University of Ljubljana
Institute of Mathematics, Physics and Mechanics
Department of Mathematics
Jadranska 19, 1000 Ljubljana, Slovenia

Preprint series, Vol. 39 (2001), 771

MULTIPROCESS TIME QUEUE

Andrej Brodnik Johan Karlsson

ISSN 1318-4865

August 16, 2001

Ljubljana, August 16, 2001

Multiprocess Time Queue

Andrej Brodnik[1,2] and Johan Karlsson[1]

[1] Department of Computer Science and Electrical Engineering,
Luleå University of Technology, Luleå, Sweden
{andrej.brodnik,johan.karlsson}@sm.luth.se
[2] Department of Theoretical Computer Science,
Institute of Mathematics, Physics, and Mechanics, Ljubljana, Slovenia

Abstract. We show how to implement a bounded time queue for two different processes. The time queue is a variant of a priority queue with elements from a discrete universe. The bounded time queue has elements from a discrete bounded universe. One process has time constraints and may only spend constant worst case time on each operation while the other process may spend more time. The time constrained process only has to be able to perform some of the time queue operations while the other process has to be able to perform all operations. We show how to do a deamortization of the `deleteMin` cost and to provide mutual exclusion for the parts of the data structure that both processes maintain.

1 Introduction

In this paper we look at a special variant of the *Priority Queue* problem which we call the *Time Queue* problem. A time queue is a queue that stores elements together with a time stamp. Newly inserted elements must have a time stamp that lies in the future. The time queue can be used in various ways. One task might be as a time-out manager, where an element has to be processed before some given time otherwise it should be considered to have timed-out and be handled specially. The time queue can also be used for the *simulation event set* problem [3] and other *scheduling* problems.

The time queue supports, given a set $\mathcal{N}$ of N elements, the ordinary operations of a priority queue, `insert`, `min` and `deleteMin`. By convention the highest priority has the lowest numerical value, hence `min`. We refer to the element with the minimum numerical priority value as the *min element* and use t_0 to denote its priority.

Usually a priority queue supports the `decrease-key` operation, which decreases the priority of an element in the queue. The `increase-key` operation is also supported by the time queue and we combine these operations into a general `update` operation, which updates the priority of an element.

Further, a general `delete` operation is also supported in the time queue. Therefore, we let `insert` return a *finger* to the inserted element, which can be used by other operation such as `delete` and `update`. It is convenient that the `min` operation also returns a finger. Since we use fingers we need operations to

get the priority and element from the finger, `value` and `data` respectively. In this paper we use the terms element and finger to an element interchangeably.

Finally, the time queue supports deletion of all elements with a priority less than a specific value, using the operation `delLessThan`. The `delLessThan` can be augmented with an additional function, $\mathcal{F}$, that is called for each deleted element. Note that this forces the `delLessThan` to take $\Omega(d \cdot F)$ time where d is the number of deleted elements and F is the running time of the function $\mathcal{F}$. Without loss of generality we assume that $F = \Theta(1)$.

In the time queue the priorities are times. We assume that time is a discrete value and hence the time queue is restricted to only support priorities that are discrete (e.g. integers). We require that for the time t_e of the newly inserted or of the updated element e must hold $t_e > t_0$, which means that the min time t_0 is non-decreasing, the time queue is *monotonic* [12]. Moreover, we require that the time for any element in the time queue is less than $t_0 + C$, where C denotes the maximum duration of any element (cf. *maximum event duration* [4]). To sum up: time is drawn from a bounded discrete universe.

The above description gives the following formal definition:

Definition 1. *The* Time Queue problem *is the problem of maintaining a set, $\mathcal{N}$, of elements to support the following operations:*

`insert(`e`,`t`):`f	*Iff $t_0 < t \leq t_0 + C$ then let $\mathcal{N} := \mathcal{N} \cup \{e\}$ and return a finger f to the newly inserted element.*
`delete(`f`)`	*let $\mathcal{N} := \mathcal{N} \backslash \{f\}$.*
`min():`f	*Find the min element and return a finger, f, to it.*
`deleteMin()`	*Delete the min element.*
`update(`f`,`t`)`	*Iff $t_0 < t \leq t_0 + C$ then change the time of f to t.*
`delLessThan(`t`, `$\mathcal{F}$`)`	*Delete all elements with time less than t and call the function '$\mathcal{F}$' for each of the deleted elements.*

where t_0 is the priority of the min element and C is the maximum duration of any element.

This research was initiated by a manufacturer of a firewall. In their firewall IP packets are processed in two different paths called *fast* and *slow* path. The fast path must not be delayed when using the time-out manager and this process needs only some of the operations of the time queue. The slow process has to be able to perform all the operations, but it is not that time sensitive.

