INTERVAL HEAPS

J. van Leeuwen, D. Wood

RUU-CS-87-1
January 1987

Rijksuniversiteit Utrecht

Vakgroep informatica

Budapestlaan 6 3584 CD Utrecht
Corr. adres: Postbus 80.012 3508 TA Utrecht
Telefoon 030-53 1454
The Netherlands

INTERVAL HEAPS

J. van Leeuwen, D. Wood

Technical Report RUU-CS-87-1
January 1987

Department of Computer Science
University of Utrecht
P.O.Box 80.012, 3508 TA Utrecht
The Netherlands

Interval Heaps *

Jan van Leeuwen[†] Derick Wood[‡]

March 31, 1988

Abstract

We present a simple, implicit data structure for implementing a double-ended priority queue. The data structure can be viewed as a natural generalization of the heap, and is different from a data structure for the same problem recently proposed by Atkinson et al. 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 and K the number of answers of a query. Several related ideas and applications are also discussed.

Keywords and phrases: implicit data structure, heap, double-ended priority queue, median searching, range searching, complementary range searching.

1 Introduction

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

1. MIN — determine the smallest value.
2. MAX — determine the largest value.
3. INSERT(x) — add a value $x \in X$ to the subset.

*This work was supported under a Natural Sciences and Engineering Research Council of Canada Grant No. A-5692

[†]Department of Computer Science, University of Utrecht, P.O. Box 80.012, 3508 TA UTRECHT, The Netherlands

[‡]Data Structuring Group, Department of Computer Science, University of Waterloo, WATERLOO, Ontario N2L 3G1, CANADA

4. DELETEMIN — remove the smallest value.

5. DELETEMAX — remove the largest value.

In this paper we will consider the problem of designing an efficient implicit data structure for double-ended priority queues, that is, a data structure for maintaining a set 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 (cf. [1,8,13]). In particular we will assume familiarity with heaps. Heaps only support the operations (1), (3) and (4) or, alternatively, the operations (2), (3) and (5) from the list above.

An interesting implicit data structure for implementing double-ended priority queues was proposed by Atkinson et al. [2] in 1984 (see also [3] The MIN/MAX heap proposed by Atkinson et al. is organised 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-levels nodes must be greater than or equal to all values at descendant nodes. Atkinson et al. show that MIN/MAX heaps can be constructed 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. (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 organised 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.

In this paper we propose a different implicit data structure for double-ended priority queues that seems to generalize heaps in a more natural way. The data structure is called an *interval heap*, and essentially consists of a heap in which each node carries a pair of values a, b (with $a \leq b$) from the current set, considered as the endpoints of an interval of X. The usual heap property is replaced by the requirement that, for each node, the interval at this 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 subset. A more precise definition of interval heaps is given in Section 2. In Section 2 we prove 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 an interesting variant of interval heaps that we call *interval *heaps*. In interval *heaps the heap property is the reverse of the one for interval heaps, namely, 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 set of values. The structure allows us to dynamically maintain the median. A 'natural' bottom-up approach to updating interval *heaps that is analogous to the approach taken for interval heaps yields an $O(\log^2 n)$ update time. Fortunately, however, a different approach yields an $O(\log n)$ update time. Interval *heaps enable us 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 multi-dimensional 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 multi-dimensional data structures and computational geometry. The complementary range searching problem is only the first of a number of problems currently being explored from this perspective.

2 Interval Heaps

There are two ways of looking at ordinary heap structures with n elements (see for example, [1,8,13]).

1. As a binary tree of n nodes with all levels completely filled from left to right except for, perhaps, the lowest level which only contains 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 below, but note that all our manipulations could be carried out by simple address-calculations and accesses on the array H (as for 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*. In this section we introduce interval heaps, which are obtained by associating (closed) intervals $[a, b]$ $(a, b \in X)$ to the nodes. Let $I(v)$ denote the interval associated with node v. Let the n^{th} (and last) node of a heap structure be called the *left-end 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 $[a, b](a, b \in X)$.*
2. *For the L-node v, $I(v)$ is either a single value a $(a \in X)$ or an interval $[a, b](a, b \in X)$.*
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 set-size is odd or even.

