Improving the efficiency of priority-queue structures

Claus Jensen

Co-authors

Six of the seven papers in the thesis have been produced in collaboration with:

- Amr Elmasry
- Jyrki Katajainen

Main focus of our study

- The efficiency of addressable priority-queue structures
- Worst-case efficiency
- Comparison complexity
- Constant factors

Priority queues (introduction)

- A priority queue is a data structure that maintains a collection of elements from a totally ordered universe
- For reasons of simplicity we do not distinguish elements from their associated priorities
- Heap-ordered trees can be used as the basic components
- A collection of heap-ordered trees can be maintained using different strategies
- The chosen strategy will affect the efficiency of the priority queue

Set of operations supported by a minimum priority queue Q

find-min(Q). Returns a reference to a node containing a minimum element of priority queue Q

insert(Q, x). Inserts a node referenced by x into priority queue Q. It is assumed that the node has already been constructed to contain an element

extract(Q). Extracts an unspecified node from priority queue Q, and returns a reference to that node. The *extract* operation is in some places called *borrow*

delete-min(Q). Removes a minimum element and the node in which it is contained from priority queue Q

delete(Q, x). Removes the node referenced by x, and the element it contains, from priority queue Q

Set of operations supported by a minimum priority queue Q

$decrease(Q, x, e)$. Replaces the element at the node referenced by x with element e. It is assumed that e is not greater than the element earlier stored in the node

$meld(Q_1, Q_2)$. Creates a new priority queue containing all the elements held in the priority queues Q_1 and Q_2, and returns a reference to that priority queue. This operation destroys Q_1 and Q_2s

Number systems (introduction)

In a positional number system represented by its digits and their corresponding weights.

A **representation** is a string of digits $\langle d_0, d_1, \ldots, d_{k-1} \rangle$ of lenght k

Let $\boldsymbol{d} = \langle d_0, d_1, \ldots, d_{k-1} \rangle$
Where d_0 is the least significant digit

$$value(\boldsymbol{d}) = \sum_{i=0}^{k-1} d_i \times w_i$$

Where w_i is the weight corresponding to d_i

b-ary: $w_i = b^i$ or $w_i = b^{i+1} - 1$ (Skew)

Number systems

Binary: $d_i \in \{0,1\}$; $w_i = 2^i$

Redundant binary: $d_i \in \{0,1,2\}$; $w_i = 2^i$

Regular binary: $d_i \in \{0,1,2\}$; $w_i = 2^i$; Every string has the form $\left(0 \mid 1 \mid 01^{*}2\right)^{*}$ [Clancy & Knuth 1977]

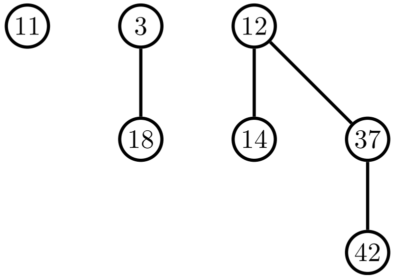
Canonical Skew binary: $d_i < j = 0$; $d_i \in \{0,1,2\}$; $d_i > j \in \{0,1\}$; $w_i = 2^{i+1} - 1$ [Myers 1983]

Zeroless regular: $d_i \in \{1,2,3\}$; $w_i = 2^i$; Every string has the form $\left(1 \mid 2 \mid 12^{*}3\right)^{*}$ [Brodal 1995]

Connection between number systems and priority queue structures

- **A binomial queue using binary representation**

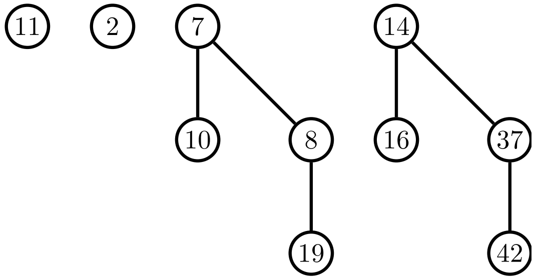
$d = \langle 111 \rangle$



Connection between number systems and priority queue structures

- **A binomial queue using redundant binary representation**

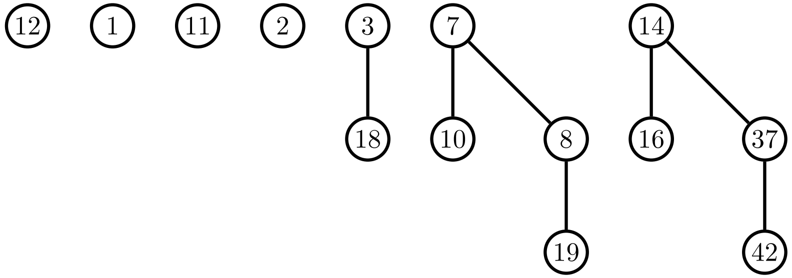
$\boldsymbol{d} = \langle 202 \rangle$



Connection between number systems and priority queue structures

- **A binomial queue using zeroless representation**

$\boldsymbol{d} = \langle 412 \rangle$



Magical skew system (paper one)

