Relaxed Fibonacci heaps: An alternative to Fibonacci heaps with worst case rather than amortized time bounds[*]

Chandrasekhar Boyapati C. Pandu Rangan

Department of Computer Science and Engineering
Indian Institute of Technology, Madras 600036, India
Email: rangan@iitm.ernet.in

November 1995

Abstract

We present a new data structure called relaxed Fibonacci heaps for implementing priority queues on a RAM. Relaxed Fibonacci heaps support the operations **find minimum, insert, decrease key** and **meld**, each in $O(1)$ worst case time and **delete** and **delete min** in $O(\log n)$ worst case time.

Introduction

The implementation of priority queues is a classical problem in data structures. Priority queues find applications in a variety of network problems like single source shortest paths, all pairs shortest paths, minimum spanning tree, weighted bipartite matching etc. [1] [2] [3] [4] [5]

In the amortized sense, the best performance is achieved by the well known Fibonacci heaps. They support **delete** and **delete min** in amortized $O(\log n)$ time and **find min**, **insert**, **decrease key** and **meld** in amortized constant time.

Fast meldable priority queues described in [1] achieve all the above time bounds in worst case rather than amortized time, except for the **decrease key** operation which takes $O(\log n)$ worst case time. On the other hand, relaxed heaps described in [2] achieve in the worst case all the time bounds of the Fibonacci heaps except for the **meld** operation, which takes $O(\log n)$ worst case

[*]Please see Errata at the end of the paper.

time. The problem that was posed in [1] was to consider if it is possible to support both **decrease key** and **meld** simultaneously in constant worst case time.

In this paper, we solve this open problem by presenting relaxed Fibonacci heaps as a new priority queue data structure for a Random Access Machine (RAM). (The new data structure is primarily designed by relaxing some of the constraints in Fibonacci heaps, hence the name relaxed Fibonacci heaps.) Our data structure supports the operations **find minimum, insert, decrease key** and **meld**, each in $O(1)$ worst case time and **delete** and **delete min** in $O(\log n)$ worst case time. The following table summarizes the discussion so far.

	delete	delete min	find min	insert	decrease key	meld
Fibonacci heaps **(amortized)**	O(log n)	O(log n)	O(1)	O(1)	O(1)	O(1)
Fast meldable heaps	O(log n)	O(log n)	O(1)	O(1)	**O(log n)**	O(1)
Relaxed heaps	O(log n)	O(log n)	O(1)	O(1)	O(1)	**O(log n)**
Relaxed Fibonacci heaps	O(log n)	O(log n)	O(1)	O(1)	**O(1)**	**O(1)**

For simplicity we assume that all priority queues have at least three elements. We use the symbol Q to denote a relaxed Fibonacci heap and n to denote the size of a priority queue it represents. Unless otherwise mentioned, all the time bounds we state are for the worst case.

In Section 1, we prove some results regarding binomial trees which will be central to establishing the $O(\log n)$ time bound for the **delete min** operation. In Section 2.1, we describe the relaxed Fibonacci heaps. In Section 3, we describe the various operations on relaxed Fibonacci heaps.

1 Some results regarding binomial trees

Consider the following problem. We are given a binomial tree B [5] whose root has degree d. The children of any node N in B are arranged in the increasing order of their degrees. That is, if the children of N are N_0, N_1, ... N_{d-1}, then $N_i.degree = i$.

We are to remove some nodes from this tree. Every time a node gets removed, the entire subtree rooted at that node also gets removed. Suppose the resulting tree is B', which is not necessarily a binomial tree. For any node $N \in B'$, $N.lost$ denotes the number of children lost by N. For any node $N \in B'$, define W_N as the weight of node N as follows: $W_N = 0$, if $N.lost = 0$. Else, $W_N = 2^{N.lost-1}$. Define weight $W = \sum W_N$ for all nodes $N \in B'$.

Given an upper bound on weight W that B can lose, let B' be the tree obtained by removing as many nodes from B as possible.

