WIRTSCHAFTS
UNIVERSITÄT
WIEN VIENNA
UNIVERSITY OF
ECONOMICS
AND BUSINESS

EFMD
EQUIS
ACCREDITED

ePub^{WU} Institutional Repository

Reinhold Hatzinger and Wolfgang Panny

Single and Twin-Heaps as Natural Data Structures for Percentile Point Simulation Algorithms

Working Paper

Single and Twin-Heaps as Natural Data Structures for Percentile Point Simulation Algorithms

Reinhold Hatzinger, Wolfgang Panny

Institut für Statistik
Wirtschaftsuniversität Wien

Forschungsberichte

Bericht 32
May 1993

http://statmath.wu-wien.ac.at/

Statistics and Computing **3** (1993) 163–170

32

Single and twin-heaps as natural data structures for percentile point simulation algorithms

R. HATZINGER

Department of Statistics, Vienna University of Economics, Augasse 2–6, A-1090 Vienna, Austria

W. PANNY

Department of Computer Science, Vienna University of Economics, Augasse 2–6, A-1090 Vienna, Austria

Submitted November 1992 and accepted May 1993

Sometimes percentile points cannot be determined analytically. In such cases one has to resort to Monte Carlo techniques. In order to provide reliable and accurate results it is usually necessary to generate rather large samples. Thus the proper organization of the relevant data is of crucial importance. In this paper we investigate the appropriateness of heap-based data structures for the percentile point estimation problem. Theoretical considerations and empirical results give evidence of the good performance of these structures regarding their time and space complexity.

Keywords: Percentile point estimation, order statistics, Monte Carlo simulation, heap data structures, twin-heaps

1. Introduction

To determine percentile points for distributions which are intractable or analytically cumbersome one often has to resort to simulation techniques. Virtually all simulation approaches presented in the literature are based on the following — also intuitively appealing — result due to Pfanzagl (1974): if nothing is known about the distribution of interest the appropriate sample order statistic provides an estimator having the smallest asymptotic variance among all translation-invariant estimators. To provide reliable and accurate results it is usually necessary to generate rather large samples, where the proper organization of the relevant data becomes crucial regarding computer time and storage requirements. A closely related, asymptotically equivalent approach which is less critical regarding its storage requirements has been suggested by Tierney (1983). However, if one needs a minimal squared error estimator there seems to be no alternative to determine precisely the corresponding sample order statistic. Even without the estimation context the latter problem deserves some attention, especially for really large samples.

The *direct* approach to store and sort all N simulated values fails for such large samples: N sample points have to be stored and sorting time is at least proportional to $N \log N$ on the average. Since it is only necessary to extract the kth sample order statistic, sorting in fact is more than is actually needed and an appropriate selection algorithm (e.g. Hoare, 1961) would do as well, thus reducing the expected time to $O(N)$. On the other hand, if memory constraints become critical one may consider storing and updating only the upper (lower) part of the sample, depending on $p \geq 0.5$ ($p < 0.5$), where x_p denotes the percentile of interest. Proceeding this way it suffices to allocate memory for Np^* sample points ($p^* = \min\{1-p, p\}$), which is particularly advantageous if p is not too far away from one of the tail regions of the distribution, e.g. $p = 0.95$. However, the proper organization of the relevant 'one-sided window' then becomes decisive, since at any stage it is necessary to know the Np^* largest (smallest) values of the sample generated so far, leading to a mean time complexity proportional to $Np^* \log(Np^*)$.

The direct approach and its refinements certainly have their merits for moderate N. But their memory requirements linearly grow with N, which eventually becomes

0960-3174 © 1993 Chapman & Hall

prohibitive whenever really large samples must be generated in order to assure a certain accuracy of the resulting percentile point estimator. To cope better with these inconveniences a *stochastic* approach building on an earlier proposal due to Krutchkoff (1986) has recently been presented by Dunn (1991). This approach relies on the fact that the number of sample points smaller or equal to the requested percentile x_p is by definition binomially distributed with parameters (N, p), which, of course, may be approximated by the normal distribution for sufficiently large N. The basic idea is to determine and administer a sufficiently large 'two-sided window' from the sample, which is very likely to contain the requested percentile point at the end of the simulation run. Appropriately using the latter approach, memory and time requirements can be reduced to an order of $O(\sqrt{N})$ and $O(\sqrt{N}\log N)$. However, even with Dunn's rather sophisticated method the technical issue of properly organizing the simulated data remains a critical issue. These technical considerations become even more important taking into account that, in general, a single large-sample percentile-point estimator is preferable to combining several smaller-sample estimators, as shown by Juritz *et al.* (1983) and Zelterman (1987).

