Distribution-Sensitive Binomial Queues

Amr Elmasry

Computer Science Department
Alexandria University
Alexandria, Egypt

Abstract. A new priority queue structure is introduced, for which the amortized time to insert a new element is $O(1)$ while that for the minimum-extraction is $O(\log \overline{K})$. $\overline{K}$ is the average, taken over all the deleted elements x, of the number of elements that are inserted during the lifespan of x and are still in the heap when x is removed. 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 [15], thus establishing a hierarchy of properties.

Following finger trees [13], 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 [11,21,22], the dequeue theorem [21], and the dynamic finger theorem [4]. Each of these theorems describes a natural class of sequences of operations, and shows that the amortized cost 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 [14,15]. 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 [14] proved that if the minimum item in a pairing heap [12] of maximum size n is to be removed, and k heap operations have been performed since its insertion, the minimum-extraction operation takes amortized time $O(\log \min(n, k))$. 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 [16] 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 there is no search tree that can have this property. A priority queue is said to be queueish if the amortized cost of the insertion is $O(1)$, and the amortized cost of the minimum-extraction of 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 structure based on the well-known binomial queues. Let K_x denote the number of elements that are inserted during the lifespan of x and are still in the heap when x is removed. Let $\overline{K}$ be the average of these K_x's over all the deleted elements. Our modified binomial queues have the property that the amortized cost of the insert operation is $O(1)$, while the amortized cost of the delete-minimum operation is $O(\log \overline{K})$. 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.

The paper is organized as follows. The next section reviews the operations of binomial queues that we use as a basic structure for our new implementation. Section 3 is an informal discussion to the problems and solutions that motivates the way we implement our structure. We describe the operations of our structure in Section 4. Some of the possible applications are given in Section 5. We conclude with an improvement that achieves better constants with respect to the number of comparisons.

2 Binomial Queues

A binomial tree [1,24] 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 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 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 to form 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 implement some operations efficiently each node may, in addition, contain a pointer to its parent. The roots of the binomial trees within a binomial queue are organized in a linked list, which is referred to as the *root-list*. The ranks of the roots strictly increase as the root list is traversed 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.

For an n-node binomial queue, the worst-case cost for the *insert* and the *delete-minimum* is $O(\log n)$. The amortized cost [23] for the *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 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 $Pre(Q)$. At the moment when an element i is to be deleted from Q, let D_i be the number of elements preceding i 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 following property at any point of time: If we sum the D_i's over all the deleted elements and get the average, this number is upper bounded by $\overline{K}$ (i.e. $\sum_i D_i \leq \sum_i K_i$). We call an operation that preserves this property an *inversion-preserving* operation. See [17] for the notion of inversions.

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 becomes 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 [8,9] for a similar notion.

The next problem is with respect to the *delete-minimum* operation. When the root with the minimum value is deleted, its sub-trees are scattered according to their ranks and merged with other sub-trees in the heap, again affecting the order in which the elements are input. Our solution to this problem is to change the way the *delete-minimum* is implemented. When the root of the minimum value is deleted, one of the nodes of this tree is promoted to replace the deleted root. The heap property is maintained by a special implementation of a *heapify* operation. Two problems will pop-up as a result. The first problem is how to implement the *heapify* operation within a logarithmic time in the size of the tree. This leads to augmenting each node of the binomial queue with an extra pointer, as will be explained in details in the next section. The second problem occurs when several nodes are repeatedly deleted from a tree, causing such a tree to lose the structural properties of binomial trees. To overcome this problem, some restructuring is performed on such trees and a relaxation to the properties of the standard binomial trees is required.

We are not on the safe side yet. Consider the case when the root of a tree T of rank $r1$ 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 $r2$, where $r1 \gg r2$. The time required by this *delete-minimum* operation can be implemented to be in $\Theta(r1)$, which is not comparable to $r2$ that represents the logarithm of the number of elements that precedes the deleted element in $Pre(Q)$. 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 $r1$ to $r2$. 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 the subsequent *delete-minimum* operations.