Lemma 1 *B' defined above has the following properties:*

1. *Let N be any node in B'. If $N.lost = 0$ and if N' is a descendant of N then $N'.lost = 0$.*
2. *Let N be any node in B' such that $N.lost = l$. Then the children lost by N are its last l children (which had the highest degrees in B).*
3. *Let N_i and N_j be two nodes that have the same parent such that $N_i.degree > N_j.degree$, in B. Then, in B', $N_i.lost \geq N_j.lost$.*
4. *Let a node N have four consecutive children N_i, N_{i+1}, N_{i+2} and N_{i+3} belonging to B'. Then it cannot be true that $N_i.lost = N_{i+1}.lost = N_{i+2}.lost = N_{i+3}.lost > 0$.*

Proof: If any of the 4 statements above are violated, then we can easily show that by reorganizing the removal of nodes from B, we can increase the number of nodes removed from B without increasing the weight W removed. In particular, if statement 4 is violated, then we can reorganize the removal of nodes by increasing $N_{i+3}.lost$ by one and decreasing $N_i.lost$ and $N_{i+1}.lost$ by one each. $\square$

Lemma 2 *Let B be a binomial tree whose root has degree d. Let $d > 4d_0 + 3$, where $d_0 = \lceil \log n_0 \rceil$, for some n_0. Let the upper bound on the weight that B can loose be $2n_0$. Then, B' will have more than n_0 nodes in it.*

Proof: Let the children of the root of B be B_0, B_1, ..., B_{d-1}. We first claim that $B_{d_0}.lost = 0$. Or else, it follows trivially from the previous lemma that $B_{4d_0+3}.lost \geq d_0 + 2$, because statement 4 implies that there can be at most three consecutive nodes which have lost the same number of children. This in turn implies that the weight lost by B is greater than $2^{d_0+1} \geq 2n_0$, which is not possible. Thus our claim holds.

Since $B_{d_0}.lost = 0$, no nodes are deleted from the subtree rooted at B_{d_0}, according to statement 1 of the previous lemma. Thus the number of nodes in B' is greater than $2^{d_0} \geq n_0$. $\square$

2 The relaxed Fibonacci heaps

Our basic representation of a priority queue is a **heap ordered tree** where each node contains one element. This is slightly different from binomial heaps [4] and Fibonacci heaps [3] where the representation is a forest of heap ordered trees.

2.1 Properties of the relaxed Fibonacci heaps

We partition the children of a node into two types, type I and type II. A relaxed Fibonacci heap Q must satisfy the following constraints.

1. A node of type I has at most one child of type II. A node of type II cannot have any children of type II.

2. With each node N we associate a field *degree* which denotes the number of children of type I that N has. (Thus the number of children of any node N is either $degree + 1$ or *degree* depending on whether N has a child of type II or not.)

   (a) The root R is of type I and has *degree* zero. R has a child R' of type II.

   (b) Let $R'.degree = k$. Let the children of R' be R_0, R_1, ..., R_{k-1} and let $R_0.degree \leq R_1.degree \leq ... \leq R_{k-1}.degree$. Then, $R'.degree = k \leq R_{k-1}.degree + 1$.

3. With each node N of type I we associate a field *lost* which denotes the number of children of type I lost by N since its *lost* field was last reset to zero.

   For any node N of type I in Q, define W_N as the weight of node N as follows: $W_N = 0$, if $N.lost = 0$. Else, $W_N = 2^{N.lost-1}$.

   Also for any node N of type I, define w_N as the increase in W_N due to N losing its last child. That is, $w_N = 0$ or 1 or $2^{N.lost-2}$ respectively depending on whether $N.lost = 0$ or 1 or greater than one.

   Define weight $W = \sum W_N$ for all N of type I in Q.

   Every relaxed Fibonacci heap has a special variable P, which is equal to one of the nodes of the tree. Initially, $P = R$.

   (a) $R.lost = R_0.lost = R_1.lost = ... = R_{k-1}.lost = 0$.

   (b) Let N be any node of type I. Let $N.degree = d$ and let the children of N of type I be N_0, N_1, ..., N_{d-1}. Then, for any N_i, $N_i.degree + N_i.lost \geq i$.

   (c) $W \leq n + w_P$.

4. Associated with Q we have a list $L_M = (M_1, M_2, ..., M_m)$ of all nodes of type II in Q other than R'. Each node M_i was originally the R' of some relaxed Fibonacci heap Q_i till some **meld** operation. Let n_i denote the number of nodes in Q_i just before that **meld** operation.

   (a) $M_i.degree \leq 4\lceil \log n_i \rceil + 4$.

   (b) $n_i + i \leq n$.