In his paper Dunn suggests that a value 'be sorted into the stored list (using a divide and conquer algorithm)'. If the window really has to be kept in sorted order, a tree-like data structure, e.g. sorted binary trees, would provide a good implementation with high performance but at the cost of increased memory requirements due to the necessary pointer fields. In this paper we investigate the appropriateness of heap-based data structures for the problem at hand. These structures do not keep the pertaining windows completely sorted but provide for efficiently updating their endpoints, in fact the only functionality needed. It will be seen that *single heaps* allow for improving the efficiency of the one-sided window variant within the direct approach where no additional memory is required. Above all, they provide a basis for the development of the *twin-heap* data structure (cf. Knuth, 1973, 5.2.3 ex.31), which naturally supports the managing of the relevant two-sided window of the sample when applying the stochastic approach.

2. Two typical algorithms

In this section two prominent simulation approaches to the percentile point estimation problem will be presented, namely the direct approach using a one-sided window and Dunn's stochastic procedure, which requires a two-sided window. The basic algorithmic schemes will be explained by C-like code. However, the main purpose of this algorithmic presentation is to identify the functions necessary to administer the corresponding windows but not to annoy the reader with intricate details.

2.1. *Direct approach with one-sided window*

As outlined in the introduction, memory requirements for the direct approach can be considerably reduced by storing only $n_l = p^* n_{\max}$ sample points, where $n_{\max} = N$ denotes the total sample size and $p^* = \min\{1 - p, p\}$. In the following algorithm we assume that $p \geq 0.5$.

```
$n_l = \lfloor n_{\max}(1 - p) + 1.5 \rfloor$;
```

n_l is the size of the one-sided window that will contain the n_l largest values of the sample at the end of the simulation run. After properly initializing the window (initialize_l) every generated value is inserted into the window by applying the insert_l-function during stage 1.

```
initialize_l( );
for ($n = 1$; $n \leq n_l$; $n = n + 1$)
    insert_l (gen_value( ));
```

Throughout stage 2, the window always contains the n_l largest values generated so far. Consequently, a new value x only has to be stored, if it exceeds the window's actual minimum, returned by the function min_value_l. In this case the window's minimum must be updated as well. The function replace_min_l(x) performs the necessary storing and updating operation.

```
for ($n = n_l + 1$; $n \leq n_{\max}$; $n = n + 1$) {
    $x$ = gen_value ( );
    if ($x >$ min_value_l( ))
        replace_min_l($x$);
}
```

Obviously, at the end of the simulation run the minimum of the window is the $pn_{\max}$ smallest value of the whole sample and thus provides the requested estimate for the percentile point of interest.

```
$\hat{x}_p$ = min_value_l( );
```

This scheme is easily adapted for the case $p < 0.5$. Incidentally, also Krutchkoff's one-sided stochastic approach could conveniently be implemented along these lines.

2.2. *Dunn's procedure with two-sided window*

The direct approach is rather appealing since it is straightforward and relatively simple. However, its storage requirements grow linearly with the sample size $n_{\max}$, which eventually becomes the bottleneck if really large samples must be generated regarding the accuracy of the resulting percentile point estimator. In contrast to this, Dunn's algorithm has a built-in 'pinching' mechanism that considerably reduces the number of stored values, i.e. memory requirements now are proportional to $\sqrt{n_{\max}}$ only.

Dunn's procedure essentially consists of three stages. Stage 1 comprises iterations 1 to n_l, stage 2 iterations

$n_1 + 1$ to n_2 and stage 3 the remaining iterations. The first two stages algorithmically coincide with the direct approach 2.1: during stage 1 all values are stored, throughout stage 2 the window size n_1 remains fixed and every new point not belonging to the largest n_1 values generated so far is dropped, where we again assume that $p \geq 0.5$. The proportion of n_1 to n_2 (line 2) assures that at the end of stage 2 the window contains the order statistics with ranks $n_2p - (n_1 - 1)/2$, $n_2p - (n_1 - 1)/2 + 1, \ldots, n_2p - 1, n_2p, n_2p + 1, \ldots, n_2p + (n_1 - 1)/2$, where for convenience we assume that n_2p and $(n_1 - 1)/2$ are integers. The determination of n_2 will be postponed until the explanation of stage 3.

```
 1  $n_2 = \lfloor(4 + c_2^2 p + \sqrt{8c_2^2 p + c_2^4 p^2})/(2(1-p))\rfloor$;
 2  $n_1 = \lfloor 2n_2(1-p) - 1\rfloor$;
 3  initialize_2( );
 4  for ($n = 1; n \leq n_1; n = n + 1$)
 5      insert_2(gen_value( ));
 6  for ($n = n_1 + 1; n \leq n_2; n = n + 1$) {
 7      $x$ = gen_value( );
 8      if ($x >$ min_value_2( ))
 9          replace_min_2($x$);
10  }
```