Hence, in our model, we have two different processes manipulating the data structure. The first process (*fast*) has to be able to perform the `min`, `value`, `data` and a restricted `update` operations. The second process (*slow*) has to be able to perform all the operations on the time queue. The *fast* process is time critical and must not be delayed, i.e., the operations it uses must run in $O(1)$ worst case time. The fast process `update` only needs to update elements in the near future, i.e., only elements with current and new time less than $t_0 + \epsilon$ (ϵ is to be defined later).

Our main goal is to implement the operations of the *fast* process to run in $O(1)$ worst case time, hence amortized or expected time is not good enough. To do this, we let only the operations `deleteMin` and the `delLessThan` change

the min element. this makes the operations `delete` and `update` more restricted, and, consequently, less complicated than `deleteMin` and the `delLessThan`. We refer to the time queue problem with these restrictions as the *restricted Time Queue problem*.

Furthermore, `delLessThan` is called by the slow, and this at least every c time units for some small value c.

To allow the two processes to share data we need mutual exclusion of the operations. For this we use locks and the interface to the locks has to provide both blocking and non-blocking locking functions. We also assume that the processes can pass messages asynchronously.

To compare different priority queues both theoretically and practically the *hold model* [10] has been used. In this model a priority queue of size N is created and a *hold* operation is performed a number of times. The `hold` operations is a sequence of `min`; `deleteMin`; and `insert` operations, hence N is not changed. The priority of the newly inserted element is $t_0 + d$ for some value d.

In the following section we look at how other solutions can be used to solve the restricted time queue problem, in particular the *Calendar Queue* by Brown [3]. In Sect. 3 we present our solution, a modification of the calendar queue, to support the operations of the fast process while Sect. 4 concludes the paper.

2 Previous Work

A number of solutions for the priority queue problem can be used to solve the time queue problem for one process with only small modifications if any. The standard *heap* described by Williams [18] can be modified to use fingers by adding a dictionary that stores the position in the heap for each element. The heap solution (*heap* in Table 1) even works if the maximum duration is unbounded and it only needs $O(N)$ space. The model used is the pointer machine model [11].

Van Emde Boas et al. proposed a data structure they call a *stratified tree* which supports the time queue operations in $O(\lg \lg C)$ time (*vEB* in Table 1) [13, 15]. However, the stratified tree needs $O(C + N)$ space. The model is the pointer machine model.

Willard shows how perfect hashing (see [6, 8]) can be used to improve the space bound to $O(N)$ for the stratified tree [17] (*vEB-W* in Table 1). The model is the RAM model [14] of the stronger cell probe model [19] due to the hashing.

More recently Andersson and Thorup improved their *exponential search* trees to achieve worst case performance of $O(\sqrt{\lg N / \lg \lg N})$ [1] (*EST* in Table 1). The model used here is the RAM model.

Brodnik et al. showed how a *split tagged tree* can be used to achieve worst case constant time for all the time queue operations (*SST* in Table 1) [2]. They use $O(C + N)$ space in the Yggdrasil implementation [2] of the RAMBO model [9].

So far we have seen the bounds in Table 1, with the Calendar queue (*CQ*) presented below.

Table 1. Time bounds for different solutions to the Time Queue problem

Operation	Heap	vEB	vEB-W
`insert`	$O(\lg N)$	$O(\lg\lg C)$	exp $O(\lg\lg C)$
`delete`	$O(\lg N)$	$O(\lg\lg C)$	exp $O(\lg\lg C)$
`min`	$O(1)$	$O(1)$	$O(1)$
`deleteMin`	$O(\lg N)$	$O(\lg\lg C)$	exp $O(\lg\lg C)$
`hold`	$O(\lg N)$	$O(\lg\lg C)$	exp $O(\lg\lg C)$
`update`	$O(\lg N)$	$O(\lg\lg C)$	exp $O(\lg\lg C)$
Space	$O(N)$	$O(C+N)$	$O(N)$

Operation	EST	STT	CQ
`insert`	$O(\sqrt{\lg N/\lg\lg N})$	$O(1)$	am $O(1)$
`delete`	$O(\sqrt{\lg N/\lg\lg N})$	$O(1)$	am $O(1)$
`min`	$O(1)$	$O(1)$	$O(1)$
`deleteMin`	$O(\sqrt{\lg N/\lg\lg N})$	$O(1)$	am exp $O(1)$
`hold`	$O(\sqrt{\lg N/\lg\lg N})$	$O(1)$	exp $O(1)$
`update`	$O(\sqrt{\lg N/\lg\lg N})$	$O(1)$	$O(1)$
Space	$O(N)$	$O(C+N)$	$O(N)$

2.1 The Calendar Queue

The *Calendar Queue* data structure described by Brown [3] and analyzed by Erickson et al. [7] is a priority queue specially designed for the event set problem. Erickson et al. give a short and good description of the calendar queue that we restate here.

