Concurrent Priority Queue Using Lock-based Skiplist

Mingbin Xu

Department of Computer Science and Engineering, York University
4700 Keele Street, Toronto, Ontario, Canada, M3J 1P3

Abstract. This paper provides an approach of implementing concurrent priority queue using a lock-based skiplist. Skiplist is a probabilistic alternative to balanced trees. Its distributed nature makes it extremely suitable for concurrency. Previous works attempted to wrap a non-blocking concurrent skiplist into a priority queue. However, modification of non-blocking algorithms is generally complex. In this paper, we will review an optimistic lock-based concurrent skiplist implementation, adapt it for concurrent priority queue and evaluate its performance.

1 Introduction

The proliferation of multi-core computers have been pervasively affecting how we develop software. It poses increasingly more programming challenges. Data structures and algorithms, while classical, are no longer practical for parallel architectures. Priority queue is one of the fundamental data structures with rich applications, ranging from bandwidth management to discrete event emulation.

Many efforts have been devoted to handling overlapping queue operations in a concurrent setting. Known solutions include a tree-based structure by Shavit and Zemach [9], a heap-based structure by Hunt et al. [5] and recently a skiplist-based structure by Shavit and Lotan [8]. One drawback of the first two approaches is that the underlying data structures require complex rebalancing and centralized lockings, which may cause bottlenecks and contention. As shown in [8], skiplist outperforms other alternatives with relatively simpler implementation. However, the underlying skiplist algorithm credited to Pugh [7] is still considered complicated compared with the one proposed by Herlihy et al. [4] whose synchronization is optimistic and lazy. This paper aims to examine concurrent priority queues based on this simpler synchronization.

2 Priority Queue

A priority queue is a set of items where each item is assigned a priority indicating its importance, conventionally the smaller the better. Item priorities are not

necessarily distinct and may be time varying. Items with a higher priority are removed before those with a lower priority, irrespective of when they were added. A priority queue must at least support two operations: the *insert* operation to add an item to the set, and the *deleteMin* operation to remove and return the item with highest priority or smallest score. Heap structures, leftist heap [1] and skewed heap [10] for instance, are the most popular backbone for priority queues, giving $O(\log(n))$ time complexity for both queue operations. Variants such as binomial heap [11] and Fibonacci heap [2] perform faster on certain operations. Besides, priority queues can be modeled by simple linear data structures such as sorted arrays and sorted lists.

Listing 1: PriorityQueue Interface

```
1  public interface PriorityQueue<K extends Comparable<K>, V> {
2    /** Associates the specified item(v) with the specified
3      * priority(k) in this priority queue.
4      * @param   k   The priority associated with the new
5      *              item.
6      * @param   v   The value of the item to be stored.
7      * @return  The return is implementation-specific. If
8      *          duplicate is allowed, this method should
9      *          always return true; otherwise, when
10     *          duplicate is hit, it does NOT put the item
11     *          in the queue and returns false. */
12     public boolean insert(K k, V v);
13
14   /** Deletes and returns an item with highest priority.
15     * @return      An item with highest priority. */
16     public V deleteMin();
17 }
```

3 SkipList

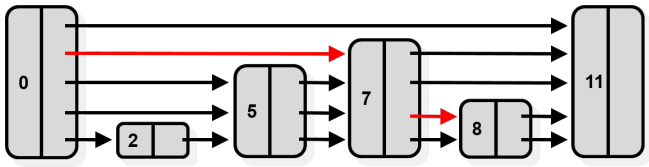


Fig. 1: This example is a skiplist of 6 items. The bottom list contains all items. The lists above are the sublists of the lists below them. In order to search for the item 8, traversal begins at the leftmost node of the topmost level. At each level, move rightwards until a key greater than or equal to what is searched for is hit. If such a key has been hit, go one layer down and repeat this process. As seen in this example, an "express" from 0 to 7 at the second topmost level skips 2 and 5.

Skiplist [6] is a probabilistic data structure that allows efficient search, insertion and removal. A skiplist maintains a collection of sorted linked hierarchy, which

mimics, with some randomization, a balanced search tree. Searching starts with the sparsest linked list in a top-down fashion, traversing until the largest item smaller than the sought is found at each level. Fast searching is made possible by skipping over a few elements at each level. Figure 1 shows a skiplist with integer keys. The higher-level lists serve as expresses to the lower-level lists. The bottom level is a regular sorted linked list with all items. Each item has a probability of 0.5 to be chosen in the level above. More formally, an item is present at level i with probability 0.5^i. For example, the nodes at level 0 have a chance of 0.5 to appear at level 1, and each node at level 1 will be promoted to level 2 with probability 0.5^2. Thus, roughly speaking, level i contains $n * 0.5^i$ items, and each link at level i skips 2^i items, where n is the number of items in the list.

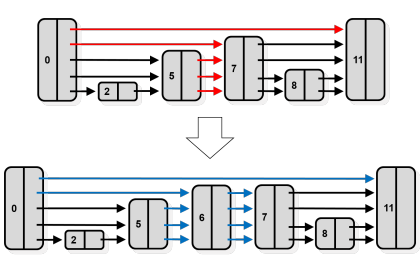
The structures of a skiplist and a skiplist node are detailed in Figure 3 (UML) and Listing 6 (Appendix). It is convenient to add two sentinels *__head* and *__tail* with priorities $-\infty$ and $+\infty$ respectively. Each skiplist node, be it sequential or concurrent, stores an array of skiplist node references which are its successors at all levels. Listing 2 shows the *__getPredecessorsAndSuccessors* method. It traverses the skiplist with *predecessors* and *successors* arrays starting from *__head* at the highest level. Given *target* to be searched, the *__getPredecessorsAndSuccessors* method descends from one level to another (Line 15-26). At each level it continuously sets *current* to be the successor of *predecessor* until it encounters a node whose priority is larger than the one sought (Line 17-20). If it locates a node with a matching priority, it records the height of the node in *levelFound* (Line 21-23). Before going one level down, *__getPredecessorsAndSuccessors* also puts *predecessor* and *successor* in the arrays of *predecessors* and *successors* (Line 24-25). If it finds *target*, *predecessor* itself is *target*; otherwise it is the largest node that is smaller than *target*. Figure 1 demonstrates the process of *search*. When performing *insert* and *remove*, one could keep track of the predecessors and the successors of the search path and then re-connect the predecessors and the successors, as depicted in Figure 2.