Digit set: $d_i \in \{0, 1, 2, 3, 4\}$

Extreme digits: $d_i \in \{0, 1, 3, 4\}$

Low digits: $d_i \in \{0, 1\}$

High digits: $d_i \in \{3, 4\}$

Weight: $w_i = 2^{i+1} - 1$ (skew)

Application: Binary heaps

- **Using the magical skew system to facilitate *insert* in a collection of pointer-based binary heaps we archive the following bounds:**

operations	**worst-case cost**
find-min, *insert*	$O(1)$
delete	$O(\lg n)$ (n size of data structure) $6 \lg n + O(1)$ element comparisons

Regular skew

Cost of a digit change: $O(j)$ at position j

Discretization: Initially, j *bricks* at position j, i.e. $b_j = j$

Digit set: $d_i \in \{0, 1, 2\}$ $\forall i$; when $b_k > 0$, d_k is said to form a *wall* ($\boxed{1}$ or $\boxed{2}$) of b_k bricks

Incremental digit changes: Remove some bricks from some walls in addition to the normal actions; do not transfer digits across any walls

Application: Binary heaps

- **Using the regular skew system to facilitate *insert* and *meld* in a collection of pointer-based binary heaps we archive the following bounds:**

operations	worst-case cost
find-min, *insert*	$O(1)$
meld	$O(\lg^2 m)$ (m size of data structure and $m < n$)
delete	$O(\lg n)$ (n size of data structure) $5 \lg n + O(1)$ element comparisons

Multipartite binomial queue (paper two)

- **A multipartite binomial queue consist of the following five components:**

Buffer: This is a binomial queue relying on the regular binary number system. The buffer is responsible for handling insertions

Reservoir: This is a single tree, initially a binomial tree, but it gradually loses its binomial structure while nodes are borrowed or deleted

