

A General Technique for Implementation of Efficient Priority Queues

Peter Høyer

Department of Mathematics and Computer Science
Odense University, Denmark

Abstract

This paper presents a very general technique for the implementation of mergeable priority queues.

The amortized running time is $O(\log n)$ for DELETEMIN and DELETE, and $\Theta(1)$ for all other standard operations. In particular, the operation DECREASEKEY runs in amortized constant time. The worst-case running time is $O(\log n)$ or better for all operations.

Several examples of mergeable priority queues are given. The examples include priority queues that are particular well suited for external storage. The space requirement is only two pointers and one information field per item.

The technique is also used to implement mergeable, double-ended priority queues. For these queues, the worst-case time bound for insertion is $\Theta(1)$, which improves the best previously known bound. For the other operations, the time bounds are the same as the best previously known bounds, worst-case as well as amortized.

1 Introduction

A *mergeable priority queue* is one of the fundamental data types. It is used for storing a set of items with keys drawn from a total order. The five basic operations are:

MAKEPRIORITYQUEUE (): create and return a new, empty priority queue.

INSERT (Q, v): insert a new item v into Q .

FINDMIN (Q): return an item with minimum key in Q .

DELETEMIN (Q): delete an item with minimum key from Q and return it.

MELD (Q_1, Q_2): create and return a new priority queue that contains all the items of Q_1 and Q_2 . This operation destroys Q_1 and Q_2 .

The operation BUILD creates and returns a new priority queue Q that contains n specified items. Generally, if INSERT has worst-case running time $\omega(1)$, BUILD is included in the collection of basic operations.

Often, the following two operations are useful:

DELETE (Q, v): delete the item v from Q .

DECREASEKEY (Q, v, δ): subtract δ from the key of item v in Q .

In a mergeable *double-ended* priority queue, two further operations are supported, FINDMAX and DELETEMAX, and often the operation INCREASEKEY is useful. They are all defined in the obvious way. When both DECREASEKEY and INCREASEKEY are supported, we use CHANGEKEY as shorthand to denote both.

Binomial queues were introduced by Vuillemin [13], and they support the basic operations above in $O(\log n)$ worst-case time, where n denotes the size of the queue. Brown [4] studied their properties in detail. Fredman and Tarjan [7] developed an extension of binomial queues called *Fibonacci heaps*, which also supports the next two operations. In a Fibonacci heap, the amortized running time is $\Theta(n)$ for BUILD, $O(\log n)$ for DELETEMIN and DELETE, and $\Theta(1)$ for the rest of the operations. *Vheaps* were introduced by Peterson [12], and they achieve the same time bounds as Fibonacci heaps up to constant factors. These time bounds were also obtained by Driscoll et al. in [6] for *relaxed heaps*, and the bound for DECREASEKEY was improved to $\Theta(1)$ worst-case time for a variant of relaxed heaps.

In this article, we provide a general technique for the implementation of mergeable priority queues, called *ranked priority queues*. A ranked priority queue efficiently supports all the basic and the next two operations above. Specifically, it achieves the same amortized time bounds as Fibonacci heaps, and all worst-case time bounds remain logarithmic or better.

A ranked priority queue is a forest of binary trees in which the nodes satisfy some ordering, called a *half-*

ordering. There is an one-to-one correspondance between half-ordered trees and forests of heap-ordered multiway trees. In section 2, we discuss some important operations on half-ordered trees. We also define a ranked priority queue and give the implementation of the data structure.

The main idea in Fibonacci heaps [7], Vheaps [12], and ranked priority queues is to maintain a forest of balanced trees. A Vheap differs from, but corresponds to a forest of half-ordered AVL-trees (see section 4.3), and these trees are kept balanced by some rotation primitives. In ranked priority queues, we also keep the trees balanced by some rotation primitives, which are, however, defined differently than in [12]. In addition, we use some other re-balancing primitives.

In ranked priority queues, we maintain a strict separation of the ordering and the balancing. We use this to implement the operations DELETEMIN, DELETE, and DECREASEKEY in an un-complicated way compared to [12]. The operations are discussed in section 3, which contains a few assumptions, that a specific implementation must fulfill. The proposed technique is very general, and several different examples of mergeable priority queues are given in section 4. The technique is extended further in sections 4.4 and 4.5.

There is a claim in [12] that that technique can be generalized from AVL-trees to any *comparable* balanced (binary) tree scheme. However, nothing more is said, so it is not clear which schemes are “comparable”.

Very recently, two new, efficient mergeable double-ended priority queues have been proposed. Ding and Weiss [5] proposed the *relaxed min-max heap*. The *double-ended binomial queue* was proposed by Khoong and Leong [10]. The worst-case and amortized running times for relaxed min-max heaps and double-ended binomial queues are the same up to constant factors. The worst-case running time is $\Theta(1)$ for MAKEPRIORITYQUEUE, FINDMIN, FINDMAX; $O(\log n)$ for INSERT, DELETEMIN, DELETEMAX, DELETE, CHANGEKEY, and MELD; and $\Theta(n)$ for BUILD. The amortized running time is $\Theta(1)$ for MAKEPRIORITYQUEUE, FINDMIN, FINDMAX, INSERT, and MELD¹; $O(\log n)$ for DELETE, DELETEMIN, DELETEMAX, and CHANGEKEY; and $\Theta(n)$ for BUILD.

