Priority Queue Testing Report

Anita Zakrzewska
Borja Zarco
Marat Dukhan
Yi Wei Cheng

In this study we perform unit tests and performance tests on four known priority queue structures: linked list, splay tree (Sleator and Tarjan, 1985), sorted-discipline calendar queue (Brown, 1988) and the ladder queue (Tang et al, 2005). We first give a brief introduction on the respective queues (except the linked list) in Part 1. In Parts 2 and 3, we present the unit test and performance test results respectively.

1. Priority Queues

The splay tree is a form of binary search tree with the capacity to "self-adjust" data structure during each operation. The self-adjustment/splay operation follows a simple restructuring rule aiming at improving the efficiency of future operations. Theoretically the splay tree have an amortized time bound of O(log n) per hold operation. As such, the splay tree is as efficient as many existing balanced trees such as B-trees and height-balance trees. However, unlike balanced trees, the splay tree is able to maintain its efficiency even for long access sequences, due to the splay operation. During each splay operation, the element that is splayed is placed at the root of tree after restructuring the tree (i.e. tree rotation). As the splay operation is performed immediately after each operation (i.e. enqueue, dequeue), the most recently accessed element is always at the root of the tree, reducing the overheads incurred during searches.

The Sorted-discipline Calendar Queue (SCQ) and the ladder queue are hashing schemes. The SCQ was proposed by Brown in 1988. Heuristically, the data structure is similar to events in a desktop calendar. Arrays of "buckets" are defined, each bucket represents a single day in a year and contains events scheduled in that particular day. Events within the bucket are sorted according to priority (e.g. time). The SCQ achieves O(1) performance when number of elements in the queue, N, and mean timestamp increment, □, are constant (Ronngren and Ayani, 1997). However, when □ and N vary, O(1) performance is no longer achieved. For example, when □ is varied while N remains constant, only O(N) performance can be achieved. The lost in performance has been attributed to the ineffectiveness of the SCQ resize trigger.

The ladder queue is developed by Tang et al (2005) to overcome the problems experienced by SCQ. The ladder queue consist of three basic parts: (1) Top; which

contains an unsorted linked list, (2) Ladder; which contains serveral rungs of buckets, each containing an unsorted linked list, and (3) bottom; which contains a sorted linked list. To overcome the problems encountered by SCQ and maintain O(1) performance the ladder queue follow four principles: (1) Sorting only occurs when high priority events are close to being dequeued, (2) During enqueue operations, only when the Bottom structure becomes too populated, then does a resize operation take place, (3) Bucketwidth refinement during dequeue operations occur on a bucket-by-bucket basis, and (4) Ladder queue generates optimal operating parameters dynamically in accordance to events in its structure.

