Title:
Relaxed weak queues: an alternative to run-relaxed heaps

Speaker:
Jyrki Katajainen

Co-workers:
Amr Elmasry and Claus Jensen

These slides as well as the underlying paper are available at http://www.cphstl.dk/.

© Performance Engineering Laboratory
Priority-Queue Operations

**insert**
input: element
output: locator

**find-min**
input: none
output: locator

**delete**
input: locator
output: none

**delete-min**
\[ p \leftarrow \text{find-min()} \]
\[ \text{delete}(p) \]

**decrease**
input: locator, element
output: none

**meld**
input: two priority queues
output: one priority queue
Various Approaches

- **winner tree**
- **loser tree**
- **heap-ordered tree**
- **weak-heap-ordered tree**
- **search tree**
- selection tree
- navigation pile
- binomial tree
- Vheap
- binary heap
- leftist heap
- weak heap
- AVL tree
## Market Analysis

<table>
<thead>
<tr>
<th>Efficiency Method</th>
<th>Binary Heap Worst Case</th>
<th>Binomial Queue Worst Case</th>
<th>Fibonacci Heap Amortized</th>
<th>Run-Relaxed Heap Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>find-min</strong></td>
<td>( \Theta(1) )</td>
<td>( \Theta(1) )</td>
<td>( \Theta(1) )</td>
<td>( \Theta(1) )</td>
</tr>
<tr>
<td><strong>insert</strong></td>
<td>( \Theta(lg , n) )</td>
<td>( \Theta(1) )</td>
<td>( \Theta(1) )</td>
<td>( \Theta(1) )</td>
</tr>
<tr>
<td><strong>decrease</strong></td>
<td>( \Theta(lg , n) )</td>
<td>( \Theta(lg , n) )</td>
<td>( \Theta(1) )</td>
<td>( \Theta(1) )</td>
</tr>
<tr>
<td><strong>delete</strong></td>
<td>( \Theta(lg , n) )</td>
<td>( \Theta(lg , n) )</td>
<td>( \Theta(lg , n) )</td>
<td>( \Theta(lg , n) )</td>
</tr>
</tbody>
</table>
| **meld**          | \( \Theta(lg \, m \times lg \, n) \) \( \Theta(min\{lg \, m, \, lg \, n\}) \) | \( \Theta(1) \)          | \( \Theta(min\{lg \, m, \, lg \, n\}) \) |}

Here \( m \) and \( n \) denote the number of elements in the priority queues just prior to the operation.
Our Work

Relaxed weak queues — an alternative to run-relaxed heaps:

- are simpler to program,
- work on a pointer machine except that \textit{meld} requires random access [pointer machine \(\approx\) C without arrays],
- are asymptotically equally fast,
- have low constant factors [\textit{delete} requires \(3 \lg n + O(1)\) element comparisons; can be improved to \(\lg n + O(\lg \lg n)\)], and
- use less space [\(3n + O(\lg n)\) extra words; \(4n + O(\lg n)\) with \textit{meld}].
Nonstandard Tree Terminology

- $p$ is the **surrogate parent** of $q$.
- $p$ is the **real parent** of $r$.
- Let $s$ be a node in a binary tree. We call every ancestor of $s$ that is a real parent of another ancestor of $s$ a **real ancestor** of $s$. 
Perfect Weak Heaps

A perfect weak heap is a binary tree having the following three properties:

1. The root has no left subtree.
2. The right subtree of the root is a complete binary tree.
3. For every node $s$, the element stored at $s$ is no smaller than the element stored at the first real ancestor of $s$.

**Fact 1.** A perfect weak heap stores $2^h$ elements for some integer $h \geq 0$.

**Fact 2.** The root of a perfect weak heap must store a minimum element.
A **weak queue** $Q$ storing $n$ elements is a collection of disjoint perfect weak heaps. Consider the binary representation of $n$

$$n = \sum_{i=0}^{\lfloor \lg n \rfloor} b_i 2^i,$$

where $b_i \in \{0, 1\}$ for all $i \in \{0, \ldots, \lfloor \lg n \rfloor\}$. In its basic form, $Q$ contains a perfect weak heap $H_i$ of size $2^i$ if and only if $b_i = 1$, i.e.

$$Q = \{H_i \mid n = \sum_{i=0}^{\lfloor \lg n \rfloor} b_i 2^i \text{ and } b_i = 1\}.$$
**Primitive Operations**

Joining and splitting two perfect weak heaps of the same size:

Note that for a binary heap a join may take logarithmic time.
Heap Store

A **heap store** is a sequence of perfect weak heaps appearing in increasing order of height.

\[ H_j \rightarrow H_k \rightarrow \ldots \rightarrow H_\ell \]

size \( O(\lg n) \)

**input:** \( H_i, \ i \leq j \)
**output:** none

**inject**

**input:** none
**output:** \( H_j \)

**eject**

**input:** \( H_\ell \) and \( H'_\ell \)
**output:** none

**replace**

**Idea.** Injections are done lazily by not doing all joins at once; we allow between zero and two perfect weak heaps of each size.

**Theorem 1.** All heap-store operations **inject**, **eject**, and **replace** take \( O(1) \) worst-case time.
Potential Violation Nodes

- A **weak-heap-order violation** occurs if the element stored at a node is smaller than the element stored at the **first** real ancestor of that node. In a **marked node** a weak-heap-order violation may occur.

- A marked node is **tough** if it is the left child of its parent and also the parent is marked.

- A chain of consecutive tough nodes followed by a single nontough marked node is called a **run**.

- All tough nodes of a run are called its **members**.

- The single nontough marked node of a run is called its **leader**.