In section 4.5, we show how to implement a mergeable double-ended priority queue. Then, we achieve *worst-case* constant running time for insertion, improving the best previously known bound. For the

¹The original article on the relaxed min-max heap states only an $O(\log n)$ amortized bound for merging. A $\Theta(1)$ bound can be proved following the ideas in [10].

other operations, we achieve the same time bounds as the best previously known bounds [5, 10] stated above. The data structure has some similarities with the one used in [10]. We discuss these and use some of the ideas from previous sections to generalize the double-ended binomial queue of [10].

2 Heap trees

2.1 Heap trees and heap tree operations

In this section, we define and discuss the underlying structure, the *heap tree* used in ranked priority queues. We summarize some well-known operations on heap trees, and we define some very useful operations which are fundamental for the implementation of ranked priority queues. They help us to introduce new ways to support the two operations DECREASEKEY and DELETE. These are explained in section 3.

Let \tilde{T} be any tree. Through the well-known *natural correspondance*, described in [11], \tilde{T} can also be represented as a binary tree T . We use this second representation, except that we switch which is the left and right child, compared to [11]. Thus in this representation of ordered trees, a node’s left child is its right sibling from the ordered tree, and its right child is its first child from the ordered tree. If \tilde{T} is heap-ordered, that is, each non-root node has key value greater than or equal to the key value of its parent, then the corresponding binary tree T is half-ordered (see definition 1).

As an example, we use the binomial tree, which is the underlying structure used in binomial queues and Fibonacci heaps. The binomial tree \tilde{B}_r of rank r is defined inductively as follows. \tilde{B}_0 is a tree consisting of just one node, and a general \tilde{B}_r is formed by linking two \tilde{B}_{r-1} together, that is, by letting the root of one of them become a child of the root of the other. In figure 1, the first few \tilde{B}_r are shown to the left, and their corresponding binary representation are shown to the right.

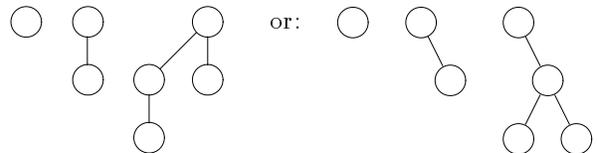


Figure 1: Two ways of looking at the binomial tree.

Obviously, the binary tree corresponding to a binomial tree consists of a root node having a perfect

binary tree (in which for all nodes v , the height of the left subtree of v equals the height of the right subtree of v) as its right subtree. Since we require that the root has no left child, the tree has minimum height. Therefore, we say that the tree is balanced.

Definition 1 A **half-ordered tree** is a binary tree that satisfies the following condition:

- for any node v , v has key value less than or equal to the key value of any node in v 's right subtree.

Definition 2 A **heap tree** is a half-ordered tree that satisfies the following condition:

- the root has at most one child, which is then a right child.

We distinguish between subtrees and sub-heap trees. A *subtree* is defined as usual. The *sub-heap tree* rooted at v consists of v and v 's right subtree. Throughout this paper, T_v denotes the sub-heap tree rooted at v . A *leftmost path* is a path in which all edges go to left children and in which the last node has no left child.

Three well-known primitives on heap trees work as follows. The first is to make a new heap tree containing a single specified node v . It consists of specifying that v has neither a left, nor a right child, nor a parent. Trivially, this can be done in constant time.

The second primitive is to create the union of two heap trees, which is done as follows. Let x and y be the roots of the two trees. Without loss of generality, we assume that the key value of x is less than or equal to the key value of y . Then x 's right subtree becomes y 's left subtree, and y becomes x 's right child (see figure 2). This primitive is called *linking*, and can trivially be done in constant time. The reverse operation, the *un-linking*, can similarly be done in constant time.

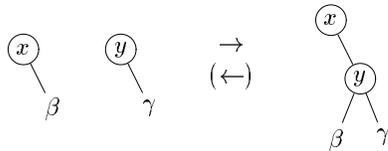


Figure 2: Linking (and un-linking) can be done in constant time. Note: $\text{Key}(x) \leq \text{Key}(y)$.

The third primitive is to *cut* out the sub-heap tree T_v rooted at some node v . It removes v 's left child x as a child of v , and makes x a child of v 's parent (see figure 3). The cutting makes v and its right subtree into a new tree T_v , and the old tree T into a smaller tree. This process can be done in constant time.

