Brief Announcement: Concurrent Double-Ended Priority Queues

Panagiota Fatourou
Department of Computer Science, University of Crete, Heraklion, Greece
FORTH ICS, Heraklion, Greece

Eric Ruppert
York University, Toronto, Canada

Ioannis Xiradakis
Department of Computer Science, University of Crete, Heraklion, Greece
FORTH ICS, Heraklion, Greece

Abstract

This work provides the first concurrent implementation of a double-ended priority queue (DEPQ). We describe a general way to add an ExtractMax operation to any concurrent priority queue that already supports Insert and ExtractMin.

2012 ACM Subject Classification Theory of computation → Concurrent algorithms; Theory of computation → Data structures design and analysis

Keywords and phrases shared-memory, data structure, double-ended, priority queue, priority deque, heap, skip list, combining

Digital Object Identifier 10.4230/LIPIcs.DISC.2025.55

Related Version *Full Version*: `https://arxiv.org/abs/2508.13399` [11]

Funding *Panagiota Fatourou and Ioannis Xiradakis:* Greek Ministry of Education, Religious Affairs and Sports through the HARSH project (project no. ΥΠ3ΤΑ - 0560901), within the framework of the National Recovery and Resilience Plan–Greece 2.0–with funding from the European Union–NextGenerationEU.
Eric Ruppert: Natural Sciences and Engineering Research Council of Canada.

1 Introduction

Priority queues, which store a set of keys and support an Insert operation that adds a key to the set and an ExtractMin operation that removes and returns the minimum key, have long been recognized as an important data structure for concurrent systems. They have been used in operating systems for job queues and load balancing, heuristic searches [29], graph algorithms (e.g., [1, 17]) and for event-driven simulations [19]. There are numerous concurrent implementations of priority queues.

In the single-process setting, there has been much research on designing a *double-ended* priority queue (DEPQ), which supports both ExtractMin and ExtractMax operations [31]. We provide a general transformation to construct a linearizable concurrent DEPQ from a concurrent (single-ended) priority queue. Implementing a *concurrent* DEPQ has not been explored previously.

We take our inspiration from the insight that some of the sequential DEPQ data structures can be viewed as being constructed from a pair of single-ended priority queues [5]. Our general construction in Section 3 uses two linearizable, concurrent priority queues to construct a linearizable *dual-consumer* DEPQ, which allows only one process to perform ExtractMin and one process to perform ExtractMax at a time. Insertions proceed concurrently with one another and with Extract operations. The construction works even if the underlying priority queues used are *single-consumer*, meaning that only one process at a time may perform ExtractMin operations. Our construction uses a new lightweight layer of synchronization between the two processes that perform ExtractMin and

39th International Symposium on Distributed Computing (DISC 2025).
Editor: Dariusz R. Kowalski; Article No. 55; pp. 55:1–55:7
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

ExtractMax operations. It preserves linearizability and lock-free progress: if the underlying priority queue is lock-free, then so is the resulting DEPQ. If the underlying priority queue also supports deletions of arbitrary elements, we provide time and space bounds for our DEPQ.

Section 4 describes how to adapt our dual-consumer DEPQ to handle concurrent Extract operations at each end using a lock-based combining technique [10]. At a high level, each process that acquires the lock for one end of the DEPQ performs a whole batch of Extract operations, which keeps the overhead for synchronization low.

Our technique is general enough to be applied to any concurrent priority queue. In the full version [11], we provide an example that applies our technique (including combining) to a simple list-based priority queue.

2 Related Work

There are many DEPQ implementations for just one process (see e.g., Chong and Sahni's survey [5]). In the concurrent setting, we are unaware of any previous work that aims to provide a linearizable concurrent DEPQ implementation. Medidi and Deo [24] described a DEPQ that supports batches of parallel operations, but their focus was on the synchronous PRAM model. Our construction uses two concurrent (single-ended) priority queues as building blocks. See [7, 26] for surveys of early work on concurrent priority queues, including many based on sequential heap data structures. Tamir, Morrison and Rinetzky [34] added support for an operation that modifies an existing key in a lock-based concurrent heap. Pugh's skip list data structure [28] has been used as the basis for several concurrent priority queues [3, 21, 32, 33]. Liu and Spear [22] gave a novel concurrent priority queue data structure based on a tree where each node stores a sorted linked list.