- A marked node that is neither a member nor a leader of a run is called a **singleton**.
Node Store

The primary purpose of a node store is to keep track of potential violation nodes, and its secondary purpose is to store the heights and types of the nodes.

mark
input: a node
output: none

unmark
input: a node
output: none

correct
input: none
output: none

effect: Unmark at least one arbitrary marked node.

Theorem 2. The node-store operations mark, unmark, and reduce take $O(1)$ worst-case time.
Primitives Used by \textit{reduce}

a)
\[
\begin{array}{c}
p \\
q \\
s \\
A \\
B \\
C \\
D \\
\end{array} \rightarrow \begin{array}{c}
p \\
s \\
q \\
A \\
D \\
C \\
B \\
\end{array}
\]

b)
\[
\begin{array}{c}
p \\
q \\
A \\
B \\
C \\
\end{array} \rightarrow \begin{array}{c}
p \\
q \\
A \\
B \\
C \\
\end{array} \text{ or } \begin{array}{c}
p \\
A \\
q \\
B \\
C \\
\end{array}
\]

c)
\[
\begin{array}{c}
p \\
q \\
s \\
A \\
B \\
C \\
D \\
\end{array} \rightarrow \begin{array}{c}
p \\
s \\
q \\
A \\
C \\
B \\
D \\
\end{array} \text{ or } \begin{array}{c}
p \\
s \\
A \\
C \\
D \\
B \\
\end{array}
\]

d)
\[
\begin{array}{c}
p \\
q \\
r \\
A \\
B \\
C \\
D \\
\end{array} \rightarrow \begin{array}{c}
p \\
r \\
B \\
D \\
\end{array} \text{ or } \begin{array}{c}
s \\
q \\
A \\
B \\
\end{array}
\]
To facilitate a fast \textit{find-min}, a pointer to the node storing the current minimum is maintained and updated by all modifying operations. This \textbf{minimum pointer} refers to a root or to a potential violation node.

The minimum pointer points to the node storing the current minimum, so this node can just be returned.

\textbf{Worst-case time:} \( \Theta(1) \); no element comparisons
1. Allocate a new node and put $e$ there.
2. Place the new node, which is also a perfect weak heap of height 0, into the heap store by invoking `inject`.
3. Correct the minimum pointer to point to the new node if $e$ is smaller than the current minimum.

**Worst-case time:** $\Theta(1)$ with at most 2 element comparisons
\textit{decrease}(p, e)

1. Make the element replacement at \( p \).

2. Make \( p \) a potential violation node by invoking \textit{mark}.

3. Reduce the number of potential violation nodes, if possible, by invoking \textit{reduce}.

4. Correct the minimum pointer if necessary.

\textbf{Worst-case time:} \( \Theta(1) \) with at most 4 element comparisons
The idea is to extract the subheap rooted at $p$ from the perfect weak heap, in which it resides, borrow another node $q$ from the smallest perfect weak heap to fill in the hole created by $p$, and put the new subheap in the place of the extracted subheap.

**Worst-case time:** $\Theta(\lg n)$ with at most $3 \lg n + O(1)$ element comparisons
delete$(p)$ — Details

1. Eject the smallest perfect weak heap from the heap store by invoking $eject$. Let $q$ be the root of that perfect weak heap.

2. Repeat until $q$ has no children:
   a) Split the perfect weak heap rooted at $q$. Let $r$ be the root of the other sub-heap created.
   b) Remove the marking of $r$, if any, by invoking $unmark$.
   c) Insert the subheap rooted at $r$ into the heap store by invoking $inject$.

3. If $p$ and $q$ are the same node, go to 11.

4. Extract the subheap rooted at $p$ from the perfect weak heap, in which it resides, and remember its neighbouring nodes.

5. Repeat until $p$ has no children:
   a) Split the subheap rooted at $p$. Let $s$ be the root of the other subheap created.
   b) Push the subheap rooted at $s$ onto a temporary stack.
6. Repeat until the temporary stack is empty:
   a) Pop the top of the stack. Let $s$ be the root of the subheap popped.
   b) Remove the marking of $s$, if any, by invoking `unmark`.
   c) Join the subheaps rooted at $q$ and $s$; independent of the outcome denote the new root $q$.

7. Put $q$ in the place of $p$.

8. Make $q$ a potential violation node by invoking `mark`.

9. If $p$ was a root, substitute the perfect weak heap rooted at $q$ for that rooted at $p$ in the heap store by invoking `replace`.

10. Remove the marking of $p$, if any, by invoking `unmark` to update the node store.

11. If the minimum pointer points to $p$, scan all roots and all potential violation nodes to find a new minimum element and update the minimum pointer.
12. Reduce the number of potential violation nodes, if possible, by invoking *reduce* twice (once because of the new potential violation node introduced and once more because of the decrement of \( n \)).

13. Free \( p \) and return.
**meld**($Q, R$)

Assume that the sizes of $Q$ and $R$ are $m$ and $n$, respectively, and that $m \leq n$.

1. Move all singleton and run-leader objects from the node store of $Q$ to the node store of $R$.

2. Eject all perfect weak heaps of $Q$ and store them to a temporary stack $S_Q$.

3. Eject all perfect weak heaps of $R$, whose height is no greater than $\lceil \lg m \rceil$, and store them to a temporary stack $S_R$.

4. Process all perfect weak heaps in $S_Q$ and $S_R$ in height order and inject them into the heap store of $R$.

5. Reduce the number of potential violation nodes in $R$ by at most $\lceil \lg m \rceil$.

6. Destroy $Q$ and return $R$.

**Worst-case time:** $\Theta(\lg m)$ with at most $5 \lg m$ element comparisons
Open Problems

- Is fast *meld* possible without random access?

- Can the number of element comparisons performed by *delete* be reduced from $\lg n + O(\lg \lg n)$ to $\lg n + O(1)$?