It is straightforward to adapt skiplist to implement a priority queue of items tagged with priorities. Skiplist is a collection of ordered items, ensuring that high-priority items appear at the front of the list. The insertion of a skiplist-based priority queue therefore reuses the interface from a skiplist without any modification. Since the bottom level contains all items, the very next node to which *__head* points, if not *__tail*, is the smallest.

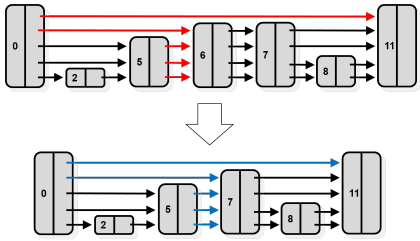
Listing 2: *__getPredecessorsAndSuccessors* of *LazySkipListPriorityQueue* and *CoarseSkipListPriorityQueue*

```
1  /** @param  target          The node to be sought.
2    * @param  predecessors    Used to retrieved the largest nodes at each
3    *                         level NOT greater than target.
4    * @param  successors      Used to retrieved the smallest nodes at each
5    *                         level greater than target.
6    * @return It guarantees to return -1 if invoked by insert() because
7    *         Node class imposes uniqueness; between 0 and 32 (inclusive)
8    *         if invoked by deleteMin() because deleteMin() removes the
9    *         very node which must exist. */
10 private int __getPredecessorsAndSuccessors (
```

```
11          Node<K, V> target, Node<K, V>[] predecessors,
12          Node<K, V>[] successors ) {
13      int levelFound = -1;
14      Node<K, V> predecessor = __head;
15      for ( int i = MAX_LEVEL; i >= 0; i-- ) {
16          Node<K, V> current = predecessor.__next[i];
17          while ( current != __tail && target.compareTo(current) > 0 ) {
18              predecessor = current;
19              current = predecessor.__next[i];
20          }
21          if ( levelFound == -1 && target.compareTo(current) == 0 ) {
22              levelFound = i;
23          }
24          predecessors[i] = predecessor;
25          successors[i] = current;
26      }
27      return levelFound;
28  }   // end of __getPredecessorsAndSuccessors
```



(a) Suppose that 6 is going to be inserted. The search algorithm gives a search path highlighted in red. The starting/finishing points of the red arrows are the predecessors/successors of the new item. Insert the new node in-between.



(b) Suppose that 6 is going to be removed. Similarly, the search algorithm returns a group of predecessors (starting point of the red arrows). Relink these predecessors' successors to 6's successors (the finishing points of outgoing arrows from 6).

Fig. 2: Insertion and Removal

4 Concurrent Skiplist-Based Priority Queue and Implementation

We now turn our attention to skiplist's concurrent implementation in Java. The same trick from [3] is applied to skiplist, that is, the *deleteMin* operation is done lazily with two steps: first label the target node, removing it logically, and second, reconnect its predecessors' next fields, removing it physically. Each node contains two *Boolean* fields, $\_\_logicallyDeleted$ and $\_\_fullyConnected$. The former is to indicate that the node has been logically deleted and is ready to be physically deleted. The latter is to notify that a node is being inserted but not

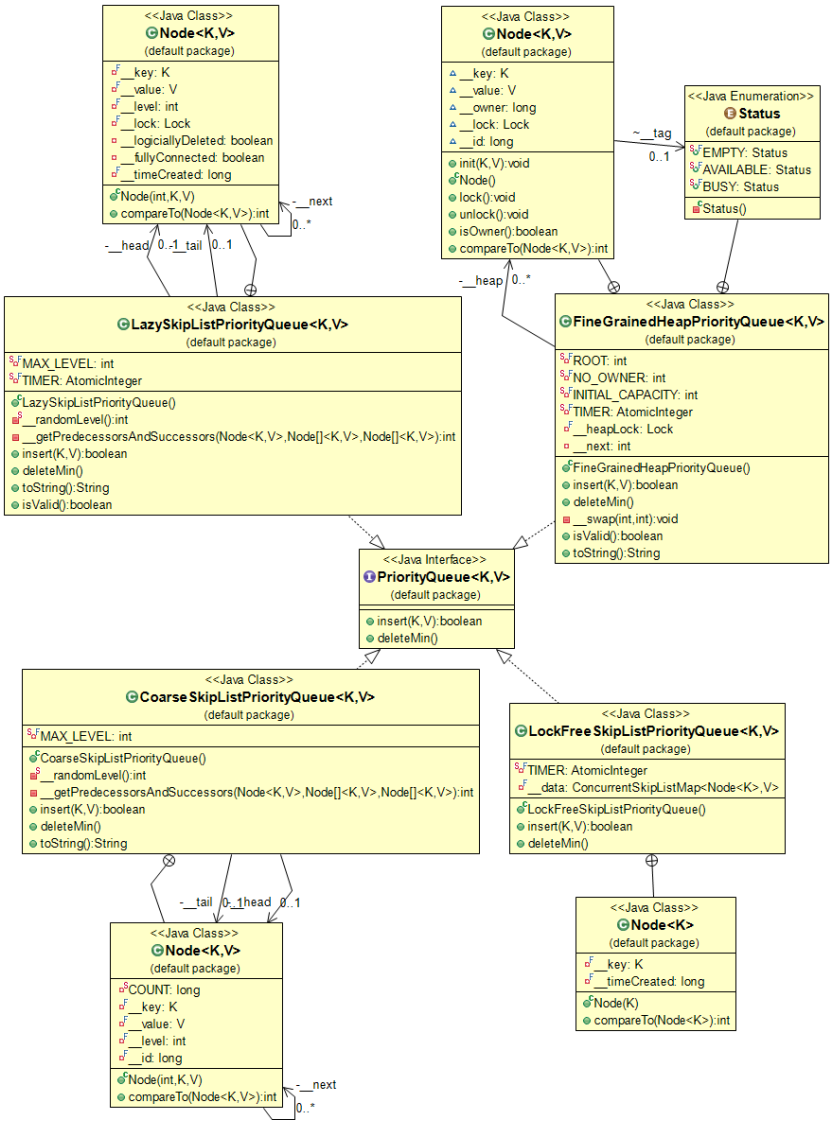


Fig. 3: Class Relationship in UML. The Java Implementations of *LazySkipListPriorityQueue* is given in the Appendix.

yet well-connected, and therefore is not considered in the list. The node to which *__head* points is not necessarily smallest since it might not be *fullyConnceted* or be marked as *logicallyDeleted*.