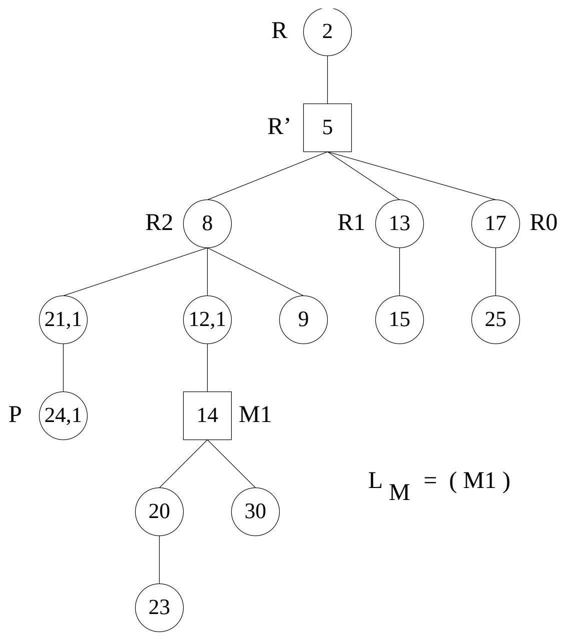


Figure 1: A relaxed Fibonacci heap

Example

Figure 1 shows a relaxed Fibonacci heap. The nodes of type I are represented by circles and the nodes of type II are represented by squares. Each node N contains either $N.element, N.lost$ or just $N.element$ if $N.lost = 0$. The node P and the list L_M are also shown.

Lemma 3 *The heap order implies that the minimum element is at the root.*

Lemma 4 *For any node N of type I, $N.degree \leq 4d_0 + 3$, where $d_0 = \lceil \log n \rceil$.*

Proof: On the contrary, say $N.degree = d > 4d_0 + 3$. If $P.lost > 1$, then W, when expanded, will contain the term $W_P = 2^{P.lost-1}$. Thus according to property 3c, $2^{P.lost-1} \leq W \leq n + w_P = n + 2^{P.lost-2}$. That is, $2^{P.lost-2} \leq n$. Hence, $W \leq 2n$. This inequality obviously holds true even if $P.lost \leq 1$. Thus the weight lost by the subtree rooted at N, say T_N, is not more than $2n$.

Let us now try to estimate the minimum number of nodes present in T_N. Let us first remove all the children of type II in T_N and their descendants. In the process, we might be decreasing the number of nodes in the tree but we will not be increasing $N.degree$.

In the resulting T_N, we know from property 3b that for any $(i+1)^{th}$ child of any node N', $N'_i.degree + N'_i.lost \geq i$. But for the binomial tree problem described in Section 1, $N'_i.degree + N'_i.lost = i$. Thus the minimum number of nodes present in T_N is at least equal to the minimum number of nodes present in a binomial tree of degree d after it loses a weight less than or equal to $2n$. But according to Lemma 2, this is more than n. Thus, T_N has more than n nodes, which is obviously not possible. $\square$

Lemma 5 *For any node N in Q, the number of children of $N \leq 4d_0 + 4$, where $d_0 = \lceil \log n \rceil$.*

Proof: If N is a node of type I, the number of children of $N \leq N.\mathit{degree} + 1 \leq 4d_0 + 4$, according to the previous lemma.

From property 2b, $R'.\mathit{degree} = k \leq R_{k-1}.\mathit{degree} + 1 \leq 4d_0 + 4$, according to the previous lemma, since R_{k-1} is of type I. Thus, number of children of $R' = R'.\mathit{degree} \leq 4d_0 + 4$.

If N is any node of type II other than R', N is equal to some M_i in L_M, according to property 4. According to property 4a, $M_i.\mathit{degree} \leq 4\lceil \log n_i \rceil + 4$. But according to property 4b, $n_i < n$. Thus the number of children of $N = M_i.\mathit{degree} \leq 4d_0 + 4$. $\square$

Remarks

The restrictions imposed by property 3c are much weaker than those in fast meldable queues [1] or in relaxed heaps [2]. But according to the above lemma, the number of children of any node is still $O(\log n)$. We believe that the introduction of property 3c is the most important contribution of this paper.

2.2 Representation

The representation is similar to that of Fibonacci heaps as described in [5]. The children of type I of every node are stored in a doubly linked list, sorted according to their degrees. Besides, each node of type I has an additional child pointer which can point to a child of type II.

To preserve the sorted order, every node that is inserted as a child of R' must be inserted at the appropriate place. To achieve this in constant time, we maintain an auxiliary array A such that $A[i]$ points to the first child of R' of degree i.

We will also need to identify two children of R' of same degree, if they exist. To achieve this in constant time, we maintain a linked list L_P of pairs of nodes that are children of R' and have same degree and a boolean array B such that $B[i]$ is true if and only if the number of children of R' of degree i is even.

Besides, we will also require to identify some node N in Q such that $N.\mathit{lost} > 1$, if such a node exists. To implement this in constant time, we maintain a list L_L of all nodes N in Q such that $N.\mathit{lost} > 1$.