The concurrent priority queues discussed above allow multiple concurrent consumers. Our construction requires only a single-consumer priority queue, which may be easier to implement. Hoover and Wei [16] gave a wait-free single-consumer priority queue where operations take $O(\log n + \log p)$ steps, where n is the number of elements in the queue and p is the number of processes accessing it.

One way to use existing data structures to build a linearizable concurrent DEPQ is to use a binary search tree (BST), where the tree is sorted by key values [20, Section 5.2.3]. There are a number of lock-free concurrent BST implementations (e.g., [8, 27]) that support insertion and deletion of keys. The Delete operation can easily be modified to delete (and return) the minimum or maximum key present in the BST to yield the Extract operations of a DEPQ. Since repeated Extract operations at one end of the DEPQ could yield a lopsided BST, it would likely be desirable to use a balanced BST, such as the concurrent chromatic tree, which has both lock-based and lock-free implementations [2, 4]. However, even with balancing, this would require each Extract operation to traverse a path of length $\Theta(\log n)$ in the BST when the DEPQ contains n elements. In contrast, if we apply our approach to a (single-ended) concurrent priority queue based on a list or skip list, Extract operations find the required key right at the beginning of the list, and less restructuring is required, compared to the rebalancing of chromatic trees. There are also concurrent implementations of (single-ended) priority queues that augment a search tree with a sorted singly-linked list of keys in the tree to expedite ExtractMin operations [18, 30].

3 Constructing a Dual-Consumer DEPQ from Two Priority Queues

Our DEPQ construction uses two (single-ended) priority queues $MinPQ$ and $MaxPQ$ organized using opposite total orders on the keys. Thus, an Extract on $MinPQ$ returns the minimum key and an Extract on $MaxPQ$ returns the maximum key. Algorithm 1 gives pseudocode for our construction.

An **Insert** operation on the DEPQ simply inserts the key into both priority queues. To coordinate extractions, each item has an associated *reserved* bit, which is initially 0. An **ExtractMin** on the DEPQ repeatedly removes the (next) minimum element from *MinPQ* and tries to set the element's *reserved* bit using a TestAndSet instruction until it successfully changes the *reserved* bit of some element. That element is then returned. If, at any time, the ExtractMin observes that *MinPQ* is empty, it terminates and indicates that the DEPQ is empty. The **ExtractMax** operation on the DEPQ is symmetric to ExtractMin, using *MaxPQ* in place of *MinPQ*.

The *reserved* bit ensures that an element cannot be returned twice by both an ExtractMin and an ExtractMax. An element removed from the DEPQ by an ExtractMin operation may remain in *MaxPQ* for some time, but if it is eventually removed from *MaxPQ* by an ExtractMax operation, the ExtractMax will skip the item because its TestAndSet operation will fail to set the item's *reserved* bit.

If the priority queue implementation we are using also supports a Delete operation, then we can add the optional lines 15 and 25 to the Extract operations. The DEPQ is linearizable regardless of whether these lines are included or not. When an ExtractMin operation removes an item from *MinPQ*, line 15 removes it from *MaxPQ*. Whether the inclusion of these lines improves performance may depend upon the underlying priority queue: if deleting an arbitrary element is not much more costly than extracting the minimum, or if performance of the priority queues would degrade significantly due to the presence of obsolete items, then it may be worthwhile to include lines 15 and 25.

3.1 Linearizability and Progress

There are challenges in showing that the DEPQ is linearizable. Since *MinPQ* and *MaxPQ* are updated separately by Insert operations, their contents may not exactly match. Moreover, since the *reserved* bit is updated *after* removing the item from *MinPQ* or *MaxPQ*, the order in which items' bits are set may be different from the order in which they are removed from the priority queues.

