DIMACS Technical Report 2004-39
August 2004

Distribution-Sensitive Priority Queues

by

Amr Elmasry
Computer Science Department, Alexandria University
Alexandria, Egypt
elmasry@alexeng.edu.eg

ABSTRACT

A new priority queue is introduced, for which the cost per insertion is constant and the cost per deletion or minimum-deletion of an item x is $O(\log k_x)$, where k_x is the number of items that are inserted during the lifespan of 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. Several applications of our structure are mentioned.

1 Introduction

A data structure is called distribution-sensitive if the asymptotic time bound 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, some structures may perform better for some sequences than others. This is analogous to a sorting algorithm running in $O(n \log n)$ for any sequence of length n, while performing better and running in $O(n)$ if the input is already sorted or inversely sorted.

In order to characterize such structures, several properties are introduced describing the behavior of these structures. These properties can be viewed as characterizations of distribution-sensitive behavior that give insights into the possibilities and limitations of these data structures. Relationships among such properties are introduced in [16], thus establishing a hierarchy of properties.

Following finger trees [14], splay trees [20] 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. Examples are: the static optimality theorem, the static finger theorem, the working-set theorem (all in [20]), the sequential access theorem [12, 21, 22], the dequeue theorem [21], and the dynamic finger theorem [5]. Each of these theorems describes a natural class of sequences of operations, and shows that the amortized cost [23] of performing any of these sequences on an n-node splay tree is $o(\log n)$ per operation. With a special interest with respect to our structure, we present the working-set property for search trees; 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 [15, 16]. Applying these properties to priority queues is more robust since the heap size and contents are allowed to dynamically change, as opposed to only analyzing access operations for search trees. Iacono [15] proved that if the minimum item in a pairing heap [13] of maximum size n is to be removed, and t heap operations have been performed since its insertion, the minimum-deletion operation takes amortized time $O(\log \min(n, t))$. Because of the similarity between this property and the working-set property, we call this property the weak working-set property for priority queues.

Iacono and Langerman [17] introduced the queueish property. The queueish property implies the complementary idea, which states that an access to an item is fast if it is one of the least recently accessed items. Formally, a data 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 a node x is $O(\log q_x)$, where q_x is the number of items that have been in the queue longer than x (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 *queap*, that has the queueish property.

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 a node 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 deletion or 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. 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 the result appeared in the preliminary version of this paper [11].

The paper is organized as follows: In the next section, we review how binomial queues, which we use as a basic structure for our new priority queue, is implemented. Section 3 is an informal discussion to the problems that motivate the way we implement our structure. In Section 4, we describe how to implement different priority queue operations to achieve the claimed bounds in the amortized sense. Then, we show how to achieve these bounds in the worst case in Section 5. Some of the possible applications are given in Section 6. We conclude with an improvement that achieves better multiplicative constants for the bounds on the number of comparisons involved in the delete and delete-minimum operations.

2 Binomial queues

A binomial tree [2, 24] 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:

- The rank of an n-node (assume n is a power of 2) binomial tree is $\log_2 n$.
- The root of a binomial tree with rank r has r sub-trees 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 elements, 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 binary representation. In such an 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. To support delete operation, each node in addition contains a pointer to its parent. 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 element 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 element is found and removed, thus leaving all the sub-trees of that element as independent trees. Trees of equal ranks are then merged until no two trees of the same rank remain.

delete. The key of the node to be deleted is repeatedly swapped with its parents up to the root of its tree. A delete-minimum is then performed to delete this node.

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

3 Discussion

Denote our queue structure by Q. We call the sequence of values obtained by a pre-order traversal of the nodes of Q the corresponding sequence of Q and denote it by $Pre(Q)$. Our traversal gives precedence to the trees of Q in a right-to-left order. Also, the precedence ordering of the sub-trees of a given node proceeds from right to left. Hence, a newly inserted element is appended as the first element in $Pre(Q)$. Let d_x be the number of elements preceding x in $Pre(Q)$. Our goal is to maintain the order in which the elements are input to the heap. What we are looking for is a set of operations that maintain the relation $d_x \leq k_x$, at the moment when x is to be deleted from Q. Consequently, our target would be to delete x in $O(\log d_x)$.