3 Operations on relaxed Fibonacci heaps

In this section, we will describe how to implement the various operations on relaxed Fibonacci heaps. But before we do that, we will describe a few basic operations namely, **link**, **add**, **reinsert** and **adjust**.

Though we will not be mentioning it explicitly every time, we will assume that whenever a node of type I loses a child, its *lost* field is automatically incremented by one unless the node is R or a child of R'. Similarly, whenever a node is inserted a child of R', its *lost* field is automatically reset to zero. We also assume that if node P gets deleted from the tree after a **delete min** operation, then P is reset to R to ensure that node P still belongs to Q.

3.1 Some basic operations

The **link** operation is similar to the linking of trees in binomial heaps and Fibonacci heaps.

Algorithm 1 : Link(Q, R_i, R_j)

```
/* Link the two trees rooted at $R_i$ and $R_j$ into one tree */
/* $R_i$ and $R_j$ are children of $R'$ and have equal degrees, say $d$ */

1. Delete the subtrees rooted at $R_i$ and $R_j$ from $Q$
2. If $R_i.element > R_j.element$ then $Swap(R_i, R_j)$
3. Make $R_j$ the last child of $R_i$
4. Make $R_i$ (whose degree now $= d + 1$) a child of $R'$ of $Q$
```

Algorithm 2 : Add(Q, N)

```
/* Add the tree rooted at $N$ to the relaxed Fibonacci heap $Q$ */

1. Make $N$ a child of $R'$ of $Q$
2. If $N.element < R'.element$ then $Swap(N.element, R'.element)$
3. If $R'.element < R.element$ then $Swap(R'.element, R.element)$
4. If among the children of $R'$ there exist any two different nodes $R_i$ and $R_j$
   such that $R_i.degree = R_j.degree$ then $Link(Q, R_i, R_j)$
```

Algorithm 3 : ReInsert(Q, N)

```
/* Remove $N$ from $Q$ and insert it as a child of $R'$ */
/* $N$ is of type I and $N \neq R$ */
/* Return the original parent of $N$ */

1. $Parent \leftarrow N.parent$
2. Delete the subtree rooted at $N$ from $Q$
3. $Add(Q, N)$
4. Return $Parent$
```

The algorithm **adjust** is called by **decrease key**, **meld** and **delete min** operations. **Adjust** restores property 3c provided some preconditions are satisfied, which will be explained shortly.

Algorithm 4 : Adjust(Q, P_1, P_2)

```
1. If $M.lost \leq 1$ for all nodes $M$ in $Q$ then return
2. If $P_1.lost > P_2.lost$ then $M \leftarrow P_1$ else $M \leftarrow P_2$
3. If $M.lost \leq 1$ then $M \leftarrow M'$ for some node $M'$ in $Q$ such that $M'.lost > 1$
4. $P \leftarrow ReInsert(Q, M)$
```

Lemma 6 *All the operations described above take constant time in the representation described in Section 2.2.*

Proof: The **add** operation needs to identify two children of R' of same degree, if they exist. This can be done using the list L_P and the array B in constant time. After removing those children and linking them into one tree, we can use the array A to insert it at the appropriate place in the child list of R'. L_P , B and A can obviously be maintained in constant time.

The **adjust** operation needs to identify some node N in Q such that $N.lost > 1$, if such a node exists. This can be done in constant time using the list L_L.

The rest of the operations trivially take constant time. □

Lemma 7 *Property 2b described in Section 2.1 is preserved under the add operation.*

Proof: After every **add** operation in which Step 4 executed, the number of children of R' does not change. Also, after every **add** operation in which Step 4 is not executed, there is at most one child of R' of each degree. Thus property 2b is preserved under the add operation. □

Lemma 8 *Property 3b is preserved under the reinsert operation.*

Proof: Let the children of a node N be N_0, N_1, ..., N_{d-1}. Whenever N loses its $(i+1)^{th}$ child N_i, then for all $j > i$, N_j now becomes N_{j-1}, even though its *degree* and *lost* fields are not changed. Thus, property 3b is never violated. □

For any variable V, let V^- denote the value of the variable before an operation and let V^+ denote the value of the variable after the operation.

Lemma 9 *In operation adjust, if $P_1 \neq P_2$ and if $W^- \leq n + w_{P_1} + w_{P_2}$, then $W^+ \leq n + w_{P^+}$.*

Proof: If the condition in Step 1 of adjust holds, then $W^+ =$ number of nodes N in Q such that $N.lost = 1$. Thus $W^+ \leq n$.

