Testing a High Performance, Random Access Priority Queue: A Case Study

James D. McCaffrey
Adrian Bonar
The Microsoft Corporation

Abstract

This paper presents a case study of the functional verification of a custom implementation of a random access priority queue which was optimized for performance. Although data structures have been used for decades few studies have examined the effectiveness of different testing strategies applied to complex data structures. In this study, four different testing approaches were used to test a priority queue. The results showed that a state transition testing approach (13 faults discovered) was clearly superior with regards to the number of faults found than the alternatives of a manual testing approach (3 faults discovered), a unit testing approach (4 faults discovered), and a classical test harness approach (6 faults discovered). Because the state transition testing approach used was in essence a modified form of random input testing, the results of this study suggest that the notion that random input testing is typically less effective than other forms of testing may be an overly broad generalization.

Keywords: Priority queue, software testing, state transition, test harness, unit testing.

1. Introduction

This paper presents a case study of the design and functional verification of a high performance, random access priority queue data structure. Although abstract data structures implemented in high level programming languages have been used for decades and many critical software systems depend upon the correctness of these data structures, there have been surprisingly few formal research studies performed which investigate the effectiveness of various testing techniques to verify the functionality of these data structures [1]. A review of the existing literature in the area suggests that most studies of the functional verification of data structure implementations are primarily feasibility studies or studies of the effectiveness of a single test strategy. A 2009 study by Deshmukh and Emeerson pointed out that testing data structures is not trivial and presents formidable challenges. That study presented a prototype test tool which had underlying functionality based on tree automata [2]. A 2006 study by Bousjjani, Habermehl, and Rogalewicz examined verification of dynamic linked data structures using shape graphs and abstract regular tree modeling [3]. A 2006 paper by Habermehlm, Iosif, and Vojnar presented a feasibility study of the verification of tree-like data structures using a classical semi-algorithmic state transition approach [4]. A 2006 study by Deshmukh, Emerson, and Gupta presented a technique based on automata theory and temporal logic to test programs which modify data structures such as linked lists and directed graphs [5].

To the best of our knowledge no previous research studies have explicitly examined the comparative effectiveness of different testing strategies for verifying the correctness of a non-trivial data structure. This case study emerged as part of a research project which analyzed shortest path metrics on very large (terabyte scale) graphs. That graph analysis project required the implementation of a custom, high performance, random access priority queue. Because each shortest path analysis could take hours or even days of processing time on a high performance computing cluster, it was important that the utility random access queue was functionally correct. Testing the correctness of the utility queue data structure was a challenging task and was approached using four different testing strategies. The results of this case study suggest that, in this one scenario at least, a state transition testing approach was clearly superior to the other three approaches.

2. Definitions and Design

An early design decision was made to implement the custom priority queue to support only a very specific shortest path scenario rather than to implement the queue in a generalized form which could be used in a variety of scenarios. Custom data structures can be implemented to emphasize performance (typically using an array approach and auxiliary lookup tables) or minimize memory usage (typically using a dynamic/pointer approach). A 2008 paper by Herbordt, Kosie, and Model presented a priority queue design which emphasized performance [6]. A 2008 paper by Dragicevic and Bauer surveyed the designs of custom priority queues used for concurrent algorithms [7]. The graph which is the target of the shortest path algorithm which uses the priority queue is assumed to be undirected and have vertices each of which has a unique, ordinal, 0-based vertex ID. Each edge between a pair of vertices in the graph is assumed to have a positive, integer-based distance value associated with the edge.

A generic priority queue abstract data type holds arbitrary items where each item has an associated priority value of some kind and supports at least three basic operations: create, enqueue, and dequeue. A generic priority queue maintains a state order invariant such that the item in the queue which has the highest priority value is always located at the virtual front of the queue, that is, the location in the queue where the next item will be removed. The create operation initializes the queue to an empty state. The enqueue operation adds an item to the queue in such a way that the queue maintains its state invariant. In general, priority queues allow items with identical priority values to exist in the queue at any given time. The dequeue operation removes the virtual front item from the queue, which by definition will be the item in the queue with the highest priority value.

