

Checking Priority Queues

Ulrich Finkler*

Kurt Mehlhorn†

Abstract

We describe a checker for priority queues. It supports the full repertoire of priority queue operations (*insert*, *del_min*, *find_min*, *decrease_p*, and *del_item*). It requires $O(1)$ amortised time per operation and uses linear additional space (i.e. the same amount as the priority queue). The checker reports an error occurring in operation i before operation $i + cN' + 1$ is completed, where N' is the number of elements in the queue at the time the error occurred and $c \leq 1$ is a constant. We show that an on-line checker, i.e., a checker that reports errors immediately, must have running time $\Omega(n \log n)$ in the worst case for a sequence of n priority queue operations. This lower bound holds in the comparison model of computation.

1 Introduction

We use the encapsulation model for checking data structures, also called the client-checker-server model [5]. In this model a checker for an abstract data type A is a layer C of software which encapsulates any instance D of A . C must provide the interface of type A , i.e., a user of type A can switch from an unchecked instance of A to a checked instance of A without change to his algorithm.

If D performs correctly on a sequence of operations Ψ then there is no change in functionality. If D performs incorrectly on Ψ in operation i then C will report an error within a delay as stated in theorem 1.2. A checker is called *time-efficient*, *time-inefficient*, and *time-superefficient*, respectively, if its running time is asymptotically no more, is more, is less, respectively, than the running time of D . The same notions apply to space. We will show two theorems in this paper.

THEOREM 1.1. *In the comparison model of computation there is no on-line, time-superefficient checker for priority queues.*

THEOREM 1.2. *There is a time-superefficient and space-efficient checker for priority queues. It requires*

time $O(n)$ to check n operations on a priority queue. The checker will report an error in operation i before the completion of operation $i + cN' + 1$. N' is the number of elements in the queue at the time the error occurred and $c \leq 1$ is a constant.

The checker uses among other data structures the linear time union-find algorithm of Gabow and Tarjan [3]. We implemented a variant of the checker described in this paper in the LEDA-framework [9]. We used a checked binary heap to sort and in Dijkstra's algorithm: in both cases running time increased by about a factor of two.

The literature on program checking is rich; the papers [1][4][5][6][7] are particularly relevant for this paper. The known checkers for priority queues are either off-line and hence space-inefficient or not time-superefficient or the priority queue is restricted either in functionality or in implementation (it has to be an ordered sequence) and therefore efficiency.

2 Time-superefficient Online Checking

Theorem 1.1 is an almost immediate consequence of work of Brodal et al. [8]. They considered sequences of *insert*, *delete* and *findany* operations and showed that a sequence of n such operations requires $\Omega(n \log n)$ comparisons in the worst case. A *findany* operation may return any element of the current set; together with the element it *must* return the rank of the element in the current set. An on-line checker of a priority queue must have a proof at hand that the result is the smallest element in the set, whenever it has to verify the result of a *del_min* operation. This observation allows to adapt the proof of [8] to a proof of theorem 1.1.

3 The Checker

LEMMA 3.1. *Let Q be a priority queue with operations *insert*, *del_min* and *del_item*. Let p_i be the priority returned by operation i (*insert* and *del_item* return $-\infty$). Let R_i be the set of items remaining in Q after operation i . Let I be a sequence of operations beginning with Q empty. If $x \geq p_i \forall i \in I \wedge \forall x \in R_i$ then the output of Q for I was correct.*

The checker maintains one item for each item in the priority queue Q in a linear list R which is ordered

*IBM T.J. Watson Research Center, Yorktown Heights, New York. Supported by the Graduiertenkolleg 'Effizienz und Komplexität von Algorithmen und Rechenanlagen', Universität Saarbrücken, Germany and Max-Planck-Institut für Informatik, Saarbrücken, Germany.

†Max-Planck-Institut für Informatik, Saarbrücken, Germany. Partially supported by ESPRIT-project ALCOM-IT.

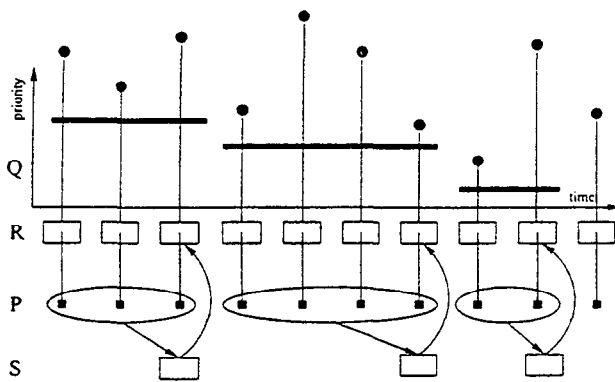


Figure 1: A snapshot of the data structure. The upper part of the figure is a plot of the priority versus time. Horizontal bars show the priorities of the blocks in the partition P .

according to the time of insertion. Each item in R is doubly linked to the corresponding item in Q . A partition P splits R into blocks P_p indexed by the common strongest lower bound p for all items in a block. All items in a block are consecutive in R . Every item $x \in P_p$ satisfies $p(x) \geq p$, where $p(x)$ is the priority of x . The blocks are maintained in a list S ordered by their priority (figure 1).