Else, Steps 2 and 3 ensure that $W_M = max(W_{P_1}, W_{P_2}, 2)$. Now, if $P_1.lost \leq 1$ then $w_{P_1} \leq 1$. Thus $W_M \geq 2w_{P_1}$. Otherwise $P_1.lost > 1$. Then $W_M \geq W_{P_1} = 2^{P_1.lost-1} = 2 \times 2^{P_1.lost-2} = 2w_{P_1}$. Similarly, $W_M \geq 2w_{P_2}$. Thus, $W_M \geq w_{P_1} + w_{P_2}$.

Thus we have, $W^+ = W^- - W_M + w_{P^+} \leq (n + w_{P_1} + w_{P_2}) - W_M + w_{P^+} = n + w_{P^+} - (W_M - w_{P_1} - w_{P_2}) \leq n + w_{P^+}$ □

Lemma 10 *In operation adjust, if $P_1 = P_2 = P^-$ and if $W^- \leq (n+1) + w_{P^-}$, then $W^+ \leq n + w_{P^+}$.*

Proof: As before, if the condition in Step 1 holds, then $W^+ =$ number of nodes N in Q such that $N.lost = 1$. Thus $W^+ \leq n$.

Else, if $P^-.lost \leq 1$ then $w_{P^-} \leq 1$. Thus $W_M \geq 2 > w_{P^-}$. Otherwise, $P^-.lost > 1$. Then, $W_M = W_P^- = 2^{P^-.lost-1} = 2 \times 2^{P^-.lost-2} = 2w_{P^-}$. Thus, $W_M > w_{P^-}$, which implies that, $W_M \geq w_{P^-} + 1$

Thus we have, $W^+ = W^- - W_M + w_{P^+} \leq (n + 1 + w_{P^-}) - W_M + w_{P^+} = n + w_{P^+} - (W_M - w_{P^-} - 1) \leq n + w_{P^+}$. □

3.2 The find min operation

Algorithm 5 : FindMin(Q)
/* Return the minimum element in the relaxed Fibonacci heap Q */

```
1. Return R.element
```

Lemma 11 *The find min operation described above returns the minimum element in Q in constant time.*

3.3 The insert operation

Binomial heaps have a very restrictive property that there will be at most one node of each degree in the root list. Hence an **insert** operation can degenerate to take $O(\log n)$ time. On the other hand, the **insert** operation in the Fibonacci heaps just puts the node in the root list and leaves all the work to **delete min**, which can take $O(n)$ worst case time as a result.

Instead, we have introduced property 2b so that the **insert** operation can be performed in $O(1)$ time and the number of children of R' still remains $O(\log n)$.

Algorithm 6 : Insert(Q, e)
/* Insert the element e into the relaxed Fibonacci heap Q */

```
1. Form a tree with a single node $N$ of type I consisting of element $e$
2. Add($Q$, $N$)
```

Lemma 12 *All the properties described in Section 2.1 are maintained under the insert operation.*

Proof: Since **insert** works by calling **add**, property 2b is preserved under the **insert** operation, according to Lemma 7. The rest of the properties are trivially preserved under **insert**. □

Lemma 13 *The insert operation takes constant time in the representation described in Section 2.2.*

Proof: Follows from Lemma 6. □

3.4 The decrease key operation

Our implementation of the **decrease key** operation is somewhat similar to that of the Fibonacci heaps. With every **decrease key** operation in the Fibonacci heaps, the corresponding node is deleted and is put in the root list. Similarly, we also delete the corresponding node and insert it as a child of R'. However, care is to be taken to see to it that too many children are not deleted, otherwise the **delete min** operation might degenerate to take $O(n)$ time in the worst case. Fibonacci heaps handle this by marking a node whenever it loses its first child and deleting the node itself (and putting it in the root list) whenever it loses a second child. However, in a particular **decrease key** operation, this effect might cascade up due to which the operation might take $O(h)$ time in the worst case, where h is the height of the tree.

Instead of imposing a local restriction as in the Fibonacci heaps (that a node can lose at most one child), we have relaxed this into a global restriction on the number of children deleted by introducing property 3c instead.

