A PRIORITY QUEUE WITH THE WORKING-SET PROPERTY

AMR ELMASRY
Department of Computer Engineering and Systems
Alexandria University, Alexandria, Egypt
elmasry@alexeng.edu.eg

ABSTRACT
A new priority queue is introduced, for which the cost to insert a new item is constant and the cost to delete the item $x$ with the minimum value is $O(\log k_x)$, where $k_x$ is the number of items that are inserted after $x$ and are still in the heap when $x$ is deleted. We achieve the above bounds in both the amortized case and the worst case.

Keywords: Data structures; binomial queues; distribution-sensitive algorithms; adaptive sorting; amortized analysis

1. Introduction
A data structure is called distribution-sensitive if the time taken by the structure to perform an operation varies according to the distribution of the input sequence. Though having upper bounds on the running time of different operations over all possible sequences, these structures perform better for some sequences than others. In order to characterize such structures, several properties are introduced describing their behavior.

Following finger trees [10], splay trees [17] is the classical example of a distribution-sensitive structure. Most of the known distribution-sensitive properties were introduced either as theorems or conjectures characterizing the performance of splay trees. With a special interest with respect to our structure, the working-set property for search trees is stated as follows [17]: The time spent to search item $x$ in a search tree is $O(\log w_x)$, where $w_x$ is the number of distinct items that have been accessed since $x$’s last access. Informally, in a data structure with the working-set property accesses to items recently accessed are faster than accesses to items that have not been accessed in a while.

Though originally formulated for the use of analyzing dictionaries, some of these properties have been applied to other structures, such as priority queues [11, 12]. Iacono [11] proved that if the minimum item $x$ in a pairing heap [9] of maximum size $n$ is to be removed, and $t_x$ heap operations have been performed since its insertion, the minimum-deletion operation takes amortized time $O(\log \min(n, t_x))$. Another distribution-sensitive priority queue is the funnel heap [2], a cache oblivious priority queue that supports minimum-deletion in amortized time $O(\log \min(n, t'_x))$, where $t'_x$ is the number of items inserted during the lifespan of $x$. Because of the similarity
between these definitions and the working-set property, we call this property the weak working-set property for priority queues (for both definitions of \( t_x \) and \( t'_x \)).

Iacono and Langerman [13] introduced the queueish property. The queueish property implies that an access to an item is fast if it is one of the least recently accessed items. Formally, a dictionary structure is said to be queueish if the time to search item \( x \) is \( O(\log (n - w_x)) \). They showed that no search tree can have this property. A priority queue is said to be queueish if the amortized cost per insertion is constant, and the amortized cost per minimum-deletion of \( x \) is \( O(\log q_x) \), where \( q_x \) is the number of items that are inserted before \( x \) and are still in the heap at the time of \( x \)'s removal. They introduced a priority queue, the \textit{queap}, having the queueish property.

In this paper, we introduce a new distribution-sensitive priority queue based on the well-known binomial queues. Let \( k_x \) denote the number of items that are inserted during the lifespan of \( x \) and are still in the heap when \( x \) is deleted. For our new priority queue, the cost per insertion is constant and the cost per minimum-deletion of \( x \) is \( O(\log k_x) \). We call this property the strong working-set property, which implies the weak working-set property (note that \( k_x \leq t'_x \leq t_x \)). We may also call this property the stack-like property, in analogy to the queueish property.

In contrast with the queap, we achieve the distribution-sensitive behavior in the worst case in addition to the amortized case. This result is an improvement over that appearing in a preliminary version of the paper [8].

Using our priority queue, a sequence of \( n \) items can be adaptively sorted as follows. A priority queue is built in \( O(n) \) by repeatedly inserting the \( n \) items in reverse order. By repeatedly deleting the minimum item from the queue, we get a sorted sequence of the input. Adding up the time for the \( n \) insertions and \( n \) minimum deletions, the time spent to sort such a sequence is \( O(\sum_{i=1}^{n} \log k_{x_i} + n) \leq O(n \log \frac{1}{n} \sum_{i=1}^{n} k_{x_i} + n) \), where \( \sum_{i=1}^{n} k_{x_i} \) is the number of inversions in the input sequence, resulting in an asymptotically optimal adaptive sorting algorithm [10, 14].