Having two trees of 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 [3,7,19]. 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 $\log n$ 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 make our data structure more efficient, achieving the claimed bounds.

4 The Data Structure

We introduce the new basic structure, which we call *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 with rank 0 is a single node.

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

Proof. 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. By induction, $n \geq 1 + 2^0 + 2^1 + \ldots + 2^{r-1} = 2^r$. □

We are now ready to describe our data structure. We use relaxed binomial trees in place of the traditional binomial trees. Our binomial 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. The following procedures are used to perform the priority queue operations:

Heapify. Given a relaxed binomial tree T of rank r, such that the heap property is valid for all the nodes except for the root. The question is how to restore this property. Applying the standard *heapify* operation will do, while maintaining the *inversion-preserving* property. Recall that the *heapify* operation proceeds by finding the node, say x, with the smallest value among the children of the root and swapping its value with that of the root. This step is repeated with the node x as the current root, until either a leaf or a node that has a value smaller than or equal to all the values of its children is reached. To show that the *heapify* operation is *inversion-preserving*, consider any two elements $x_i, x_j \in Pre(T)$, where $i < j$. If these two elements were swapped during the *heapify* operation, then $x_i > x_j$. Since x_i precedes x_j in $Pre(T)$, we conclude that this swap decreases the number of inversions.

It remains to investigate how the *heapify* operation is implemented. Finding the minimum value within a linked list requires linear time. This may lead to an $O(r^2)$ time for the *heapify* operation. We can do better, however, by maintaining with every node an extra pointer that points to the node with the smallest value among all its right siblings, including itself. We call this pointer, the pointer for the prefix minimum (*pm*). The *pm* pointer of the leftmost child of a node will, therefore, point to the node with the smallest value among all the children of the parent node. To maintain the correct values in the *pm* pointers, whenever the value of a node is updated all the *pm* pointers of its left siblings, including itself, have to be updated. This is accomplished by proceeding from right to left; the *pm* pointer of a given node x is updated to point to the smaller of the value of x and the value of the node pointed to by the *pm* pointer of the right sibling of x. A *heapify* at a node with rank $r1$ reduces to a *heapify* at its child with the smallest value whose rank is $r2$ after $O(r1 - r2)$ time and at most $3(r2 - r1)$ comparisons. The time spent by the *heapify* on T is, therefore, $O(r)$.

If we are concerned with constant factors, the number of comparisons can still be reduced as follows. First, the path from the root to a leaf, where every node has the smallest value among its siblings, is determined by utilizing the *pm* pointers. No comparisons are required for this step. Next, the value at the root is compared with the values of the nodes of this path bottom up, until the correct position of the root is determined. The value at the root is then inserted at this position, and all the values at the nodes above this position are shifted up. The *pm* pointers of the nodes whose values moved up and those of all their left siblings are updated. The savings are due to the fact that at each level of the queue (except possibly for the level of the final destination of the old value of the root) either a comparison with the old value of the root takes place or the *pm* pointers are updated, but not both. Then, the number of comparisons is at most $2r$. See [10] for a similar description to this procedure.

Merge. Given two relaxed binomial trees of the same rank r whose roots are adjacent in the *root-list* of the binomial queue, the two trees can be 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 *pm* pointer of the linked node is updated. The roots of the two trees are removed from the *root-list*, and the root of the new tree is inserted in their position. The *merge* operation takes constant time.

Insert. The new element is added to the forest as the rightmost tree whose height (rank) is 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.*

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. If this bit was set to 1 (as a result of a previous *merge* operation), we make the root of the first tree the right sibling of the root of the second tree, otherwise we make the root of the first tree the left sibling of the root of the second tree. The two roots are inserted in place of the root of T in the *root-list.* The *split* operation takes constant time, and no comparisons are needed.