The following very useful primitives, *rotations* and *swaps*, are used to modify the internal structure of

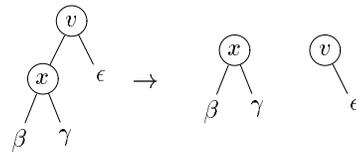
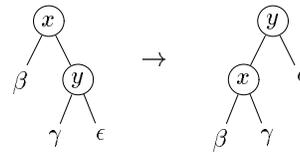
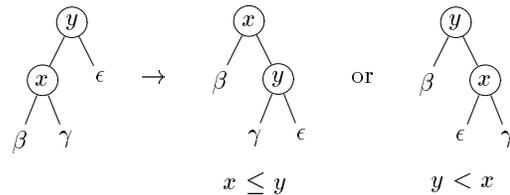


Figure 3: The operation $\text{CUT}(v)$ removes the link between v and its parent p , and the link between v and its left child x . A new link between p and x is made. The cutting makes v and its right subtree into a new tree.

heap trees. Consider any internal node y that is a right child of a node x . A *left rotation* adjusts the heap tree so that x becomes the left child of y . The operation preserves the half-ordering. A *right rotation* is a bit more tricky. Consider any internal node x that is a left child of a node y . If x has key value less than or equal to y 's key value, a right rotation is just the inverse of a left rotation. Otherwise, y remains the root of the subtree, and x becomes y 's right child. This ensures that a right rotation preserves the half-ordering, too.

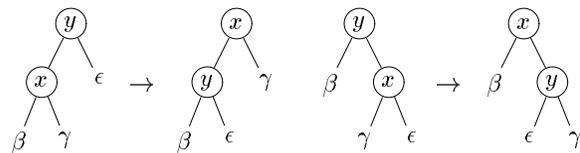


Left rotation



Right rotation

Unlike for binary search trees, we can define one more useful primitive due to the half-ordering. A *left swap* changes the order of two nodes on a left path. Let x be a left child of y . Then a left swap turns y into a left child of x . A *right swap* is defined similar, but it does not preserve the half-ordering. However, a right rotation can be viewed as a traditional binary tree rotation perhaps followed by a right swap.



Left swap

Right swap

A crucial step in the development of ranked priority queues is to separate the ordering and the balancing. The ordering is maintained by linking and cutting, while the balancing is maintained by rotations and swaps.

2.2 Ranked priority queues

Intuitively, since a heap tree is half-ordered, the number of nodes on the leftmost paths in the tree gives a measure for how well-balanced the tree is. We formalize our intuitive idea of well-balanced in the following definition. We use the notation $s(v)$ for the size of the subtree rooted at v . Additionally, $l(v)$ denotes the number of nodes on the leftmost path from v . We define the size $s(T)$ of a heap tree T to be the size of the root of the tree, and $l(T)$ to be $\max_{v \in T} l(v)$.

Definition 3 *A tree node v is said to be α -left-height-balanced for some α , $1/2 \leq \alpha < 1$, if it satisfies:*

$$l(v) \leq \log_{(1/\alpha)}(s(v) + 1).$$

A tree is said to be α -left-height-balanced if all nodes in it are α -left-height-balanced.

As shorthand, we use $h_\alpha(v)$ to denote $\log_{(1/\alpha)}(s(v) + 1)$, and similarly $h_\alpha(T)$ to denote $\log_{(1/\alpha)}(s(T) + 1)$. To maintain α -left-height-balance, we assign a positive integer $r(v)$ to each node v , called the *rank* of the node. We define the rank $r(T)$ of a heap tree T to be the rank of the root of the tree.

Definition 4 *A ranked priority queue Q is a forest of heap trees that satisfies, for some fixed constants α and κ , $1/2 \leq \alpha < 1$ and $\kappa \geq 1$, the following **balancing conditions**:*

For any node v in any heap tree in the forest,

- (i) *if v is a root with no child, $r(v) = 1$,*
- (ii) *if v is a root with a right child, $r(v) = r(\text{right}(v)) + 1$,*
- (iii) *if v is a non-root node, $l(v) \leq \kappa r(v) \leq h_\alpha(v)$.*

Let n denote the number of nodes in Q , and τ denote the number of trees in Q . In addition, let λ denote the maximum number of nodes on any leftmost path in any tree in Q , and $r_{max}(Q)$ denote the maximum rank of any tree in Q .

Let v be any node in any tree. The most important property of ranked priority queues is (iii), which bounds $r(v)$ from above and below. Due to the inequality $l(v) \leq \kappa r(v)$, $r(v)$ gives an upper bound on the number of nodes on the leftmost path from

v . In particular, note that for any heap tree T , $l(T) \in O(h_\alpha(T))$, and thus $\lambda \in O(\log_{(1/\alpha)} n)$.

Due to the inequality $\kappa r(v) \leq h_\alpha(v)$, $r(v)$ gives a lower bound on the number of nodes in the heap tree rooted at v . Most importantly, note that for any heap tree T , $r(T) \in O(h_\alpha(T))$, and thus $r_{max}(Q) \in O(\log_{(1/\alpha)} n)$. So, $l(T)$ and $r(T)$ are logarithmically bounded by the size of the tree.

In section 4, we provide several *balancing schemes* for maintaining the balancing conditions. Each scheme specifies the constants α and κ , and gives rules for assigning ranks to the nodes. In each scheme, the primitive, linking, can be done in constant time, and it maintains the balancing conditions. This is an important property since this primitive includes re-computing ranks of nodes. In section 3, we assume that linking preserves the balancing conditions and still can be executed in constant time. Creation of a new heap tree always preserves the balancing conditions by assigning 1 as the rank of the new root (see balancing condition (i)). Un-linking also always preserves the balancing conditions, by re-computing the rank of the two new roots using balancing conditions (i) and (ii).