"A *calendar queue* has M buckets numbered 0 to $M-1$, a current bucket with index i_0, a bucket width δ, and a current time t_0. We have the relationship that $i_0 = (t_0 \operatorname{div} \delta) \bmod M$. For each element e in the calendar queue, $t_e \geq t_0$, and element e is located in bucket i if and only if $i \leq (t_e \operatorname{div} \delta) \bmod M < (i+1)$."

The calendar queue is implemented as an array of lists, which we denote `buckets`. Depending on *bucket discipline* the lists in the buckets are either sorted or unsorted. In unsorted buckets `insert` takes constant time and `min` takes time proportional to the number of elements in the bucket. On the other hand, in sorted buckets `min` (`deleteMin`) takes constant time and `insert` time proportional to the log of the number of elements in the bucket. In Brown's and Erickson's descriptions all buckets use the same bucket discipline.

Brown [3] suggests to use a doubling technique to adjust the number of buckets M to be $\Theta(N)$ where N is the number of elements in the queue. Hence, when inserting an element and N becomes greater than M, we allocate $2M$ new buckets, copy all the elements to the new buckets and deallocate the old buckets. When deleting an element and N becomes less than $M/4$, we allocate $M/2$ buckets, copy all the elements and deallocate the old buckets. We see that, if a doubling of the number of buckets occurs when there are N_0 elements, at least N_0 new elements has to be inserted into the queue before the next doubling will occur. Hence the copying cost of the $2N_0$ elements at the second double can be charged to the insertion of the N_0 elements. Similarly for deletes and the copying cost when halving the number of buckets. The bucket width δ should be

adjusted to match the average distance between elements in the queue in order to get an expected constant number of elements in each bucket. Hence, insert and delete can be done in expected $O(1)$ amortized time. Brown gave empirical evidence that the calendar queue achieves expected constant time for the `hold` operation. In other words, if we choose δ and M properly, the number of elements in each bucket will be $O(1)$.

Erickson et al. (see "Optimizing Static Calendar Queues" [7] for details, *Static* here means that the number of elements in the queue is unchanged, not that all events have to be known in advance) analyzed the calender queue with unsorted buckets. They describe how to choose δ and M under the assumption that only the `hold` operation is used (the case for which Brown gave empirical evidence). The value d in the `hold` operations is here defined by a random variable with probability density e. In essence, choose $\delta = \sqrt{2}\frac{\mu}{N}$ where μ is a function of e. Using this bucket width and infinitely many buckets the expected time is constant for the `hold` operation. Given a maximum duration C, choosing $M \geq C \operatorname{div} \delta + 1$ will guarantee no loss of performance over choosing infinitely many buckets. If a small degradation of the performance is acceptable one can choose $M = rN$, where r depends on the allowed degradation.

A variation of the time queue problem has been studied by Varghese and Lauck [16]. They look at the problem of providing a timer facility for an operating system. In the timer facility problem the `delLessThan` operation is called once for each time t (i.e., $c = 1$). Also even if $t < t_0$. The solution suggested by Varghese and Lauck, called *Hashed and Hierarchical Timing Wheel*, is very similar to the Calendar Queue.

3 Our Solution

We will now modify the calendar queue to achieve $O(1)$ worst case time for the `min`, `update`, `value` and `data` operations, and see under what conditions we can expect `deleteMin` and `delLessThan` to run in $O(1)$ time per deleted element. As Erickson et al. we will use the *unsorted* bucket discipline to achieve $O(1)$ worst case time for insertion into a bucket. We use lists of doubly linked nodes in each bucket and let a finger be a reference to the node that stores the element. Given a finger to the element, this achieves $O(1)$ worst case time for deletion in a bucket.

As pointed out by Thorup [12] we can always, in any monotonic priority queue, make the `min` operation run in $O(1)$ worst case time by remembering the element (and its priority) that was deleted by the last `deleteMin` and consider it part of the priority queue. We implement this by letting `deleteMin` find the element that will be min when the current min is deleted and store a finger to this element.

Since an `update` is a `delete` followed by an `insert`, if we can support `delete` and `insert` in $O(1)$ worst case time we also have `update` in $O(1)$ worst case time. The reason for the amortization in the calendar queue is the copying of elements when M is changed. If we never need to copy any elements during an `insert`

(`delete`) the time for these operations is worst case. Hence, if M and δ are fixed the copying is never needed and we achieve $O(1)$ worst case time for the `min`, `update`, `value` and `data` operations.