The algorithm is optimistic. Similar to [3], searching for the insertion spot or the victim to be removed does not acquire locks. During *insert* and *deleteMin*, an item is spliced in or out only when its predecessors are fully connected and not logically deleted, and their successors remain unchanged; it retries otherwise. Predecessors are retrieved by a top-down search. However, *deleteMin* scans the bottom level from the list head. If duplicated priority is allowed, the former finds the highest match. Contrarily, the latter stops as soon as it hits the first valid match. They are not necessarily the same. Uniqueness is enforced by assigning a creation time to each node upon its construction so as to avoid inconsistent search.

The *insert* method, shown in Listing 3, initializes the *predecessors* and the *successor* arrays before calling *__getPredecessorsAndSuccessors* (Line 5-6). *__getPredecessorsAndSuccessors* determines the predecessors and the successors of a new node and put them into the said arrays respectively. These two arrays are not reliable because they may not be accurate when the nodes are accessed. The thread executing *insert* acquires locks in descending priority (bottom-up) and validates each of the predecessors (Line 18-30). The validation at each level checks whether *predecessor* is still adjacent to *sucessor* and neither is marked as *logicallyDeleted*. If validation fails, there must be some contention introduced by other threads. The current thread releases the locks it holds and simply retries. If the current thread manage to locks all the predecessors, it splices in *target* between *predecessors* and *successors* (Line 35-38). When all re-connections are done, the field *__fullyConnected* of *target* is set to true, which indicates that *target* is inserted in the queue (Line 41).

Listing 3: *insert* of *LazyConcurrentSkipListPriorityQueue*

```
1  /** {@inheritDoc} */
2  public boolean insert(K key, V value) {
3      int newHeight = __randomLevel();
4      Node<K, V> target = new Node<K, V>(newHeight, key, value);
5      Node<K, V>[] predecessors = (Node<K, V>[]) new Node[MAX_LEVEL + 1];
6      Node<K, V>[] successors = (Node<K, V>[]) new Node[MAX_LEVEL + 1];
7
8      while ( true ) {
9          int levelFound = __getPredecessorsAndSuccessors(
10                                 target, predecessors, successors );
11         assert levelFound == -1;// Uniqueness is enforced by Node class.
12
13         // keep track of the lock height, so as to know where to unlock
14         int lockHeight = -1;
15
16         try {
17             boolean valid = true;
18             Node<K, V> predecessor, successor;
19             for ( int i = 0; valid && i <= newHeight; i++ ) {
20                 predecessor = predecessors[i];
21                 successor = successors[i];
22                 predecessor.__lock.lock();
23                 lockHeight = i;
```

```
24
25                  // 1. If predecessor.__logicallyDeleted, predecessor
26                  //    is being removed.
27                  // 2. If successor.__logicallyDeleted, successor is
28                  //    being removed.
29                  // 3. If predecessor.__next[i] != successor, sth has
30                  //    been inserted in-between
31                  // In either case, the predecessor or successor is invalid.
32                  valid = !predecessor.__logicallyDeleted &&
33                          !successor.__logicallyDeleted &&
34                          predecessor.__next[i] == successor;
35              }
36              if ( !valid ) {
37                  continue;
38              }
39
40              for ( int i = 0; i <= newHeight; i++ ) {
41                  target.__next[i] = successors[i];
42                  predecessors[i].__next[i] = target;
43              }
44
45              // linearization point, the node is considered "inserted".
46              target.__fullyConnected = true;
47              return true;
48          }   // end of try
49          finally {
50              for ( int i = 0; i <= lockHeight; i++ ) {
51                  predecessors[i].__lock.unlock();
52              }
53          }   // end of finally
54      }   // end of while ( true )
55  }   // end of insert
```

One can disallow *deleteMin* to remove a node whose creation time is earlier than the function invocation if the FIFO (first in first out) convention of a queue needs to be preserved. More formally, let dm be a *deleteMin* operation, Q be the item set whose elements invoke *insert* before dm, D be the item set that is removed by other *deleteMin* operations that precede or are concurrent with dm, and I be the item set that is concurrently inserted during dm's execution. If timestamps are preserved, dm returns the minimum from $Q - D$; $Q + I - D$ otherwise.

The *deleteMin* method appears in Listing 4. At the beginning (Line 3), it records the invocation time, nodes of creation time greater than which will not be considered. Similarly two arrays are initialized to retrieve the predecessors and the successors (Line 5-6). *deleteMin* scans the bottom list to get the smallest item (Lines 11-24). If *current* is ready to be deleted, *deleteMin* acquires lock, sets the field *__logicallyDeleted* to true (Line 14-20); otherwise *current* becomes the predecessor of *current* (Line 22-23). Note that new arrivals may be inserted before *current*, but their creation time cannot be earlier than *deleteMin*'s invocation time. If *__tail* is encountered, the queue is empty before the start of *deleteMin*, though it might not be the case during the execution of *deleteMin*. *__getPredecessorsAndSuccessors* is called to retrieve *target*'s ostensible predecessors and successors. In order to avoid deadlock, locks are acquired and released in the same manner as *insert* (bottom-up, Line 38-44, 54-58). *predecessor* is ver-

ified as valid when it points to *target* and is not logically deleted (Line 42-43). Once all levels are locked and valid, *target* is spliced out (Line 45-50).

A small modification is made in Listing 5 where time restriction on *deleteMin* is not imposed. When scanning the bottom list, the *while* loop is broken by *timeInvoked* in Line 13 Listing 4 but is not in Line 8 Listing 5. Another change is in Line 17 Listing 5 where search is retried from the head of the list. We will see in the experiment section that this impacts the throughput.