2.3 The data structure

The data structure for handling the forest contains, for each r , $1 \leq r \leq r_{max}(Q)$, a *forest pointer* Q_r , which may be NIL, to a tree of rank r . The remaining trees are kept in the *tree list*: a list of pairs of trees of the same rank. Initially, the tree list is empty. In addition, there is a pointer to the root which contains the minimum key in Q .

To insert a tree T with rank r into the forest, we use a subroutine `ADDTREE`, which works as follows. If $Q_r = \text{NIL}$, then Q_r gets the value T ; otherwise the value of Q_r and T are made into a pair which is added to the tree list and Q_r becomes NIL. The number of trees τ in the forest has just been incremented. Now `ADDTREE` attempts to decrement τ . The subroutine checks whether the tree list is empty. If the tree list is not empty, it removes the first pair from the tree list and links the two trees to form a tree T' with rank r' , $r' \geq r$. T' is inserted; either as the value of $Q_{r'}$ or as part of a pair added to the tree list.

If $r' > r_{max}(Q)$, `ADDTREE` creates and initializes $r' - r_{max}(Q)$ new forest pointers, which can be done in linear time in the number of new pointers. Finally, `ADDTREE` updates the pointer to the root which contains the minimum key.

Since linking can be done in constant time, the worst-case running time for `ADDTREE` is $\Theta(\max(r' -$

$r_{max}(Q, 1)$). If, for some constant $\Delta > 0$, $r' - r \leq \Delta$ for all r , then `ADDTREE` has constant worst-case running time. In section 3, we assume that such a constant exists. For all schemes mentioned in section 4, $\Delta = 1$. By the assumption that linking preserves the balancing conditions, `ADDTREE` preserves them, too. By induction, we immediately get that $\tau \leq r_{max}(Q)$. That is, the maximum rank of the trees is an upper bound on the number of trees. Since $r_{max}(Q) \in O(\log_{(1/\alpha)} n)$, we have $\tau \in O(\log_{(1/\alpha)} n)$.

The idea of adding a tree list to the data structure was introduced by Driscoll et al. [6], and is here used somewhat modified. Its obvious advantage is to support `ADDTREE` in worst-case constant time rather than amortized constant time.

3 Priority queue operations

Theorem 1 *Let Q be a ranked priority queue. Assume linking can be done in constant time, and assume for some constant $\Delta > 0$, $r' - r \leq \Delta$ for all r , where r' denotes the rank of the new tree after linking two trees of rank r . Then the worst-case running times are $\Theta(1)$ for `MAKEPRIORITYQUEUE`, `INSERT` and `FINDMIN`; $O(\log_{(1/\alpha)} n)$ for `DELETEMIN`; $\Theta(\min(\log_{(1/\alpha)} n_1, \log_{(1/\alpha)} n_2))$ for `MELD`; and $\Theta(n)$ for `BUILD`.*

Proof

`MAKEPRIORITYQUEUE`

Operation `MAKEPRIORITYQUEUE` initializes the tree list. In addition, the operation only initializes the first forest pointer, since we dynamically create new forest pointers when $r_{max}(Q)$ is increased by an `ADDTREE` operation.

`INSERT` and `BUILD`

To insert a node v , we make v into a heap tree T and execute `ADDTREE(T)`. This can be done in constant time as explained in section 2.3. By repeatedly calling `INSERT`, a ranked priority queue can be built in linear time.

`FINDMIN`

By maintaining a pointer to the root of the tree holding the minimum key, this node can be accessed in constant time. Whenever `ADDTREE` or `DELETEMIN` is executed, we update the pointer.

`MELD`

Let n_1 and n_2 be the number of nodes in the two ranked priority queues, Q_1 and Q_2 , and without loss of generality assume $r_{max}(Q_1) \geq r_{max}(Q_2)$. We assume that the two priority queues share the same scheme

for maintaining the balancing conditions. For each tree T_2 in Q_2 , the operation performs `ADDTREE(T_2)` in Q_1 . Since we have to examine each forest pointer in Q_2 and each pair in Q_2 's tree list, this takes time $\Theta(r_{max}(Q_2))$.

`DELETEMIN`

Let T be the tree with root v , holding the minimum key. The operation removes T from Q . If T was in the tree list, the other tree in the pair is re-inserted by `ADDTREE`. If the root v in T has a right child, T is unlinked, the second tree is re-inserted by `ADDTREE`, the first tree becomes T , and the process is repeated until the root v in T has no right child. By each un-linking, the rank of the two root nodes are recomputed as described in section 2.2. The operation finds a new root node with minimum key by searching through all roots in the forest. Finally, v is returned and the operation terminates.