Under what conditions can we expect `deleteMin` to run in $O(1)$ time, and can we improve these conditions in some way? The approach with fixed M and δ is what Brown started with in his description of calendar queue. He noted that this will lead to inefficient space use if $N << M$. Moreover, if $N << M$, `deleteMin` may have to search many empty buckets to find the next element, `deleteMin` takes $O(M)$ time in the worst case. On the other hand, if $N >> M$, the current bucket may contain many elements, `deleteMin` takes $O(N)$ time in the worst case. However, on the average $O(M/N + N/M)$ time is needed. From the discussion above we conclude that we can expect $O(1)$ time for `deleteMin` if $N = \Theta(M)$ and the elements are evenly distributed among the buckets.

To improve these conditions we will focus on the case where $N = \Omega(M)$, and the main problem of `deleteMin` is to find the next element in the current bucket. Since the elements in the buckets are unsorted it takes time proportional to the number of elements in the bucket to find the new min.

One way of reducing the time could be to keep the current bucket sorted, then deletion of the min element in the bucket would take $O(1)$ time. Each element would then be involved in one sorting and the amortized cost per element would be $s(k)$ where $s(k)k$ is the cost of sorting k elements (cf. equivalence between sorting and priority queues [12]). This makes `update` of elements within the bucket i_0 too expensive for the *fast* process.

Instead, we use δ buckets of width 1, implemented as an array of doubly linked lists denoted `head`. We store the elements of the current bucket i_0 in `head[`j`]` where $j = t_e \bmod \delta$. Hence, each list in the head only stores elements with the same priority. Now `update`, `insert`, `delete` and `min` are still $O(1)$ in the worst case even though the constant is a bit higher.

In the analysis of `deleteMin` we denote the number of elements in a bucket i by B_i and the number of elements in the `head` by H. Finding the new min in the head is similar to finding the next non empty bucket in the calendar queue, which is done in $O(M/N)$ time on the average. Hence in a head with more than one element it takes $O(\delta/H + 1)$ time on the average. If $H = \Omega(\delta)$ this is $O(1)$. When the last element of the head is deleted, and all the δ buckets are empty, we need to move all the B_{i_0+1} elements of bucket $i_0 + 1$ into the head and increase i_0 by one. The cost of the copying is $O(B_{i_0+1})$, which indicates that the worst case cost of `deleteMin` is $O(N)$. However in an amortized analysis the cost of copying an element can be charged to the operation that deletes the element from the head, which makes the amortized time $O(1)$ for `deleteMin`.

Finally, we do a deamortization of the `deleteMin` operation to achieve $O(1)$ expected time instead of amortized time. The deamortization is done by using a second head denoted `head2` and move $\lceil B_{i_0+1}/H \rceil$ elements from bucket $i_0 + 1$ into `head2` in each `deleteMin` operation. When the last element is deleted from the head the rest of the elements are moved from bucket $i_0 + 1$ into `head2` and the two heads are swapped. If an element should be inserted (updated) into bucket

$i_0 + 1$ it will instead be inserted into `head2`. Hence B_{i_0+1} will never increase and therefore the number of elements in bucket $i_0 + 1$ will be $O(1)$ when the last element is deleted from the head. If $H = \Omega(B_{i_0+1})$ the cost is $O(1)$ for copying the elements.

Now if $N = \Omega(M)$, $H = \Omega(\delta)$ and $H = \Omega(B_{i_0+1})$ we have $O(1)$ expected time for `deleteMin`. If not, the time for `deleteMin` and `hold` is exp $O(M/N + B_{i_0+1}/H + \delta/H)$ and $O(M+N+\delta)$ worst case. If we choose $\delta = \Theta(M)$ the above conditions reduce to $H = \Omega(\delta)$ (since $N \geq H$) and $H = \Omega(B_{i_0+1})$ where the second condition depends on the distribution of the elements among the buckets.

Now let us see what conditions are needed if a sequence of `delLessThan` calls are used instead of `deleteMin`. First we note that $O(\delta/c)$ calls are made between two changes of heads. Consequently, if $\lceil B_{i_0+1}/(\delta/c) \rceil$ elements are moved each time `delLessThan` is called, all the elements in bucket $i_0 + 1$ will be moved to the second head before the next head swap. If $\delta = \Omega(B_{i_0+1})$ then only a constant number of elements are moved each time. If the min element is deleted, `delLessThan` needs to find the next element to be min which is done in $O(1)$ time on the average if $H = \Omega(\delta)$. However, assume that p empty buckets have to be scanned to find the next element to be min, when the min element is deleted, then approximately p/c `delLessThan` calls are made before the min is deleted again. Hence on the average c buckets are scanned by `delLessThan` even if $H = o(\delta)$. Note that this is true even if we need to scan the `buckets` array to find the next non empty bucket. The condition we are left with is $\delta = \Omega(B_{i_0+1})$ which is fulfilled if there is only a constant number of elements with equal priority.