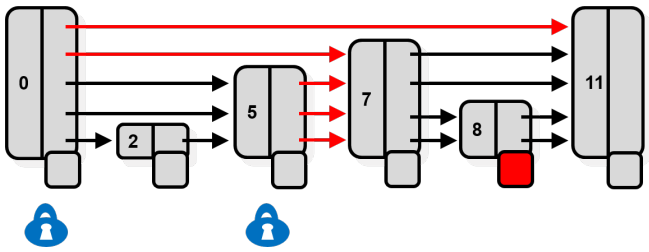
Listing 4: *deleteMin* of *LazyConcurrentSkipListPriorityQueue*

```
1  /** {@inheritDoc} */
2  public Node<K,V> __deleteMin() {
3      long timeInvoked = TIMER.get();
4      int targetHeight = -1;
5      Node<K, V>[] predecessors = (Node<K, V>[]) new Node[MAX_LEVEL + 1];
6      Node<K, V>[] successors = (Node<K, V>[]) new Node[MAX_LEVEL + 1];
7      Node<K, V> predecessor, current, target = null;
8
9      predecessor = __head;
10     current = __head.__next[0];
11     while ( current != __tail ) {
12         if ( current.__fullyConnected && !current.__logicallyDeleted
13                 && current.__timeCreated > timeInvoked ) {
14             current.__lock.lock();
15             if ( !current.__logicallyDeleted ) {
16                 current.__logicallyDeleted = false;
17                 current.__lock.unlock();
18                 break;
19             }
20             current.__lock.unlock();
21         }
22         predecessor = current;
23         current = predecessor.__next[0];
24     }
25
26     if ( current != __tail ) {
27         while ( true ) {
28             int levelFound =
29             __getPredecessorsAndSuccessors(
30                             current, predecessors, successors );
31             target = successors[levelFound];
32             assert (  levelFound != -1 && current == target );
33
34             target.__lock.lock();
35             int lockHeight = -1;
36             try {
37                 boolean valid = true;
38                 for ( int i = 0; valid && i <= targetHeight; i++ ) {
39                     predecessor = predecessors[i];
40                     predecessor.__lock.lock();
41                     lockHeight = i;
42                     valid = !predecessor.__logicallyDeleted &&
43                                 predecessor.__next[i] == target;
44                 }
45                 if ( !valid ) {
46                     continue;
47                 }
48                 for ( int i = 0; i <= targetHeight; i++ ) {
49                     predecessors[i].__next[i] = target.__next[i];
50                 }
51                 target.__lock.unlock();
52                 return target;
53             }
54             finally {
```

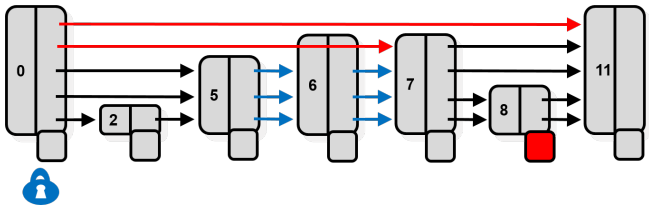
```
55                  for ( int i = 0; i <= lockHeight; i++ ) {
56                      predecessors[i].__lock.unlock();
57                  }
58              }
59          }   // end of while ( true )
60      }   // end of if ( current != __tail )
61
62      return null;
63  }   // end of deleteMin
```

Listing 5: *deleteMin* of *LazyConcurrentSkipListPriorityQueue*

```
1   /** {@inheritDoc} */
2   public Node<K,V> __deleteMin() {
3       // see listing 4
4       // ......
5       predecessor = __head;
6       current = __head.__next[0];
7       while ( current != __tail ) {
8           if ( current.__fullyConnected && !current.__logicallyDeleted ) {
9               current.__lock.lock();
10              if ( !current.__logicallyDeleted ) {
11                  current.__logicallyDeleted = false;
12                  current.__lock.unlock();
13                  break;
14              }
15              else {
16                  current.__lock.unlock();
17                  current = __head;
18              }
19          }
20          predecessor = current;
21          current = predecessor.__next[0];
22      }
23      // ......
24      // see listing 4
25  }
```

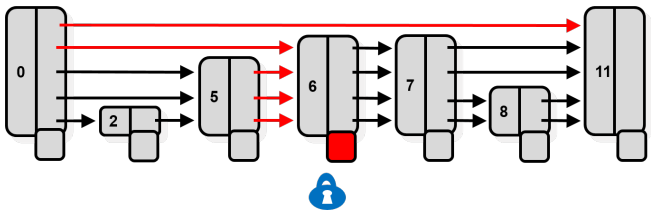


(a) Suppose that 6 is going to be inserted. Step 1: Identical to a sequential skiplist, the search algorithm returns the predecessors and the successors of the new node, which is indicated by the red arrows. Step 2: From right to left, each predecessor, if valid (fully connected, not logically deleted, and still pointing to the successor returned by the search algorithm), is locked; otherwise, insertion unlocks these predecessors and retries from Step 1.

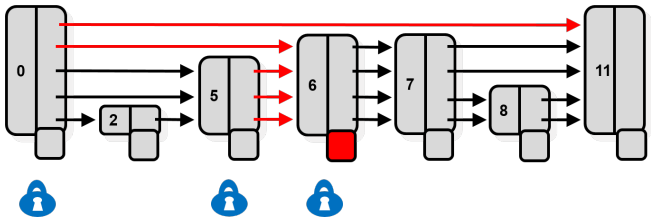


(b) Step 3: Splice in the new node from right to left. As depicted here, 6 has been linked between 5 and 7 in Level 0, Level 1 and Level 2. 0 still points to 7. 6 is not fully connected, and thus is not considered in the skiplist.

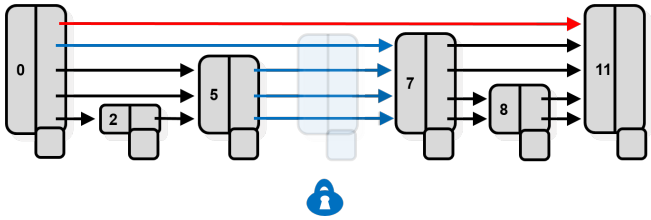
Fig. 4: Concurrent *insert*



(a) Suppose 0, 2 and 5 were inserted after *deleteMin*'s invocation and 6 was inserted before *deleteMin*. Step 1: Scan the bottom list from left to right until the first node inserted earlier than *deleteMin*'s invocation and not logically deleted is found. In this example, 6 hasn't been logically deleted. *__remove* sets it as logically deleted (red) and locks it.



(b) Step 2: from right to left, each predecessor, if valid (not logically deleted and still pointing to the successor returned by the search algorithm), is locked; otherwise, *deleteMin* unlocks those predecessors and retries from Step 1.



(c) Step 3: from right to left, link each predecessor to 6's successor at the same level. Unlock the predecessors when all done.

Fig. 5: Concurrent *deleteMin*

5 Experiment

5.1 Experiment Setup

The proposed algorithm is implemented as a class *LazySkipListPriorityQueue* in the Java programming language. Its performance is evaluated by comparing with a priority queue wrapping *ConcurrentSkipListMap* class from the *java.util.concurrent* package[1]. In addition, we also provide a baseline implementation in which methods are *synchronized* to guarantee thread-safety and a dummy priority queue whose methods are *synchronized* but without any actual implementation. Since *deleteMin* introduces heavy contention on the head of a list, we run the algorithms with insert-to-deleteMin ratio of 5:5, 7:3 and 9:1 in the priority range from 0 up to 100, 1000 and 10000, and examine how *deleteMin* affects performance. All experiments starts with empty data structures with priorities selected from a uniform distribution. An operation is decided randomly with bias according to the insert-to-deleteMin ratio. The experiments are conducted in the Manycore Testing Lab (MTL)[2]. It is a remote system equipped with 256GB memory and 4 CPUs of Intel® Xeon® E7-4860 @ 2.27GHz. In our experiments, we constrained the CPU usage on the first 16 logical processing units.[3] and parallel executions up to 32 threads.