The iterative calls of `ADDTREE` take time $O(\lambda)$ and the final search takes time $O(r_{max}(Q))$, making `DELETEMIN` run in $O(\lambda + r_{max}(Q))$ time. \square

The amortized running time for the operation `MELD` is $\Theta(1)$, which is proved as follows. As the potential function for a ranked priority queue, we use $r_{max}(Q) + \Delta\tau$. The initial potential is zero, and the potential is always non-negative. The potential before the operation `MELD` is $r_{max}(Q_1) + r_{max}(Q_2) + \Delta(\tau_1 + \tau_2)$. With an `ADDTREE` operation, either τ_1 or $r_{max}(Q_1)$ does not increase, so in each step, $r_{max}(Q_1) + \Delta\tau_1$ increases by at most Δ . Thus, after the operation `MELD` the potential is at most $r_{max}(Q_1) + \Delta(\tau_1 + \tau_2)$. Examining the forest pointers and the tree list takes time $\Theta(r_{max}(Q_2))$. Hence, the overall amortized running time is $\Theta(1)$. The amortized running time for any other operation is the same as the worst-case running time, up to a constant factor.

The basic operations all maintain the ordering, while the balance problem and hence the running times have been hidden by the two numbers: λ and $r_{max}(Q)$. The balancing conditions bound these numbers, and thus make the separation of the ordering and balancing possible.

The operations `DECREASEKEY` and `DELETE` modify the internal structure of a heap tree, and the result may violate the balancing conditions. To restore these, we use a subroutine `REBALANCE`. The subroutine re-establishes the balancing conditions in amortized constant time by the use of rotations and swaps. The implementation of `REBALANCE` depends on the particular scheme for maintaining the balancing conditions, just as different balanced search trees use dif-

ferent rebalancing operations. The schemes in section 4 include implementations of the rebalancing operation.

Theorem 2 *Let Q be a ranked priority queue. Assume linking can be done in constant time, and assume for some constant $\Delta > 0$, $r' - r \leq \Delta$ for all r , where r' denotes the rank of the new tree after linking two trees of rank r . In addition, assume REBALANCE can be done in $O(\log_{(1/\alpha)} n)$ worst-case time, and $\Theta(1)$ amortized time. Then the worst-case running times are $O(\log_{(1/\alpha)} n)$ for DECREASEKEY and DELETE. The amortized running time is $\Theta(1)$ for DECREASEKEY, and $O(\log_{(1/\alpha)} n)$ for DELETE.*

Proof

DECREASEKEY

The operation DECREASEKEY (Q, v, δ) subtracts δ from the key of v in tree T in Q . Let x be v 's left child. First decrease v 's key by δ , and then execute CUT(v). Set $r(v) = r(\text{right}(v)) + 1$, and execute ADDTREE (T_v), that is, insert the new heap tree rooted at v into the forest. Recall that ADDTREE updates the pointer to the root which contains the minimum key. Finally, rebalance T by executing the subroutine REBALANCE(x).

If v is a right child and the tree after the decrease-step satisfies the half-ordering, the cutting is unnecessary. Instead the operation checks whether v is a root. If so, the operation updates the pointer to the root which contains the minimum key. It then stops.

DELETE

The operation DELETE (Q, v) is implemented as a combination of DECREASEKEY and DELETEMIN. First, the key of v is made smaller than the minimum of Q by a DECREASEKEY. Then v is removed from the forest by a DELETEMIN. \square

4 Specific schemes for maintaining the balancing conditions

To use theorem 1, one must show that linking preserves the balancing conditions and runs in constant time. For the particular balancing schemes in this section, we prove that in two steps. First, we show that linking preserves the balancing scheme. Then, we show that the balancing scheme preserves the balancing conditions.

However, it is also of interest how much one can prove about the linking primitive, given none or just little knowledge about the particular balancing

scheme. For the general case (that is, not specifying the particular scheme), we can show the necessary condition stated in lemma 3. For a class of schemes, we can show the result stated in lemma 4.

Lemma 3 *Linking two trees of equal rank preserves α -left-height-balance.*

Lemma 4 *If $1 \leq \kappa \leq \log_{(1/\alpha)} 2$, then linking two trees of equal rank can be implemented such that it runs in constant time and preserves the balancing conditions.*

In lemmas 3 and 4, we assume that the two trees have equal rank. Note that in theorems 1 and 2, we do only perform the link primitive within the ADDTREE operation, which only links trees of equal rank.

In the rest of this section, we consider different schemes for maintaining the balancing conditions. A scheme consists of the constants α and κ , and a set of rank properties. For each scheme, we check the necessary conditions that must be satisfied to implement a ranked priority queue. These conditions are: linking preserves the balancing conditions and runs in constant time, and REBALANCE runs in $O(r_{max}(Q))$ worst-case and $\Theta(1)$ amortized time.

4.1 Red-black priority queues

Our first example uses a slight modification of the weighted height properties from red-black trees [2, 8] as the rank properties. A *red-black priority queue* is a forest of heap trees with the constants $\alpha = 1/\sqrt{2}$ and $\kappa = 2$, that satisfies the following rank properties, called r_{rb} :

For any node v in any heap tree in the forest,

- if v is a root with no child, $r_{rb}(v) = 1$,
- if v is a root with a right child, $r_{rb}(v) = r_{rb}(\text{right}(v)) + 1$,
- if v is a non-root node with at most one child, $r_{rb}(v) = 1$,
- if v is a node with a parent, $r_{rb}(v) \leq r_{rb}(p(v)) \leq r_{rb}(v) + 1$,
- if v is a node with a grandparent, $r_{rb}(v) < r_{rb}(p(p(v)))$.