Since we know the maximum duration C of all the elements, we can choose δ and M to cover this range, hence $C = M\delta$. We choose $M, \delta = O(\sqrt{C})$ to have $\delta = \Theta(M)$. This gives a space bound of $O(\sqrt{C}+N)$ since the number of buckets and the number of buckets in the heads are $O(\sqrt{C})$. We note that when using $\delta = M = \sqrt{C}$ both mod$\sqrt{C}$ and div$\sqrt{C}$ can be computed fast if $C = 2^h$ and even if $C \neq 2^h$ the approximation $\delta = 2^{\frac{\lg C}{2}}$ is good enough. The above analysis leads to the time and space bounds in Table 2.

Table 2. Time bounds and space for our solution to the Time Queue problem

Operation	Our modified CQ
`insert`	$O(1)$
`delete`	$O(1)$
`min`	$O(1)$
`deleteMin`	exp $O(1)$ if $H = \Omega(\delta), H = \Omega(B_{i_0+1})$
`hold`	exp $O(1)$ if $H = \Omega(\delta), H = \Omega(B_{i_0+1})$
`update`	$O(1)$
`delLessThan`	exp $O(1)$ if $\delta = \Omega(B_{i_0+1})$
Space	$O(\sqrt{C} + N)$

3.1 Representation and Algorithms

The data structure consists of buckets, two heads of buckets, a finger to the min element and an index into the current bucket (see Algorithm 1 and Fig. 1 for an example). Both the buckets and the two heads of buckets are arrays of lists of doubly linked nodes. We let a finger be a reference to a node.

```
typedef void * Element;
typedef struct node {
  struct node * prev;
  struct node * next;
  Element       e;
  int           t;
  Bool          inHead;
} NODE,
typedef struct list {
  NODE *        first;
  NODE *        last;
  int           nrOfElements;
} LIST;
typedef struct tq {
  LIST          buckets[M];
  LIST          head[δ];          int     H;
  LIST          head2[δ];         int     H2;
  NODE *        min;
  int           i0;
} TQ;
```

Algorithm 1: Representation of the Time Queue

We first look at the algorithms for only one process (the slow one) and later see what modifications are needed for the fast process.

- During `insert` (Algorithm 3) we calculate the bucket index for the new element and check if the element should be in either of the heads. If it is we calculate the head index j and insert the new element at the end of the list, otherwise we insert it at the end of the list of the proper bucket.
- In the `delete` (Algorithm 4) we have the finger the element and we can easily delete it from the appropriate list. If it is the last element in the list we mark the bucket as empty.
- The `update` (Algorithm 5) is, as said, a deletion followed by an insertion.
- The `min` returns the stored min finger.
- The `deleteMin` (Algorithm 6) first deletes the min element. Then it searches for the next element that should be min and updates the min reference. Finally, the routine moves some of the elements from the next bucket into `head2`.

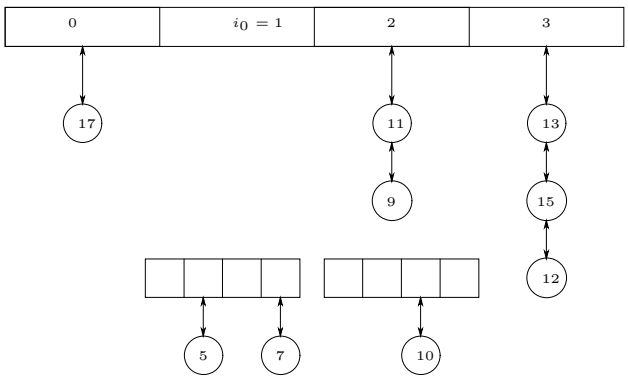

Fig. 1. Time Queue representation with: $C = 16$, $N = 9$, $M = \delta = 4$, $B_0 = 1$, $B_1 = 0$, $B_2 = 2$, $B_3 = 3$, $H = 2$, $H2 = 1$, $t_0 = 5$ and $i_0 = 1$

If the deleted element was the last element in the head, it first search for the next non empty bucket, moves the remaining elements from that bucket to `head2`, swaps the two heads, and increases i_0. Then it continues with the search for the next element to be min.

- As long as the time t_0 of the min element is less than the specified time, `delLessThan` (Algorithm 8) gets min, calls the function $\mathcal{F}$ on it, deletes it and finds the new min. Finally, it moves some elements from the next bucket into `head2`.
  If `delLessThan` deletes the last element in the head it search for the next non empty bucket, moves the remaining elements from that bucket to `head2`, swaps the two heads, and increases i_0.
- The operations to get the data and time from a finger, `data` and `value` respectively, only returns the data and time from the linked list node.