Promote. Given a relaxed binomial tree T with a deleted root of rank r, the purpose of this procedure is to promote a node to replace the root, while maintaining the structural properties of relaxed binomial trees together with the *inversion-preserving* property. The procedure starts by promoting the single node representing the rightmost child, making it the new root of the binomial tree. As a result, there may become no tree of rank 0. To maintain the properties of relaxed binomial trees, assume that before performing the following iterative step there is no child of T with rank i. We call the following iterative step *gap(i).* The rightmost 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: This case is terminal.
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 leftmost trees are merged into one tree with rank $i+1$. This case is also terminal.
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 *gap(i+1)*).

If the iterative step is repeated until there is no tree of rank $r-1$, the iterative step terminates and the promoted root is assigned a rank of $r-1$. Otherwise, the promoted root is assigned a rank of r.

To maintain the *pm* pointers of the children of the promoted root without performing extra comparisons, the following trick is made. Before the *promote*, if the value of the single node representing the rightmost child is smaller than the value of its left sibling, the two nodes are swapped. As a result the *pm* pointers of the other children will not need to be changed. The time spent by the *promote* is $O(r)$, and the number of comparisons performed is $O(1)$. After the *promote*, a *heapify* must be called to maintain the heap property for the promoted root.

Fill-gaps. Given a relaxed binomial tree T of rank $r1$ such that the rank of the tree to its right in the queue is $r2$, where $r1 > r2 + 1$, several *split* operations are performed on T. A tree of rank i can be split into two or three trees of rank $i - 1$ by performing one or two *split* operations, respectively. While the ranks of the trees resulting from the split are greater than $r2$, a *split* is repeatedly performed on the right tree among these trees. As a result, there will be at most one tree of rank $r1$ (if there was two before this procedure), one or two trees of each of the ranks $r1-1, r1-2, \ldots, r2+2$, and two or three trees of rank $r2+1$. The possibility of having three trees of the same rank violates the rules. If this happens, the leftmost two trees of rank $r2+1$ are merged to form a tree of rank $r2 + 2$. This violation may propagate while performing such merge operations, until there are no three trees of the same rank; a case that is insured to be fulfilled if the result of the merge is a tree of rank $r1$. As a final result of this procedure, there will be at most two trees of rank $r1$, one or two trees of each of the ranks $r1-1, r1-2, \ldots, r2+1$. The time spent by the *fill-gaps* procedure is $O(r1 - r2)$.

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. The solution is to reuse the idea of the prefix minimum pointers. A pointer is used with every root, in the *root-list*, that points to the node with the smallest value among the roots to its left, including itself. We call these pointers the suffix minimum (*sm*) pointers of the roots. The *sm* pointer of the rightmost root points to the root with the 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). This process is crucial to achieve the claimed bounds for the heap operations. A more efficient procedure to implement this step would improve the constants of the heap operations, as explained in Section 6.

Delete-minimum. First, the root of the tree T with the minimum value is removed. A *promote* is performed on T, followed by a *heapify*. As a result of the *promote*, the rank of T may decrease by one, and there may be three trees with the same rank. In this case, a *merge* operation is performed on T and the tree to its right, restoring the property of having at most two trees with the same rank. Next, a *fill-gaps* is performed on the tree T. The final step is to perform a *maintain-minimum* to keep track of the new minimum.

Theorem 1. *Starting with an empty distribution-sensitive binomial queue, the amortized cost of the insert operation is* $O(1)$*, and that of the delete-minimum is* $O(\log \overline{K})$*. 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 for the roots of the trees in the queue there is either $0, 1,$ *or* 2 trees. After the i^{th} operation, let N_0^i be the number of rank values that is not represented by any trees, N_1^i be the number of rank values that is represented by one tree, and N_2^i be the number of rank values that is 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 $N_2^{i+1} - N_2^i \leq -t + 1$ and $N_0^{i+1} - N_0^i \leq 0$. 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 c_2.

