AN EMPIRICAL COMPARISON OF PRIORITY-QUEUE AND EVENT-SET IMPLEMENTATIONS

DOUGLAS W. JONES
Basic Priority Queue Operations

• Enqueue (Insert)
  – Places an item in the priority queue

• Dequeue (Delete-min)
  – Removes and returns the highest priority item from queue
Relation to Simulation

- Priorities represent event times in discrete event simulation
  - Enqueue Schedules Events
  - Dequeue Finds Next Pending Event (lowest numbered time)
Measuring Performance

- Hold Method
  - Based on simple discrete-event simulation
  - All events cause scheduling of one new event
    - Keeps constant queue size
    - Direct measure of \( \frac{\text{queuesize}}{\text{performance}} \)
    - Random priority value, like next-event simulation
  - Repeatedly dequeue and enqueue items
  - Divide by total time by number of trials
Measuring Performance (Cont.)

- 5 priority increment distributions were used
- Measurements based on 1000 trials

<table>
<thead>
<tr>
<th>Distribution</th>
<th>Expression to compute random values</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Exponential</td>
<td>$-\ln(u)$</td>
</tr>
<tr>
<td>2. Uniform 0.0-2.0</td>
<td>$2 \times u$</td>
</tr>
<tr>
<td>3. Biased 0.9-1.1</td>
<td>$0.9 + 0.2 \times u$</td>
</tr>
<tr>
<td>4. Bimodal</td>
<td>$0.95238 \times u + \text{if } u &lt; 0.1$ \text{ then } 9.5238 \text{ else } 0$</td>
</tr>
<tr>
<td>5. Triangular</td>
<td>$1.5 \times u^{0.5}$</td>
</tr>
</tbody>
</table>

$u$ is a Uniform(0,1) call
Implementations

- Linear List
- Implicit Heaps
- Leftist Trees
- Two List
- Henriksen’s
More Implementations

- Binomial Queues
- Pagodas
- Skew Heaps
- Splay Trees
- Pairing Heaps
Linear List

- Singly linked list searching from the head at insertion

- Favors LIFO behavior

- Minimizes storage requirements
  - Only one pointer per item

- \( O(n) \) sequential search for enqueue, \( O(1) \) dequeue

- Best implementation for 10 or less item queues
Implicit Heaps

- $O(\log n)$ performance

- Fast, but many newer queue implementations faster

- Represented as binary tree with heap invariant

- Any item has higher priority than its children

- Stored as an array
  - Location 1 is root
  - $2i$ and $2i + 1$ are children of location $i$
Implicit Heaps (Cont.)

- Enqueue operation
  - Search begins from leaf at upper bound of heap
  - Search toward root
  - Passed items are demoted to make space for new item

- Dequeue operation
  - Returns the root
  - Promotes other items while searching for new place for the most distant leaf.
Leftist Trees

- Heap structure explicitly represented with pointers from parents to their children

- Enqueue operation
  - Item initialized as one node tree
  - Then merged with original tree

- Dequeue operation
  - Root returned
  - Right and Left subtrees then merged
Leftist Trees (cont.)

- Merge operation
  - Merge rightmost branches of the 2 trees
  - Distance to the nearest leaf is recorded for each item
  - 2 children sorted so that path to nearest leaf is always through the right child
  - This guarantees $O(\log n)$ bound

- About 30% slower than implicit heaps in tests
Queues Favoring Discrete-Event Simulation

- Two List and Henrickson’s implementations

- Stable queue behavior
  - 2 events scheduled to occur at same time are FIFO

- Most other priority queues cannot guarantee this
Two List

- One short sorted list of items near the head of the queue

- One long unsorted list of more distant events

- Enqueued item compared with a threshold priority to determine correct list to put it in

- Dequeued items just removed from sorted list

- When sorted list is empty
  - Advance threshold and search unsorted list for items to move to sorted list
  - Keeps an average of $n^{0.5}$ items in sorted list
Two List (cont.)