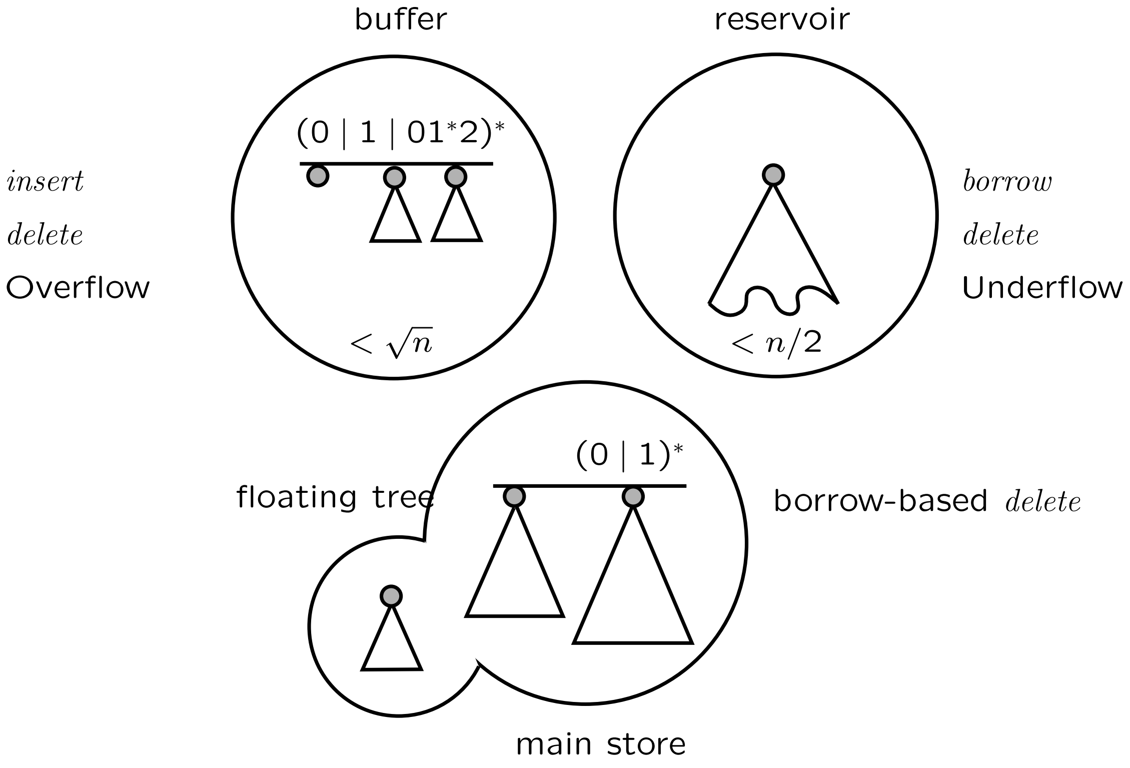
Multipartite binomial queue

Main store: This is a binomial queue relying on the binary number system. The large portion of the n elements is stored here

Upper store: This is a circular doubly-linked list that maintains the order among the roots in the main store. The order is maintained using prefix-minimum pointers. For a given rank a prefix-minimum pointer points to the root which holds the minimum element among the roots which have equal or smaller rank

Floating tree: This is a single binomial tree. It is needed to regulate the traffic between the buffer and the main store

Multipartite binomial queue



Multipartite binomial queue operations

find-min: Compares the minimum candidates from the following components:

- Upper store, the prefix-minimum pointer of the tree of the largest rank in the main store
- The floating tree if such exists
- The buffer
- The reservoir

insert: All new elements are inserted into the buffer

delete-min: If the minimum is in the main store a node is borrowed in the reservoir and the structure of the tree is re-established. The prefix-minimum pointers in the upper store are updated

delete: The node is swapped with its parent until it becomes a root, after which the procedure used in *delete-min* is followed

Multipartite binomial queue bounds

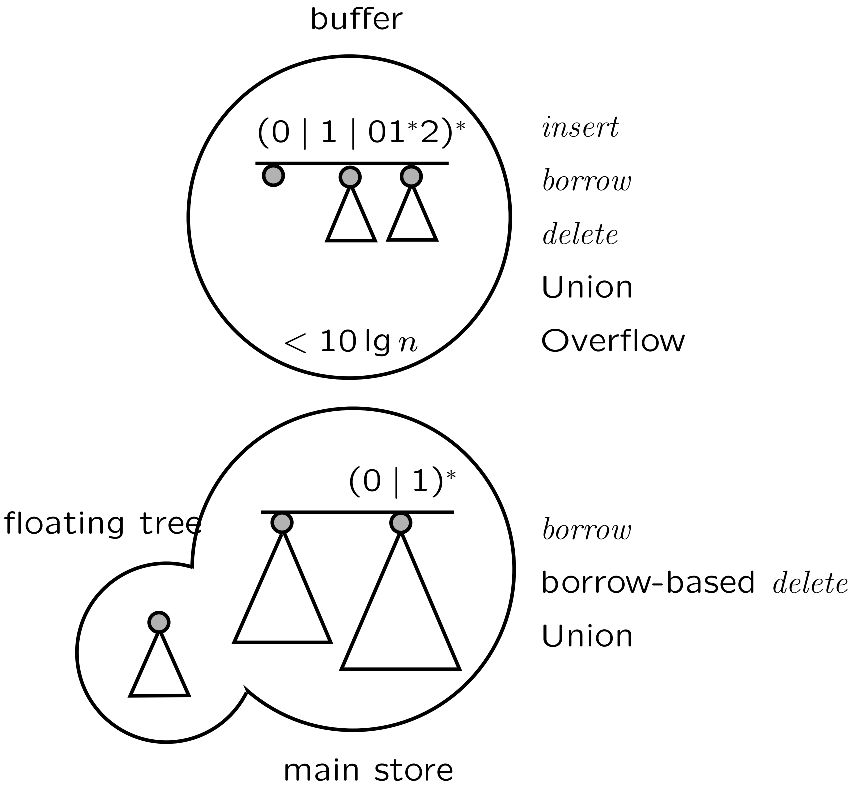
- **Multipartite binomial queue have the following worst-case comparison-complexity bounds for the operations:**

operations	**worst-case cost**
find-min, *insert*	$O(1)$
delete	$O(\lg n)$ (n size of data structure) $\lg n + O(1)$ element comparisons