During stage 3 a two-sided window of order statistics $x_l, x_{l+1}, \ldots, x_{u-1}, x_u$ has to be maintained. This window is chosen to give a high probability that the interesting order statistic with rank np is contained at the nth iteration, $n_2 \leq n \leq n_{\max}$. The determination of the window relies on the fact that the number of sample points less than or equal to the percentile point x_p follows a binomial distribution with parameters (n,p). Hence the ranks of the bounding order statistics at iteration n can be defined essentially by $l_2 = np - c_2\sqrt{np(1-p)}$ and $u_2 = np + c_2\sqrt{np(1-p)}$ using the central limit theorem (cf. lines 18, 19), which also determines n_2 (line 1). Dunn suggests using rather large values for c_2 (4 or even 4.5), such that the actual window is very likely to contain the order statistics x_{l_1} and x_{u_1} also, which define a confidence interval for the true percentile point x_p, where $l_1 = \lfloor np - c_1\sqrt{np(1-p)} + 0.5\rfloor$ and $u_1 = \lfloor np + c_1\sqrt{np(1-p)} + 1.5\rfloor$.

At the beginning of stage 3 the actual window is bounded by the $(n_2 - n_1 + 1)$th and n_2th order statistics, of course (line 11). If the actual sample value x does not belong to the window (because $x < x_l$ or $x > x_u$), it is dropped (lines 15, 37) and both l and u are updated accordingly. If x belongs to the window the bounding indices l_2, u_2 of the 'theoretical' window are updated (lines 18,19) and x has to be stored in the actual window (lines 21, 24 or 26). The formation of the actual window is now governed by l_2 and u_2: if the actual window protrudes to the left ($l < l_2$, line 20) or to the right ($u \geq u_2$, line 23) the actual window must also be adjusted by dropping the pertaining boundary point x_l or x_u. The necessary storing and updating operation is performed by the functions replace_min_2(x) and replace_max_2(x), respectively. Otherwise, x is simply inserted into the window by the function insert_2(x).

It can be shown by an inductive argument that the length $u_2 - l_2 + 1$ of the theoretical window is always greater or equal to the length $u - l + 1$ of the actual window. This is important for the proper working of the algorithm: it implies that the actual window can only protrude to one side, on the other hand it guarantees that we can never run out of memory, since enough storage is provided for the theoretical window at iteration $n_{\max}$, viz. $\lceil 2c_2\sqrt{np(1-p)} + 3\rceil$.

```
11  $l = n_2 - n_1 + 1$; $u = n_2$; term_ind = 0;
12  for ($n = n_2 + 1; n \leq n_{\max}; n = n + 1$) {
13      $x$ = gen_value( );
14      if ($x <$ min_value_2( ))
15          {$l = l + 1$; $u = u + 1$;}   /* drop left */
16      else
17          if ($x \leq$ max_value_2( )) {
18              $l_2 = np - c_2\sqrt{np(1-p)} - 1$;
19              $u_2 = np + c_2\sqrt{np(1-p)} + 2$;
20              if ($l < l_2$)
21                {replace_min_2($x$); $l = l + 1$; $u = u + 1$; }
22              else
23                if ($u + 1 \geq u_2$)
24                    replace_max_2($x$);
25                else
26                    {insert_2($x$); $u = u + 1$; }
27              if (($np < l$)||($np > u$))
28                {term_ind = 2; break; }
29              if (max_value_2( )-min_value_2( )< $e$) {
30                $l_1 = \lfloor np - c_1\sqrt{np(1-p)} + 0.5\rfloor$;
31                $u_1 = \lfloor np + c_1\sqrt{np(1-p)} + 1.5\rfloor$;
32                if ($l_1 \geq l$ && $u_1 \leq u$)
33                    {term_ind = 1; break; }
34              }
35          }
36      else
37          ; /*                    drop right */
38  }
39  if (($\lfloor np + 0.5\rfloor < l$)||($\lfloor np + 0.5\rfloor > u$))
40      term_ind = 2;
```

Whenever a new value has been stored into the window some checks must be done. It may happen that the npth order statistic has shifted out of the actual window. In this case an abnormal termination occurs (lines 27–28, 39–40, 43). On the other hand the desired precision e might have been attained. In that case we terminate prematurely provided that x_{l_1} and x_{u_1} are also contained in the window (lines 29–34, 42). Otherwise the algorithm terminates after $n_{\max}$ iterations (line 41).

```
switch (term_ind) { case 0: /* n_max iterations        */
        break;
                    case 1: /* precision e achieved */
        break;
                    case 2: /* x_np out of window   */
        break;
}
```

An obvious disadvantage of this stochastical procedure is that it may fail because the *np*th order statistic has shifted out of the window (case 2). However, if c_2 is large enough this is rather unlikely to occur, which is also supported by Dunn's and our empirical results. In any case, if the procedure terminates normally (cases 0 or 1) the window is guaranteed to contain the order statistic of interest. Moreover, in case 1 the window also contains a $(2\Phi(c_1) - 1)$%-confidence interval, which allows us to assess the accuracy of the resulting percentile-point estimator. This could also be enforced in case 0 at the expense of a slightly increased proportion of unsuccessful runs.