We build on the notion of binomial queues trying to obtain an implementation that is distribution-sensitive. When a new element is inserted, a single-node tree is added as the rightmost tree in the queue. The first problem we face as a result of this 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 tree to-the-right is linked as the leftmost child of the root of the tree to-the-left. This case immediately affects the order in which the elements are input. To keep track of this order, we add an extra bit to each node of the binomial 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 [9, 10] for a similar notion.

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 sub-trees are scattered according to their ranks and merged with other sub-trees, again affecting the order in which elements 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, that we call relaxed 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 sub-tree 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 nodes 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, 8, 19]. 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 element is the smallest element that will be immediately deleted afterwards. Each of the insert operations requires $\log_2(n+1)$ merges. The claimed bounds for our structure imply that both operations must 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 *relaxed binomial trees*, as an alternative to binomial trees.

Relaxed binomial trees. The children of the root of a relaxed binomial tree of rank r are relaxed 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 relaxed binomial tree of rank 0 is a single node.

lemma 1 *The rank of an n-node relaxed binomial tree is at most* $\log_2 n$*.*

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 relaxed binomial tree. Using the induction hypothesis, $n \geq 1 + 2^0 + 2^1 + \ldots + 2^{r-1} = 2^r$. □

We are now ready to describe our priority queue. We use relaxed binomial trees in place of the traditional binomial trees. Our priority queue may have up to two $(0, 1,$ *or* $2)$ relaxed 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 denote the number of trees in the queue with rank r, where $n_r \in \{0, 1, 2\}$. The following procedures are used to perform the priority queue operations:

merge. Given two relaxed binomial trees of the same rank r, the two trees are merged into one tree of rank $r+1$ by making the root with the larger value the leftmost child of the root of the other tree. If the right sibling is linked to its left sibling its reverse bit is set to 1, otherwise the reverse bit of the linked node (the left sibling) is set to 0. The merge operation takes constant time.

split. A relaxed binomial tree T of rank r can be split into two trees as follows. The first tree is the sub-tree of the leftmost child of the root of T, and the second tree is the rest of T. The rank of the first tree is $r-1$, and the rank of the second tree is either r or $r-1$ (depending on the rank of its current leftmost child). The reverse bit of the root of the first tree is checked, and the position of these two trees with respect to each other is decided accordingly. The split operation takes constant time, and no comparisons are needed.

reconstruct-tree. Given a relaxed binomial tree T with a deleted root of rank r, the purpose of this procedure is to reconstruct the sub-trees of this deleted root to maintain the structure of a relaxed binomial tree. The procedure starts by imposing that the two rightmost sub-trees have rank 0. If there is only one sub-tree with rank 0, we perform the following iterative step. In the general step, assume that before performing the iterative step, *split(i+1)*, there is no child of T with rank i. The rightmost sub-tree with rank $i+1$ is split, and three cases may take place depending on the ranks of the resulting two trees:

1. The left tree has rank $i+1$ and the right tree has rank i: Do nothing.

2. The left tree has rank i and the right tree has rank $i+1$: The right tree is split into two trees each with rank i (this is the only possibility for this second split). Now, there are three trees each with rank i. The two left trees are merged into one tree with rank $i+1$.

3. Both of the resulting two trees have rank i: If there was another tree of rank $i+1$, the iterative step terminates. If there was only one tree of rank $i+1$, there is none after the split. The iterative step is performed with no trees of rank $i+1$ (i.e. call *split(i+2)*).