The paper is organized as follows: In the next section, we review the binomial queue operations. Section 3 is an informal discussion to the problems that motivate the way we implement our structure. In Section 4, we describe how to achieve the claimed bounds in the amortized sense. Then, we show how to achieve these bounds in the worst case in Section 5.

2. Binomial queues

A binomial tree [3, 19] of rank (height) \( r \) is constructed recursively by making the root of a binomial tree of rank \( r - 1 \) the leftmost child of the root of another binomial tree of rank \( r - 1 \). A binomial tree of rank 0 is a single node. The following properties follow from the definition:

- A binomial tree of rank \( r \) has \( 2^r \) nodes.
- The root of a binomial tree of rank \( r \) has \( r \) subtrees each of which is a binomial tree, having respective ranks 0, 1, \ldots, \( r - 1 \) from right to left.
To represent a set of \( n \) items, where \( n \) is not necessarily a power of 2, we use a forest having a tree of height \( i \) whenever the binary representation of the number \( n \) has a 1-bit in the \( i \)-th position. A binomial queue is such a forest with the additional constraint that every node contains a data value smaller than or equal to those stored in its children.

Each binomial tree within a binomial queue is implemented using the following standard implementation. Every node has two pointers: one pointing to its left sibling and the other to its leftmost child. The sibling pointer of the leftmost child points to the rightmost child forming a circular list. Given a pointer to a node, both its rightmost and leftmost children can be accessed in constant time. The list of its children can be sequentially accessed from right to left. The roots of the binomial trees within a binomial queue are organized in a doubly-linked list, which is referred to as the root-list. The ranks of the roots strictly increase as the root list is traversed from right to left.

Two binomial trees of the same height can be merged in constant time, by making the root of the tree that has the larger value the leftmost child of the other root. The following operations are defined on binomial queues:

- **insert.** The new item is added to the forest as a tree of rank 0, and successive merges are performed until there are no two trees of the same rank. (This is equivalent to adding 1 to the number in the binary representation.)

- **delete-minimum.** The root with the smallest item is found and removed, thus leaving all the subtrees of that item as independent trees. Trees of equal ranks are then merged until no two trees of the same rank remain.

For an \( n \)-node binomial queue, the worst-case cost per insert or delete-minimum is \( O(\log n) \). The amortized cost per insert is \( O(1) \).

### 3. Discussion

We build on the notion of binomial queues targeting a distribution-sensitive implementation. Denote our queue structure by \( Q \). We call the sequence of items obtained by a pre-order traversal of the nodes of \( Q \) the corresponding sequence of \( Q \) and denote it by \( \text{Pre}(Q) \). Our traversal gives precedence to the trees of \( Q \) in a right-to-left order. Also, the precedence ordering of the subtrees of a given node proceeds from right to left. When a new item is inserted, a single-node tree is added as the rightmost tree in the queue. Hence, a newly inserted item is appended as the first item in \( \text{Pre}(Q) \). Let \( d_x \) be the number of items preceding \( x \) in \( \text{Pre}(Q) \). Our goal is to maintain the order in which the items are input to the heap. What we are looking for is a set of operations that maintain the relation \( d_x \leq k_x \) (\( k_x \) denote the number of items that are inserted during the lifespan of \( x \) and are still in the heap), at the moment when \( x \) is to be deleted from \( Q \). Consequently, our target would be to delete \( x \) in \( O(\log d_x) \).

The first problem we face, as a result of an insertion, is when two trees with the same rank are merged such that the root of the tree to the right is larger than the root of the tree to the left. As a result, the root of the right tree is linked as the
leftmost child of the root of the left tree. This case immediately affects the order in which the items are input with respect to \( Pre(Q) \). To keep track of this order, we add an extra bit to each node of the queue, and call it the reverse bit. When a node is linked to its left sibling, the reverse bit of this node is set to 1 indicating what is called a rotation. See [6, 7] for a similar notion. In other words, the reverse bit of a node reflects the order of the subtree rooted at this node when it was linked below its parent.