3. Heap-based data structures for implementing the windows

In the preceding section we have presented two typical algorithms for the percentile point estimation problem and tried to identify the functionality needed to administer the pertaining windows. For the one-sided window we have the functions initialize_1( ), insert_1(x), min_value_1( ) and replace_min_1(x). For the two-sided window the functions initialize_2( ), insert_2(x), min_value_2( ), max_value_2( ), replace_min_2(x) and replace_max_2(x) must be implemented. At first sight one would probably think to use sorted arrays for this purpose.

Another more sophisticated approach would consist of considering sorted binary trees for implementation. In this section the suitability of heap-based structures will be discussed, which seem to constitute natural data structures for the problems at hand. The adequacy of these structures is due to the fact that they do not provide any additional functionality not really needed, as already outlined in the introduction.

3.1. *One-sided window*

In computer science heaps are well-known as efficient data structures for implementing priority queues. A *priority queue* may be seen as a generalization of a *simple queue*, where a simple queue has a 'first in, first out' behavior, in the sense that every deletion removes the oldest remaining item. A related structure is a *stack*, where a 'last in, first out' discipline has to be regarded in the sense that every deletion removes the youngest item. *Priority queues* show a 'smallest in, first out' behavior, where every deletion removes the item having the smallest key. The name priority queue is due to the fact that the key of an element represents its priority to leave the queue. Hence, a 'largest in, first out' discipline is possible as well. In this latter case we speak of a maximum priority queue, as opposed to the former minimum priority queue. It is not difficult to see that the one-sided window within the direct approach is just an instance of a priority queue. To be more specific, we have to maintain a minimum or maximum priority queue depending on $p \geq 0.5$ or $p < 0.5$, respectively, which can most conveniently be implemented by the heap data structure (cf. Williams, 1964; Knuth, 1973; Sedgewick, 1983; Gonnet, 1984). In this paper we also use the term *single heap* to distinguish better the heap data structure from the more elaborate *twin-heap* data structure brought into play for the implementation of the two-sided window.

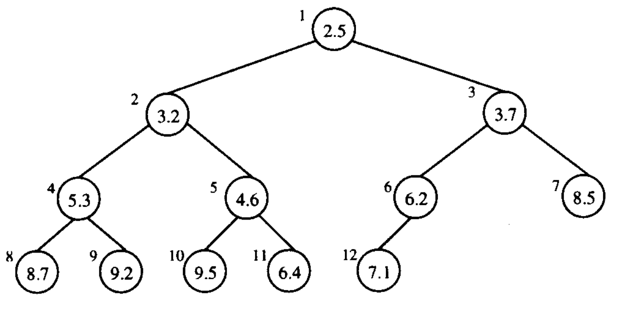

Fig. 1. *Example of a min-heap-tree*

A heap is a perfect binary tree represented implicitly in an array. The nodes of a min-heap are subject to the *min-heap condition*: the key in each node should be smaller or equal to the keys in its descendants. This of course entails that the smallest key is in the root. Since a heap is a binary tree it can naturally be represented in an array: the root is located in position 1, the left son in position 2, the right son in position 3, ... In general, the left son of the node located in position j is stored in position $2j$, the right son is in position $2j + 1$. This linearization is dense, since a heap is perfect. This is a great advantage over general binary trees because no extra memory for link fields is required. Figure 1 shows a min-heap-tree. The corresponding array representation is:

1	2	3	4	5	6	7	8	9	10	11	12
2.5	3.2	3.7	5.3	4.6	6.2	8.5	8.7	9.2	9.5	6.4	7.1

To *insert* a new key x into the heap by insert_1(x) we first put x to the next free location. If the heap condition is violated (because the new key is smaller than its father) the new key is exchanged with its father. This may, in turn, cause a violation, which can be mended in the same way, etc. This process eventually stops at a father node whose key is

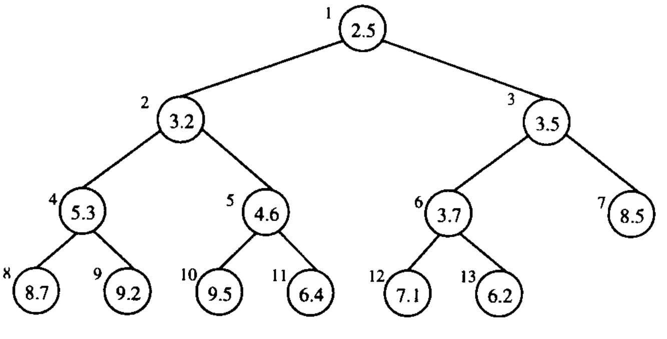

Fig. 2. *min-heap-tree after insertion*

smaller or equal to x, or after x has migrated to the root. Figure 2 shows the state of the min-heap-tree after insertion of the new key $x = 3.5$.