We must show three things. First, we show that r_{rb} satisfies the balancing conditions in definition 4. Then, we implement the link operation, such that it runs in constant time and satisfies the definition of r_{rb} . Finally, we show how to implement REBALANCE in $O(r_{max}(Q))$ worst-case and $\Theta(1)$ amortized time,

such that DECREASEKEY preserves the definition of r_{rb} . By theorems 1 and 2, this gives the following theorem:

Theorem 5 *A red-black priority queue is a ranked priority queue. The worst-case running time is $\Theta(1)$ for MAKEPRIORITYQUEUE, INSERT, and FINDMIN; $\Theta(\min(\log(n_1), \log(n_2)))$ for MELD; $O(\log n)$ for DELETEMIN, DELETE, and DECREASEKEY; and $\Theta(n)$ for BUILD. The amortized running time is $\Theta(1)$ for MAKEPRIORITYQUEUE, INSERT, FINDMIN, MELD, and DECREASEKEY; $O(\log n)$ for DELETE and DELETEMIN; and $\Theta(n)$ for BUILD.*

The rank properties r_{rb} are the same as the weighted height properties from red-black trees, but a minor difference: the root is treated separately. The following lemma shows that the choice of r_{rb} is appropriate.

Lemma 6 *The rank properties r_{rb} satisfy the balancing conditions.*

The primitive for linking two trees of equal rank increases the rank of the new root by one, which is sufficient to satisfy the definition of r_{rb} .

Let T be a tree in the red-black queue, and let v be a node in T with a parent. The operation CUT(v) replaces v by v 's left child x (see figure 3). The rank of x is one or two less than that of its new parent. If the difference is two, we say that x is a *bad node*. The operation REBALANCE(x) executes a sequence of *decrease steps*. In each decrease step, the rank of x 's parent is decreased by one, so the one violation is pushed up the tree, just as in a red-black tree (see case (a) for red-black priority queues in the appendix).

The sequence of decrease steps is followed by at most two *terminating steps*. A terminating step performs one or two rotations or swaps, or decreases the rank of a node. In each step, we distinguish between whether x is a left child or a right child (see the appendix).

Since the decrease steps do not alter the tree's structure and since rotations and swaps preserve the half-ordering, the rebalance operation preserves the half-ordering, too. After the terminating steps, the rank of the root may have been decreased by one. If so, we remove the tree T from the forest and re-insert it by ADDTREE(T). Note that T may be in the tree list. In that case, we also re-insert the other tree in the pair.

Lemma 7 REBALANCE runs in $O(\log n)$ worst-case time, and $\Theta(1)$ amortized time.

Proof By lemma 6 and the discussion above, the number of decrease steps is $O(\log n)$, each step taking constant time. Hence, the worst-case running time for REBALANCE is $O(\log n)$.

To perform an amortized analysis, we use the number of siblings (in the binary tree) with equal rank as the potential function. The initial potential is zero, and the potential is always non-negative. Each decrease step which causes another decrease step, decreases the potential function by one. (For example, suppose the bad node is a right child, we have case (a) in the appendix, and this decrease step causes another decrease step. Then A has a sibling, and this sibling has rank $r + 2$.) Each terminating step changes the potential by at most a constant. So, the amortized running time is $\Theta(1)$ for REBALANCE. \square

The linking and un-linking may also change the potential, but only by a constant. The amortized running time for any other operation is thus affected by at most a constant factor.

This completes the proof of theorem 5.

4.2 One-step priority queues

In a red-black priority queue, the height of any node is logarithmically bounded. This is a sufficient condition, but not a necessary condition. Actually, the more nodes in the right subtree, the better. This can be allowed by relaxing the conditions for the right subtree, as, for example, in the following definition of the rank properties r_o :

For any node v in any heap tree in the forest,

- if v is a root with no child, $r_o(v) = 1$,
- if v is a root with a right child, $r_o(v) = r_o(\text{right}(v)) + 1$,
- if v is a non-root node with at most one child, $r_o(v) = 1$,
- if v is a left child, $r_o(p(v)) = r_o(v) + 1$,
- if v is a right child, $r_o(p(v)) \leq r_o(v) + 1$.

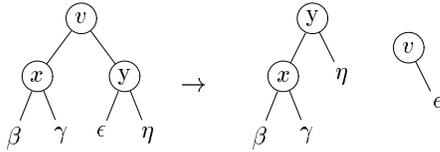
A *one-step priority queue* Q_o is a forest of heap trees with the constants $\alpha = 1/2$ and $\kappa = 1$, satisfying the rank properties r_o . The equation $r_o(p(v)) = r_o(v) + 1$ is the source of the name *one-step priority queue*. It states, that $r_o(v) = l(v)$. At first, it seems difficult to maintain this equation, but together with the inequality $r_o(p(v)) \leq r_o(v) + 1$, it gives some very simple rebalancing operations, which can be seen in the appendix.

The amortized running time is $\Theta(1)$ for REBALANCE, choosing the number of siblings with equal

rank as the potential function. By the fact that $r_o(v) \leq h_\alpha(v)$ and the above discussion, r_o satisfies the balancing conditions. When linking two trees of equal rank, we increase the rank of the new root by one. The linking runs in constant time and satisfies the definition of r_o . In conclusion, Q_o is a ranked priority queue. The running times for Q_o are the same as for a red-black priority queue up to constant factors (see theorem 5).