A random access priority queue is a priority queue which supports two additional operations: remove and modify. The remove operation deletes a specified item from the queue, which may or may not be located at the virtual front position, and maintains the state order invariant. The modify operation changes one or more of the fields of a specified item in the queue, and maintains the state order invariant. Note that both the remove and modify operations impose the constraints that items contained in the queue must have some unique identification field and the existence of a location function which returns the position of a particular item within the queue. The random access priority queue tested in this study was designed to emphasize performance; the enqueue, dequeue, modify, and remove operations were designed to have O(lg n) performance and the location function was designed to have O(1) performance.

There are several ways commonly used to design a priority queue. One typical design choice is to use a binary heap in part because there is a very close relationship between the priority queue state order invariant and the heap state order invariant. A binary heap is similar to a binary tree structure and has two important characteristics. First, the heap structure is ordered with respect to some item key field in such a way that the key field of any node is less than the key fields of both of its child nodes, that is, the heap satisfies a state invariant such that for any node n in the heap, if node n is the parent of c then n.key >= c.key. In the case of the shortest path algorithm, the graph node distance field acts as the priority value. This means a node with the smallest distance value (where distance represents the distance from the algorithm start vertex to the reference node) will always be located at the root of the binary heap. A second binary heap property is that the tree structure is complete meaning that, visually, the tree is filled level-by-level, from left to right. This shape property allows a binary heap to be efficiently implemented using an array.

The following pseudocode illustrates the five operations supported by the random access priority queue:

```
(0) graphSize := 15;
(1) pq := NewQueue(graphSize);
(2) pq.Enqueue(NodeInfo(8, 110));
(3) pq.Enqueue(NodeInfo(2, 120));
(4) pq.Enqueue(NodeInfo(6, 100));
(5) if (pq.Contains(2) = true)
(6)   Print("Node with ID = 2 is in queue");
(7) pq.Modify(8, 90);
(8) pq.Dequeue();
(9) pq.Remove(2);
```

Because the priority queue does not support dynamic resizing, the size of the associated graph must be known before the queue is instantiated as shown in lines 0 and 1. Line 2 creates a node with ID = 8 and distance = 110 and adds it (as the root node) to the queue. Line 3 adds a node with ID = 2 and distance = 120. Because this node has greater distance (i.e., lower priority) than the previously added node, it is added below the existing root node. Line 4 adds another node but because it has a smaller distance the heap is modified so that this new node with distance 100 is now the root node. Line 5 and 6 illustrate the search function, which will take only O(1) time. Line 7 changes the distance value of the node with ID = 8 to 90. Because this is now the smallest distance, the heap is modified so that this node becomes the root node. Line 8 removes the root node. Line 9 removes the node with ID = 2.

3. Implementation

Figure 1 illustrates the implementation of the random access priority queue which was the target of the testing approaches examined in this case study. The top part of Figure 1 represents a binary heap associated with the shortest path algorithm for a graph which has 15 vertices, numbered from 0 through 14. The data in each node represent a graph vertex ID and the distance from the shortest path start vertex to the node vertex. So, the root node values mean that graph vertex with ID = 6 has distance = 100 to the start vertex. Note that not every vertex in the graph is necessarily represented in the binary heap.