An interval heap represents the set or, more precisely, the multi-set 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 will be that interval heaps are a viable implicit data structure for implementing double-ended priority queues. We begin by considering the repertoire of operations given in Section 1.

Proposition 2.1 *Interval heaps support MIN and MAX operations in $\Theta(1)$ time.*

Proof: Let I be an interval heap and r be the root node of I. If r is the L-node of I and contains only a single value a ($a \in X$), then the (multi-)set represented by I is the singleton set $\{a\}$ and MIN = MAX = a. Otherwise, let $[a, b]$ be the interval at r. By the heap property it follows that the entire set represented by I must be contained in the interval $[a, b]$ and that (hence) MIN = a and MAX = b. (If r does not exist, then I is empty and MIN = MAX = undefined.) Thus MIN and MAX can be determined in $\Theta(1)$ time. □

Lemma 2.2 *Interval heaps support INSERT operations in* $\Theta(\log n)$ *time.*

Proof: Let I be an interval heap and x a value to be added to I. If I is empty, initiate a root node r and let x be the single value at r. If I is nonempty and its L-node v contains a single value a ($a \in X$), then proceed as follows. Compare a and x, and assume, without loss of generality, that $a \leq x$. Replace the value a at v by the interval $[a, x]$. (Note that the new interval may violate the heap property at the parent of v.) Assume inductively that we are considering a node v and its parent w, and that we have just replaced the value or interval at v by the interval $[a, x]$ (extending the original value or interval to the right). Let the interval associated with w be $[c, d]$ where, necessarily, $a \in [c, d]$. When $x \leq d$, the heap property is satisfied and I has been restored to a valid interval heap. When $x > d$, replace the interval at v by $[a, d]$ and the interval at w by $[c, w]$ (which means that the heap property is satisfied everywhere except, perhaps, on the path from w to the root), and observe that the inductive assumption now holds at w and its parent (if it has one). Working upwards along the path from the L-node to the root, the heap property is restored at the expense of only $\Theta(1)$ time per node and (hence) $\Theta(\log n)$ time total.

If I is nonempty and the L-node contains an interval, then proceed as follows. Allocate the next node of the heap structure (call it v) and let x be the single value at v. Let w be the parent of v, and let the interval at w be $[a, b]$. If $x \in [a, b]$ then I is a valid interval heap. Otherwise, assume, without loss of generality, that $b < x$. Now x can trickle up along the path towards the root exactly as before, and the heap property is gradually restored within $\Theta(\log n)$ time. □

Observe that an n-node interval heap represents a set of $2n - 1$ or $2n$ elements. By inserting elements into an initially empty interval heap, the lemma shows that every finite set can be represented as an interval heap. Also, the insertion procedure outlined in the lemma is essentially the procedure for inserting 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, while that given by removing all left-end values is a MAX heap. We call these the *underlying MIN and MAX heaps.* It follows that the techniques of Gonnet and Munro [7] immediately apply to obtain a faster insertion algorithm. It is a simple exercise to prove that interval heaps also support the remaining operations for double-ended priority queues.

Proposition 2.3 *Interval heaps support DELETEMIN and DELETEMAX operations in* $\Theta(\log n)$ *time.*

Proof: We only sketch the argument for DELETEMIN operations. (DELETEMAX operations are 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 (cf. the proof of Proposition 2.1.). Let's assume that the root node is not the L-node of I, and that $[a, b](a, b \in X)$ 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. Delete a and fill its place by following the usual technique for heaps. It means that the empty place is moved down in the tree and the heap property restored until the empty place reaches the bottom of the tree. In the interval heap it means that we have reached a leaf node v whose interval has just lost its left-end value. In case v is the L-node of I, we are done: the new heap structure is again a valid interval heap. (This holds even in the case that v contained just a single value a, which it lost because the a moved up.) If v is not the L-node of I, then pick one of the (at most) two values at the L-node and add it to v (thus restoring the object at v to be an ordinary interval). Now use the procedure outlined in the proof of Lemma 2.2 starting at v, to restore the heap property of I. It is clear that the DELETEMIN operation takes $\Theta(\log n)$ time. □

Theorem 2.4 *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 proofs it should be clear that interval heaps essentially have 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 set of $2n - 1$ or $2n$ values in arbitrary order. A linear time algorithm for it is obtained by applying the well-known construction of Floyd[6], as follows. The crucial observation is contained in the following lemma.