4.3 Other balancing schemes and variants

The weighted height properties from *AVL-trees* [1] can be used in a balancing scheme r_{avl} , too, if we treat the root separately as for red-black priority queues. The $CUT(v)$ operation is slightly modified: If $r_{avl}(\text{left}(v)) = r_{avl}(v) - 1$, we remove T_v . Otherwise, we only remove a part of T_v :



Thus, the balancing condition for node x (respectively node y) in the remaining tree T after a cut is violated by at most one. This makes it easy to rebalance T using a straightforward implementation of the $REBALANCE$ operation. The running times are the same as for a red-black priority queue up to constant factors (see theorem 5).

We note that for AVL priority queues, it is possible to link two trees when the ranks of the two root nodes differ by one. This can be done in constant time such that the resulting tree satisfies the r_{avl} conditions.

The AVL and the red-black balancing scheme share the property that the height of any node is logarithmically bounded. This is not a necessary condition and variants for such balancing schemes can be defined as follows. We *relax* the scheme by removing any upper bound on the rank of any node's right subtree. This relaxation is used in the one-step priority queue.

The standard implementation of ranked priority queues uses three pointers and one information field per node. A variant of the ranked priority queue can be implemented using only two pointers and one information field per node. A tree is implemented using *child-sibling* links. Each node contains a left pointer to its left child (if any), and a sibling pointer, which points to its right sibling, if any, and otherwise to its parent (if any).

Variants of ranked priority queues can be implemented using simpler data structures than forest

pointers and a tree list. This can easily be done without affecting the amortized time bounds, and may thus be preferred in applications where only the amortized time bounds are of interest. One variant is to drop the tree-list. This affects only the worst-case time bounds for $INSERT$ and $MELD$. Another variant is to use *lazy melding* [7] instead of the tree-list. This improves the worst-case time bound for $MELD$ at the expense of the worst-case time bound for $DELETEMIN$ and $DELETE$.

4.4 (a, b) -priority queues

The (a, b) -trees [9] are another class of balanced search trees. If $b = 2a - 1$, (a, b) -trees are known as B-trees, which were introduced by Bayer and McCreight [3]. In this section, we show how to implement efficient, mergeable priority queues using the ideas from (a, b) -trees and the previous sections. These priority queues are particularly well suited for external storage.

We use node-oriented storage instead of the usual leaf-oriented storage scheme. We define the *rank* of a node to be the height of the node. A node contains the following: the rank of the node, a parent pointer, a *leftmost child-pointer*, and $b - 1$ *sets*, each containing a key value, data, and a child-pointer. We denote a child, which is not a leftmost child, a *right child*.

Definition 5 An (a, b) -heap tree, $a \geq 2$, $b \geq 2a - 1$, is a tree that satisfies the following conditions:

- (i) the root node has at most one child, which is then a right child,
- (ii) for any non-root node v , v has zero or between a and b children.
- (iii) all nodes with no children have the same depth, and have between $a - 1$ and $b - 1$ sets,
- (iv) for any key value v , v is less than or equal to any key value in the associated subtree,

Definition 6 An (a, b) -priority queue is a forest of (a, b) -heap trees.

An example of an (a, b) -heap tree is shown to the left in figure 4. Note that, through the "binarization" described in [8] and the natural correspondance described in section 2.1, T can be represented as a heap-ordered tree \tilde{T} . In this tree, for any node v , we have:

- if the rank of v is one, then v has no children,
- if the rank of v is $r > 1$, then v has between $a - 1$ and $b - 1$ children of each rank s , $s = 1, 2, \dots, r - 1$.

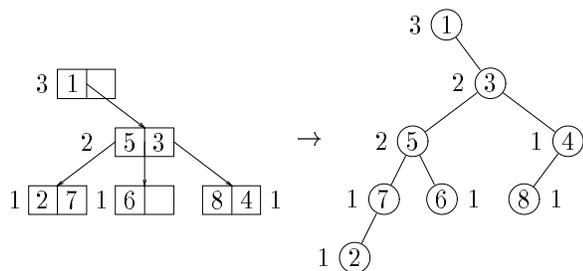


Figure 4: An example of a $(2,3)$ -heap tree is shown to the left. Data are stored with the key values and are not shown. The binarized representation of the tree is shown to the right.

Using this data structure, we can achieve the same time bounds as for red-black priority queues up to constant factors (see theorem 5). Our re-balancing operations are somewhat similar to the re-balancing operations used in (a, b) -trees. In (a, b) -heap trees, we do, however, need to make a sharp distinction between leftmost children and right children.

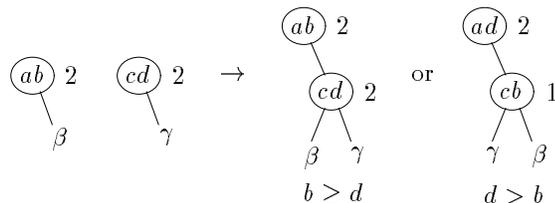
A relaxation of (a, b) -heap trees can be defined similarly to the definition of relaxation of ranked priority queues in section 4.3.