3.2 Support for Concurrent Processes

We assume that there are just two processes: one fast process and one slow process. The fast process only has to update elements in the near future ϵ while the remaining updates are sent to the slow process. Without loss of generality we assume that $\epsilon \leq \delta$ and hence only elements in bucket i_0 and bucket i_0+1 may be updated by the fast process. The elements in bucket i_0 are stored in `head` while the elements in bucket $i_0 + 1$ may be in both `buckets`$[i_0 + 1]$ and `head2`. We will use three locks to ensure mutual exclusive access to these entities. The assumption that $\epsilon \leq \delta$ is not really a restriction since if this is not true we only need to add more locks for the buckets that need protection.

Whenever `head`, `H`, i_0 or `min` is read or written we acquire `headLock`. Similarly for `head2Lock` and `bucketLock`. Since only the slow process modifies i_0 and `min` it does not need to acquire the `headLock` in order to read these variables. The fast process always has to acquire the corresponding lock. Algorithm 9 shows

the deletion by the slow process with the proper locks. To avoid deadlocks, we choose to break the circular chain condition by imposing a linear order of the locks [5]. If a process needs more than one lock it has to acquire them in the following order: `headLock`, `head2Lock` and `bucketLock`. The representation of the time queue includes these locks (Algorithm 2). The `update` (Algorithm 10)

```
typedef struct tq {
  LIST          buckets[M];
  LIST          head[δ];          int     H;
  LIST          head2[δ];         int     H2;
  NODE *        min;
  int           i0;
  LOCK          headLock;
  LOCK          head2Lock;
  LOCK          bucketLock;
} TQ;
```

Algorithm 2: Representation of the Time Queue with locks

for the fast process is only allowed to move elements to/from the heads and corresponding buckets. If the element should be moved to/from another bucket it passes a request to the slow process.

If there are more than one process of each kind special care is needed for the slow processes. We need also to acquire the lock when reading variables in the slow process and have locks for all the different parts of the time queue data structure. More than one fast process can be handled without any special care.

4 Conclusion

We have proposed a solution for two different processes to simultaneously maintain a time queue. One of the processes performs only a subset of the operations in $O(1)$ worst case time, while the other process shall perform all operations. All operations except `deleteMin` and `delLessThan` are performed in $O(1)$ worst case time. `deleteMin` is performed in $O(1)$ expected time and `delLessThan` is performed in $O(1)$ expected time per deleted element.

The main difference from the Hashed and Hierarchical Timing Wheels by Varghese and Lauck [16] is the deamortization of the `deleteMin` and the concurrent solution.

Furthermore, we have shown how to allow one fast and one slow process to maintain our data structure by using locks to provide mutual exclusion.

References

1. Arne Andersson and Mikkel Thorup. Tight(er) worst-case bounds on dynamic searching and priority queues. In *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing*, pages 335–342, Portland, Oregon, US, May 21–23 2000. ACM Press.
2. Andrej Brodnik, Svante Carlsson, Johan Karlsson, and J. Ian Munro. Worst case constant time priority queue. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 523–528, Washington, DC, US, 7–9 January 2001.
3. Randy Brown. Calendar queues: A fast $O(1)$ priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31(10):1220–1227, October 1988.
4. Boris V. Cherkassky, Andrew V. Goldberg, and Craig Silverstein. Buckets, heaps, lists, and monotone priority queues. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 83–92, New Orleans, Louisiana, US, 5–7 January 1997.
5. E. G. Coffman, Jr., M. J. Elphick, and A. Shoshani. System deadlocks. *ACM Computing Surveys*, 3(2):67–78, June 1971.
6. Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–791, 1994.
7. K. Bruce Erickson, Richard E. Ladner, and Anthony LaMarca. Optimizing static calendar queues. In *35th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 732–742. IEEE, 20–22 November 1994. Also published as tech. report TR-94-09-02 `ftp://ftp.cs.washington.edu/tr/1994/09/UW-CSE-94-09-02.PS.Z`.
8. Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, July 1984.
9. Michael L. Fredman and Michael E. Saks. The cell probe complexity of dynamic data structures. In *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing*, pages 345–354, Seattle, Washington, US, May14–17 1989. ACM Press.
10. Douglas W. Jones. An empirical comparison of priority-queue and event-set implementations. *Communications of the ACM*, 29(4):300–311, April 1986.
11. Arnold Schönhage. Storage modifications machines. *SIAM Journal on Computing*, 9(3):490–508, August 1980.
12. Mikkel Thorup. On RAM priority queues. In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 59–67, Atlanta, Georgia, US, 28–30 January 1996.
13. P. van Emde Boas. Preserving order in a forest in less than logarithmic time. In *16th Symposium on Foundations of Computer Science*, pages 75–84, 1975.
14. P. van Emde Boas. Machine models and simulations. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, pages 3–66. Elsevier/MIT Press, Amsterdam, 1990.
15. P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.
16. George Varghese and Anthony Lauck. Hashed and hierarchical timing wheels: Efficient data structure for implementing a timer facility. *IEEE/ACM Transaction on Networking*, 5(6):824–834, December 1997.

17. Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Information Processing Letters*, 17(2):81–84, 24 August 1983.
18. J. W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7:347–348, 1964.
19. Andrew Chi-Chih Yao. Should tables be sorted? *Journal of the ACM*, 28(3):614–628, July 1981.

A Pseudo Code

The time queue, `TQ`, (Algorithm 1 and Algorithm 2) consists of the arrays, `buckets`,`head`, and `head2`, of lists. These lists are doubly linked lists and each list has a counter, `nrOfElements`, of the number of elements in it. The lists consists of nodes, `NODE`, which stores the element and its time together with an indication, `inHead`, whether the element is in either of the two heads or not. The node also stores references to its neighbours in the list. The `TQ` also stores the number of elements in each of the two heads, `H` and `H2` respectively. Further the current bucket index, i_0, indicates in which of the buckets the min element is. It also has a reference, `min`, to the node storing the min elements and its priority.

```
NODE *
insert(TQ tq, Element e, int t) {
  int    bucket, index;
  NODE  *n;
  Create new node
    n = createNode();
    n->e = e;
    n->t = t;
    n->next = n;
    n->prev = n;
    n->inHead = false;
  bucket = (n->t  div δ) mod M;
  index = n->t mod δ;
  if (bucket == tq.i0) {
    Append n at the end of tq.head[index]
      if (tq.head[index].last != NULL) {
        tq.head[index].last->next = n;
        n->prev = tq.head[index].last;
        n->next = tq.head[index].last->next;
      } else {
        tq.head[index].first = n;
      }
      tq.head[index].last = n;
      tq.head[index].nrOfElements++;
    n->inHead = true;
    tq.H++;
  } else if (bucket == ((tq.i0+1) mod M)) {
    Append n at the end of tq.head2[index]
    n->inHead = true;
    tq.H2++;
  } else {
    Append n at the end of tq.buckets[bucket]
  }
  return n;
}
```

Algorithm 3: Insertion into Time Queue

```
void
delete(TQ tq, NODE * finger) {
  int    bucket, index;
  bucket = (finger->t div δ) mod M;
  if (finger->next == finger) {                    /* Last element in list */
    if (!finger->inHead) {
      Clear list tq.buckets[bucket]
        tq.buckets[bucket].first = NULL;
        tq.buckets[bucket].last = NULL;
        tq.buckets[bucket].nrOfElements = 0;
    } else {
      index = finger->t mod δ;
      if (bucket == tq.i0) {
        Clear list tq.head[index]
        tq.H--;
      } else {                          /* (bucket == ((tq.i0+1) mod M)) */
        Clear list tq.head2[index]
        tq.H2--;
      }
    }
  } else {
    Remove node from its list
      finger->next->prev = finger->prev;
      finger->prev->next = finger->next;
      if (!finger->inHead) {
        tq.buckets[bucket].nrOfElements--;
      } else {
        index = finger->t mod δ;
        if (bucket == tq.i0) {
          tq.head[index].nrOfElements--;
          tq.H--;
        } else {                        /* (bucket == ((tq.i0+1) mod M)) */
          tq.head2[index].nrOfElements--;
          tq.H2--;
        }
      }
  }
  free(finger);
}
```

Algorithm 4: Deletion from Time Queue

```
void
update(TQ tq, NODE * finger, int t) {
  int    bucket, index;
  bucket = (finger->t div δ) mod M;
  if (finger->next == finger) {                    /* Last element in list */
    if (finger->inHead == false) {
      Clear list tq.buckets[bucket]
    } else {
      index = finger->t mod δ;
      if (bucket == tq.i0) {
        Clear list tq.head[index]
      } else {                                     /* (bucket == ((tq.i0+1) mod M)) */
        Clear list tq.head2[index]
      }
    }
  } else {
    Remove node from its list
  }
  finger->t = t;
  finger->inHead = false;
  bucket = (finger->t div δ) mod M;
  index = finger->t mod δ;
  if (bucket == tq.i0) {
    Append finger at the end of tq.head[index]
    finger->inHead = true;
    tq.H++;
  } else if (bucket == ((tq.i0+1) mod M)) {
    Append finger at the end of tq.head2[index]
    finger->inHead = true;
    tq.H2++;
  } else {
    Append finger at the end of tq.buckets[bucket]
  }
}
```