Since we have assumed that the implementations of *MinPQ* and *MaxPQ* are linearizable, the composability property of linearizability [15] allows us to consider operations applied to each of these priority queues as atomic steps. For the sake of simplicity in our presentation, we assume that all keys inserted into the DEPQ are distinct. This allows us to talk about the Insert that inserted the key extracted by some ExtractMax or ExtractMin operation without ambiguity.

Fix an execution α. We now describe how to linearize operations in α. We linearize each ExtractMin and ExtractMax when it performs a successful TestAndSet (or when it sees that *MinPQ* or *MaxPQ* is empty, in the case of operations that return *nil*). An Insert(x) adds x to *MinPQ* and then to *MaxPQ*. By default, we linearize the Insert when it adds x to *MaxPQ*. However, if an ExtractMin removes x from *MinPQ* and returns it before x is added to *MaxPQ*, we must shift the linearization point of the Insert earlier.

We now describe this linearization more formally. Consider an ExtractMin or ExtractMax operation e. If e never performs an Extract at line 12 or 22, it is not linearized, since it never accesses shared memory. If e does perform an Extract at line 12 or 22, let $last_e$ be the last step where e does so, and let $result_e$ be the key returned by this last ExtractMin performed by e.

L1. Linearize an ExtractMin or ExtractMax operation e at $last_e$ if either $result_e = nil$ or e performs a successful TestAndSet at line 14 or 24.

L2. Linearize each Insert operation at the earlier of

- **a.** its insertion into *MaxPQ* at line 8, or
- **b.** immediately before the ExtractMin operation on DEPQ that returns its item.

If neither of these events occur, then the Insert is not linearized.

Algorithm 1 Generic construction of a dual-consumer DEPQ from two single-consumer priority queues.

```
1: class item
2:     Key key
3:     boolean reserved
4: end item
5: function Insert(Key x)
6:     item i := new item with key = x, reserved = 0
7:     MinPQ.Insert(i)
8:     MaxPQ.Insert(i)
9: end function
10: function ExtractMin : Key
11:     while true do                              ▷ repeatedly extract element from MinPQ
12:         item x := MinPQ.Extract
13:         if x = nil then return nil             ▷ DEPQ is empty
14:         else if TestAndSet(x.reserved) = 0 then ▷ return when TestAndSet succeeds
15:             MaxPQ.Delete(x)                    ▷ this line is optional
16:             return x.key
17:         end if
18:     end while
19: end function
20: function ExtractMax : Key
21:     while true do                              ▷ repeatedly extract element from MaxPQ
22:         item x := MaxPQ.Extract
23:         if x = nil then return nil             ▷ DEPQ is empty
24:         else if TestAndSet(x.reserved) = 0 then ▷ return when TestAndSet succeeds
25:             MinPQ.Delete(x)                    ▷ this line is optional
26:             return x.key
27:         end if
28:     end while
29: end function
```

In the full version [11], we prove every operation that terminates is assigned a linearization point, and that the linearization points of operations are within their execution intervals. Combining these claims with Lemma 1, which is the cornerstone of our proof, we get Theorem 2, proving linearizability.

▶ **Lemma 1.** *In the sequential execution defined by the linearization, each* ExtractMin *or* ExtractMax *operation e returns $result_e$.*

▶ **Theorem 2.** *If $MinPQ$ and $MaxPQ$ are linearizable (single-consumer) priority queues, then Algorithm 1 is a linearizable dual-consumer DEPQ.*

The Insert operation on the DEPQ is lock- or wait-free if insertions on the underlying priority queues are. The ExtractMin and ExtractMax operations on the DEPQ are lock-free if extractions (and deletions, if lines 15 and 25 are included) on the underlying priority queues are. This is because the TestAndSet on line 14 or 24 can fail only if some other ExtractMin or ExtractMax operation has successfully performed the TestAndSet, and that other operation is guaranteed to terminate.