[1] We also implemented and compared with heap-based priority queue credited to Hunt et al. [5]. However, as concurrency rose, a single operation sometimes did not finish within 1 second. It failed to provide a valuable reference. Moreover, it became slower than a sequential heap when the thread number exceeded 3.

[2] Intel Corporation has set up a special remote system that allows faculty and students to work with computers with lots of cores, called the Manycore Testing Lab. Users intentionally write programs that take advantage of multi-core parallelism and explore the issues in parallelism and concurrency that arise.

[3] Intel® Xeon® E7-4860 each has 10 cores and meanwhile supports hyper-threading. The system therefore is of 80 logical processing units.

5.2 Results & Analysis

Provided that the monitor baseline and the lock-free wrapper do not take into account the temporal relationship between *insert* and *deleteMin*, we would like to investigate the throughput with and without the timestamps assigned to an item and the invocation of *deleteMin* to ensure fairness. Results are shown in Figure 2-4. Graphs in the right columns depict the throughput of various algorithms and the left columns detail the number for retrying and the ratio of the accumulative numbers of nodes scanned at the bottom list by *deleteMin* to the number of *deleteMin*. The following conclusions have been reached[4]:

- The *deleteMin* operation brings in a great deal of contention. As the ratio of *insert* grows, contention is alleviated and thus the throughput increases. It conforms to our expectation that *deleteMin* is centralized around the list head while *insert* is well distributed.
- Overhead is primarily in the form of locking. Retrying, though important to ensure correct concurrency, is a rare case which introduces little overhead. As we can see in the left columns, retrying is outnumbered several thousands to one by operation count.
- Priority range does not give rise to contention, because uniqueness is enforced on the *Node* class by an atomic counter. Experiments are divided into three groups in the right columns. They behave similarly in all three different ratio settings.
- If *deleteMin* is not allowed to remove items whose creation time is earlier than the invocation of *deleteMin*, *deleteMin* scans a significantly longer path possibly leading to the tail of the list. At a high concurrent level, context switch is likely to take place right after *deleteMin*'s invocation and many new arrivals jump the queue. However, these new arrivals are close to the head, thus does not result in a lengthy search path. It explains why *LazySkipListPriority* with timestamps is of lower throughput.
- *LazySkipListPriorityQueue* yields comparable but more predictable result to that of the lock-free wrapper. There is no obvious winner when *deleteMin* dominates (5:5 ratio). Both lock-free and lock-based implementation are close to monitor in such situation.

6 Conclusion

This paper explores a friendly implementation of concurrent priority queue. *search*, *insert* and *remove* of concurrent skiplist, in common usage, spread all

[4] The fluctuation of the lock-free wrapper is so far not explainable. It might be a hardware problem. A reasonable guess of monitor's performance gain is that the optimizer moves some local statements out of the synchronized block.

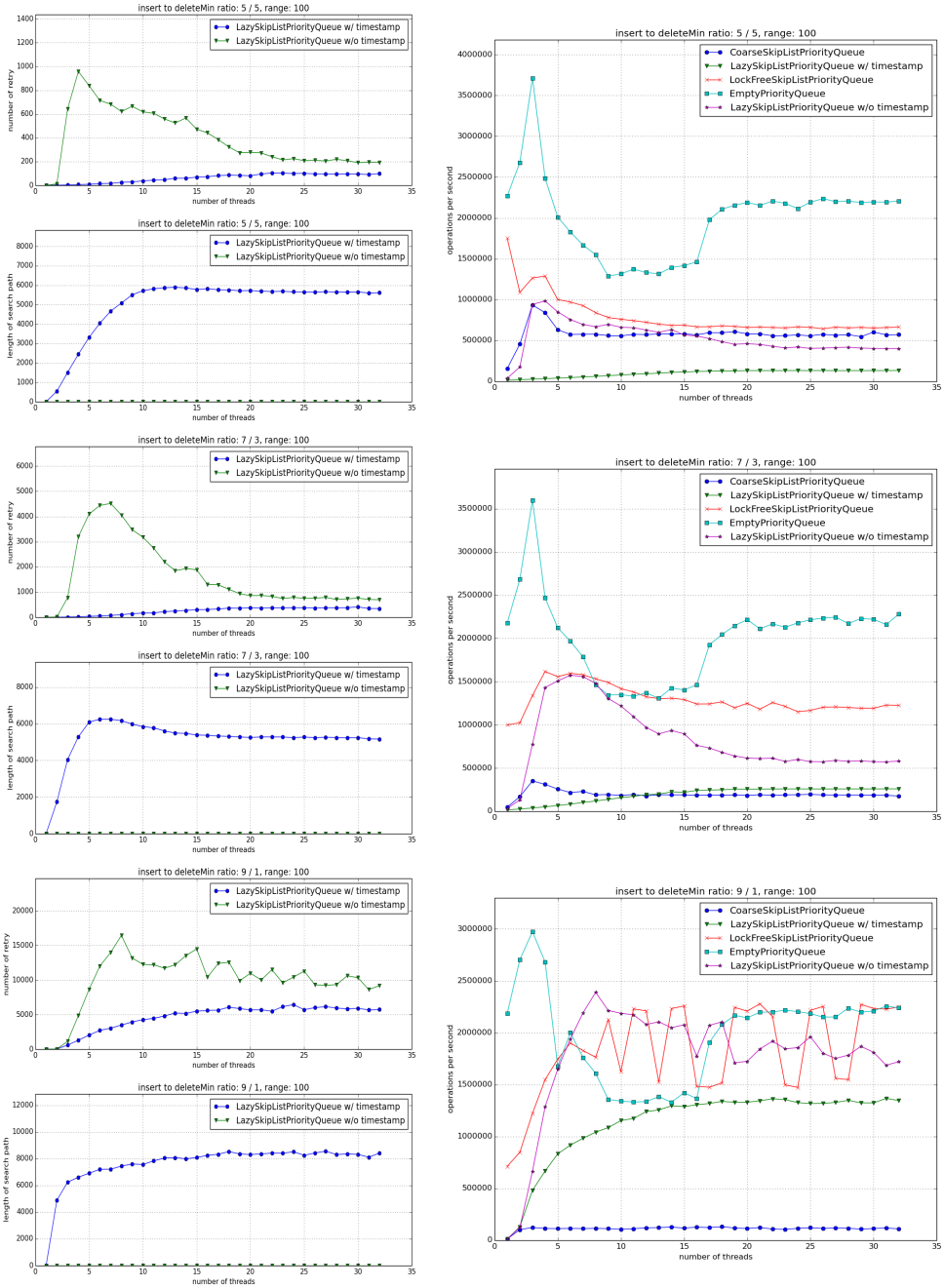


Fig. 6: Throughput within 1 second in a range of 100 and the corresponding number of retrying and the number of nodes scanned by *deleteMin* in the bottom list