Next, assume that the operation $i+1$ is a *delete-minimum* performed on the root of a tree T of rank $r1$. The actual cost is $O(r1)$. Let $r2$ 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 D_m, where m is the number of the current *delete-minimum* operation (D_m is the number of elements preceding this deleted element in $Pre(Q)$ at this moment). As a result of the *fill-gaps* procedure: $N_0^{i+1} - N_0^i \leq -(r1 - r2 - 2)$ and $N_2^{i+1} - N_2^i \leq r1 - r2 - 1$. Hence, the amortized cost is bounded by $O(r1) - (c_1 - c_2)(r1 - r2 - 1) + c_1$. 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 $O(r2)$ which is $O(\log D_m)$. It follows that the cost of these m *delete-minimum* operations is $O(\sum_{i=1}^{m} \log D_i)$. Jensen's inequality implies $\sum_{i=1}^{m} \log D_i \leq m \log\left(\frac{1}{m} \sum_{i=1}^{m} D_i\right)$. Since all our procedures have the *inversion-preserving* property, then $\frac{1}{m} \sum_{i=1}^{m} D_i \leq \overline{K}$. It follows that the amortized cost of the *delete-minimum* operation is $O(\log \overline{K})$. □

5 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 binomial 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 $O(n \log \overline{K})$. If the elements are inserted in reverse order, $\overline{K}$ will be the average number of inversions in the input sequence, and our algorithm is optimal [13,17,18]. 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 [5] 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 [6].

6 Improving the Constants

The constant factor of the number of comparisons of the *heapify* in the $O(\log \overline{K})$ is 2, and that of the *maintain-minimum* is 2, for a total of at most $4 \log_2 \overline{K} + O(1)$ comparisons per *delete-minimum*. Next, we sketch the way to implement *maintain-minimum* in $O(\log \log \overline{K})$, achieving an overall bound of $2 \log_2 \overline{K} + O(\log \log \overline{K})$ for the number of comparisons. The roots of the trees are kept in a set of heaps, such that all the nodes 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 *sm* pointers 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 *sm* 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 $O(\log \log \overline{K})$. 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 bounds 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, it is first checked 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, which insures that a marked node will never become the minimum of a heap before being reinserted.

References

1. M. Brown. *Implementation and analysis of binomial queue algorithms.* SIAM J. Comput. 7 (1978), 298–319.
2. M. Brown and R. Tarjan. *Design and analysis of data structures for representing sorted lists.* SIAM J. Comput. 9 (1980), 594–614.
3. S. Carlsson and J. I. Munro. *An implicit binomial queue with constant insertion time.* 1st SWAT. In LNCS 318 (1988), 1–13.
4. R. Cole. *On the dynamic finger conjecture for splay trees. Part II: The proof.* SIAM J. Comput. 30 (2000), 44–85.
5. M. De Berg, M. Kreveld, M. Overmars and O. Shwarzkopf. *Computational geometry-algorithms and applications.* Springer-Verlag, (1997)
6. L. Devroye. *Nonuniform random variate generation.* Springer-Verlag, (1986).
7. E. Doberkat. *Deleting the root of a heap.* Acta Informatica, 17 (1982), 245–265.
8. R. Dutton. *Weak-Heapsort.* BIT, 33 (1993), 372–381.
9. S. Edelkamp and I. Wegener. *On the performance of weak-heapsort.* STACS. In LNCS 1770 (2000), 254–260.
10. A. Elmasry. *Priority queues, pairing and adaptive sorting.* 29th ICALP. In LNCS 2380 (2002), 183–194.
11. A. Elmasry. *A new proof for the sequential access theorem for splay trees.* WSES, ADISC. In Theoretical and Applied Mathematics, (2001) 132–136.
12. 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.
13. L. Guibas, E. McCreight, M. Plass and J. Roberts. *A new representation of linear lists.* ACM STOC 9 (1977), 49–60.
14. J. Iacono. *Improved upper bounds for pairing heaps.* 7th SWAT. In LNCS (2000), 32–45.
15. J. Iacono. *Distribution sensitive data structures.* Ph.D. thesis, Rutgers University. (2001).
16. J. Iacono and S. Langerman. *Queaps.* International Symposium on Algorithms and Computation. In LNCS 2518 (2002) 211–218.
17. D. Knuth. *The Art of Computer Programming. Vol III: Sorting and Searching.* Addison-wesley, second edition (1998).
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.