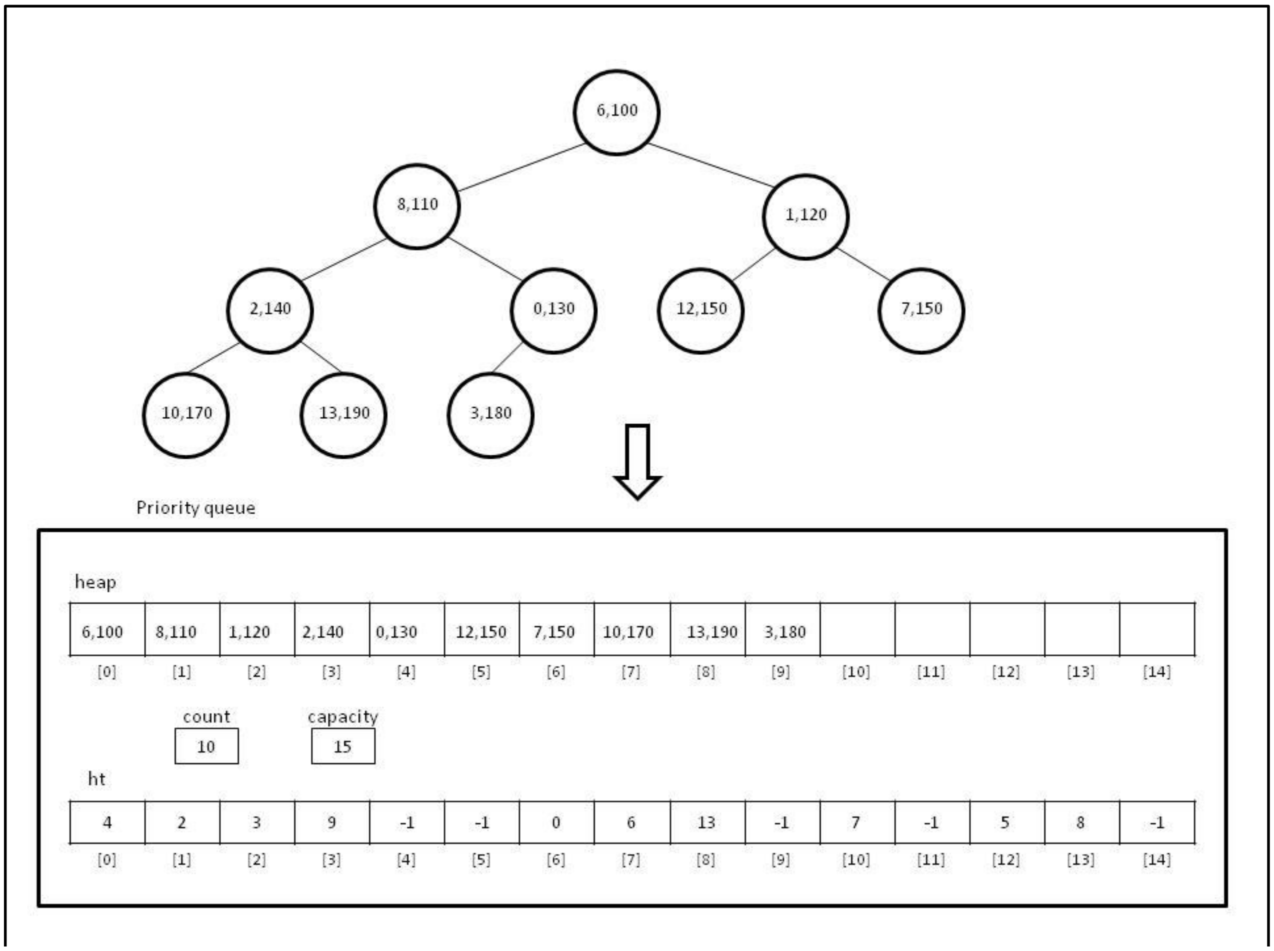

Figure 1. Random access priority queue.

The bottom part of Figure 1 represents the implementation of a priority queue which corresponds to the binary heap. Node values are stored in an array named heap which has size equal to the number of vertices in the associated heap. Notice that for any node located at index i in array heap, the index of the left child is given by i * 2 + 1, and that the index of the right child is given by i * 2 + 2, and that the index of the parent is given by (i – 1) / 2. In order to support an O(1) location function an auxiliary lookup array named ht is maintained. The index values of ht represent node IDs and the values in ht represent indexes on the heap array. So, in Figure 1 because ht[0] = 4, the node with ID = 0 is located at position [4] in the heap array. Values of -1 in ht indicate that a node is not in the heap array. And in Figure 1 because ht[4] = -1, there is no node with ID = 4 in the heap array.

The bottom part of Figure 1 illustrates that there are six implementation-dependent state invariants of the priority queue implementation which can be verified programmatically to establish the correctness of the queue. First, the distance value of the node which is located in position [0] of the heap array must be less than or equal to the distance value of all nodes at positions [1] through [count-1]. Second, the number of -1 values in the ht array must equal the quantity capacity – count. This invariant can be expressed equivalently as the number of non-negative values in the ht array must equal the count value. Third, there cannot be any duplicate non-negative values in the ht array. Fourth, each of the integer values in the set [0..count-1] must appear exactly once in the ht array. Fifth, for each non-negative value i located at index [j] in the ht array, the ID value at index [i] in the heap array must equal j. Sixth, the heap property in the heap array must hold for all values at positions [0] through [count-1], that is, the distance values at indexes [i*2+1] and [i*2+2] must be greater than or equal to the distance value at index [i].

The random access priority queue was implemented with an object oriented approach using the C# language as a class library which in turn was realized as a DLL file suitable for use by programs written in any .NET-compliant language. However, the implementation details are independent of the design for the most part and the priority queue could have been implemented using any modern language, such as C++ or Java.