Algorithm 7 : DecreaseKey(Q, N, e)

```
/* Decrease the value of the element in node $N$ of $Q$ by $e(> 0)$ */

1. $N.element \leftarrow N.element - e$
2. If ($N = R$ or $R'$) and ($R'.element < R.element$) then
   $Swap(R'.element, R.element)$; return
3. If ($N$ is of type II) and ($N.element < N.parent.element$) then
   $Swap(N.element, N.parent.element)$; $N \leftarrow N.parent$
4. If $N.element \geq N.parent.element$ then return
5. $P' \leftarrow ReInsert(Q, N)$
6. $Adjust(Q, P, P')$
```

Lemma 14 *All the properties described in Section 2.1 are maintained under the decrease key operation.*

Proof: Since any node is finally inserted as a child of R' only through the **add** operation, property 2b is preserved, according to Lemma 7. Also, since **decrease key** works by calling **reinsert**, property 3b is preserved according to Lemma 8.

Just before Step 5, the weight $W \leq n + w_P$, according to property 3c. Hence, just after Step 5, weight $W \leq n + w_P - W_N + w_{P'} \leq n + w_P + w_{P'}$. Therefore according to Lemma 9, just after Step 6, weight $W \leq n + w_P$. Thus property 3c is preserved under the **decrease key** operation.

The rest of the properties are trivially preserved under the **decrease key** operation. □

Lemma 15 *The decrease key operation takes constant time in the representation described in Section 2.2.*

Proof: Follows from Lemma 6. □

3.5 The meld operation

The **meld** operation essentially adds the root R of one relaxed Fibonacci heap, say Q_2, as a child of R' of the other relaxed Fibonacci heap Q_1. Whenever a variable name might cause confusion as to whether the variable belongs to Q_1 or Q_2 or Q, we will prefix it appropriately.

Algorithm 8 : Meld(Q_1, Q_2)

```
/* Meld the two relaxed Fibonacci heaps $Q_1$ and $Q_2$ into $Q$ */
/* Return $Q$ */

1a. If $Q_1.R'.element > Q_2.R'.element$ then $Swap(Q_1, Q_2)$
1b. $Add(Q_1, Q_2.R)$
1c. If $Q_2.R.element < Q_1.R'.element$ then $Swap(Q_2.R.element, Q_1.R'.element)$
1d. If $Q_1.R'.element < Q_1.R.element$ then $Swap(Q_1.R'.element, Q_1.R.element)$

2a. Add the node $Q_2.R'$ to the tail of $Q_1.L_M$
2b. Concatenate $Q_2.L_M$ with $Q_1.L_M$ by adding the head of $Q_2.L_M$ after the tail of $Q_1.L_M$

3. $Adjust(Q_1, Q_1.P, Q_2.P)$
4. **Return** $Q_1$
```

Lemma 16 *Property 4b is preserved under the meld operation.*

Proof: Let $Q_1.L_M = (M_1^1, M_2^1, ..., M_{m_1}^1)$. Since each M_i and its parent form unique nodes in Q, there are at least two unique nodes in Q per M_i. Thus, $Q_1.n > 2m_1 \geq m_1$. Or, $Q_1.n \geq m_1 + 1$.

Also, let $Q_2.L_M = (M_1^2, M_2^2, ..., M_{m_2}^2)$. Then, $Q.L_M = (M_1^1, ..., M_{m_1}^1,\ Q_2.R',\ M_1^2, ..., M_{m_2}^2) = (M_1, ..., M_{m_1+m_2+1})$.

We will prove the lemma by considering the three separate cases namely $1 \leq i \leq m_1$, $i = m_1 + 1$ and $m_1 + 2 \leq i \leq m_1 + m_2 + 1$.

If $1 \leq i \leq m_1$ then $Q.n_i + i \leq Q_1.n < Q.n$, since these elements originally belonged to Q_1 and they still remain in the same position in L_M.

If $i = m_1 + 1$ then $Q.n_{m_1+1} + (m_1 + 1) = Q_2.n + (m_1 + 1) \leq Q_2.n + Q_1.n = Q.n$.

If $m_1 + 2 \leq i \leq m_1 + m_2 + 1$ then let $j = i - (m_1 + 1)$. Then $Q.n_i + i = Q_2.n_j. + j + (m_1 + 1) \leq Q_2.n + (m_1 + 1) \leq Q_2.n + Q_1.n = Q_n$. □

Lemma 17 *All the properties described in Section 2.1 are maintained under the meld operation.*

Proof: Since $Q_2.R$ is inserted as a child of $Q_1.R'$ through the **add** operation, property 2b is preserved, according to Lemma 7.

According to property3c, just before Step 3, the weight $W = Q_1.W + Q_2.W \leq Q_1.n + w_{Q_1.P} + Q_2.n + w_{Q_2.P} \leq n + w_{Q_1.P} + w_{Q_2.P}$. Therefore according to Lemma 9, just after Step 3, weight $W \leq n + w_P$. Thus property 3c is preserved under the **meld** operation.