To summarize, if there is only one sub-tree with rank 0, call *split*(1) and proceed with the iterative step. After performing such iterative step there will be two sub-trees of rank 0, and one or two sub-trees of ranks from 1 up to $r-1$ or $r-2$. We next perform a gluing step, which starts by merging the two sub-trees of rank 0 into a tree of rank 1, and merging this resulting tree with the one or two sub-trees of rank 1 to form a relaxed binomial tree of rank 2, and repeating this merging until we get the final relaxed binomial tree, whose rank is either r or $r-1$. (To merge three trees with the same rank, the two roots with the larger values are linked to the other roots and the reverse bits of the linked roots are set correctly.) The time required by this reconstruction is $O(r)$.

split?. A tree of rank r is split into two trees of rank $r-1$ if it had one child of rank $r-1$ before performing the split. Alternatively, it is split into three trees of rank $r-1$ if it had two children of rank $r-1$ before performing the splits.

fill-gaps. Given a relaxed 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 all the roots 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 node, including itself. We call these pointers the prefix minimum (*pm*) pointers of the roots. The pm pointer of the rightmost root points to the root with the overall minimum value. After deleting the minimum node, 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). A more efficient procedure that achieves $O(\log r)$ is given in Section 7.

insert. The new element 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. The merging must be done while maintaining the ordering of the elements. More specifically, if there are three trees with the same rank, the two leftmost trees are merged and the root of the resulting tree replaces the roots of these two trees in the root-list. The pm pointers of the affected nodes are maintained.

delete-minimum. First, the root of the tree T with the minimum value is removed. This is followed by a reconstruction of T as a relaxed 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 u or 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:

1. If $r_t = r_u > r_v$: Do nothing.
2. If $r_t = r_u = r_v$: Merge the tree of t with the tree of u.
3. If $r_t = r_u - 1$: Split the tree of u to two or three trees of rank r_t. Merge the tree of t with the leftmost tree among the trees resulting from the split. If there are more than two trees of rank r_t remaining, merge the leftmost two.

Next, a fill-gaps procedure is performed on T if necessary. The final step is to perform a maintain-minimum that fixes the affected pm pointers to keep track of the new minimum. The time required by this procedure is $O(r_t)$.

delete. If the node to-be-deleted is a root of a tree, perofrm a similar procedure to the delete-minimum on this node. Otherwise, let T be the sub-tree whose root is to be deleted. after the deletion of the root of T, a reconstruction of T to a relaxed binomial tree is performed. Let t be the new root of T. Let u be the right sibling of t, v be the right sibling of u and s be the left sibling of t (special cases where u, v or s do not exist are easier to handle). If, as a result, the rank of T decreases by one, the following iterative step, *low-rank(t)*, is performed.

1. If $r_s - 1 = r_t = r_u = r_v + 1$: Do nothing.
2. If $r_t = r_u = r_v$: Merge the tree of t with the tree of u.

3. If $r_t = r_u - 1$: Split the tree of u to two or three trees of rank r_t. Merge the tree of t with the leftmost tree among the trees resulting from the split. If there are more than two trees of rank r_t remaining, merge the leftmost two.

4. If $r_s - 2 = r_t = r_u = r_v + 1$: An iterative step is initiated by calling *split(r_s)*. If as a result the rank of the parent of t decreases by one, one of these cases needs to be called iteratively (i.e. call *low-rank(parent(t))*).

Let T' be the tree that contained the deleted node. A fill-gaps procedure is performed on T'. If necessary, a maintain-minimum is performed. If r is the rank of T', the actual time required by the delete is $O(r)$. This follows from the fact that after finishing the work on a node of rank r_2, the work done to move to its parent whose rank is r_1 is $O(r_1 - r_2)$, until we reach the root of T'.

Theorem 1 *Starting with an empty distribution-sensitive binomial queue, the amortized cost per insert is $O(1)$, and that per delete or 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, the fact that the rank of any tree is $O(\log n)$, and that the number of trees in the heap is $O(\log n)$.