2. Unit Test Results

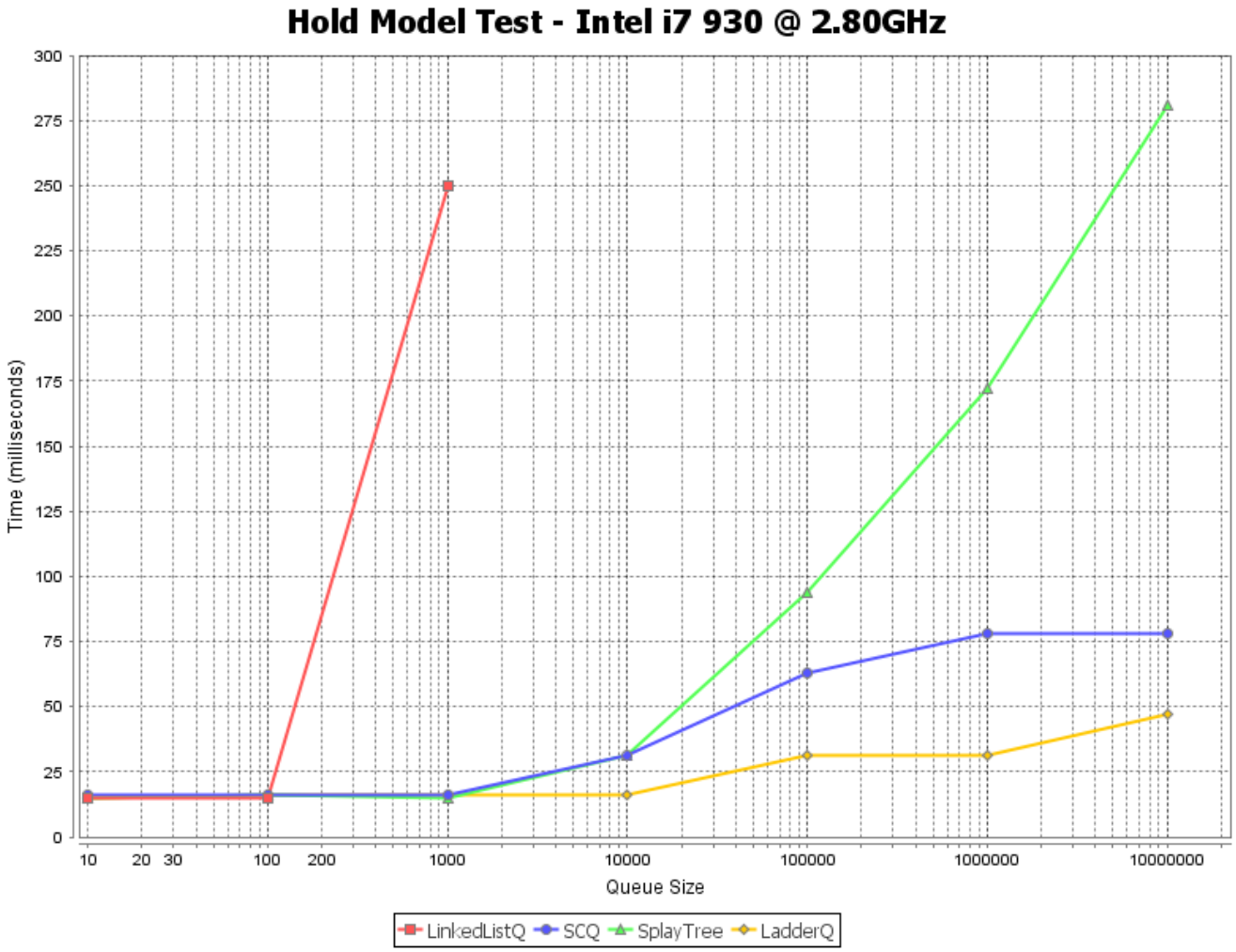
We utilize a model that consist of enqueue and dequeue operations to test the priority queues that we have coded. In this model, fifteen events are initialized with random priorities and indices. First, these fifteen events are enqueue into the priority queue under testing. Next, the events with the smallest priorities are sequentially dequeued until the priority queue becomes empty. If the priority queue is functioning correctly, the result will be a sorted sequence of events from low priority to high priority.