Bipartite binomial queue

- A new result which is not in the thesis (a collaboration with Amr Elmasry and Jyrki Katajainen)
- Simplifies multipartite binomial queue
- All trees are binomial
- Now supports $meld$ at logarithmic worst-case cost

Bipartite binomial queue



buffer

$(0 \mid 1 \mid 01^*2)^*$

$< 10 \lg n$

insert

borrow

delete

Union

Overflow

floating tree

$(0 \mid 1)^*$

main store

borrow

borrow-based *delete*

Union

Bipartite binomial queue bounds

- **Bipartite binomial queue have the following worst-case comparison-complexity bounds for the operations:**

operations	**worst-case cost**
find-min, *insert*	$O(1)$
meld	$O(\lg n)$ (m and n are the sizes of data structures and $m < n$) $\lg n + O(\lg \lg n)$ element comparisons
delete	$O(\lg n)$ (n size of data structure) $\lg n + O(1)$ element comparisons

Two-tier relaxed heaps - Relaxed heaps (paper three)

- A relaxed binomial tree is a almost heap-ordered binomial tree
- Some nodes can be marked active indicating that there may be a heap-order violation
- A relaxed heap supports *decrease* and the number of active nodes is at most $\lfloor \lg n \rfloor$ (Driscoll, Gabow, Shrairman og Tarjan 1988)
- A **singleton** is an active node which has no active siblings
- A **run** is a sequence of consecutive active siblings
- The number of active nodes can be reduced using violation-reducing transformations

Two-tier relaxed heaps

- **Two-tier relaxed heaps consist of the following two components:**

Upper store: This is a modified relaxed heap whose nodes contain pointers to the following nodes in the lower store:

  - current roots and current active nodes
  - former roots and former active nodes which are only marked for deletion in the upper store

Lower store: This is a modified relaxed heap containing the elements

Zeroless regular system

- **To support *insert* and *extract* at worst-case constant cost, a zeroless regular number system is used**

Digit set: $d_i \in \{1, 2, 3, 4\}$

Extreme digits: $d_i \in \{1, 4\}$

Weight: $w_i = 2^i$

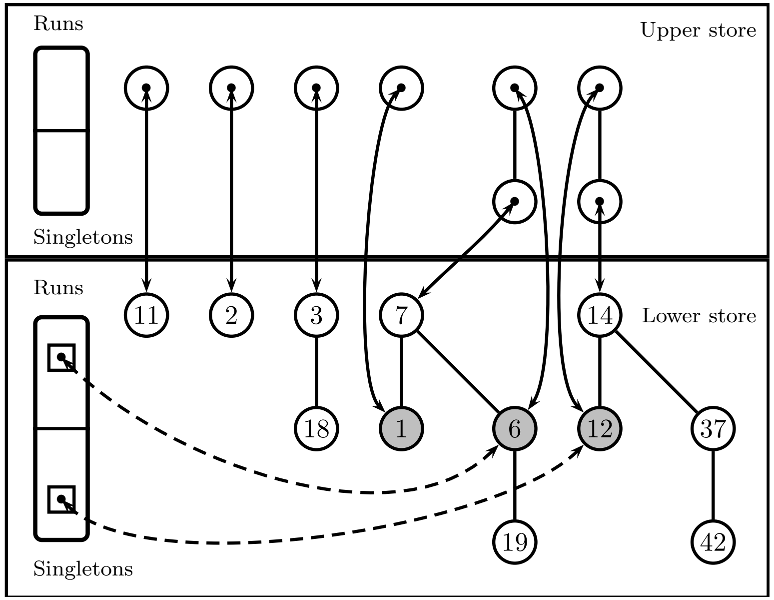
Regularity: Between any two digits equal to 4 there is a digit other than 3, and between any two digits equal to 1 there is a digit other than 2, except when one of the digits equal to 1 is the most significant digit

Two-tier relaxed heaps

- The upper store uses lazy deletions (marking) when, a join is done or an active node is made non-active in the lower store, as a normal deletion would be too expensive
- Incremental global rebuilding is used to remove the markings when the number of marked nodes become to large