The next problem is with respect to the delete-minimum operation. When the root with the minimum value is deleted from a binomial queue its subtrees are scattered according to their ranks and merged with other subtrees, again affecting the order in which items are input. The way to solve this problem is that when the root with the minimum value is deleted we reconstruct its tree in time logarithmic with respect to its size. This requires a relaxation to the properties of the standard binomial trees to allow trees to lose some nodes. This relaxation permits such restructuring in the claimed time bound. We introduce this tree structure, which we call \((2, 3)\) binomial trees, in the next section.

We are not on the safe side yet; consider the case when the root \( x \) of a tree \( T \) of rank \( r_1 \) is the minimum node that is required to be deleted from the heap, such that the rank of the tree to the right of \( T \) is \( r_2 \), where \( r_1 \gg r_2 \). The actual cost required by this delete-minimum operation can be implemented to be in \( \Theta(r_1) \), which is not comparable to \( r_2 = O(\log d_x) \). Our solution towards the claimed amortized cost is to perform several split operations on \( T \). The split operation is, in a sense, the opposite of the merge operation. A binomial tree is split into two binomial trees, by cutting the leftmost subtree of the given tree and adding it to the root-list either to the left or to the right of the rest of the tree depending on the value of the reverse bit. As a result, there will be, instead of \( T \), several trees whose ranks are in the range from \( r_1 \) to \( r_2 \). The idea is to reduce such gaps among the ranks of adjacent roots in the root-list in order to reduce this extra cost for subsequent delete-minimum operations. For the worst-case bounds, we impose an extra constraint on the ranks of the trees of the queue to prevent the above situation from popping up. In particular, we should have either \( r_1 = r_2 + 1 \) (no gaps) or \( r_1 = r_2 + 2 \) (only a gap of 1 rank, i.e. there is no tree of rank \( r_2 + 1 \)).

Having two trees with the same rank is not permitted in the standard implementation of binomial queues. In our new structure, we allow the existence of at most two trees of any rank. This is similar to using a redundant binary representation. The redundant number system has the base two, but in addition to using zeros and ones we are allowed to use twos as well. Any number can be represented using this number system (see [4, 5, 16]). The usage of a redundant number representation is crucial to achieve the required bounds. Consider the usage of the normal binary number representation instead, with the following nasty situation. Suppose that the size \( n \) of the entire structure is one less than a power of two, and suppose that we have a long alternating sequence of insert and delete-minimum, such that every time the inserted item is the smallest item that will be immediately deleted afterwards. In such case, each insert operation requires \( \log_2 (n + 1) \) merges. The claimed
bounds for our structure imply that both the insertion and the minimum-deletion operations have to be implemented in constant time, which is not achievable with the normal binary number representation. It is the savings of the carry operations in the redundant binary representation that makes our data structure more efficient, achieving the claimed bounds.

4. The data structure

We introduce \((2,3)\) binomial trees, as an alternative to binomial trees.

\((2,3)\) binomial trees. The children of the root of a \((2,3)\) binomial tree of rank \(r\) are \((2,3)\) binomial trees. There are one or two children having each of the respective ranks \(0,1,\ldots,r-1\). The number of these children is, therefore, between \(r\) and \(2r\) inclusive. The ranks of these children form a non-decreasing sequence from right to left. A \((2,3)\) binomial tree of rank 0 is a single node.

Lemma 1 The rank of an \(n\)-node \((2,3)\) binomial tree is at most \(\lfloor \log_2 n \rfloor\).

Proof. We prove the lemma by induction. The fact that a single-node tree has rank 0 establishes the base case. Let \(r\) be the rank of an \(n\)-node \((2,3)\) binomial tree. Using the induction hypothesis, \(n \geq 1 + 2^0 + 2^1 + \ldots + 2^{r-1} = 2^r\).

In our priority queue we use \((2,3)\) binomial trees in place of the traditional binomial trees, and may have zero, one or two \((2,3)\) binomial trees with the same rank. The order of the roots of the trees is important within the root-list. The ranks of these roots form a non-decreasing sequence from right to left. Let \(n_r \in \{0,1,2\}\) be the number of trees of the queue with rank \(r\).

We maintain the following invariants defining a total order on the set of the heap items according to the relation defined by the time these items were inserted:

Invariant 1. If a tree \(T\) is to the right of a tree \(T'\), then the items of \(T\) were inserted after the items of \(T'\).