4.5 Double-ended priority queues

To implement a mergeable double-ended priority queue, we make some changes to the heap tree. Instead of storing a single item per node, we store two items per node. Thus, a node contains two key values. If the number of items in the priority queue is odd, the key value of one of the items is contained twice in a specified node (to simplify the following discussion). The tree is *twin-half-ordered*:

- Any node is either a *type 1* or a *type 2* node. The root is a type 2 node.
- If a node v , contains key values a and b , $a \leq b$, the key b is greater than or equal to any key in v 's right subtree. If v is a type 1 (type 2) node, a is less than or equal to any key in v 's left (right) subtree.

The linking can be done in constant time:



The un-linking can similarly be done in constant time. If we restrict ourselves to looking only at binary trees which arise from a conversion from a binomial tree, then this data structure has many similarities with the one used in [10]. Using this data structure, we can achieve the following time bounds:

Theorem 8 *With the above modifications, one can implement a mergeable double-ended priority queue. The worst-case running time is $\Theta(1)$ for MAKEPRIORITYQUEUE, INSERT, FINDMIN, and FINDMAX; $\Theta(\min(\log(n_1), \log(n_2)))$ for MELD; $O(\log n)$ for CHANGEKEY, DELETEMIN, DELETEMAX, and DELETE; and $\Theta(n)$ for BUILD. The amortized running time is $\Theta(1)$ for MELD, and the same as the worst-case bounds for the other operations.*

All operations but CHANGEKEY are implemented as in section 3 with the obvious modifications. The CHANGEKEY operation can be implemented in several ways. A simple implementation is to use a *bubble-up* and *trickle-down* technique (see [14]), where the trickle-down operation is implemented in a bottom-up fashion. The bubble-up operation can be replaced by a cut-and-rebalance technique (see section 3). The trickle-down operation can be replaced by a cut-and-rebalance technique combined with a repeated un-link technique (see the DELETEMIN in section 3).

These bounds match the previously best known bounds [5, 10] for all operations but insertion. Due to the tree list, we improve the time bound to $\Theta(1)$ for insertion. It is possible to remove the restriction that the trees must arise from conversions from binomial trees. We do this using the ideas presented in section 2.2.

Acknowledgements

The author is very grateful to Joan Boyar for many valuable discussions and for constructive criticism. The author would also like to thank Kim S. Larsen for his encouragement and useful comments. The author thanks Rolf Fagerberg for useful comments and for pointing out reference [12] just before the deadline, for the final version of this paper, for this conference.

References

- [1] G. M. Adel'son-Vel'skiĭ and E. M. Landis. An Algorithm for the Organisation of Information.

Dokl. Akad. Nauk SSSR, 146:263–266, 1962. In Russian. English translation in *Soviet Math. Dokl.*, 3:1259–1263, 1962.

- [2] R. Bayer. Symmetric Binary B-Trees: Data Structure and Maintenance Algorithms. *Acta Informatica*, 1:290–306, 1972.
- [3] R. Bayer and E. McCreight. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica*, 1:173–189, 1972.
- [4] Mark R. Brown. Implementation and analysis of binomial queue algorithms. *SIAM Journal on Computing*, 7:298–319, 1978.
- [5] Yuzheng Ding and Mark Allen Weiss. The relaxed min-max heap — A mergeable double-ended priority queue. *Acta Informatica*, 30(3):215–231, 1993.
- [6] James R. Driscoll, Harold N. Gabow, Ruth Shrairman, and Robert E. Tarjan. Relaxed Heaps: An Alternative to Fibonacci Heaps with Applications to Parallel Computation. *Communications of the ACM*, 31:1343–1354, 1988.
- [7] Michael L. Fredman and Robert Endre Tarjan. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *Journal of the ACM*, 34:596–615, 1987.
- [8] Leo J. Guibas and Robert Sedgewick. A Dichromatic Framework for Balanced Trees. In *19th IEEE FOCS*, pages 8–21, 1978.
- [9] Scott Huddleston and Kurt Mehlhorn. A New Data Structure for Representing Sorted Lists. *Acta Informatica*, 17:157–184, 1982.
- [10] C. M. Khoong and H. W. Leong. Double-Ended Binomial Queues. In *ISAAC '93 — Algorithms and Computation*, volume 762 of *Lecture Notes in Computer Science*, pages 128–137. Springer-Verlag, 1993.
- [11] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, 1968.
- [12] Gary L. Peterson. A Balanced Tree Scheme for Meldable Heaps With Updates. Technical Report GIT-ICS-87-23, School of Information and Computer Science, Georgia Institute of Technology, 1987.
- [13] Jean Vuillemin. A Data Structure for Manipulating Priority Queues. *Communications of the ACM*, 21:309–315, 1978.
- [14] J. W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7:347–348, 1964.

Appendix

Rebalancing steps used in red-black priority queues.

Squared nodes are bad nodes. If the bad node is a *right* child: (a) Push the problem towards the root. (b) - (d) Produce a terminating case. In case (b) and (d), perform a right swap if $a < b$.

If the bad node is a *left* child: (e) Push the problem towards the root. (f) - (h) Produce a terminating case.