We use a potential function [23] to derive the amortized bounds. For each possible rank value r of the trees of the queue, we have $n_r \in \{0, 1, 2\}$. After the i-th operation, for a given $j \in \{0, 1, 2\}$, let $N_j^i = || \{r \ : \ n_r = j, \ \forall r\} ||$. In other words, let N_0^i be the number of rank values that are not represented by any trees, N_1^i be the number of rank values that are represented by one tree, and N_2^i be the number of rank values that are represented by two trees. Let Φ^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 Φ^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 $N_2^{i+1} - N_2^i \le -t + 1$ and $N_0^{i+1} - N_0^i \le 0$. 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. The number of nodes of this tree is upper-bounded by k_x, implying $r_2 = O(\log k_x)$. This follows from the fact that all the items of the trees to the right of T must have been inserted after x; a fact that is a result of the way we implemented all our operations to preserve the ordering of the input items. As a result of the fill-gaps procedure: $N_0^{i+1} - N_0^i \le -(r_1 - r_2 - 2)$ and $N_2^{i+1} - N_2^i \le r_1 - r_2 - 1$. 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 $O(r_2) = O(\log k_x)$. The amortized bound on the delete operation follows similarly. □

5 The structure with the worst-case bounds

To implement the insert operation in a worst-case constant cost, the same techniques of the implicit priority queue of Carlsson et al. [4] can be 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 to pick up the work on the next operation. To facilitate this, we maintain a logarithmic number of pointers to the merges in progress and their structures, kept as a stack of pointers. With every insertion we make progress on the merge of the two smallest trees with the same rank. From the amortized analysis of binomial queues it is clear that we must do at least two comparisons per insertion, performing the merges.

To implement delete and delete-minimum in the claimed worst-case bounds, we apply more regularity constraints on the ranks of the trees of the queue. See [1, 14] for other regularity constraints used in different contexts. The intuition is that the only reason for our bounds being only in the amortized sense is the presence of gaps between the ranks of adjacent tree roots. Once we eliminate these gaps, the worst-case bounds are achieved. Our amortized solution was to allow such gaps to occur until a deletion takes place in a tree, where the gaps in ranks between this tree and its right neighbor are filled by several splits. For the worst-case solution, we impose the following regularity constraint:

$$\forall i, j \ : (i < j \ \& \ n_i = n_j = 0 \implies \exists k : i < k < j \ \& \ n_k = 2),$$

which implies 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 any two adjacent roots in the root-list is at most 2. Maintaining this constraint immediately results in achieving the claimed worst-case bounds. The question is: when will the above constraint be violated? If we write a sequence of ternary digits representing the ranks of the trees of the queue in the same order as they appear in the root-list, the constraint is violated once this sequence has as a subsequence 01^*0, where 1^* means zero or more 1's. Let i be the rank of the right 0, and j be the rank of the left 0 in the subsequence 01^*0 (i.e. $n_i = n_j = 0$ and $n_k = 1$, for $i < k < j$). This violation would only be a result of a tree whose rank was i and decreases to $i - 1$ in effect of a delete or a delete-minimum operation. To maintain the constraint, the tree to the left of the subsequence 01^*0 is to be split. Note that this tree has rank $j + 1$, since n_{j+1} must have been non-zero before this violation. If n_{j+1} was 1 before the violation, the subsequence 101^*0 results in either the subsequence 021^*0 or 111^*0 after the split. If n_{j+1} was 2 before the violation, the subsequence 201^*0 results in either the subsequence 121^*0 or 211^*0 after the split. (If as a result of the split the left tree has rank j and the right tree has rank $j + 1$, the right tree is split to two trees of rank j and the left tree is merged with the left tree resulting from this second split.) For each of the resulting sequences, the above constraint is restored. One obstacle is to efficiently identify the tree to be split (of rank $j + 1$) when the violation occurs while deleting a node from a tree of rank i. Our solution is that for every subsequence of adjacent trees whose ranks satisfy ($n_i = n_j \neq 1$ and $n_k = 1$, for $i < k < j$), we maintain a block pointer from the tree of rank $i + 1$ to the tree of rank j. First, to check if a violation took place as a result of a delete or a delete-minimum in a tree whose rank k decreases resulting in $n_k = 0$, we walk to the right from the root of rank k by the trees in the root-list until we reach a tree with rank i, where $n_i \neq 1$ (this cost is covered by the cost of the delete operation), use the block pointer from the tree of rank $i+1$ to reach the tree with rank j. 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. We adjust the block pointer of the tree of rank $i+1$ to point to the tree of rank $k-1$. The block pointer of the tree of rank $k+1$ is adjusted according to the above case analysis resulting from fixing the violation. We also need to adjust the block pointers with other delete or delete-minimum operations that reduce the ranks of the corresponding trees. Note that adjusting these block pointers after a delete or a delete-minimum operation requires only constant time, plus the time to traverse the root-list to the right to reach one such pointer (this cost is covered by the cost of the delete operation).

