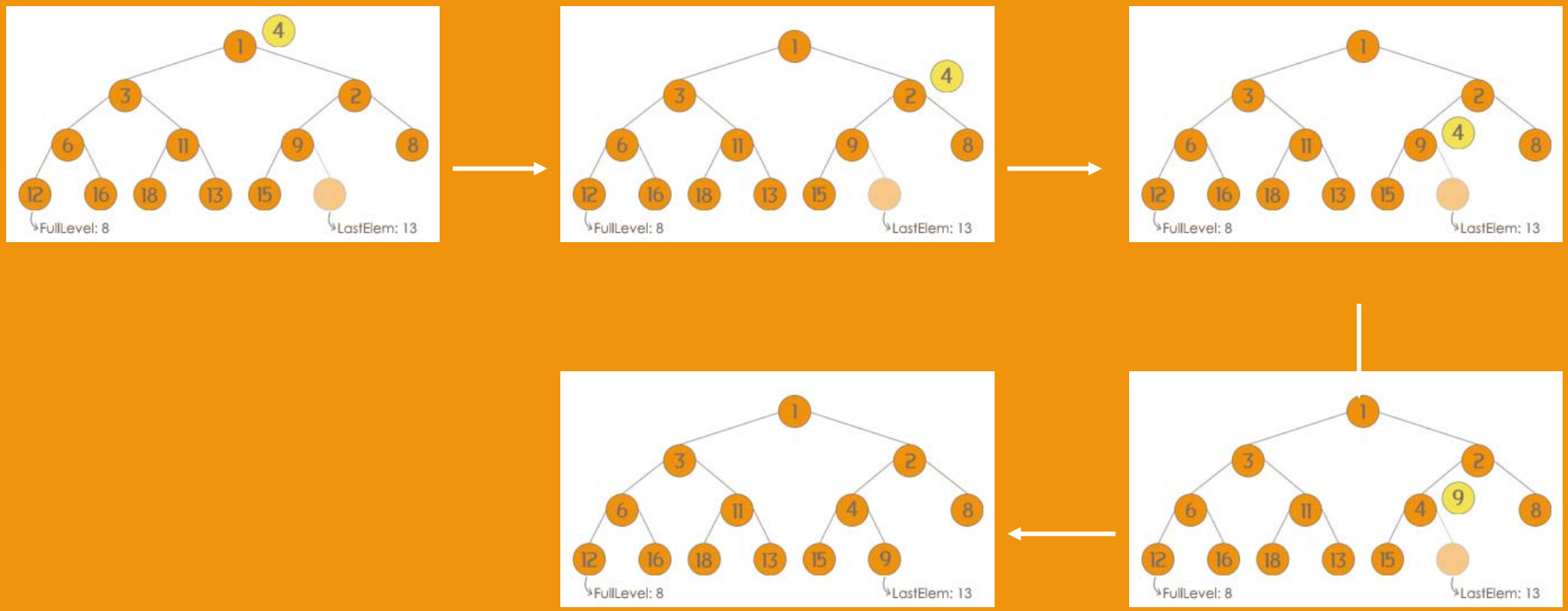
Concurrent Access of Priority Queues

R.V. Nageshwara and V. Kumar

Modified Binary Heap

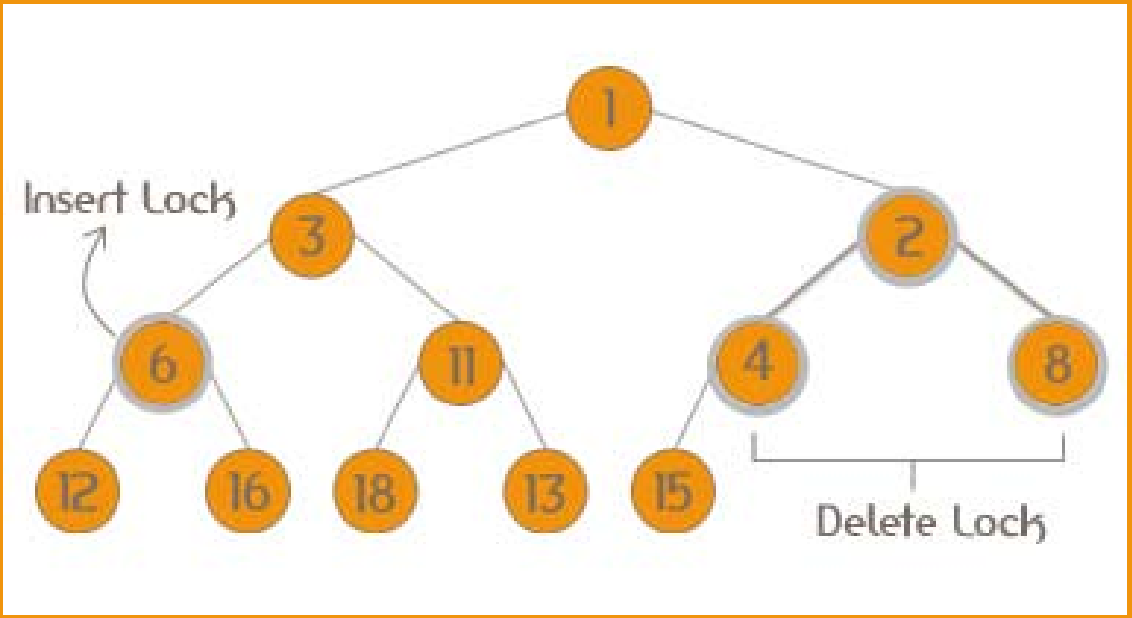
In order to allow for concurrent access in a binary heap the insert routine must be modified to allow for a top down insert (so that it matched the top down delete).



Concurrent Binary Heap

With the top down insert procedure it now becomes possible to open the heap to multiple processes and lock it on a node by node basis.

- **Insert Processes** would lock only one node
- **Delete Processes** would lock 3 nodes: The parent and the 2 children



Implementation Details

- The heap was implemented as an array of size N+1, elements were stored in indices 1 to N+1.
- Each node/element consisted of a wrapper class that had the node value, the status and the node lock.
- The insert process proceeded in a top down manner, locking a node and it's two children.
- The delete process cleared the root node and waited for the replacement node to be made available, then would proceed down the tree to fix up the node.

Validation
Details

In order to verify the java implementation of a concurrent heap the following properties were examined:

- deadlock freedom
- race conditions and lock necessity
- correct behavior
- access to node status

JPF was used extensively to verify various properties however it was found that the implementation of the concurrent heap as was discussed in the paper was not one that was easy to check with JPF.

Validation
Details

Deadlock Freedom:

- Default property of JPF
- Using BFS and DFS search on the execution path tree no deadlocks reported
- By definition this implementation can’t possibly have any deadlocks, only data races because of the way that inserts and deletes are structured

Validation
Details

Race Conditions and Lock Necessity:

- Removing any of the locks causes the JPF Race Listener to raise exceptions so the number of locks present is the minimal amount that can be used to implement the heap
- Oddly enough no race conditions were found on the status of the root node using both BFS and DFS for JPF

Validation
Details

Correct Behavior :

- Asserts added before and after insert routine to check that the heap structure is maintained correctly
- Asserts also added before and after delete routine to check that the heap structure is maintained correctly
- Loop added to implementation to verify that after an insert or delete or a series of both the heap order property and heap structure was maintained
- For the checking loops, exceptions thrown if something is not correct so JPF can trigger the uncaught exception property

Conclusion

Questions?

This implementation was not one that was easy to check. As mentioned before only part of the execution tree was checked by JPF because even with a small number of trials and threads the state space seemed to explode exponentially.

For the part of the execution tree that was checked though, the data structure checked out ok and functioned properly.