Concurrent Access of Priority Queues

Andrew Milner
Overview

• Review

• New Results

• Java Pathfinder Results
Inserting and Deleting

• Issues arise when performing insertions and deletions at the same time.

• Deadlocks will occur as a result.
Solution

• Nageshwara and Kumar suggest insertions should be completed from the top down.

• Both procedures now perform actions from the top down and eliminate the potential for a deadlock to occur.
Top Down Insertions

- Two values are required in order to predicate the path the node takes through the heap.
  - LastElem: The node location of the last element + 1.
  - FullLevel: The node location of the first element at the deepest level of the heap.

- The difference between the two values gives the path.
  - This is possible because the heap is a binary tree.
Top Down Insertions

• The difference between LastElem and FullLevel is represented as a binary value.

• Each digit in the binary value represents a direction of travel through the heap.

• 1 indicates a right movement and 0 indicates a left movement.
Top Down Insertions

NodeLoc = (LastElem + 1) – FullLevel = 5
NodeLoc = 101
The path of the node through the heap is right, left, right.

FullLevel = 8
LastElem = 12
Top Down Insertions

NodeLoc = (LastElem + 1) – FullLevel = 5
NodeLoc = 101
The path of the node through the heap is right, left, right.
Top Down Insertions

NodeLoc = (LastElem + 1) – FullLevel = 5
NodeLoc = 101
The path of the node through the heap is right, left, right.
Top Down Insertions

\[ \text{NodeLoc} = (\text{LastElem} + 1) - \text{FullLevel} = 5 \]
\[ \text{NodeLoc} = 101 \]
The path of the node through the heap is right, left, right.
Top Down Insertions

NodeLoc = (LastElem + 1) – FullLevel = 5
NodeLoc = 101
The path of the node through the heap is right, left, right.
Top Down Insertions

NodeLoc = (LastElem + 1) – FullLevel = 5
NodeLoc = 101
The path of the node through the heap is right, left, right.
Deleting/Inserting

• Only a small portion of the heap is locked.

• This locking window consists of three nodes for deletion (the parent and child nodes) and one node for insertions.

• Anytime a node is accessed it is locked in order to maintain mutual exclusion.

• FullLevel and LastElem are modified only during the initialization, after the root has been locked.
Deleting/Inserting

• One issue still exists which prevents insertions and deletions from working together.

• The deletion operation is forced to wait for an insertion operation to complete before it can begin.
Deleting/Inserting

• The problem is solved by associating a status field with each of the nodes.

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRESENT</td>
<td>A key exists at the node.</td>
</tr>
<tr>
<td>PENDING</td>
<td>An insertion is in progress which will ultimately insert a key at the node.</td>
</tr>
<tr>
<td>WANTED</td>
<td>A deleter is waiting for the key.</td>
</tr>
<tr>
<td>ABSENT</td>
<td>No key is present at the node.</td>
</tr>
</tbody>
</table>
Deleting/Inserting

• When an insertion operation begins the target status is set to PENDING.

• If the deletion operation is invoked while an insertion operation is still in progress, the status of the target is changed to WANTED.

• During each loop, the insertion operation checks to see if the target node’s status is WANTED.
Implementation

- The implementation of the concurrent heap was taken from Nageshwara and Kumar.

- Pseudo code was provided in the original paper.
Design Choices

• Pseudo code presented by Nageshwara and Kumar uses an integer based implementation.

• Array based heap data structure.
  – Parent location is represented by \( i \).
  – The children are located at positions \( i \times 2 \) and \( i \times 2 + 1 \)
Testing

• The correctness of the algorithm was tested using a single thread.

• A series of insertion and deletion operation were performed.

• The heap sizes ranged from 10 to 10,000.
Testing

- Check that the heap property was always maintained.

- Check for the occurrence of deadlock during insertion/deletion operations.

- Check if the node status was updated correctly.
Testing

• Initial insertion and random test heap size 5,000.

• Initial deletion test heap size 15,000.

• Values were randomly determined between 1 and the max value.

• Work was split up evenly between threads.

• Each thread test was run 100 times.
Semaphore Insertion Results
Lock Insertion Results
Lock Deletion Results
Semaphore Insert/Delete Results
Lock Random Results

![Graph showing lock random results with values ranging from 35 to 95.](image-url)
Results

• Results did not support the predicated performance by Nageshwara and Kumar.

• Performed only 1000 operations.

• Results did predict a fall off in performance as the number of threads increased.
Java Pathfinder

• Testing was completed to verify:
  – Deadlock freedom
  – Race conditions
  – Removal of locks
  – Correctness of behaviour

• Execution time was much longer than expected.
  – Even for small heaps verification took a very long time.
Deadlock Free

• Checks were performed for the occurrence of deadlocks.

• No deadlocks reported!

• The risk of deadlock was unlikely as the code implementation prevents this.
Inserting and Deleting
Inserting and Deleting

• The algorithm attempts to perform and insertion and deletion at the same time.
Inserting and Deleting

- The top node is deleted but cannot get the last node until it is unlocked.
Inserting and Deleting

- The last node is unlocked and moved to the top. The newly inserted node continues to move up through the heap.
Inserting and Deleting

• The two nodes are now deadlocked! Both nodes are waiting for the other to unlock and the algorithm is now stuck.
Race Conditions

• Likely the only problem to exist.
  – Nodes are not protected correctly.

• Node status codes could be cause a problem.

• Both the deletion and insertion operation have access to the status.
  – Status code is updated when a node is wanted.
Race Conditions

- Race conditions probably will not occur.
  - The root and target are locked when a status update occurs.

- No exceptions were raised by the race condition listener.
Removal of Locks

• Does the data structure already utilize the minimum number of locks?

• One lock is associated with each node.
  – Global variables are only accessed when the root is locked.

• Data races were present in the execution.
Removal of Locks

• The test showed that the heap uses the minimum number of locks.

• The heap cannot function properly without one lock per node.
Behaviour Correctness

• Several assertion tests were created to check the data structure.
  – Insertion operation
  – Deletion operation
  – Heap property
Behaviour Correctness

• Insertion tests
  – Heap size increased
  – Node status updates
  – Target location is filled
  – Heap property maintained

• No exceptions were raised during execution.
Behaviour Correctness

• Deletion tests
  – Heap size decreased
  – Root node is removed
  – Node status updates
  – Heap property maintained

• Again no exceptions were raised.
Behaviour Correctness

• Was the heap property maintained?

• A heap property test was run during the insertion and deletion loops.
  – Also run after the operation was completed.

• No exceptions were raised.
Conclusions

• Execution time was very very very very long.

• The data structure functioned as expected.
  – Still not as effective as a serial implementation.
Sources

Questions?