Lemma 2.5 *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 of 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 set) 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 set) and leave an open place at some leaf node w. Observe that $a' \leq b'$ 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 2.6 *Interval heaps can be created in $\Theta(n)$ time and constant extra space.*

Proof: Arrange all values as intervals at the nodes of a heap structure, with a single value perhaps at its L-node. Now heapify the structure, level by level, working from the lowest level upwards as in Floyd's method for ordinary heaps, using lemma 2.5 in every node. This turns the structure into a valid interval heap, and the total time is $\Theta(n)$ by the same analysis as for Floyd's method. □

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 in Atkinson et al.[3]. We mention a number of different applications to problems for which no efficient, implicit data structures were given before.

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 (that is, 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 multi-dimensional data structures and computational geometry (see for example, Preparata and Shamos[12]). In this section we only study the 1-dimensional version of it and, in fact, consider the complementary problem first. This problem is defined as follows: given a set 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 (cf.[8]) 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) 1-dimensional complementary range query problem that requires $\Theta(n)$ time to build and supports queries in $\Theta(K)$ time.*

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 pre-order 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 only have reached v 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 it easily extends to interval heaps.

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

Theorem 3.2 *1. There is an implicit data structure for the (static) 1-dimensional range query problem that requires $\Theta(n \log n)$ time to build and supports queries in $\Theta(\log n + K)$ time.*

2. There is an implicit data structure for the (dynamic) 1-dimensional halfspace query problem that requires $\Theta(n)$ time to build 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 in-order 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; see for example[8].) 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 only sketch 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, $x \leq T(w)$, 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, cf. [12].)

2. Observe that 1-dimensional halfspace queries can be viewed as 1-dimensional complementary range queries, because the complement of a 1-dimensional halfspace is a 1-dimensional halfspace (that is, a range). The technique of Theorem 3.1 applies without any change. □

Munro [11] has developed a dynamic data structure that allows insertions and deletions in $O(\log^2 n)$ time and range queries in $O(K + \log^2 n)$ time, where K is the number of answers. At present, we are unable to match these bounds using interval heaps. We can, based on the dynamization technique of [14], derive a dynamic implicit data structure for range queries as follows. We represent a set of n values using $\lceil n/c \rceil$ blocks of size c, for some $c > 0$, where the last block may be smaller than c. The i^{th} block appears in positions $(i-1)c + 1$ to ic of the array. All values in any block are greater than all values in earlier blocks and less than the values in later blocks. Finally, each block is organized as an interval heap.

To insert an element x into the set, we first locate its block, say, i. Then we delete the maximum value y from block i and insert x into block i (unless x *is* the maximum value, when we continue with the next step). The maximum value z in block $i+1$ is now deleted and the value y is inserted into block $i+1$. We now repeat this rippling process with $i := i+1$ and $y := z$ until we reach the last block. At the last block we simply insert y into its interval heap. Once the last block grows sufficiently large we reorganize the structure to have larger size blocks. Deletions are carried out in a similar manner.

For a given range $[x, y]$, we perform a range query for $[x, y]$ by first determining the blocks x and y fall into. Let these be blocks i and j, respectively.

If $i \neq j$, the range query is easily decomposed into two complementary range queries (one for block i and one for block j) and $j - i - 1$ traversals. On the other hand, if $i = j$, we can do little better than a traversal of the interval heap in block i to determine those values that fall into the range $[x, y]$. Thus, the block size determines the efficiency of range queries. If $c = O(\log n)$, then we obtain logarithmic query time, in the worst case, but updates can cause a ripple through every one of the $\lceil n/\log n \rceil$ blocks at a cost of $O(\log \log n)$ per block. If we choose $c = O(\sqrt{n \log n})$, then both query and update time become $O(\sqrt{n \log n})$ (plus $O(K)$ for queries).

It is an open problem whether one can use interval heaps more effectively for the range query problem.