6 Applications

We expect our data structure to be useful for several applications, from which we mention some examples:

Adaptive sorting. Given a sequence of n elements, a distribution-sensitive priority queue is built in $O(n)$ by repeatedly inserting these elements. By repeatedly deleting the minimum node from the queue we get a sorted sequence of the input. The time spent to sort such a sequence is $O(n \log \frac{1}{n} \sum_{i=1}^{n} k_{x_i})$. If the elements are inserted in reverse order, $\sum_{i=1}^{n} k_{x_i}$ will be the number of inversions in the input sequence, and our algorithm is optimal [14, 18]. Our heap structure is more flexible since it allows interleaved insertions and minimum-deletions. Hence, it can be used in on-line adaptive sorting and order statistics.

Geometric applications. There are several geometric applications that require the usage of a heap structure. For example, in the *sweep-line* paradigm [6] the usage of a priority queue is essential. Our heap may be used if the events to be handled follow some specific distributions; a case where deleting the minimum of an n-node heap may require $o(\log n)$. The existence of some problems, where the geometric nature implies that the expected time that the inserted events spend in the heap before being deleted is small, needs to be investigated.

Discrete event simulation. e.g. future event set algorithms. In such applications a list of future events is to be maintained, and at every step the next occurring event in the list is processed inserting some new events. These events may follow some probability distribution, and hence their processing may be faster using our structure. For a survey on discrete event simulation, see [7].

7 Improving the constants

The multiplicative constant factor for the number of comparisons involved in the reconstruction of the tree of x in the $O(\log k_x)$ is 2, and that for maintain-minimum is 2, for a total of at most $4 \log_2 k_x + O(1)$ comparisons per delete or delete-minimum. Next, we sketch the way to implement maintain-minimum in $O(\log \log k_x)$, achieving an overall bound of $2 \log_2 k_x + O(\log \log k_x)$ for the number of comparisons. The roots of the trees are kept in another set of heaps, such that all the roots whose ranks are in the range from 2^i to $2^{i+1} - 1$, for possible integers i, are kept in the same heap. These heaps are arranged in an increasing order of their sizes, maintaining pm pointers involving the minimum node of each of these heaps from right to left. (The constant in the smaller terms may even be improved by having a hierarchy of levels of heaps instead of using the pm pointers at this level.) Deleting the minimum among these heaps takes $O(\log r)$ if the rank of the deleted node is r, implying a bound of $O(\log \log k_x)$. We need to maintain this set of heaps whenever the roots of the main trees change. This requires inserting and deleting such nodes in and from the heaps whenever necessary. Using our original analysis, it follows that the number of the main operations is an upper bound on the number of such heap operations. Our goal is to insert or delete an element in these heaps in $O(1)$. We can use any of the heap implementations that perform insert in $O(1)$ and delete-minimum in $O(\log n)$. We use a method of delayed deletions. Whenever a node needs to be deleted from this second level of heaps it is marked. Before inserting a new node, we first check if it already exists as a marked node, and hence unmarking it. Whenever the number of the marked nodes reaches half the total number of nodes in one of these heaps, this heap is rebuilt getting rid of the marked nodes. Achieving an $O(1)$ is possible for the deletion because of the nature of the application that ensures that a marked node will never become the minimum of a heap before being reinserted.