According to the previous lemma, property 4b is preserved under the **meld** operation.The rest of the properties are trivially preserved under the **meld** operation. □

Lemma 18 *The meld operation takes constant time in the representation described in Section 2.2.*

Proof: Follows from Lemma 6. □

3.6 The delete min operation

Algorithm 9 : DeleteMin(Q)

```
/* Delete and return the minimum element in the relaxed Fibonacci heap $Q$ */

1a. $MinElement \leftarrow R.element$
1b. $R.element \leftarrow R'.element$
1c. $R'' \leftarrow$ The child of $R'$ containing the minimum element among the children of $R'$
1d. $R'.element \leftarrow R''.element$

2a. Delete the subtree rooted at $R''$ from $Q$
2b. **For** all children $N$ of type I of $R''$ **do** make $N$ a child of $R'$ of $Q$

3a. **If** $R''$ has no child of type II **then goto** Step 4.
3b. Let $M'$ be the child of type II of $R''$. $Insert(Q, M'.element)$
3c. **For** all children $N$ of $M$ **do** make $N$ a child of $R'$ of $Q$

4.  $Adjust(Q, P, P)$
```

```
5a. **If** $L_M$ is empty **then goto** Step 6
5b. $M \leftarrow Head(L_M)$; $L_M \leftarrow Tail(L_M)$
5c. Delete $M$ from $Q$
5d. $Insert(Q, M.element)$
5e. **For** all children $N$ of $M$ **do** make $M$ a child of $R'$ of $Q$

6.  **While** among the children of $R'$ there exist any two different nodes $R_i$ and $R_j$
    such that $R_i.degree = R_j.degree$ **do** $Link(Q, R_i, R_j)$

7.  **Return** MinElement
```

Lemma 19 *Property 4b is preserved under the delete min operation.*

Proof: Property 4b requires that $n_i + i \leq n$. After each **decrease key** operation, the value of n decreases by one. But each M_i also at least becomes M_{i-1}. Thus, the property is preserved under the **delete min** operation. □

Lemma 20 *All the properties described in Section 2.1 are maintained under the delete min operation.*

Proof: Since after any **delete min** operation R' has at most one child of each degree, property 2b holds after a **delete min** operation.

In a **delete min** operation, the number of nodes n decreases by one. Thus just before Step 4, $W \leq (n+1) + w_P$, according to property 3c. Therefore according to Lemma 9, just after Step 3, weight $W \leq n + w_P$. Thus property 3c is preserved under the **delete min** operation.

According to the previous lemma, property 4b is preserved under the **delete min** operation. The rest of the properties are trivially preserved under the **delete min** operation. □

Lemma 21 *It is easy to see that in the representation described in Section 2.2, the delete min operation takes $O(R'.degree + R''.degree + M'.degree + M.degree)$ $= O(\log n)$ time, according to Lemma 5.*

3.7 The delete operation

Algorithm 10 : Delete(Q, N)
/* Delete the node N from the priority queue Q and return $N.element$ */

```
1. $Element \leftarrow N.element$
2. $DecreaseKey(Q, N, \infty)$
3. $DeleteMin(Q)$
4. **Return** $Element$
```

Lemma 22 *It follows from the proofs of correctness and complexity of the decrease key and delete min operations that the delete operation described above deletes the node N from Q in $O(\log n)$ time.*

Conclusion

We can summarize the discussions in the previous sections by the following theorem.

Theorem 1 *An implementation of priority queues on a RAM exists that supports the operations find minimum, insert, decrease key and meld, each in $O(1)$ worst case time and delete and delete min in $O(\log n)$ worst case time.*

The central idea in designing this data structure has been the identification of the weak global constraint on the number of children lost(property 3c). The classification of the nodes into type I and type II has been done to facilitate the **meld** operation.

It is easy to see that these time bounds are optimal for any comparison based algorithm that performs the **meld** operation in sub-linear time.