To *insert* a new key x into the heap and *deleting* the current minimum at the same time by replace_min_1(x) we may proceed as follows: we first put x to the root location. If the heap condition is violated (because the new root is larger than a son) this can be fixed by exchanging the new root with its smallest son. Again, this process possibly has to be repeated until eventually two sons with key-values $\geq x$ are encountered or until the new key x has migrated to a leaf location. Figure 3 shows the state of the min-heap-tree after performing replace_min_1(x), where $x = 7.3$.

The operation to delete the smallest element involves almost the same process. It can be done essentially by moving the element from the last location into the root. Then the last location is freed by decrementing the location counter. The heap-condition can be re-established in the same way as for the replace_min_1-function.

The three update-functions outlined above have an $O(\log n)$ behavior, even in the worst case. The minimum of the heap can be inspected by simply accessing the root element, i.e. location 1 of the pertaining array. Hence, the function min_value_1 has an $O(1)$ behavior. The same is true for the function initialize_1, which only initializes the current location counter to zero. Moreover, an appropriate sentinel key ('$-\infty$') can be stored in location 0, which slightly simplifies the insert_1-function.

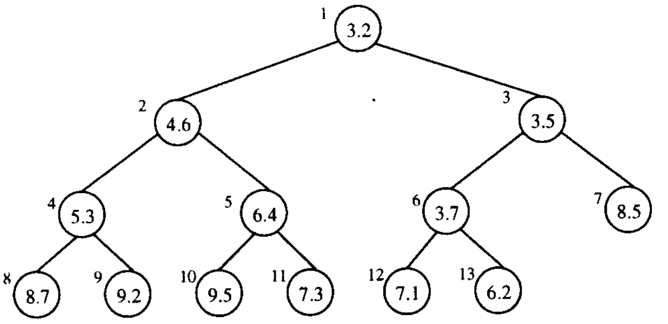

Fig. 3. *min-heap-tree after replace-min*

A max-heap can be implemented analogously. One only has to replace the $\leq$ relation by $\geq$.

3.2. *Two-sided window*

In the previous section we mentioned priority queues as generalizations of simple queues and stacks. In particular, we have shown that the one-sided window within the direct approach may be seen as an instance of a priority queue. Another fundamental structure is the so-called *deque*, which incorporates the 'first in, first out' discipline of a simple queue and the 'last in, first out' behavior of a stack simultaneously. This means that one has the option of either removing the oldest or the youngest element by a deletion. The deque concept can be generalized to a *priority deque*, where the simple time context of the deque is replaced by a key concept representing the priority of each element. Accordingly, one now has the option to delete either the element with the highest or the lowest key value. Hence, the two-sided window within the stochastic approach is just an instance of a priority deque. In Knuth (1973, 5.2.3 ex.31) the basic ideas of an efficient, heap-based implementation are sketched and the name *twin-heap* for the corresponding data structure has been introduced.

We define a twin-heap to consist of both a min-heap and a max-heap tied together by the following two conditions:

(a) Let us assume that the twin-heap contains n elements at a given time. Then $\lceil n/2 \rceil$ elements are stored in the min-heap and the remaining $\lfloor n/2 \rfloor$ elements are stored in the max-heap.

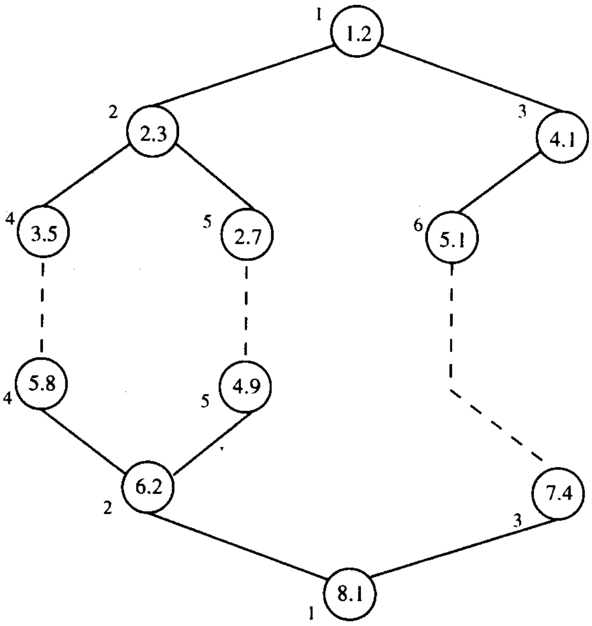

Fig. 4. *Example of a twin-heap*

(b) Let a_j and b_j denote corresponding elements in the min- and max-heap, respectively. Then the relation $a_j \leq b_j$ must hold for $1 \leq j \leq \lfloor n/2 \rfloor$. If n is odd, $n = 2m + 1$, say, then b_{m+1} does not exist. In this case node $b_{\lfloor (m+1)/2 \rfloor}$ is defined to be the counterpart of a_{m+1}, i.e. the relation $a_{m+1} \leq b_{\lfloor (m+1)/2 \rfloor}$ must also be true. Note that $b_{\lfloor (m+1)/2 \rfloor}$ is the 'father of the missing node b_{m+1}'.