Unit Test Results			
LinkedListQ	**SCQ**	**SplayTree**	**LadderQ**
ID: 1 ; Priority: 3.75	ID: 1 ; Priority: 3.75	ID: 1 ; Priority: 3.75	ID: 1 ; Priority: 3.75
ID: 2 ; Priority: 6.56	ID: 2 ; Priority: 6.56	ID: 2 ; Priority: 6.56	ID: 2 ; Priority: 6.56
ID: 3 ; Priority: 6.78	ID: 3 ; Priority: 6.78	ID: 3 ; Priority: 6.78	ID: 3 ; Priority: 6.78
ID: 4 ; Priority: 8.4	ID: 4 ; Priority: 8.4	ID: 4 ; Priority: 8.4	ID: 4 ; Priority: 8.4
ID: 5 ; Priority: 8.4	ID: 5 ; Priority: 8.4	ID: 5 ; Priority: 8.4	ID: 5 ; Priority: 8.4
ID: 6 ; Priority: 11.35	ID: 6 ; Priority: 11.35	ID: 6 ; Priority: 11.35	ID: 6 ; Priority: 11.35
ID: 7 ; Priority: 12.35	ID: 7 ; Priority: 12.35	ID: 7 ; Priority: 12.35	ID: 7 ; Priority: 12.35
ID: 8 ; Priority: 12.35	ID: 8 ; Priority: 12.35	ID: 7 ; Priority: 12.35	ID: 8 ; Priority: 12.35
ID: 9 ; Priority: 12.35	ID: 9 ; Priority: 12.35	ID: 9 ; Priority: 12.35	ID: 9 ; Priority: 12.35
ID: 10 ; Priority: 12.45	ID: 10 ; Priority: 12.45	ID: 10 ; Priority: 12.45	ID: 10 ; Priority: 12.45
ID: 11 ; Priority: 13.35	ID: 11 ; Priority: 13.35	ID: 11 ; Priority: 13.35	ID: 11 ; Priority: 13.35
ID: 12 ; Priority: 35.3	ID: 12 ; Priority: 35.3	ID: 12 ; Priority: 35.3	ID: 12 ; Priority: 35.3
ID: 13 ; Priority: 65.35	ID: 13 ; Priority: 65.35	ID: 13 ; Priority: 65.35	ID: 13 ; Priority: 65.35
ID: 14 ; Priority: 85.2	ID: 14 ; Priority: 85.2	ID: 14 ; Priority: 85.2	ID: 14 ; Priority: 85.2
ID: 15 ; Priority: 114.67	ID: 15 ; Priority: 114.67	ID: 15 ; Priority: 114.67	ID: 15 ; Priority: 114.67
Is queue empty? true	Is queue empty? true	Is queue empty? true	Is queue empty? True

Results show that both the SCQ and the splay trees are able to perform the enqueue and dequeue operations well.

3. Performance Test Results

We utilize the hold model to measure the performance of four priority-queues: linkedlist, SCQ, splaytree and LadderQ. In the hold model, the enqueue operation schedules events into the priority queue. The dequeue operation searches for the event with the lowest/highest priority and removing it from the priority queue. In one iteration, the hold model dequeues an event, randomly alters the priority of the event and enqueue the event back into the priority queue. This process is iterated for N times, where N is the size of the priority queue. At the end of the simulation, average time for the operation is evaluted by normalizing total simulation time by N. We evaluate the average operation time for queue sizes ranging from 10 to 10000000 to obtain performance curve for each priority queue. We then compare the performance of the four priorities by plotting all the performance curve together.



As expected results show that the Linkedlist queue has the worst performance.

Performance time of the Linkedlist queue increases exponentially with size, and quickly goes beyond 300 ms when queue size becomes greater than 1000. As expected, the LadderQ has the best performance. This is consistent with results from Tang et al (2005), which showed that the LadderQ has better performance than both the splaytree and SCQ. Performance between splaytree and calendar queue is similar for queue sizes less than 10000. However, beyond 10000, simulation time of the splaytree increases at a much faster rate than the SCQ. At queue size 10000000, the performance time of the splaytree is >3X that of the SCQ and ~7X that of the LadderQ.

4. Conclusion

Base on the unit test and the performance results, we choose the the LadderQ as the main form of priority queue that we will implement in our traffic simulation model.

5. References

Brown, R. 1988. Calendar queues: A fast *O*(1) priority queue implementation for the simulation event set problem. *Commun. ACM 31*, 10 (Oct.), 1220–1227.

R¨onngren, R. and Ayani, R. 1997. A comparative study of parallel and sequential priority queuealgorithms. *ACM Trans. Model. Comput. Simul. 7*, 2 (Apr.) 157–209.

Sleator, D.D.. and Tarjan, R.E. Self-adjusting binary trees. In Proceedings of the ACM SIGACT Symposium on Theory of Computing (Boston, Mass., Apr. 25-27). ACM, New York, 1983, pp. 235-245.

Tang, W. T and Goh, R . S. M and Thng, I. L (2005). Ladder queue: An o(1) priority queue structure for large-scale discrete event simulation. ACM Transactions on Modeling and Computer Simulation. Vol 15, No. 3, 175–204.