4. Testing

As described in the introduction section of this paper, there were few guidelines available to suggest how best to verify the functional correctness of the random access priority queue. Previous studies have suggested that in general a pure random testing approach is less effective than other testing techniques such as equivalence partition based testing but there are few studies which provide practical guidance for testing complex data structures [8].

Four different testing approaches were used to test the priority queue data structure described in the previous section of this study. The number of test cases created using each of the four techniques was determined by the number of cases which could be created by an experienced software engineer in four hours. The first approach was to use a manual coding technique. In this approach, a short program was created where the queue was created and one or two operations were manually coded. The program was executed and the final state of the queue was examined using a utility display function. The program was then manually modified and executed again, and the process repeated. A total of 36 test cases were created in this way.

The second testing approach was a unit test technique. In this approach small auxiliary blocks of code called unit tests were embedded into the implementation of the random access priority queue. Each code block corresponded to a single test case and the unit tests were run via a unit test harness program. A total of 23 test cases were created in this way. The third testing approach was a test harness approach. Here an external file of test case data was created, and a small test harness program was written. The test harness program read each test case, performed the indicated operations on the queue based on test case input, and programmatically determined a pass/fail result based on test case data expected value. A total of 36 test cases were created in this way. The fourth testing approach was a state transition approach. Here a dedicated test program was created. The test program repeatedly selected a queue random operation, performed that operation, and then performed a state validation check. Because this approach is effectively a type of random input testing, the equivalent of a total of 500,000 test cases were executed.

An example of code in the manual coding test approach is suggested by the following pseudocode:

```
pq := NewQueue(8);
pq.Enqueue(Node(0,20));
pq.Enqueue(Node(1,10));
pq.Show();
```

Here a node with ID = 0 and distance/priority = 20 is added, and then a second node with ID = 1 and distance/priority = 10 is added. The resulting queue was then examined manually to verify that the second node was at the front of the queue.

An example of code in the unit test approach is suggested by the following pseudocode:

```
[Test]
public void TestCase001()
{
  pq := NewQueue(8);
  pq.Enqueue(Node(0,20));
  pq.Enqueue(Node(1,10));
  Assert.AreEqual(1, pq.PeekID());
}
```

An example of test case data used in the test harness approach is suggested by the following:

```
001:e0,20:e1,10#p1
002:e0,20,e1,10#c2
```

The first line of test case data is interpreted by the test harness program to mean, read test case 001, enqueue a node with ID = 0 and distance/priority = 20, then enqueue a node with ID = 1 and distance/priority = 10, then verify the peek (front) node has ID = 1.

An example of code in the state transition test approach is suggested by the following pseudocode:

```
loop
  operation = RandomOperation();
  if (operation = "enqueue") enqueue;
  else if (operation = "dequeue") dequeue;
  else if (operation = "modify") modify;
  else if (operation = "remove") remove;
  check internal state of queue
end loop
```

The key to the state transition approach is the existence of a function which checks the priority queue for internal consistency. This verifier function performed four checks which correspond to four of the six implementation invariants described in the previous section of this paper. First, the verifier checked that the highest priority / minimum distance in the queue is located at the root node of the underlying binary heap. Second, the verifier checked that the auxiliary ht / location array values are consistent with the values in the heap array. Third, the verifier checked that the fundamental heap ordering property holds for all value in the heap array. Fourth, the verifier checked that there are no nodes which have duplicate ID values in the heap array.

A 2005 paper by de Nivelle and Piskac present a formal specification for priority queues. The authors point out that when data structures are used within a computer program, state verification of the data structure can be performed in what they call an on-line or off-line manner [9]. That paper defines on-line priority queue verification as verification which takes place immediately after any operation is performed, and off-line verification a verification which takes place at some later point in time, possible after several operations have been performed. In this context, the state transition testing approach used in this case study was performing an on-line verification.

5. Results

Because this study was an empirical case study rather than an experimental study, there was no a priori hypothesis with regards to which of the four testing techniques would be the most effective. An initial, beta version implementation of the random access priority queue was tested using each technique. This beta version was created as a production version and not modified because it was intended to be part of a case study. The numbers of faults in each of the four priority queue operations detected by each testing strategy are shown in Table 1.