Figure 4 gives an example of a twin heap containing $n = 11$ elements. The corresponding array representation is:

	1	2	3	4	5	6
a:	1.2	2.3	4.1	3.5	2.7	5.1

	1	2	3	4	5
b:	8.1	6.2	7.4	5.8	4.9

If a new key x is *inserted* into the twin-heap by insert_2(x), two cases must be considered according to the parity of the actual value of n.

$n = 2m + 1$: According to condition (a), the next free location is b_{m+1}. This location determines a unique path from the root of the min-heap to the root of the max-heap, viz. $a_1, \ldots, a_{m+1}, b_{m+1}, b_{\lfloor (m+1)/2 \rfloor}, \ldots, b_1$. Note that $a_1 \leq \ldots \leq a_{m+1} \leq b_{\lfloor (m+1)/2 \rfloor} \leq \ldots \leq b_1$ as a consequence of condition (b). If $a_{m+1} \leq x \leq b_{\lfloor (m+1)/2 \rfloor}$, x is stored in location b_{m+1} and we are done (e.g. $x = 6.7$, referring to Fig. 4). The case $x > b_{\lfloor (m+1)/2 \rfloor}$ amounts to a regular insertion of x into the max-heap (e.g. $x = 9.3$). The case $x < a_{m+1}$ can be dealt with by removing a_{m+1} from the min-heap and storing its content to b_{m+1}. x is then regularly inserted into the min-heap. This latter case is illustrated by Fig. 5, which shows the state of the twin-heap after inserting $x = 2.0$ into the twin-heap of Fig. 4.

$n = 2m$: The next free location is now a_{m+1}, which determines the unique path $a_1, \ldots, a_{\lfloor (m+1)/2 \rfloor}, a_{m+1}, b_{\lfloor (m+1)/2 \rfloor}, \ldots, b_1$, where again $a_1 \leq \ldots \leq a_{\lfloor (m+1)/2 \rfloor} \leq b_{\lfloor (m+1)/2 \rfloor} \leq \ldots \leq b_1$. If $a_{\lfloor (m+1)/2 \rfloor} \leq x \leq b_{\lfloor (m+1)/2 \rfloor}$, x is stored in location a_{m+1} and no further action is needed (e.g. $x = 6.3$, referring to Fig. 5). The case $x < a_{\lfloor (m+1)/2 \rfloor}$ amounts to a regular insertion of x into the min-heap (e.g. $x = 1.0$). If $x > b_{\lfloor (m+1)/2 \rfloor}$, the content of $b_{\lfloor (m+1)/2 \rfloor}$ is stored in a_{m+1} and x is regularly inserted into the max-heap starting at location $b_{\lfloor (m+1)/2 \rfloor}$. This latter case is illustrated by Fig. 6, which shows the state of the twin-heap after inserting $x = 8.1$ into the twin-heap of Fig. 5.

The implementation of the functions replace_min_2(x) and replace_max_2(x) can be accomplished in a similar way. The same is true for the corresponding *delete* functions. The crucial point consists of maintaining the integrity constraints (conditions (a) and (b)) of the twin-heap while operating on the two single heaps involved. Again, all these update functions show an $O(\log n)$ behavior, even in the worst case, whereas the inspection functions min_value_2( ) and max_value_2( ) and the initialize_2( )-function are only $O(1)$.