ExtractMin and ExtractMax operations on the DEPQ are not wait-free, even if the underlying priority queue is wait-free because one such operation may repeatedly fail its TestAndSet in every iteration. Modifying the construction to make it wait-free would probably require some coordination between the two processes performing extractions at opposite ends of the DEPQ.

We provide amortized bounds on step complexity, provided that lines 15 and 25 are included. We say that the amortized step complexity is $O(T_i)$ for insertions and $O(T_e)$ for extractions if, for every (finite) execution in which m_i Inserts and m_e ExtractMin and ExtractMax operations are invoked, the total number of steps in the execution is $O(m_i \cdot T_i + m_e \cdot T_e)$. Let n be the maximum number of elements in the DEPQ at any time, if operations are performed sequentially in the order of the linearization. Let c be the maximum point contention, that is, the maximum number of operations that are active at any one time. Assume that the underlying priority queues have space complexity $S(n, c)$ and that the (amortized) step complexity for their Insert, Extract and Delete operations are $T'_i(n, c), T'_e(n, c)$ and $T'_d(n, c)$, respectively. The maximum number of elements that are ever in *MinPQ* or *MaxPQ* is $O(n+c)$. Thus, the amortized step complexity for each insertion and extraction operation on the DEPQ is $O(T'_i(n + c, c))$ and $O(T'_e(n + c, c) + T'_d(n + c, c))$, respectively.

If we have bounds on the *expected* step complexity of the underlying data structure (for example, if the data structure is randomized, like a skip list) then the bounds described above hold for the expected amortized step complexity of operations on the DEPQ. If T_i is a bound on the worst-case time per insertion into the underlying priority queue, rather than an amortized bound, then the bound for insertions on the DEPQ is also a worst-case bound.

4 Constructing a Multi-Consumer DEPQ from a Dual-Consumer DEPQ

The construction provided in Section 3 permits a single Extract operation at each end of the DEPQ at a time. If multiple processes wish to perform Extract operations, we could use two locks: one for each end. A process must acquire the lock for one end of the DEPQ before performing an Extract on that end. Insert operations can proceed regardless of the locks.

To reduce the overhead required by locking, we can use software combining, wherein the process that acquires the lock performs its own work as well as the work of other processes that failed to acquire the lock. Combining is particularly useful for data structures like stacks, queues, deques and priority queues, which have hot spots of contention (e.g., [10, 13, 14, 9]). Early versions of this approach [12, 35] used a tree to combine requests for work. Hendler et al. [14] developed a more efficient approach called flat combining.

We use the CC-Synch combining algorithm of Fatourou and Kallimanis [10] for our DEPQ. This combining algorithm has improved efficiency due to an integrated scheme for both locking the data structure and organizing the requests for operations on it. CC-Synch implements a combining-friendly variant of a queue lock [6, 23, 25]. The queue that implements the lock is also used to store the active operations in FIFO order. Processes use this lock to choose a *combiner* process, which is delegated to perform a batch of pending operations. After announcing its operation, each process p that is not chosen as the combiner simply waits (by performing local spinning) until a combiner informs p that its requested operation has been completed and provides the result of its operation.

We use two instances of CC-Synch, one for each end of the DEPQ. To perform an Extract operation on the DEPQ, a process adds its operation to the FIFO queue of the appropriate instance of CC-Synch. When CC-Synch chooses a combiner process to perform a batch of Extract operations from the FIFO queue, the combiner performs the Extracts one by one using Algorithm 1. An Insert simply runs Algorithm 1 directly, without using CC-Synch. CC-Synch ensures that only one process acts as a combiner at a time, satisfying Algorithm 1's requirement that there be only one process performing Extract operations at each end of the DEPQ. This approach improves locality in accessing the data structure, since the combiner process performs an entire batch of Extract operations. It also avoids having a hotspot of contention because multiple Extract operations would typically access the same part of the data structure. We remark that there is little contention on the *reserved* bits of items: only the two Extract operations that extract an item from *MinPQ* and *MaxPQ* can ever access its *reserved* field, and the *reserved* field is ignored by *Insert* operations.