Invariant 2. If the reverse bit of a non-root node \(x\) is set to one, then the items of the subtree rooted at \(x\) were inserted after the items of the subtrees of all the right siblings of \(x\) as well as the item at \(x\)'s parent. Otherwise, if the reverse bit of \(x\) is set to zero, then the items of the subtree of \(x\) were inserted before these items.

The following five procedures are used as subroutines to implement the insert and delete-minimum priority-queue operations:

merge. Given two or three \((2,3)\) binomial trees of the same rank \(r\), the given trees are merged into one tree of rank \(r+1\) by making the root with the smallest value the root of the resulting tree and the other one or two roots are linked to it as its leftmost children. The reverse bits of the linked roots are set appropriately to maintain the above ordering invariant. The merge operation takes constant time. (Note that we used the name \((2,3)\) trees because of the fact that we can merge two or three such trees of rank \(r\) to form one tree of rank \(r+1\).)
split. A \((2,3)\) binomial tree \(T\) of rank \(r + 1\) can be split into two or three trees of rank \(r\), by cutting the one or two subtrees of the root of \(T\) having rank \(r\). These trees together with the rest of \(T\) compromise the resulting trees. The reverse bits of the roots of the resulting trees are checked, and the position of these trees with respect to each other is decided accordingly to maintain the above ordering invariant. The split operation takes constant time, and no comparisons are needed.

reconstruct-tree. Given a \((2,3)\) binomial tree with a deleted root of rank \(r\), our purpose is to reconstruct the subtrees of this deleted root as a \((2,3)\) binomial tree.

Consider any iteration during the reconstruction procedure. Let \(x\) be the root of the rightmost subtree among the remaining subtrees. Let \(y\) be the left sibling of \(x\) and \(z\) be the left sibling of \(y\). Let \(r_x\), \(r_y\) and \(r_z\) be the ranks of \(x\), \(y\) and \(z\), respectively. If the reverse bit of \(y\) is 1, move the subtree of \(y\) to the right of the subtree of \(x\). Three cases are possible:

- If \(r_x = r_y - 1\): Split the subtree of \(y\). If the result of the split is two subtrees, merge these two subtrees with the subtree of \(x\) (this would be the only case that may result in this same scenario where \(r_x = r_y - 1\) for the next iteration). If the result of the split is three subtrees, these together with the subtree of \(x\) form four subtrees; merge the two right subtrees, and merge the two left subtrees setting the reverse bit of the root of the resulting subtree to 0.
- If \(r_x = r_y = r_z - 1\): Merge the subtree of \(x\) and the subtree of \(y\).
- If \(r_x = r_y = r_z\): If the reverse bit of \(z\) is 1, move the subtree of \(z\) to be the rightmost subtree. Merge the three subtrees into one subtree.

We repeat the above iteration until we get the final \((2,3)\) binomial tree, whose rank is either \(r\) or \(r - 1\). The time required by this reconstruction is \(O(r)\).

fill-gaps. Given a \((2,3)\) binomial tree of rank \(r_1\) such that the rank of the tree to its right is \(r_2\), where \(r_1 > r_2 + 1\), several split operations are performed on this tree. While the ranks of the trees resulting from the splits are greater than \(r_2 + 1\), a split is repeatedly performed on the right tree among these trees. As a result, there will be at most one tree of rank \(r_1\) (if there was two before this procedure), one or two trees of each of the ranks \(r_1 - 1, r_1 - 2, \ldots, r_2 + 2\), and two or three trees of rank \(r_2 + 1\). If there are three trees of rank \(r_2 + 1\), the left two of these three trees are merged to form a tree of rank \(r_2 + 2\). This violation may propagate while performing such merge operations, until there are no three trees of the same rank; a case that is ensured to be fulfilled if the result of the merge is a tree of rank \(r_1\). As a final result of this procedure, there will be at most two trees of rank \(r_1\), one or two trees of each of the ranks \(r_1 - 1, r_1 - 2, \ldots, r_2 + 1\). The time required by this procedure is \(O(r_1 - r_2)\).
**maintain-minimum.** After deleting the minimum node we need to keep track of the new minimum. Checking the values of the roots of all the trees in the queue leads to a $\Theta(\log n)$ cost for the delete-minimum operation, where $n$ is the size of the queue. Instead, a pointer is used with every root in the root-list pointing to the node with the smallest value among the nodes to the left of this root, including itself. We call these pointers the *prefix-minimum* pointers of the roots. In other words, the prefix-minimum pointer of a root $x$ points to the node with the minimum value between either $x$ or the node pointed to by the prefix-minimum pointer of the left sibling of $x$. The prefix-minimum pointer of the rightmost root points to the root with the overall minimum value. After deleting the minimum node and reconstructing its tree, maintaining the affected pointers (the pointers to the right of the deleted root) can be done from left to right. If the rank of the deleted root is $r$, the number of the affected pointers is at most $2(r+1)$ (there may be two trees of each possible rank value).