Algorithm 5: Updates in Time Queue

```
void
deleteMin(TQ tq) {
  int    bucket, index;
  int    n, j;
  NODE  *finger;
  LIST  *head;
  Delete the min element
    finger = min(tq);
    j = finger->t mod δ
    delete(finger);
    finger = NULL;
  if (tq.H == 0) {                          /* Last element in head deleted */
    Find next non empty bucket
      if (tq.H2 != 0) {
        bucket = (tq.i0 +1) mod M;
      } else {
        bucket = (tq.i0 +1) mod M;
        while ((tq.buckets[bucket].nrOfElements == 0) &&
               (bucket != tq.i0)) {
               bucket = (bucket+1) mod M;
        }
      }
      if (bucket == tq.i0) {                        /* No more elements */
        tq.min = NULL;
        return;
      }
    Move all elements in buckets[bucket] to head2
      while (tq.buckets[bucket].nrOfElements > 0) {
        finger = tq.bucket[bucket].first;
        update(tq, finger, finger->t);
      }
    Swap heads
      head = tq.head;
      n = tq.H;
      tq.head = tq.head2;
      tq.H = tq.H2;
      tq.head2 = head;
      tq.H2 = n;
    Increase i0
      tq.i0 = bucket;
    j = 0;
  }
continued in Algorithm 7
```

Algorithm 6: Deletion of min from Time Queue

```
  Search for next min
    while (tq.head[j].nrOfElements == 0) {
      j++;
    }
  Update the min reference
    tq.min = tq.head[j].first;
  Move B_{i0+1}/H elements from bucket i0 + 1 to head2
    bucket = (tq.i0 +1) mod M;
    for (i = 1;
         i <= ⌈tq.buckets[bucket].nrOfElements/tq.H⌉;
         i++) {
      finger = tq.buckets[bucket].first;
      update(tq, finger, finger->t);
    }
}
```

Algorithm 7: Deletion of min from Time Queue (cont.)

```
void
delLessThan(TQ tq, int t, func F) {
  NODE  *finger;
  int    bucket;
  int    index;
  int    i, n;
  LIST  *head;
  while (t < min()->t) {
    F(min());
    Delete the min element
    if (tq.H == 0) {                          /* Last element in head deleted */
      Find next non empty bucket
      Move all elements in buckets[bucket] to head2
      Swap heads
      Increase i0
    }
    Search for next min
    Update the min reference
  }
  Move B_{i0+1}/(δ − (t mod δ)) elements from bucket i0 + 1 to head2
}
```

Algorithm 8: Deletion of all elements with time less than t from Time Queue

```
void
delete(TQ tq, NODE * finger) {
  int    bucket;
  LOCK  *l;
  bucket = (finger->t  div δ) mod M;
  if (!finger->inHead) {
    if(bucket == ((tq.i0+1) mod M)) {
      l = bucketLock;
    } else {
      l = NULL;
    }
  } else {
    if (bucket == tq.i0) {
      l = headLock;
    } else {                              /* (bucket == ((tq.i0+1) mod M)) */
      l = head2Lock;
    }
  }
  Acquire(l, BLOCK);
  Delete as in Algorithm 4
  Release(l);
  free(finger);
}
```

Algorithm 9: Deletion from Time Queue with Mutual Exclusion

```
void
update(TQ tq, NODE * finger, int t) {
  int    bucket, bucket2, index;
  bool   gotH2Lock = false;
  bool   gotbucketLock = false;
  if (Acquire(headLock, NOBLOCK)) {
    bucket = (finger->t div δ) mod M;
    bucket2 = (t div δ) mod M;
    if (!(((bucket == i0) || (bucket == (i0 + 1 mod M))) &&
          ((bucket2 == i0) || (bucket2 == (i0 + 1 mod M))))) {
      Release(headLock);
      goto SEND;
    }
    if (((bucket == (i0 + 1 mod M)) && finger->inHead) ||
        (bucket2 == (i0 + 1 mod M))) {
      if (Acquire(head2Lock, NOBLOCK))
        gotH2Lock = true;
      else {
        Release(headLock);
        goto SEND;
      }
    }
    if ((bucket == (i0 + 1 mod M)) && !finger->inHead) {
      if (Acquire(bucketLock, NOBLOCK))
        gotbucketLock = true;
      else {
        if (gotH2Lock)
          Release(head2Lock);
        Release(headLock);
        goto SEND;
      }
    }
    Update as in Algorithm 5
    if (gotbucketLock)
      Release(bucketLock);
    if (gotH2Lock)
      Release(head2Lock);
    Release(headLock);
    return;
  }
SEND:                                   /* Send request to the slow process */
  SendMesg(update, tq, finger, t);
}
```

Algorithm 10: Fast process updates in Time Queue