References

1 D. P. Bertsekas, F. Guerriero, and R. Musmanno. Parallel asynchronous label-correcting methods for shortest paths. *Journal of Optimization Theory and Applications*, 88(2):297–320, February 1996. doi:10.1007/BF02192173.

2 Joan Boyar, Rolf Fagerberg, and Kim S. Larsen. Amortization results for chromatic search trees, with an application to priority queues. *Journal of Computer and System Sciences*, 55(3):504–521, 1997. doi:10.1006/jcss.1997.1511.

3 Anastasia Braginsky, Nachshon Cohen, and Erez Petrank. CBPQ: high performance lock-free priority queue. In *Proc. 22nd International Conference on Parallel and Distributed Computing*, volume 9833 of *LNCS*, pages 460–474. Springer, 2016. doi:10.1007/978-3-319-43659-3_34.

4 Trevor Brown, Faith Ellen, and Eric Ruppert. A general technique for non-blocking trees. In *Proc. 19th ACM Symposium on Principles and Practice of Parallel Programming*, pages 329–342, 2014. doi:10.1145/2555243.2555267.

5 Kyun-Rak Chong and Sartaj Sahni. Correspondence-based data structures for double-ended priority queues. *ACM J. Exp. Algorithmics*, 5, December 2000. doi:10.1145/351827.351828.

6 Travis S. Craig. Building FIFO and priority-queuing spin locks from atomic swap. Technical Report 93-02-02, Department of Computer Science and Engineering, University of Washington, 1993.

7 Kristijan Dragičević and Daniel Bauer. A survey of concurrent priority queue algorithms. In *Proc. 22nd IEEE International Symposium on Parallel and Distributed Processing*, pages 1–6. IEEE, 2008. doi:10.1109/IPDPS.2008.4536331.

8 Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *Proc. 29th ACM Symposium on Principles of Distributed Computing*, pages 131–140, 2010. doi:10.1145/1835698.1835736.

9 P. Fatourou and P. Papadogiannakis. Double-ended queues based on software combining. Manuscript, 2025.

10 Panagiota Fatourou and Nikolaos D. Kallimanis. Revisiting the combining synchronization technique. In *Proc. 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 257–266, 2012. doi:10.1145/2145816.2145849.

11 Panagiota Fatourou, Eric Ruppert, and Ioannis Xiradakis. Concurrent double-ended priority queues. *CoRR*, abs/2508.13399, 2025. URL: https://arxiv.org/abs/2508.13399.

12 James R. Goodman, Mary K. Vernon, and Philip J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *Proc. 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 64–75, April 1989. doi:10.1145/70082.68188.

13 O. Grimes, A. Hassan, P. Fatourou, and R. Palmieri. PIPQ: A strict insert-optimized concurrent priority queue. In *Proc. 39th International Symposium on Principles of Distributed Computing*, 2025.

14 Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proc. 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 355–364, 2010. doi:10.1145/1810479.1810540.

15 Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990. doi:10.1145/78969.78972.

16 Kenneth D. Hoover and Yuanhao Wei. A fast single-extractor wait-free priority queue. Manuscript. A brief overview in the *Review of Undergraduate Computer Science* is available from https://rucs-uoft.github.io/theory-of-computation/a-fast-single-extractor-wait-free-priority-queue, 2017.

17 Qin Huang and W.E. Weihl. An evaluation of concurrent priority queue algorithms. In *Proc. 3rd IEEE Symposium on Parallel and Distributed Processing*, pages 518–525, 1991. doi:10.1109/SPDP.1991.218255.

18 Theodore Johnson. A highly concurrent priority queue. *J. Parallel Distributed Comput.*, 22(2):367–373, 1994. doi:10.1006/JPDC.1994.1097.