**insert.** The new item is added as the rightmost single-node tree with rank 0, and successive merges are performed until there are no three trees of the same rank. More specifically, among any three trees with the same rank, the two leftmost trees are merged and the root of the resulting tree replaces these two trees in the root-list. The prefix-minimum pointers of the affected nodes are maintained. These are the roots to the right of the root whose tree was merged last (and includes this node). As above, maintaining these pointers is performed by traversing these roots from left to right and updating the pointers accordingly. The number of comparisons performed to maintain such pointers is proportional to the number of merges performed.

**delete-minimum.** First, the root of the tree $T$ with the minimum value is removed (we use the prefix-minimum pointer of the rightmost root to identify the tree $T$). This is followed by a reconstruction of $T$ as a $(2,3)$ binomial tree with one less node. Let $t$ be the new root of $T$. Let $u$ be the right sibling of $t$ and $v$ be the right sibling of $u$ (special cases where $v$ or both $u$ and $v$ do not exist are easier to handle). Let $r_t$, $r_u$ and $r_v$ be the ranks of $t$, $u$ and $v$, respectively. If as a result of this deletion the rank of $T$ decreases by one then:

- If $r_t > r_u$ or $r_t = r_u > r_v$: Do nothing.
- If $r_t = r_u = r_v$: Merge $T$ with the tree rooted at $u$.
- If $r_t = r_u - 1$: Split the tree of $u$ to two or three trees of rank $r_t$. If the split results in two trees, merge $T$ with these two trees. If the split results in three trees, merge $T$ with the leftmost tree among the trees resulting from the split and merge the other two trees together.

If $r_t > r_u + 1$, a fill-gaps procedure is performed on $T$. The final step is to perform a maintain-minimum that fixes the affected prefix-minimum pointers to keep track of the new minimum. The time required by this procedure is $O(r_t)$.
Theorem 1  Starting with an empty distribution-sensitive binomial queue, the am-

torized cost per insert is \( O(1) \), and that per delete-minimum of an item \( x \) is \( O(\log k_x) \).
The worst-case cost of these operations is \( O(\log n) \).

Proof. The worst-case cost follows from the way the operations are implemented
and the fact that both the number of trees and their distinct ranks are \( O(\log n) \).

We use a potential function [18] to derive the amortized bounds. For each
possible rank value \( r \leq r_{\text{max}} \), where \( r_{\text{max}} \) is the maximum rank of a tree in the
queue, \( n_r \in \{0, 1, 2\} \) is the number of trees of the queue with rank \( r \). After the
\( i \)-th operation, for a given \( j \in \{0, 1, 2\} \), let \( N_j^i = | \{ r : n_r = j, \forall r \leq r_{\text{max}} \} | \). In other words, let \( N_j^i \) be the number of rank values that are not represented by
any trees, \( N_j^i \) be the number of rank values that are represented by one tree, and
\( N_j^i \) be the number of rank values that are represented by two trees. Let \( \Phi^i \) be the
potential function, such that \( \Phi^i = c_1 N_0^i + c_2 N_2^i \), where \( c_1 \) and \( c_2 \) are constants to be
determined. The value of \( \Phi^0 \) is 0.

First, assume that the operation \( i + 1 \) is an insert operation that involved \( t \)
merges. If as a result of this insertion two trees with the same rank are merged, then
there should have been two trees with this rank before the insertion and only one
remains after the insertion. This implies that \( N_2^i+1 - N_2^i \leq -t+1 \) and \( N_0^i+1 - N_0^i \leq 0 \).
Hence, the amortized cost is bounded by \( O(t) - c_2 t + c_2 \). By selecting \( c_2 \) greater
than the constant involved in the \( O() \) notation in this relation, the amortized cost
of the insertion is at most \( c_2 \).