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 $[a, b]$ $(a, b \in X)$.*
2. *For the L-node v, $I(v)$ is either a single value a $(a \in X)$, considered as the "interval" $[a, +\infty]$, or an interval $[a, b]$ $(a, b \in X)$.*
3. *For all nodes v and w, if v is a child of w then $I(w) \subseteq I(v)$.*

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

Lemma 4.1 *Let I be an interval *heap representing a set 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 $(n \geq 1)$ and let $|V| \geq 2$. (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 n^{th} 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 $c \leq a$, then a and b are the n^{th} and $(n+1)^{st}$ value of V as before, otherwise (that is, when $b \leq c$) they are the $(n-1)^{st}$ and n^{th} value in the sorted order of V. □

It follows that the median of a set 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 set. 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 multi-dimensional 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 set. 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 $\mid V \mid = 2n$.) Determine a and b as the n^{th} and $(n+1)^{st}$ elements in the sorted order of V using the $\Theta(n)$ median-finder of Blum et al.[4] as modified by Lai and Wood [9,10] to work implicitly. Let $V_a \subseteq \{x \in V \mid x \leq a\}$ and $V_b = \{x \in V \mid b \leq x\}$ be such that $V_a \cup V_b = V$ and $\mid V_a \mid = \mid V_b \mid = n$. ($V_a$ and V_b are easily found implicitly in $\Theta(n)$ time as well.) Now organise the elements of V_a as a heap under the $\geq$-ordering and the elements of V_b as a heap under the $\leq$-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 $\mid V \mid = 2n + 1$,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.

It actually is quite intriguing that interval *heaps are dynamic. We first consider an apparently rather costly, bottom-up scheme for insertions.

Lemma 4.2 *Interval *heaps support INSERT operations in* $\Theta(\log^2 n)$ *time.*

Proof: Let I be an n-node interval *heap and x a value to be added to I. If I is empty or contains only one value, then the insertion procedure (at the root) proceeds trivially in $\Theta(1)$ time. Thus, let I be non-trivial ($n \geq 1$) and assume that the L-node (call it v) already contains the value c. Let w be the parent of v, u the sibling of v (if it exists), $I(w) = [a, b]$ and $I(u) = [d, e]$ (if applicable). More generally, the procedure will be applied at any node v where the old interval was just updated to $[x, c]$ or $[c, x]$, with c an endpoint of the original interval $I(v)$, x some new value, the *heap property restored in the subtree rooted at v and $[a, b]$ (still) contained in all intervals appearing **below** v. (This is the invariant at v.) We now distinguish 10 cases depending on the relative ordering of the values involved. Let s be the parent of w (if it exists).

Case I: $x < d < c < a < b < e$. Leave $[d, e]$ assigned to u, but assign $[x, b]$ to v and $[c, a]$ to w. Now observe the following. By assigning $[x, b]$ to v the *heap property may be violated again at the top of v's subtree (the original interval of $[x, c]$ is widened). But, recall that $[a, b]$ and $[x, c]$ were contained in all intervals below v. Thus, $[x, b]$ must be contained in these intervals, and the *heap property is not endangered. Note that the *heap property is restored in the entire subtree of w. In fact, because $I(s) \subseteq [a, b]$ we also have that $I(s) \subseteq I(u)$ (unchanged) and $I(s) \subseteq I(v)$. Thus the invariant applies at w, and the algorithm proceeds upwards.

Case II: $d < x < c < a < b < e$. Leave $[d, e]$ assigned to u, assign $[x, b]$ to v and $[c, a]$ to w. In the same way as in Case I this establishes the invariant at w, and the algorithm proceeds upwards.

Case III: $d < c < x < a < b < e$. Leave $[d, e]$ assigned to u, assign $[c, b]$ to v and $[x, a]$ to w. As in Case I this establishes the invariant at w, and the algorithm proceeds upwards.

Case IV: $d < c < a < x < b < e$. Leave $[d, e]$ assigned to u, assign $[c, b]$ to v and $[a, x]$ to w. As in Case I this establishes the invariant at w, and the algorithm proceeds upwards.

Case V: $d < c < a < b < x < e$ or $d < c < a < b < e < x$. Observe that the *heap property has apparently been restored in the entire tree, and the algorithm can stop!