- Average enqueue time of $O(n^{0.5})$

- Worst-case dequeue $O(n)$, but most are done in $O(1)$ time

- Average dequeue of $O(n^{0.5})$

- Good performance for queues up to a few hundred items

- Very poor with Bimodal distribution
Henriksen’s

- Uses Simple linear list
- Auxiliary array of pointers into list
- Allows $O(\log n)$ binary search to find range of entries where enqueued items should be placed
- Significant cost of maintaining array and searching subsection of list pointed to by array entry
- Average performance bounded by $O(n^{0.5})$
- Performed well comparatively
Binomial Queues

- A forest of binomial trees where the number of elements in each tree is an exact power of 2

- Height $n$ Binomial Tree
  - Root has $n - 1$ children
  - Children are binomial trees with heights $n - 1$, $n - 2$, ..., 0

- Performs extremely well

- Varies for small queue size changes based on binary representation of size

Binomial trees of heights 0, 1, 2, and 3.
Pagodas

- Based on heap ordered binary trees
- Primary pointers lead from leaves toward root
- Secondary pointers point down to item’s left- and rightmost descendants
Pagodas (cont.)

- Enqueue and dequeue operations
  - Merge the right branch of one pagoda with left branch of another

- Insertions occur in constant time

- No balancing effort made, resulting in infinite sequences of $O(n)$ per operation

- Arbitrary deletions occur in $O(\log n)$ time
  - All branches circularly linked

- Performs about as well as Binomial Queues
Skew Heaps

- Similar to leftist tree, but no record of path length to nearest leaf

- Children of each item visited on the merge path are exchanged to randomize the tree structure

- Per operation cost never exceeds $O(\log n)$ over a sufficiently long sequence of operations

- Performs faster than implicit heaps
Splay Trees

- Set up as binary search trees
  - All items in left subtree smaller than root
  - All items in right subtree larger than root

- Dequeue operation simply removes the leftmost item

- Blindly performs pointer rotations
  - The basic balancing operation
  - Avoids keeping and testing balancing records
  - Causes increased number of rotations
Splay Trees (cont.)

- Stable - Equal priority items are FIFO

- Like Henriksen’s performed exceptionally well for the biased distribution

- Overall faster than Henriksen’s implementation

- In a sense optimal
Pairing Heaps

- Heap-ordered tree

- Constant time Enqueue
  - Can make new item root
  - Or adds new item as additional child of root

- Dequeue returns root then searches for new root

- Key to pairing heaps is method of finding new root

- Link successive children of old root in pairs, then link each pair to the last pair produced
Pairing Heaps (cont.)

- Combining two pairing heaps
  - Adds heap with lower priority root as child of other heap

- Performed about the same as bottom-up skew heap

- Ran especially well on the biased distribution
Conclusions

- Linked list is best implementation for < 10 items
- Two-list performs well up to a couple hundred items except for some distributions
- Leftist trees don’t perform well enough for any application
- Henricksen’s acceptable for all queue sizes
- Splay trees challenge it where stable behavior is required
Conclusions (cont.)

- Implicit Heaps one of worst for less than 20 items

- Binomial queues are erratic and most complex to code

- Skew heaps, pairing heaps, and pagodas all almost as good as splay trees

- Top-down skew heap is very simple

- When other operations are needed like arbitrary deletions or priority changes
  - Bottom-up skew heaps, splay trees, and pairing heaps are best alternatives
## Summary

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Relative Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linked list</td>
<td>11</td>
</tr>
<tr>
<td>Implicit heap</td>
<td>8</td>
</tr>
<tr>
<td>Leftist tree</td>
<td>9 – 10</td>
</tr>
<tr>
<td>Two List</td>
<td>9 – 10</td>
</tr>
<tr>
<td>Henriksen’s</td>
<td>1 – 7</td>
</tr>
<tr>
<td>Binomial Queue</td>
<td>1 – 7</td>
</tr>
<tr>
<td>Pagoda</td>
<td>4 – 8</td>
</tr>
<tr>
<td>Skew heap</td>
<td>4 – 7</td>
</tr>
<tr>
<td>Splay Tree</td>
<td>1 – 3</td>
</tr>
<tr>
<td>Pairing Heap</td>
<td>3 – 6</td>
</tr>
</tbody>
</table>

1 is fastest; 11 is slowest