Next, assume that the operation \( i + 1 \) is a delete-minimum performed on the root
\( x \) of a tree \( T \) whose rank is \( r_1 \). The actual cost is \( O(r_1) \). Let \( r_2 \) be the rank of the tree
to the right of \( T \) before the operation is performed. As a consequence of Invariant 1,
the number of nodes of this tree is upper-bounded by \( k_x \), implying \( r_2 = O(\log k_x) \).
As a result of the fill-gaps procedure: \( N_0^i+1 - N_0^i \leq -(r_1 - r_2 - 2) \) and \( N_2^i+1 - N_2^i \leq
r_1 - r_2 - 1 \). Hence, the amortized cost is bounded by \( O(r_1) - (c_1 - c_2)(r_1 - r_2 - 1) + c_1 \).
By selecting \( c_1 \) such that \( c_1 - c_2 \) is greater than the constant in the \( O() \) notation
in this relation, the amortized cost of the delete-minimum is \( O(r_2) \leq O(\log k_x) \). \( \square \)

5. The structure with the worst-case bounds

To implement the insert operation in a worst-case constant cost, techniques
similar to those used for the implicit priority queue of Carlsson et al. [4] and the
partial rebuilding of Overmars [15] are used. After adding a new node of rank 0, we
cannot afford to perform all the necessary merges at once. Instead, we do a constant
number of merges with each insertion. Hence, merges will be left partially completed
until the next operation. To accommodate this, we need to allow the number of
trees \( n_r \) with a given rank \( r \) to be at most 3 instead of 2, i.e. \( n_r \in \{0, 1, 2, 3\} \).

With every insertion, we make progress on the merge of the leftmost two trees
among the three smallest trees with the same rank. We maintain a logarithmic
number of pointers to the merges in progress and their structures, kept as a stack of
pointers. More closely, each pointer points to a root among a triple of roots having
the same rank. In one merge step: the pointer at the top of the stack is popped,
the two roots are removed from the root-list, the corresponding trees are merged, and the root of the resulting tree is put in the place of the roots of the two merged trees. If it happens that there exist another two trees with the same rank as the resulting tree, a pointer indicating this triple is pushed onto the stack. If one merge is done in connection with every insert operation, the on-going merges are already disjoint meaning that none of these merges would result in producing a tree with the same rank as an already existing triple of roots. To illustrate this fact, we point out that the following regularity constraint, which guarantees that \( \forall r : n_r \leq 3 \), is maintained by the priority-queue operations. (If \( n_r = 3 \), merging two trees with rank \( r \) results in \( n_r = 1 \) and an increment of \( n_{r+1} \).)

\[
(\forall i, j : i < j \& n_i = n_j = 3) \implies (\exists k : i < k < j \& n_k \in \{0, 1\}).
\]

Following an observation made in [4], the size of the stack can be significantly reduced if two merge steps are executed with every insert operation, instead of one.

Consider the effect of performing the aforementioned delete-minimum operation. The number of trees \( n_r \) having any rank \( r \) cannot increase to 3. A possible increase of \( n_r \) from 1 to 2 will be accompanied with a decrease of \( n_{r+1} \) by 1. It follows that the above regularity constraint will never be violated as a result of a delete-minimum operation. If \( n_r \) decreases from 3 to 2, the pointer indicating such a triple is to be removed from the stack.

To implement the delete-minimum operation in the claimed worst-case bounds, we apply another regularity constraint on the ranks of the trees of the queue. (See [1, 10] for other regularity constraints used in different contexts.) The intuition is that the only reason for our bounds being in the amortized sense is the differences between the ranks of adjacent tree roots. Our amortized solution was to allow such differences in ranks to occur until a minimum-deletion takes place, where these differences in ranks between a tree and its right neighbor are filled by several splits of such tree. If we eliminate these gaps, the worst-case bounds are achieved and we do not need to apply the fill-gaps procedure accompanying a delete-minimum, which is responsible for the amortized bound. Following these ideas, in order to achieve the worst-case bounds, we impose the following regularity constraint:

\[
(\forall i, j : i < j \& n_i = n_j = 0) \implies (\exists k : i < k < j \& n_k \in \{2, 3\}).
\]

This constraint indicates that if \( n_i = 0 \) then both \( n_{i-1} \) and \( n_{i+1} \) are non-zeros. In other words, the difference between the ranks of two adjacent roots is at most 2. Maintaining this constraint immediately implies the claimed worst-case bounds.