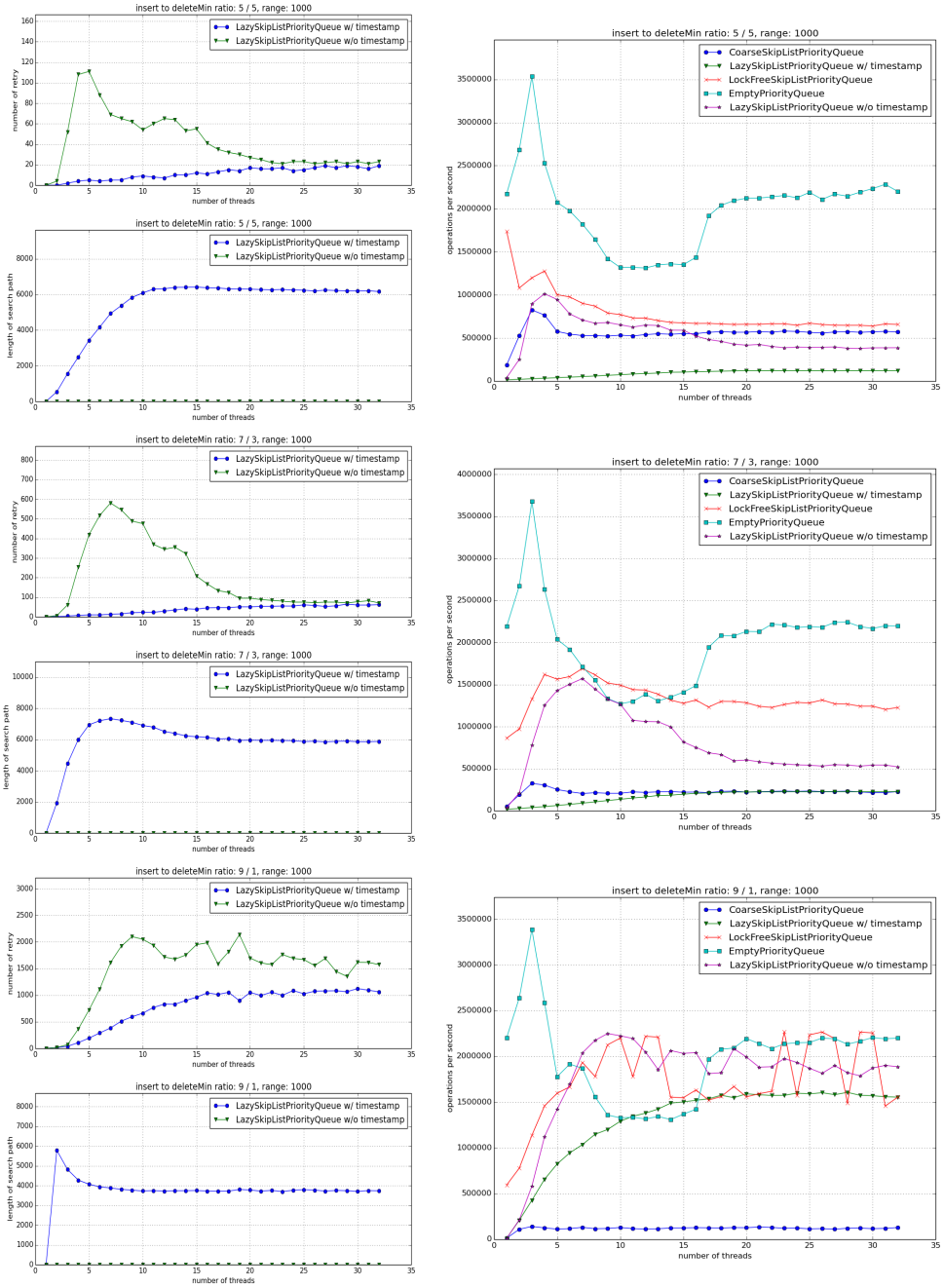


Fig. 7: Throughput within 1 second in a range of 1000 and the corresponding number of retrying and the number of nodes scanned by *deleteMin* in the bottom list