Case VI: $x < c < d < a < b < e$. This case is slightly more involved. Assign $[c, e]$ to u, $[x, b]$ to v and $[d, a]$ to w. Because $[x, c]$ and $[a, b]$ are contained in every interval appearing below v, the *heap property is certainly maintained in the subtree headed by v. Because the interval assigned to u is widened from $[d, e]$ to $[c, e]$, the *heap property may no longer hold in the subtree of u. But it can be restored in $\Theta(\log n)$ time by letting c trickle downwards in the heap of left-end values. It will replace $[c, e]$ at u by $[y, e]$ for some y with $c \leq y$ but, because of the original *heap constraint with $[d, e]$ at u, $y \leq d$. It follows that $[d, a] \subseteq [y, e]$ and the *heap property is indeed restored at w (and in its subtree). Because $I(s) \subseteq [a, b]$ we also have $I(s) \subseteq I(u)$ and $I(s) \subseteq I(v)$, and it follows that the invariant is restored at w. Thus the algorithm can proceed one node further upwards.

Case VII: $c < x < d < a < b < e$. Assign $[c, e]$ to $u, [x, b]$ to v and $[d, a]$ to w. In the same way as in Case VI the invariant can be made to hold at node w at the expense of $\Theta(\log n)$ time, and the algorithm can proceed upwards.

Case VIII: $c < d < x < a < b < e$. Leave $[d, e]$ assigned to u, assign $[c, b]$ to v and $[x, a]$ to w. As in Case III this establishes the invariant at w, and the algorithm can proceed one step upwards.

Case IX: $c < d < a < x < b < e$. Leave $[d, e]$ assigned to u, assign $[c, b]$ to v and $[a, x]$ to w. As is Case IV this restores the invariant at w, and the algorithm can proceed one step upwards.

Case X: $c < d < a < b < x < e$ or $c < d < a < b < e < x$. As in Case V the *heap property is now restored in the entire tree and the algorithm can stop!

Clearly this does not exhaust all possible cases that must be considered. But the (remaining) cases with c between b and e or beyond e are all symmetric to the cases discussed above. Also, the values need not all be different for the argument to go through. Finally, there is the case to consider that the inductive argument starts at an L-node v which does not contain an initial value, that is, v is a newly allocated node where the value x is inserted. If $I(w) \subseteq x$ (that is, $x \notin [a, b]$), then the *heap property is preserved. If $x \in [a, b]$, then assign b to v and $[a, x]$ to w and proceed with the preceding argument starting at w.

It is clear that by proceeding towards the root, the *heap property will eventually be restored in at most $\Theta(\log^2 n)$ time. □

The above maintainance algorithm for *heaps parallels the one given for interval heaps in Section 2; in this sense, it is the natural one. Unfortunately, as we have seen, it requires $O(\log^2 n)$ update time. We now demonstrate that by considering the two underlying heaps separately, we can obtain an $O(\log n)$ update time. We are able to use a top-down method that avoids the problems appearing in the bottom-up algorithm given above.

Lemma 4.3 *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 n^{th} 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: $a \leq x \leq b$. 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 4.4 *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 set 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 set 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 4.5 *There is an implicit data structure for the (dynamic) median range query problem that requires* $\Theta(n)$ *time to build and supports median 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 $a \leq b \leq c \leq d$, 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 (multi-)sets $V \subseteq X^d$ which supports the following operations:

1. $\mathrm{MIN_i}$—determine the element with the smallest i^{th} coordinate.

2. $\mathrm{MAX_i}$—determine the element with the largest i^{th} coordinate.

3. INSERT—add a value $x \in X^d$ to the subset.

4. $\mathrm{DELETEMIN_i}$—remove the element with the smallest i^{th} coordinate.

5. $\mathrm{DELETEMAX_i}$—remove the element with the largest i^{th} 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 $d \geq 2$.

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 set of points $W \subseteq X^d$. Assume first that $| W | \geq 2d$.