Table 1. Effectiveness of testing strategies.

	enqueue	dequeue	remove	modify	
manual	0	1	1	1	3
unit	0	2	1	1	4
harness	1	1	2	2	6
state	3	3	4	3	13

Each of the four testing strategies was performed independently, on the same initial version of the priority queue implementation and all test approaches were created before any of the testing approaches were used to eliminate any bias introduced by running one set of tests before another. The set of faults detected by the state transition testing strategy was a proper superset of the faults found by the combined fault sets of the other three testing strategies, or in other words, the manual test strategy and the unit test strategy and the classical test harness strategy together did not find any faults which were not detected by the state transition strategy.

Because the state transition testing strategy used in this study generates test scenarios where a priority queue operation is selected at random, the state transition testing approach can be viewed as an intelligent form of pure random input testing. In pure random input testing, pseudorandom input is fed to the system under test and in most situations there is no explicit expected value associated with input. Therefore, the goal of pure random input testing is typically to cause a severe system fault which will halt the system [10]. The state transition testing approach used in this study essentially used a global, dynamic meta expected value in the form of a consistent internal state of the priority queue under test. In order to evaluate the extent to which state transition testing is related to pure random testing, a pure random input test harness was created and executed against the beta implementation of the priority queue. The pure random testing approach revealed a total of 4 program faults suggesting that state transition testing is in fact more than simply a variation of pure random input testing.

6. Conclusions

In terms of the number of faults discovered, the state transition testing approach was clearly the best technique in this particular case study. The state transition approach revealed the most faults for each of the four fundamental operations. In fact, after this case study concluded, the priority queue examined here was used for extensive graph analyses over the course of several months and no additional faults were revealed suggesting that the state transition testing approach revealed all significant faults in the priority queue.

Even though the state transition testing approach was able to execute vastly most test cases than the other three testing approaches, the fact that the state transition approach discovered the most faults is somewhat surprising because this approach is a relatively unsophisticated approach meaning that little software testing principles were required to create the implicit test cases used with this approach. On the other hand, because the concept of state is such an inherent part of an abstract data structures, the results are perhaps not so surprising. Because this was a case study rather than an experimental study, no broad conclusions can be drawn from the study's results. This case study is perhaps best viewed as a preliminary investigation which lays the groundwork for experimental investigations of the effectiveness of different strategies for testing complex data structures.

7. References

[1] J.H. Andrews, "A Case Study of Coverage-Checked Random Data Structure Testing", *Proceedings of the 19th International Conference on Automated Software Engineering*, September 2004, pp. 316-319.

[2] J. Deshmukh and E.A. Emerson, "Verification of Recursive Methods on Tree-Like Data Structures", *Formal Methods in Computer-Aided Design*, November 2009, pp. 33-40.

[3] A. Bouajjani, P. Habermehl, and A. Rogalewicz, "Abstract Regular Tree Model Checking of Complex Dynamic Data Structures", *Proceedings of the Static Analysis Symposium (SAS)*, 2006, pp. 52-70.

[4] P. Habermehl, R. Iosif, and T. Vojnar, "Automata-Based Verification of Programs with Tree Updates", *Proceedings of TACAS, Springer Lecture Notes in Computer Science*, 2006, vol. 3920, pp. 350-364.

[5] J. Deshmukh, E. Emerson, and P. Gupta, "Automatic Verification of Parameterized Data Structures", *Proceedings of the 16th International Symposium on Field-Programmable Custom Computing Machines*, April 2008, pp. 248-257.

[6] M.C. Herbordt, F. Kosie, and J. Model, "An Efficient O(1) Priority Queue for Large FPGA-Based Discrete Event Simulations of Molecular Dynamics", *Proceedings of the 19th International Conference on Automated Software Engineering*, September 2004, pp. 316-319.

[7] K. Dragicevic and D. Bauer, "A Survey of Concurrent Priority Queue Algorithms", *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing*, April 2008, pp. 1-6.

[8] J.D. McCaffrey, "An Empirical Study of the Effectiveness of Partial Antirandom Testing", *Proceedings of the 18th International Conference on Software Engineering and Data Engineering*, June 2009, pp. 260-265.

[9] H. de Nivelle and R. Piskac, "Verification of an Off-Line Checker for Priority Queues", *Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods*, September 2005, pp. 210-219.

[10] J.D. McCaffrey, *.NET Test Automation Recipes: A Problem-Solution Approach*, Apress Publishing, New York, 2006.