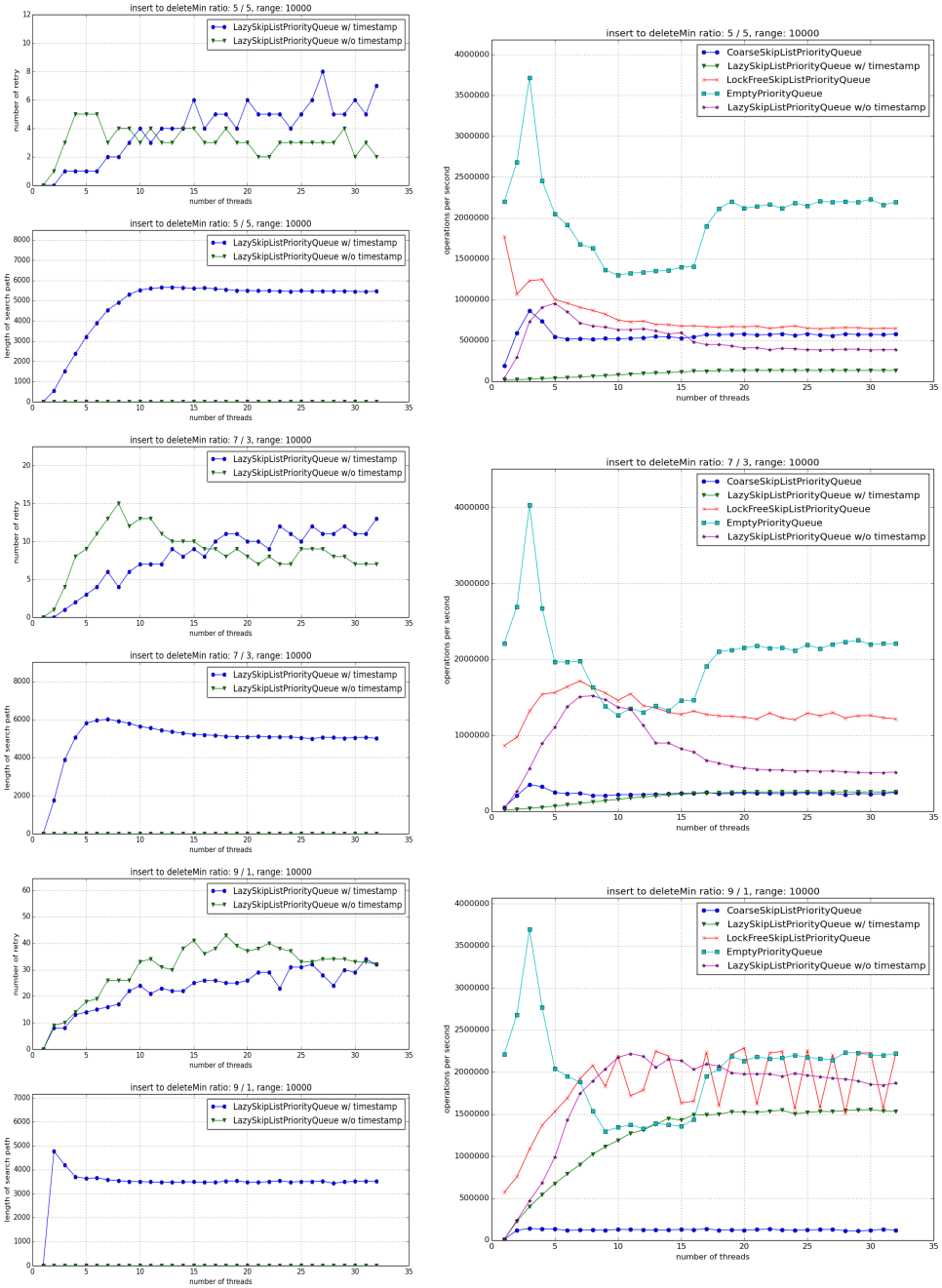


Fig. 8: Throughput within 1 second in a range of 10000 and the corresponding number of retrying and the number of nodes scanned by *deleteMin* in the bottom list

along the list. Therefore they gain performance as a matter of course. In contrast, a skiplist-based priority queue centralizes its removal on the very first item, which is heavily contentious. If $deleteMin$ is not allowed to remove items whose creation time is earlier than the invocation of $deleteMin$, $deleteMin$ scans a seriously longer path. Time is spent on an actual traversal instead of concurrent overhead. Both implementations do not benefit much from multi-threads when $insert$ and $deleteMin$ are equally mixed.

Acknowledgement

We thank the team at Intel's Multicore Testing Lab for providing us access to their machine.

References

1. Clark Allan Crane, *Linear lists and priority queues as balanced binary trees*, (1972).
2. Michael L Fredman and Robert Endre Tarjan, *Fibonacci heaps and their uses in improved network optimization algorithms*, Journal of the ACM (JACM) **34** (1987), no. 3, 596–615.
3. Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N Scherer III, and Nir Shavit, *A lazy concurrent list-based set algorithm*, Principles of Distributed Systems, Springer, 2006, pp. 3–16.
4. Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit, *A simple optimistic skiplist algorithm*, Structural Information and Communication Complexity, Springer, 2007, pp. 124–138.
5. Galen C Hunt, Maged M Michael, Srinivasan Parthasarathy, and Michael L Scott, *An efficient algorithm for concurrent priority queue heaps*, Information Processing Letters **60** (1996), no. 3, 151–157.
6. William Pugh, *Skip lists: a probabilistic alternative to balanced trees*, Communications of the ACM **33** (1990), no. 6, 668–676.
7. ———, *Concurrent maintenance of skip lists*, (1998).
8. Nir Shavit and Itay Lotan, *Skiplist-based concurrent priority queues*, Parallel and Distributed Processing Symposium, IEEE, 2000, pp. 263–268.
9. Nir Shavit and Asaph Zemach, *Diffracting trees*, ACM Transactions on Computer Systems (TOCS) **14** (1996), no. 4, 385–428.
10. Daniel Dominic Sleator and Robert Endre Tarjan, *Self-adjusting heaps*, SIAM Journal on Computing **15** (1986), no. 1, 52–69.
11. Jean Vuillemin, *A data structure for manipulating priority queues*, Communications of the ACM **21** (1978), no. 4, 309–315.