4. Empirical results and discussion

To investigate empirically the behaviour of the heap-based

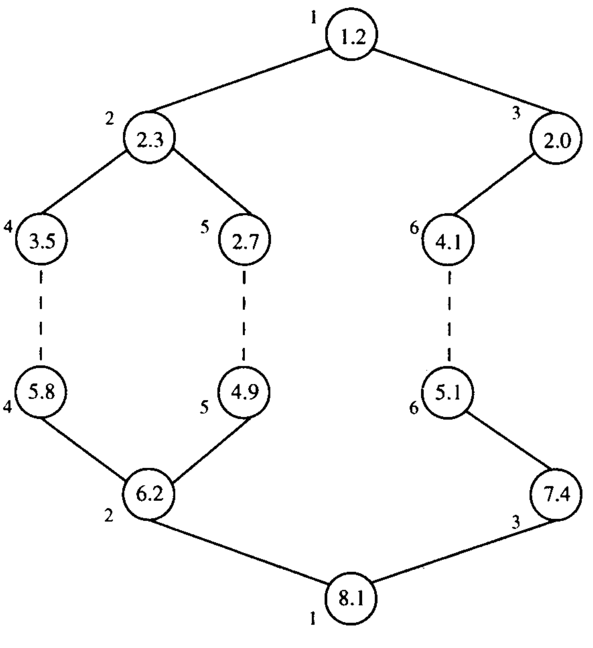

Fig. 5. *twin-heap after insertion of* $x = 2.0$

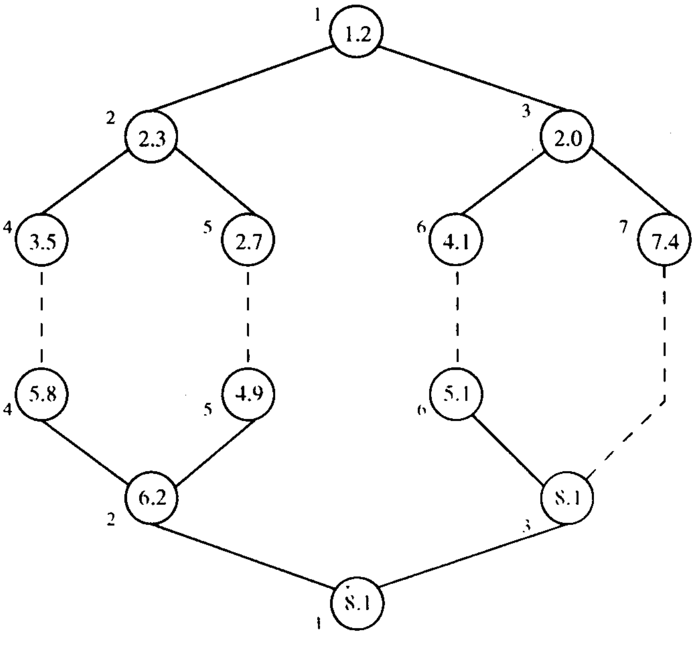

Fig. 6. *twin-heap after insertion of* $x = 8.1$

Table 1. *One-sided window, uniform distribution, p* = 0.95

	$n_{\max} = 100\,000$			$n_{\max} = 1\,000\,000$			$n_{\max} = 10\,000\,000$		
	t_{tot}	t_{net}	*mem*	t_{tot}	t_{net}	*mem*	t_{tot}	t_{net}	*mem*
OSH	1.02	0.12	39	10.59	1.59	391	118.92	28.90	3906
OST	1.21	0.31	117	14.31	5.31	1172	171.37	81.35	11721
OSA	13.34	12.44	39	1341.96	1332.96	391	136 835.03	136 745.01	3906

Table 2. *One-sided window, uniform distribution, p* = 0.5

	$n_{\max} = 100\,000$			$n_{\max} = 1\,000\,000$			$n_{\max} = 10\,000\,000$		
	t_{tot}	t_{net}	*mem*	t_{tot}	t_{net}	*mem*	t_{tot}	t_{net}	*mem*
OSH	1.28	0.37	391	15.88	6.88	3906	197.05	107.03	39 063
OST	2.59	1.68	1172	36.16	27.16	11721	–	–	117 188
OSA	448.96	448.05	391	45 520.62	45 511.62	3906	–	–	39 063

Table 3. *Two-sided window, uniform distribution, p* = 0.95

	$n_{\max} = 100\,000$			$n_{\max} = 1\,000\,000$			$n_{\max} = 10\,000\,000$		
	t_{tot}	t_{net}	*mem*	t_{tot}	t_{net}	*mem*	t_{tot}	t_{net}	*mem*
TSH	0.94	0.04	4	9.33	0.33	14	93.00	2.98	43
TST	0.95	0.05	13	9.38	0.38	41	93.12	3.10	129
TSA	0.98	0.08	4	9.74	0.74	14	96.99	6.97	43

Table 4. *Two-sided window, uniform distribution, p* = 0.5

	$n_{\max} = 100\,000$			$n_{\max} = 1\,000\,000$			$n_{\max} = 10\,000\,000$		
	t_{tot}	t_{net}	*mem*	t_{tot}	t_{net}	*mem*	t_{tot}	t_{net}	*mem*
TSH	0.96	0.05	10	9.40	0.40	31	93.56	3.54	99
TST	0.98	0.07	30	9.48	0.48	94	93.85	3.83	297
TSA	1.16	0.25	10	11.49	2.49	31	114.54	24.52	99

data structures they have been compared to sorted binary trees and simple sorted arrays. Accordingly we have written three programs for both the direct approach and for Dunn's procedure. The one-sided window (i.e. direct approach) was organized as a single heap (OSH), as a sorted binary tree (OST) and as a sorted array (OSA). The corresponding programs for Dunn's procedure are TSH, TST and TSA, where the two-sided window was organized as a twin-heap, as a sorted binary tree and as a sorted array. The implementation of the sorted array methods is straightforward and we think the reader can easily figure out what must be done. In principle the same is true for the sorted binary tree implementation which, however, may be less obvious because the underlying data structure is more sophisticated. A good description of this data structure and the related access functions can be found in Sedgewick (1983, pp. 178–184).