Definition 5.1 *The d-*interval *defined by a set of points $W \subseteq X^d$ is the* box *enclosed by (that is, 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 1^{st} coordinate, H_2 is the 1-hyperplane defined by a point $h_2 \in W - \{h_1\}$ with largest 1^{st} coordinate.*

2. *For $1 < i \leq d$, H_{2i-1} is the i-hyperplane defined by a point $h_{2i-1} \in W - \{h_1, \ldots, h_{2i-2}\}$ with least i^{th} coordinate, and H_{2i} is the i-hyperplane defined by a point $h_{2i} \in W - \{h_1, \ldots, h_{2i-1}\}$ with largest i^{th} coordinate.*

(For simplicity we use the term *i-hyperplane* to denote any hyperplane of points with fixed i^{th} 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 sets $W \subseteq X^d$ 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. By abuse of 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 $1 \leq i \leq 2d$).*
2. *For the L-node v, $I(v)$ is either a set of values from X^d with $1 \leq |I(v)| < 2d$ or a d-interval $[a_1, \ldots, a_{2d}]$ ($a_i \in X^d$ for $1 \leq i \leq 2d$).*
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 ordinary (1-)interval heaps, cf. Section 2. A d-interval heap represents the (multi-)set 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 set. 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* $\mathrm{MIN_i}$ *and* $\mathrm{MAX_i}$ *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 $x \in I(w)$, that is, 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 $I(w_0) = [a_1, \ldots, a_{2d}]$, where clearly $x \notin I(w_0)$. Now replace $I(w_0)$ by the d-interval defined by the set $\{a_1, \ldots, a_{2d}, x\}$. The defining boundary points will consist of x and all of the points $a_1, \ldots, a_{2d}$ 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 set 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 set. 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 only sketch the argument for DELETEMIN$_i$ operations. (DELETEMAX$_i$ operations are 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's assume that the root is not the L-node of I and that $[a_1, \ldots, a_{2d}]$ is the d-interval at the root. Clearly MIN$_i$ is determined as one of the points a_j $(1 \leq j \leq 2d)$ with least i^{th} coordinate. (Note that it is not necessarily the point a_{2i-1}, because of the way d-intervals are determined.) Let $a_j =$ 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 one-dimensional 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 $a_1, \ldots, a_{j-1}$ will be determined as before, but a_j leaves a hole. The point with extreme j^{th} 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 ($j < k \leq 2d$). In the former case, we have a suitable replacement and can complete the d-interval with $a_{j+1}, \ldots, a_{2d}$ 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 $a_{j+1}, \ldots, a_{k-1}$. 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 is requires $\Theta(d)$ steps of $\Theta(d)$ work to complete the construction. The resulting d-interval indeed encloses both $I(u)$ and $I(v)$, that is, 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 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 set of n points in X^d. A linear time construction can be given in the same way as in Theorem 2.6.

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

Proof: (We only give a brief outline of the construction.) Essentially, we follow the same algorithm as Floyd's 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 set 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 multi-dimensional data structures. In particular, consider the d-dimensional variant of the complementary range query problem: Given a set V of n points in d-dimensional space and a d-interval B (an iso-oriented 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 $\Theta(d^3 n)$ time to build and supports queries in $\Theta(d^2 K)$ 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 pre-order traversal of I, only visiting 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 $\Theta(d^2K)$ time. □

Note that the implicit data structure in the proof of Theorem 4.6 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 4.6 holds also for complementary partial match querying.

References

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Publishing, Co., Reading, Massachusetts, 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*, 29:996–1000, 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*, 7:448–461, 1973.

[5] S. Carlsson. The deap: A double-ended heap to implement double-ended priority queues. *Information Processing Letters*, 26:33–36, 1987.

[6] R.W. Floyd. Algorithm 245: Treesort 3. *Communications of the ACM*, 7:701, 1964.

[7] G.H. Gonnet and J.I. Munro. Heaps on heaps. *SIAM Journal on Computing*, 15:964–971, 1986.

[8] D.E. Knuth. *The Art of Computer Programming, Vol.3: Sorting and Searching.* Addison-Wesley Publishing, Co., Reading, Massachusetts, 1973.

[9] T.W.H. Lai. *Space Bounds for Selection.* Master's thesis, Department of Computer Science, University of Waterloo, 1987.

[10] T.W.H. Lai and D. Wood. *Implicit Selection.* Technical Report ?, Department of Computer Science, University of Waterloo, 1988.