported points (if any) or to the parent node (where it must have done some reporting). All testing requires $\Theta(d^2)$ time per visit, and $\Theta(d)$ per point reported. Thus, the algorithm takes at most $\Theta(d^2 K)$ time. ■

Note that the implicit data structure in the proof of Theorem 5.6 is dynamic in the following sense: Insertions are supported in $\Theta(d \log n)$ time, but deletions (in a bound of $\Theta(d^2 \log n)$ time) only if the location of the element to be deleted is known. Observe also that the query interval B does not have to be fully d-dimensional. It follows, for example, that Theorem 5.6 holds also for complementary partial match querying.

ACKNOWLEDGEMENTS

This work was supported under a Natural Sciences and Engineering Research Council of Canada Grant No. A-5692 while the second author was at the University of Waterloo.

REFERENCES

[1] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA (1974).
[2] M. D. Atkinson, J.-R. Sack, N. Santoro and Th. Strothotte, *An Efficient, Implicit Double-Ended Priority Queue*, Technical Report SCS-TR-55, Carleton University, School of Computer Science (1984).
[3] M. D. Atkinson, J.-R. Sack, N. Santoro and Th. Strothotte, Min–max heaps and generalized priority queues, *Communications of the ACM*, **29**, pp. 996–1000 (1986).
[4] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest and R. E. Tarjan, Time bounds for selection, *Journal of Computer and System Sciences*, **7**, pp. 448–461 (1973).
[5] S. Carlsson, The deap: a double-ended heap to implement double-ended priority queues, *Information Processing Letters*, **26**, pp. 33–36 (1987).
[6] R. W. Floyd, Algorithm 245: Treesort 3, *Communications of the ACM*, **7**, pp. 701 (1964).
[7] G. H. Gonnet and J. I. Munro, Heaps on heaps, *SIAM Journal on Computing*, **15**, pp. 964–971 (1986).
[8] R. Klein, O. Nurmi, Th. Ottmann and D. Wood, The dynamic fixed windowing problem, *Algorithmica*, **4**, pp. 530–550 (1989).
[9] D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison-Wesley, Reading, MA (1973).
[10] T. W. H. Lai, *Space Bounds for Selection*, Master's thesis, Department of Computer Science, University of Waterloo (1987).
[11] T. W. H. Lai and D. Wood, *Implicit Selection*, Technical Report 32, Department of Computer Science, University of Waterloo (1988).
[12] J. I. Munro, An implicit data structure supporting insertion, deletion, and search in $O(\log^2 n)$ time, *Journal of Computer and System Sciences*, **33**, pp. 66–74 (1986).
[13] F. P. Preparata and M. I. Shamos, *Computational Geometry*, Springer-Verlag, New York (1985).
[14] T. A. Standish, *Data Structure Techniques*, Addison-Wesley, Reading, MA (1980).
[15] J. van Leeuwen and D. Wood, Dynamization of decomposable searching problems, *Information Processing Letters*, **10**, pp. 51–56 (1980).
[16] J. W. J. Williams, Algorithm 232, *Communications of the ACM*, **7**, pp. 347–348 (1964).