References

[1] G. Brodal. *Fast meldable priority queues.* 4th WADS, LNCS 955 (1995), 282-290.

[2] M. Brown. *Implementation and analysis of binomial queue algorithms.* SIAM J. Comput. 7 (1978), 298-319.

[3] M. Brown and R. Tarjan. *Design and analysis of data structures for representing sorted lists.* SIAM J. Comput. 9 (1980), 594-614.

[4] S. Carlsson, J. I. Munro and P. Poblete. *An implicit binomial queue with constant insertion time.* 1st SWAT, LNCS 318 (1988), 1-13.

[5] R. Cole. *On the dynamic finger conjecture for splay trees. Part II: The proof.* SIAM J. Comput. 30 (2000), 44-85.

[6] M. De Berg, M. Kreveld, M. Overmars and O. Shwarzkopf. *Computational geometry-algorithms and applications.* Springer-Verlag, (1997)

[7] L. Devroye. *Nonuniform random variate generation.* Springer-Verlag,(1986).

[8] E. Doberkat. *Deleting the root of a heap.* Acta Informatica, 17 (1982), 245-265.

[9] R. Dutton. *Weak-Heapsort.* BIT, 33 (1993), 372-381.

[10] S. Edelkamp and I. Wegener. *On the performance of weak-heapsort.* 17th STACS, LNCS 1770 (2000), 254-260.

[11] A. Elmasry. *Distribution-sensitive binomial queues.* 8th WADS, LNCS 2748 (2003), 103-113.

[12] A. Elmasry. *On the sequential access theorem and deque conjecture for splay trees.* Theoretical Computer Science 314(3) (2004), 459-466.

[13] M. Fredman, R. Sedgewick, D. Sleator, and R. Tarjan. *The pairing heap: a new form of self_adjusting heap.* Algorithmica 1,1 (1986), 111-129.

[14] L. Guibas, E. McCreight, M. Plass and J. Roberts. *A new representation of linear lists.* 9th ACM STOC (1977), 49-60.

[15] J. Iacono. *Improved upper bounds for pairing heaps.* 7th SWAT, LNCS (2000), 32-45.

[16] J. Iacono. *Distribution sensitive data structures.* Ph.D. thesis, Rutgers University, (2001).

[17] J. Iacono and S. Langerman. *Queaps.* 13th ISAAC, LNCS 2518 (2002), 211-218.

[18] H. Mannila. *Measures of presortedness and optimal sorting algorithms.* IEEE Trans. Comput. C-34 (1985), 318-325.

[19] Th. Porter and I. Simon. *Random insertion into a priority queue structure.* IEEE Trans. Software Engineering, 1 SE (1975), 292-298.

[20] D. Sleator and R. Tarjan. *Self-adjusting binary search trees.* J. ACM 32(3) (1985), 652-686.

[21] R. Sundar. *On the deque conjecture for the splay algorithm.* Combinatorica 12 (1992), 95-124.

[22] R. Tarjan, *Sequential access in splay trees takes linear time.* Combinatorica 5 (1985), 367-378.

[23] R. Tarjan. *Amortized computational complexity.* SIAM J. Alg. Disc. Meth. 6 (1985), 306-318.

[24] J. Vuillemin. *A data structure for manipulating priority queues.* Comm. ACM 21(4) (1978), 309-314.