Two-tier relaxed heaps

- **A two-tier relaxed heap storing 12 integers**



Two-tier relaxed heaps operations

find-min: Use a minimum pointer in the upper store

insert, *extract*: Utilizes the zeroless number system

decrease: Make the node active, after which violation-reducing transformations may be used to reduce the number of active nodes

delete: Borrow a node using *extract* and re-establish the structure of the tree

Two-tier relaxed heaps bounds

- **Two-tier relaxed heaps have the following worst-case comparison-complexity bounds for the operations:**

operations	**worst-case cost**
find-min, *insert*, *extract*, *decrease*	$O(1)$
meld	$O(\min\{\lg m, \lg n\})$ (m and n are the sizes of data structures)
delete	$O(\lg n)$ (n size of data structure) $\lg n + O(\lg\lg n)$ element comparisons

Pruned binomial queue (paper four)

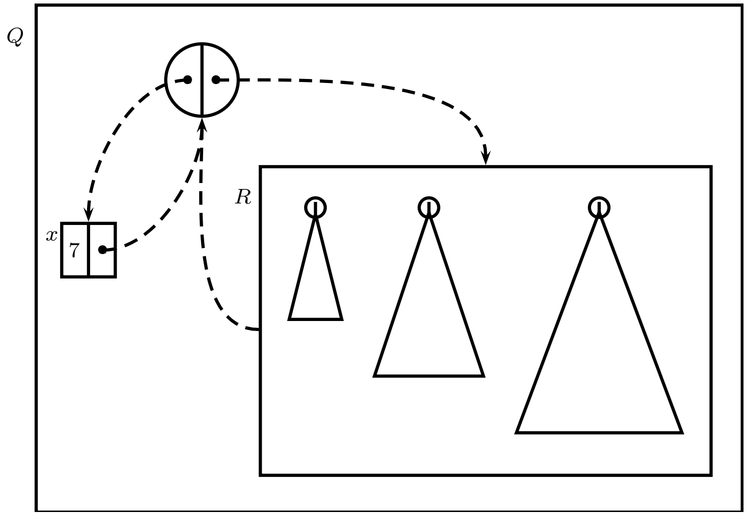
- A priority queue where structural violations are used instead of heap-order violations
- The bounds are obtained by mimicking heap-order violations using a shadow structure
- A violating node is replaced with a placeholder node and moved to the shadow structure
- Using structural violations it is possible to obtain worst-case comparison-complexity bounds comparable to those obtained using heap-order violations in two-tier relaxed heaps

Meldable heaps relying on bootstrapping (paper five)

- Use a binomial heap that supports $insert$ at constant worst-case cost and $meld$ at logarithmic worst-case cost
- Modify the binomial heap using data-structural bootstrapping
- Results in a structure where binomial heaps contain binomial heaps

Meldable heaps relying on bootstrapping

- **A simplified view of a bootstrapped heap**



Meldable heaps relying on bootstrapping bounds

- **The bounds for a meldable heaps relying on bootstrapping**

operations	worst-case cost
find-min, *insert*, *meld*	$O(1)$
delete	$O(\lg n)$ (n size of data structure) $3 \lg n + O(1)$ element comparisons

Strictly-regular number system (paper six)

Digit set: $d_i \in \{0, 1, 2\}$

Strict regularity: Every sting has the form $\left(1^+ \mid 01^*2\right)^*\left(\varepsilon \mid 01^+\right)$

Extreme digits: 0 and 2

Weight: $w_i = 2^i$

Increment

***fix-carry*$(\boldsymbol{d}, i)$:** Assert that $d_i \geq 2$. Perform $d_i \leftarrow d_i - 2$ and $d_{i+1} \leftarrow d_{i+1} + 1$

Algorithm ***increment*$(\boldsymbol{d}, i)$:**

```
$d_i \leftarrow d_i + 1$
Let $d_b$ be the first extreme digit before $d_i$, $d_b \in \{0, 2, undefined\}$
Let $d_a$ be the first extreme digit after $d_i$, $d_a \in \{0, 2, undefined\}$
if $d_i = 3$ or $(d_i = 2$ and $d_b \neq 0)$
    $fix\text{-}carry(\boldsymbol{d}, i)$
else if $d_a = 2$
    $fix\text{-}carry(\boldsymbol{d}, a)$
```

Decrement

fix-borrow$(\boldsymbol{d}, i)$: Assert that $d_i \leq 1$. Perform $d_{i+1} \leftarrow d_{i+1} - 1$ and $d_i \leftarrow d_i + 2$