The checker performs the following actions in parallel to the operations in the queue:

insert(x): A new block $P_{-\infty} = \{x\}$ is appended to the right end of S and linked to the item in Q .

del-min(): Q returns x , $P_{p_\alpha} \ni x$ is found and $p(x) \geq p_\alpha$ is checked. x is deleted from R and P_{p_α} . All blocks $P_\beta : \beta \leq p(x)$ are unioned into $P_{p(x)}$.

del-item(x): W.l.o.g. x is assumed to be not the minimum. $P_{p_\alpha} \ni x$ is found, $p(x) \geq p_\alpha$ is checked. x is deleted from R and P_{p_α} .

decrease-p(x, p): Synthesized by a *del-item(x)* and an *insert(p)*.

find-min(): Synthesized by a *del-min()* and an *insert*.

check(): $p(x) \geq p_\alpha \forall x \in P_{p_\alpha}$ is verified for all $P_{p_\alpha} \in P$.

periodic-check(): Starting with a check after operation 1, a *check()* is performed in $j + cN_j + 1$ if the last check occurred in operation j . N_j is the size of the queue after operation j .

Let the last check before the error be the check after step j . If an error occurs in operation $j + i$, $i \geq 1$, there were $N_{j+i} \geq N_j - i + 1$ elements in the queue at the time the error occurred and the next check occurs in operation $j + cN_j + 1 \leq j + i + cN_{j+i} + 1$ for $c \leq 1$. Thus, the periodic checks and one check as last operation on the queue ensure that an error is reported with the delay bound as in theorem 1.2. Note that $N_j = 0$ is possible.

4 Complexity

Consider a sequence I of N checker operations without a *check()*. Since every *union* of two blocks reduces the number of blocks by one, at most N *unions* are performed during I . Additionally, $O(N)$ *union-find* and *makeBlock* operations are performed. Hence, I has the same complexity as a sequence of $O(N)$ *union*, *makeBlock* and *union-find* operations, which takes time $O(N\alpha(N))$ with *union by rank with path compression* [2]. In fact, the special case of *incremental tree set union* [3] is applicable such that I requires $O(N)$ time. Thus the amortized complexity of each checker operation corresponding to a priority queue operation is $O(1)$.

A *check()* after N queue operations takes $N' \leq N$ operations, N' is the number of elements in the queue at the time of the check. The periodic check partitions I into pieces for each of which we account separately. At the beginning of each piece, i.e. after the last check, the queue contains N' items, before the final check it contains at most $N' + cN' + 1$ items, such that the check requires $O(N')$ time, which is distributed over N' operations. Thus, the time spend in *periodic-check()* during a sequence I of N operations is $O(N)$, which concludes the proof of theorem 1.2.

Note, that our *union-find* data structure needs to support deletions of items. This is easily incorporated into both versions by deleting lazily and rebuilding the structure as soon as more than half of the elements are marked as deleted.

5 Acknowledgements

We would like to thank Bernhard Moret and the SODA '99 committee for suggesting to bound the delay in terms of the number of elements in the priority queue at the time of an error.

References

- [1] Aho, Hopcroft, Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
- [2] R.E. Tarjan. *Journal of the ACM*, 22(2):215–225, 1975.
- [3] H.N. Gabow and R.E. Tarjan. *Journal of Computer and System Sciences*, 30(2):209–221, 1985.
- [4] G.F. Sullivan, G.M. Masson. FTCS-21: 21st Symposium on Fault-Tolerant Computing, 240–247, 1991.
- [5] N.M. Amato, M.C. Loui. FTCS-21, 164–173, 1994.
- [6] J.D. Bright, G.F. Sullivan. FTCS-24, 144–153, 1994.
- [7] J.D. Bright, G.F. Sullivan. FTCS-25, 392–401, 1995.
- [8] G. Brodal, S. Chaudhuri, J. Radhakrishnan. *Nordic Journal of Computing*, 3(4):337–351, 1996.
- [9] K. Mehlhorn, S. Näher. *The LEDA Platform for Combinatorial and Geometric Computing*. Cambridge University Press.