The programs have been coded in C and numerous runs have been made on a HP 9000/720 workstation, where sample size $n_{\max}$, distribution type (standard uniform, exponential, normal) and percentile p of interest have been varied. Tables 1–4 summarize the results of some runs, where the total CPU-time t_{tot} (in seconds) and the necessary memory *mem* (in Kbyte) for the different windows are shown. Also the net CPU-time t_{net} (in seconds) is recorded. t_{net} is obtained from t_{tot} by subtracting the time necessary to generate all $n_{\max}$ sample points.

Tables 1 and 2 show the superiority of the heap organization for the one-sided window. As expected, computation time for the sorted array (OSA) grows quadratically with the sample size $n_{\max}$. The more sophisticated binary tree organization (OST) can better compete with the heap (OSH) regarding computing time but needs three times more memory, due to the necessary link fields. Memory requirements and net time do not change for other distributions, whereas the total time, of course, reflects the complexity of the value generating algorithm. Thus we have a clear victory for the heap organization, when the direct approach with a one-sided window is applied. However, the empirical results also exhibit the limitations of the direct approach. Its memory requirements of $O(n_{\max})$ become prohibitive for really large sample sizes, especially if p is far away from one of the tail regions of the distribution and that is one reason why Table 2 is incomplete. OSA obviously fails due to time considerations.

Tables 3 and 4 give the corresponding figures for Dunn's procedure using a two-sided window. The three approaches to organize the two-sided window show the same ranking as before. However, the differences between them are not as remarkable as before.

Again, the sorted binary tree variant (TST) uses three times more memory than the two other approaches. But memory requirements grow at a rate of only $O(\sqrt{n_{\max}})$ now, so this is not really a bottleneck. The total computation time is clearly dominated by its linear term due to the generation of the sample values. The time advantage of the heap-based organization decreases for several reasons: The number of values contained in the window is significantly reduced using Dunn's procedure. For instance, if $n_{\max} = 10\,000\,000$ and $p = 0.5$, the maximal window size is only about 12 000. Also the number of sample points which require an update of the actual window is rather small. For the above example about 22 000 update operations are necessary, which is a rate of only 0.0022%. Additionally, the t_{net} figures in Tables 3 and 4 also comprise a rather large overhead due to properly managing Dunn's procedure. This overhead cannot be attributed to the window updating functions but it is difficult to eliminate within our time monitoring. We know from other experiments that the true net time factor between TST and TSH should be at least 1.7.

Summing up, it may be said that the heap-based variant (TSH) is clearly superior to its competitors but this superiority has no tremendous effect within Dunn's procedure, since Dunn's approach to the percentile point estimation problem is so clever. It should perhaps also be mentioned that Dunn reports several of his runs terminating abnormally, i.e. the order statistics of interest had drifted out of the actual window. During the many runs we did (for each of the three distributions 70 runs with $n_{\max} = 10\,000\,000$), no such behavior was observed.

Finally, single heaps as well as twin heaps can easily be implemented in a conventional programming language like FORTRAN, since no link fields or recursive functions are necessary. So it seems that heap-based structures are actually natural for the percentile point estimation problem and they might prove their appropriateness also for related problems in computational statistics.

Acknowledgement

We are indebted to the referee for his valuable comments and suggestions which helped to improve the exposition.

References

Dunn, C. L. (1991) Precise simulated percentiles in a pinch. *The American Statistician*, **45**, 207–211.

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

Hoare, C. A. R. (1961) Algorithm 65 (FIND). *Communications of the ACM*, **4**, 321–322.

Juritz, J. M., Juritz, J. W. F. and Stephens, M. A. (1983) On the accuracy of simulated percentage points. *Journal of the American Statistical Association*, **83**, 441–444.

Knuth, D. E. (1973) *The Art of Computer Programming*: *Sorting and Searching*. Addison-Wesley, Reading, MA.

Krutchkoff, R. G. (1986) Percentiles by simulation: reducing time and storage. *Journal of Statistical Computation and Simulation*, **25**, 304–305.

Pfanzagl, J. (1974) Investigating the quantile of an unknown distribution, in *Contributions to Applied Statistics* (Dedicated to Arthur Linder), Birkhäuser, Basel, pp. 111–126.

Sedgewick, R. (1983) *Algorithms*. Addison-Wesley, Reading, MA.

Tierney, L. (1983) A space-efficient recursive procedure for estimating a quantile of an unknown distribution. *SIAM Journal on Scientific and Statistical Computing*, **4**, 706–711.

Williams, J. W. J. (1964) Algorithm 232 (HEAPSORT). *Communications of the ACM*, **7**, 347–348.

Zelterman, D. (1987) Estimating percentage points by simulation. *Journal of Statistical Computation and Simulation*, **27**, 107–125.