Algorithm *decrement*$(\boldsymbol{d}, i)$:

```
Let $d_b$ be the first extreme digit before $d_i$, $d_b \in \{0, 2, undefined\}$
Let $d_a$ be the first extreme digit after $d_i$, $d_a \in \{0, 2, undefined\}$
if $d_i = 0$ or ($d_i = 1$ and $d_b = 0$ and $i \neq r - 1$)
    fix-borrow$(\boldsymbol{d}, i)$
else if $d_a = 0$
    fix-borrow$(\boldsymbol{d}, a)$
$d_i \leftarrow d_i - 1$
```

Other operations

$cut(\boldsymbol{d}, i)$**:** Cut $rep(\boldsymbol{d})$ into two strings having the same value as the numbers corresponding to $\langle d_0, d_1, \ldots, d_{i-1} \rangle$ and $\left\langle d_i, d_{i+1}, \ldots, d_{k-1} \right\rangle$. Transform $\left\langle d_i, d_{i+1}, \ldots, d_{k-1} \right\rangle$ into a strictly-regular form, if necessary

$concatenate(\boldsymbol{d}, \boldsymbol{d}')$**:** Concatenate $rep(\boldsymbol{d})$ and $rep(\boldsymbol{d}')$ into one string that has the same value as $\left\langle d_0, d_1, \ldots, d_{k-1}, d'_0, d'_1, \ldots, d'_{k'-1} \right\rangle$. Transform $\left\langle d_0, d_1, \ldots, d_{k-1}, d'_0, d'_1, \ldots, d'_{k'-1} \right\rangle$ into a strictly-regular form, if necessary

$add(\boldsymbol{d}, \boldsymbol{d}')$**:** Construct a string $\boldsymbol{d}''$ of strictly-regular form such that $value(\boldsymbol{d}'') = value(\boldsymbol{d}) + value(\boldsymbol{d}')$

Application: Meldable priority queues

- **Meldable priority queues using the strictly-regular number system have the following bounds:**

operations	worst-case cost
find-min, *insert*, *meld*	$O(1)$
delete	$O(\lg n)$ (n size of data structure) $2\lg n + O(1)$ element comparisons

Two new transformations to construct double-ended priority queues (paper seven)

- **A double-ended priority queue can extend the set of operations supported by a priority queue by the following operations:**

find-max(Q). Returns a reference to a node containing a maximum element of Q

delete-max(Q). Removes a maximum element and the node in which it is contained from Q

First transformation

- A special pivot element is used to partition the elements of the double-ended priority queue into three collections
- The three collections (maintained as priority queues) contain the elements smaller than, equal to, and larger than the pivot element
- Using this partition of the elements, we can delete an element only touching one priority queue
- To maintain the partitioning balanced the data structure is rebuild after a linear number of operations
- The rebuilding is done incrementally to obtain worst-case bounds

Application of first transformation

- **Using the first transformation together with the multipartite binomial queue the following bounds can be obtained:**

operations	**worst-case cost**
find-min/*find-max*, *insert*, *extract*	$O(1)$
delete	$O(\lg n)$ (n size of data structure) $\lg n + O(1)$ element comparisons

Second transformation

- Use the total correspondence approach
- The fact that the underlying priority queue supports $insert$, $extract$, and $decrease$ ($increase$) at $O(1)$ cost
- The transformation replaces the two priority queue $delete$ operations used in the standard total correspondence approach with one $delete$ operation and some operations having $O(1)$ cost

Application of second transformation

- **Utilizing the fact that *insert*, *extract*, and *decrease* (*increase*) in two-tier relaxed heaps has a constant worst-case cost the following bounds are obtained:**

operations	**worst-case cost**
find-min/*find-max*, *insert*, *extract*	$O(1)$
meld	$O(\min\{\lg m, \lg n\})$ (m and n are the sizes of data structures)
delete	$O(\lg n)$ (n size of data structure) $\lg n + O(\lg\lg n)$ element comparisons

Main results

- We devised a priority queue that for *find-min*, *insert*, and *delete* has a comparison-complexity bound that is optimal up to the constant additive terms, while keeping the worst-case cost of *find-min* and *insert* constant
- We introduced a priority queue that for *delete* has a comparison-complexity bound that is constant-factor optimal (i.e. the constant factor in the leading term is optimal), while keeping the worst-case cost of *find-min*, *insert*, and *decrease* constant
- We described two new data-structural transformations to construct double-ended priority queues from priority queues
- We introduced three new number systems

In total, we introduced seven priority queues, two double-ended priority queues, and three number systems.