19 Douglas W. Jones. Concurrent simulation: an alternative to distributed simulation. In *Proc. 18th Winter Simulation Conference*, pages 417–423. ACM, 1986. doi:10.1145/318242.318468.

20 Donald E. Knuth. *The Art of Computer Programming*, volume 3. Addison Wesley Longman, third edition, 1998.

21 Jonatan Lindén and Bengt Jonsson. A skiplist-based concurrent priority queue with minimal memory contention. In *Proc. 17th International Conference on Principles of Distributed Systems*, volume 8304 of *LNCS*, pages 206–220. Springer, 2013. `doi:10.1007/978-3-319-03850-6_15`.

22 Yujie Liu and Michael F. Spear. Mounds: Array-based concurrent priority queues. In *Proc. 41st International Conference on Parallel Processing*, pages 1–10. IEEE Computer Society, 2012. `doi:10.1109/ICPP.2012.42`.

23 Peter S. Magnusson, Anders Landin, and Erik Hagersten. Queue locks on cache coherent multiprocessors. In *Proc. 8th International Symposium on Parallel Processing*, pages 165–171, 1994. `doi:10.1109/IPPS.1994.288305`.

24 Muralidhar Medidi and Narsingh Deo. Parallel min-max-pair heap. In *Proc. 13th IEEE International Phoenix Conference on Computers and Communications*, pages 322–328, 1994. `doi:10.1109/PCCC.1994.504133`.

25 John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991. `doi:10.1145/103727.103729`.

26 Mark Moir and Nir Shavit. Concurrent data structures. In Dinesh P. Mehta and Sartaj Sahni, editors, *Handbook of Data Structures and Applications*, chapter 48.8.1. Chapman and Hall/CRC, second edition, 2018. `doi:10.1201/9781315119335`.

27 Aravind Natarajan, Arunmoezhi Ramachandran, and Neeraj Mittal. FEAST: a lightweight lock-free concurrent binary search tree. *ACM Transactions on Parallel Computing*, 7(2):1–64, May 2020. `doi:10.1145/3391438`.

28 William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of ACM*, 33(6):668–676, 1990. `doi:10.1145/78973.78977`.

29 V. Nageshwara Rao and Vipin Kumar. Concurrent access of priority queues. *IEEE Trans. Computers*, 37(12):1657–1665, 1988. `doi:10.1109/12.9744`.

30 Adones Rukundo and Philippas Tsigas. TSLQueue: An efficient lock-free design for priority queues. In *Proc. 27th International Conference on Parallel and Distributed Computing*, volume 12820 of *LNCS*, pages 385–401. Springer, 2021. `doi:10.1007/978-3-030-85665-6_24`.

31 Sartaj Sahni. Double-ended priority queues. In Dinesh P. Mehta and Sartaj Sahni, editors, *Handbook of Data Structures and Applications*, chapter 8. Chapman and Hall/CRC, second edition, 2018. `doi:10.1201/9781315119335`.

32 Nir Shavit and Itay Lotan. Skiplist-based concurrent priority queues. In *Proc. 14th International Parallel & Distributed Processing Symposium*, pages 263–268. IEEE Computer Society, 2000. `doi:10.1109/IPDPS.2000.845994`.

33 Håkan Sundell and Philippas Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. *J. Parallel Distributed Comput.*, 65(5):609–627, 2005. `doi:10.1016/J.JPDC.2004.12.005`.

34 Orr Tamir, Adam Morrison, and Noam Rinetzky. A heap-based concurrent priority queue with mutable priorities for faster parallel algorithms. In *Proc. 19th International Conference on Principles of Distributed Systems*, volume 46 of *LIPIcs*, pages 15:1–15:16, 2015. `doi:10.4230/LIPICS.OPODIS.2015.15`.

35 Pen-Chung Yew, Nian-Feng Tzeng, and Duncan H. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Transactions on Computers*, 36(4):388–395, 1987. `doi:10.1109/TC.1987.1676921`.