A Java Implementation of Data Structure Discussed

Listing 6: LazyConcurrentSkipListPriorityQueue

```
1  import java.io.*;
2  import java.util.concurrent.locks.*;
3  import java.util.concurrent.atomic.*;
4  import java.util.*;
5
6  /**
7  * A class which implements PriorityQueue interface and whose underlying
8  * data structure is SkipList. This implementation allows duplicated
9  * priority.
10 * @param <K>   The type of priority that is associated to an item.
11 * @param <V>   The type of item stored in the priority queue.
12 * */
13 public class LazySkipListPriorityQueue <K extends Comparable<K>, V>
14 implements PriorityQueue<K, V> {
15
16     /*
17     * Each node is associated with a timestamp. The reason is as follow:
18     * In the first stage of deleteMin(), it begins traversal at the
19     * bottom; therefore it finds the very FIRST match. In the second
20     * stage of deleteMin(), it starts traversal at the top; therefore it
21     * finds the HIGHEST match, which is NOT necessary. the first match.
22     * By adding a sorted timestamp, uniqueness is enforced, i.e., the
23     * underlying data structure has no duplicate.
24     * hashCode() though enforces uniqueness, sort order is not guaranteed.
25     * */
26     final private static class Node<K, V> implements Comparable<Node<K, V>> {
27
28         final private K __key;              // the client-provided priority
29         final private V __value;
30         final private int __level;          // the height of the node
31         final private Node<K,V>[] __next;
32         final private Lock __lock;
33
34         // if true, the node is not considered existed
35         // false by default; once changed to true by deleteMin,
36         // it cannot become false again
37         volatile private boolean __logiciallyDeleted;
38
39         // if true, the node is considered in the list
40         // false by default, once changed to true by insert,
41         // it cannot become false again
42         volatile private boolean __fullyConnected;
43
44         // linearized point at which method "takes effect"
45         final private long __timeCreated;   // also serves as a unique id
46
47         @SuppressWarnings("unchecked")
48         /**
49         * Create a node whose height is level, priority is key and item
50         * stored is value
51         * @param level     Height of the node.
52         * @param key       Priority.
53         * @param value     Item to be stored.
54         * */
55         public Node( int level, K key, V value ) {
56             __level = level;
57             __key = key;
58             __value = value;
59             __timeCreated = TIMER.getAndIncrement();
60             __logiciallyDeleted = false;
61             __next = (Node<K, V>[]) new Node[level + 1];
62             __lock = new ReentrantLock();
```

```
63              __fullyConnected = false;
64          }   // end of Node
65
66
67          @Override
68          /** {@inheritDoc} */
69          public int compareTo(Node<K, V> o) {
70              // only the keys of __head and __tail are null
71              // only __tail could be passed to this function
72              // __tail is a sentinel node which is always the largest
73              if ( o.__key == null ) {
74                  return -1;
75              }
76
77              @SuppressWarnings("unchecked")
78              int result = ((Comparable<K>)__key).compareTo(o.__key);
79              if ( result == 0 ){
80                  if ( __timeCreated < o.__timeCreated ) {
81                      return -1;
82                  }
83                  else if ( __timeCreated > o.__timeCreated ) {
84                      return 1;
85                  }
86                  else {
87                      return 0;
88                  }   // end of if ( __timeCreated )
89              }
90              else {
91                  return result;
92              }   // end of if ( result )
93          }   // end of compareTo
94      }   // end of class Node
95
96
97
98      // The list is sorted by a combination of (key, timeinserted).
99      final private static int MAX_LEVEL = 32;
100     final private static AtomicInteger TIMER = new AtomicInteger();
101
102     // 2 sentinel nodes. __head is "smallest". __tail is "largest".
103     final private Node<K, V> __head;
104     final private Node<K, V> __tail;
105
106
107     /**
108     * Default Constructor which creates an empty concurrent priority
109     * queue.
110     * */
111     public LazySkipListPriorityQueue() {
112         __head = new Node<K, V>(MAX_LEVEL, null, null);
113         __tail = new Node<K, V>(MAX_LEVEL, null, null);
114         for ( int i = 0; i < __head.__next.length; i++ ) {
115             __head.__next[i] = __tail;
116         }
117     }   // end of LazySkipList
118
119
120     /**
121     * Generates a random integer between 0 and 32, inclusive. The
122     * probability that i is returned is 0.5^i if i is not 32 and
123     * 0.5^31 if i is 32.
124     * @return  A random integer.
125     * */
126     private static int __randomLevel() {
127         int height = 0;
128         while ( height < MAX_LEVEL && Utility.randomBoolean() ) {
129             height++;
130         }
```

```
131         return height;
132     }   // end of randomHeight
133
134
135     /** Please look at Listing 2 */
136     private int __getPredecessorsAndSuccessors ( Node<K, V> target,
137     Node<K, V>[] predecessors, Node<K, V>[] successors ) {
138         // please look at Listing 2
139     }   // end of __getPredecessorsAndSuccessors
140
141
142     @SuppressWarnings("unchecked")
143     @Override
144     /** {@inheritDoc} */
145     public boolean insert(K key, V value) {
146         // Please look at Listing 3
147     }   // end of insert
148
149
150     @SuppressWarnings("unchecked")
151     @Override
152     /** {@inheritDoc} */
153     public V deleteMin() {
154         // Please look at Listing 4
155     }   // end of deleteMin
156
157
158
159     /**
160     * Return the string representation of the current object. This
161     * method is written for test purpose and is NOT thread-safe.
162     * @return  The string representation of the current object, of
163     *          which is a sorted sequence. */
164     public String toString() {
165         StringBuffer sb = new StringBuffer();
166         sb.append("[ ");
167         for ( Node<K, V> current = __head.__next[0];
168                 current != __tail; current = current.__next[0] ) {
169             sb.append('(').append(current.__key).append(", ")
170               .append(current.__timeCreated).append(", ")
171               .append(current.__value).append(") ");
172         }
173         sb.append(']');
174         return sb.toString();
175     }   // end of toString()
176
177
178
179     public boolean isValid() {
180         TreeSet<K> prevLevel = new TreeSet<K>();
181         TreeSet<K> curLevel = null;
182
183         for ( int i = 0; i <= MAX_LEVEL; i++ ) {
184             curLevel = new TreeSet<K>();
185             Node<K, V> prev = __head;
186             Node<K, V> cur = __head.__next[i];
187             boolean result = true;
188
189             while ( cur != __tail) {
190             if ( !cur.__fullyConnected || cur.__logiciallyDeleted ) {
191                 System.err.println(
192                     i + " Fail because of invalid boolean flags..." );
193                 result =  false;
194             }
195             if ( prev != __head && prev.compareTo(cur) >= 0 ) {
196                 System.err.println(
197                     i + " Fail because of sort order..." );
198                 result = false;
```

```
199             }
200             if ( !result ) {
201                 // this piece of code sees how sort order is violated
202                 // the console has only limited buffer. in order to
203                 // observe the result, we have to write it to a file
204                 File file = new File("./logging");
205                 BufferedWriter bw = null;
206                 try {
207                     if ( !file.exists() ) {
208                         file.createNewFile();
209                     }
210                     bw = new BufferedWriter(
211                             new FileWriter(file.getAbsoluteFile()));
212
213                     Node<K, V> previous = __head;
214                     Node<K, V> current = __head.__next[i];
215                     while ( current != __tail ) {
216                         bw.write( current.__key + "  " +
217                                 current.__timeCreated + "\n" );
218                         if ( previous != __head &&
219                                 previous.compareTo(current) >= 0 ) {
220                             System.err.println("violate skiplist property");
221                             bw.write("violate skiplist property\n");
222                         }
223                         previous = current;
224                         current = current.__next[i];
225                     }
226                     bw.close();
227                     System.exit(1);
228                 }
229                 catch (IOException e) {
230                     // let it throw...
231                 }
232                 return false;
233                 }   // end of if ( !result )
234
235                 curLevel.add( cur.__key );
236                 prev = cur;
237                 cur = cur.__next[i];
238             }   // end of while ( cur != __tail)
239
240             if ( i > 0 ) {
241                 // check if higher level is subset of the lower level
242                 if ( !prevLevel.containsAll( curLevel ) ) {
243                     System.err.println( i + " Fail because of xxoo" );
244                     return false;
245                 }
246             }
247             prevLevel = curLevel;
248         }   // end of for
249         return true;
250     }   // end of isValid
251
252 }   // end of LazySkipListPriorityQueue
```