The question is: when will this last constraint be violated? To answer this question, we denote by the rank-sequence the sequence of digits representing the number of trees for each of the possible ranks within the queue (including all integers starting from the minimum rank of a tree to the maximum rank of a tree) in the same order as they appear in the root-list. The constraint is violated once the rank-sequence has as a subsequence the string \( 01^*0 \), where \( 1^* \) means zero or more 1’s. There are two scenarios that result in the violation of the above regularity constraint.
The first scenario is when the rank-sequence has a subsequence the string 01*11, and a delete-minimum is performed on the tree whose rank is represented by the 1 next to the rightmost 1 in this subsequence causing the rank of this tree to decrease and resulting in the string 01*0 being a subsequence of the rank-sequence. The second scenario is when the rank-sequence has a subsequence the string 01*21*0, and as a result of performing a delete-minimum operation the rank of a tree decreases resulting in the string 011*0. Note that the regularity constraint will never be violated as a result of a merge that is performed in effect of an insertion and the existence of three trees having the same rank. (If \( n_r = 3 \), merging two trees with rank \( r \) results in \( n_r = 1 \) and an increment of \( n_{r+1} \).)

To maintain this regularity constraint, the tree to the left of the trees whose ranks are represented by the subsequence 01*0 is to be split. If this split results in three trees, the leftmost two of these trees are merged. If we start with the subsequence 101*0, we get either the subsequence 021*0 or 111*0 after the split. If we start with the subsequence 201*0, we get either the subsequence 121*0 or 211*0 after the split. If we start with the subsequence 301*0, we get either the subsequence 221*0 or 311*0 after the split. For all such sequences, the regularity constraint is restored.

One obstacle is to efficiently identify the tree to be split once the violation occurs. Our solution is that for every subsequence of adjacent trees whose ranks satisfy \( n_i = n_j \neq 1 \), and for \( n_k = 1 \) for \( i < k < j \), we maintain a block pointer associated with the tree of rank \( i + 1 \) and another associated with the tree of rank \( j - 1 \), where each such pointer points to the root of the other tree.

To check if a violation took place as a result of a delete-minimum, we consider the two scenarios that may result in the violation. For the first scenario when the root of a tree whose rank \( r \) decreases, we walk to the right from the root of the tree of rank \( r \), traversing the trees in the root-list, until we reach a tree with rank \( i + 1 \), where \( n_i \neq 1 \) (this cost is covered by the cost of the delete-minimum operation), use the block pointer of the tree of rank \( i + 1 \) to reach the tree with rank \( j - 1 \). If \( n_j = 0 \), this indicates a violation. In such case, we split the tree to the left of this tree (of rank \( j + 1 \)) as mentioned above. The block pointer of the tree of rank \( i + 1 \) is adjusted to point to the tree of rank \( r - 2 \), as \( n_{r-1} \) now equals 2 (if \( r = i + 2 \), make this block pointer point to null). The block pointer of the tree of rank \( r + 1 \) is adjusted according to the above case analysis resulting from fixing the violation. If \( n_j \geq 2 \), the constraint is not violated but we need to adjust the block pointers; the block pointer of the tree of rank \( i + 1 \) is made to point to the tree of rank \( r - 2 \), and the block pointer of the tree of rank \( r + 1 \) is made to point to the tree of rank \( j - 1 \). A similar approach is used for the second scenario that results in the constraint violation. In addition to maintaining the regularity constraint as above, we need to modify the block pointers to include the longer subsequence of consecutive 1’s.

Finally, with every other operation (merge or split) that does not violate the constraint, there may be a need to update the block pointers according to the resulting subsequence of rank values. This can be done in constant time per operation.
Acknowledgements

I would like to thank the anonymous referee for the rich and detailed comments that helped to improve the presentation of the paper.

References