References

[1] Gerth Stoling Brodal. *"Fast meldable priority queues"*. Proc. **4th** International Workshop, WADS, 282-290 (1995)

[2] James R. Driscoll, Harold N. Gabow, Ruth Shrairman and Robert E. Tarjan. *"Relaxed heaps: An alternative approach to Fibbonacci heaps with applications to parallel computing"*. Comm. ACM **31(11)**, 1343-1354 (1988)

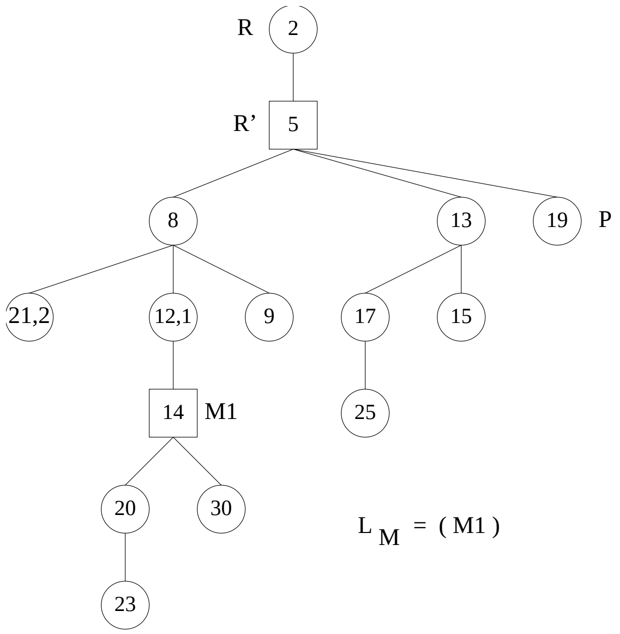
[3] Michael L. Fredman and Robert E. Tarjan. *"Fibonacci heaps and their uses in improved network optimization algorithms"*. Proc. **25th** Ann. Symp. on Foundations of Computer Science, 338-346 (1984)

[4] Jean Vuillemin. *"A data structure for manipulating priority queues"*. Comm. ACM **21(4)**, 309-315 (1978)

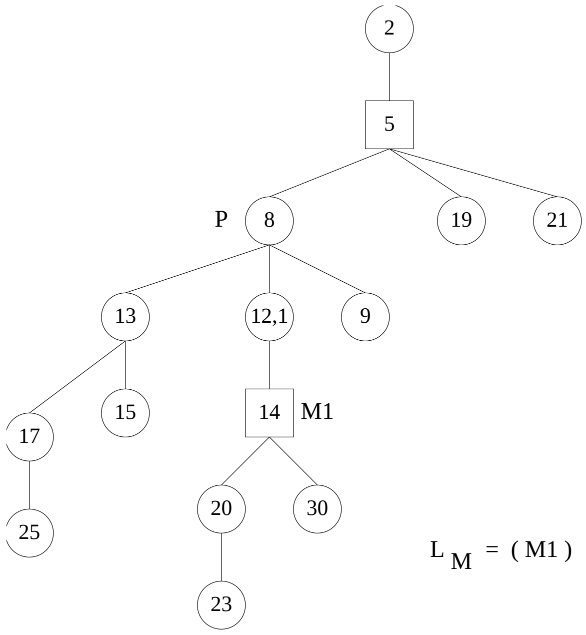
[5] Thomas H. Corman, Charles E. Leiserson and Ronald R. Rivest. *"Introduction to algorithms"*. The MIT Press, Cambridge, Massachusetts. (1989)

A Decreasing the value of the element 24 to 19 in the tree in Figure 1

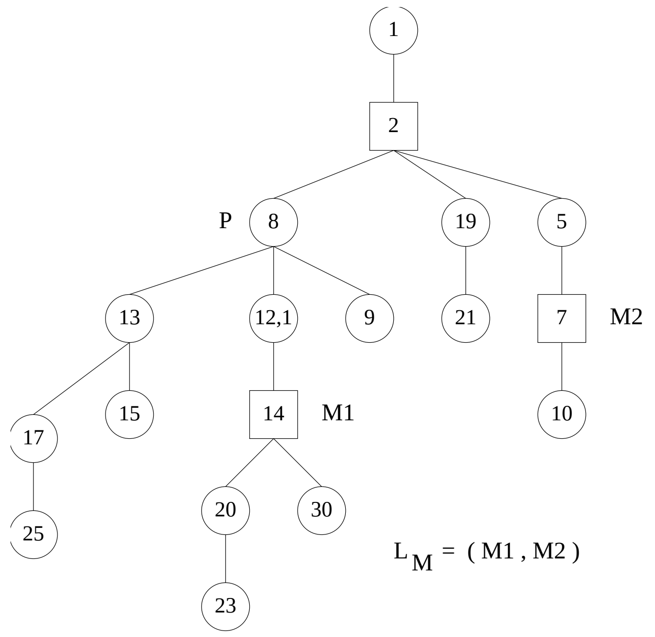
A.1 After Step 5 of decrease key operation



A.2 After the decrease key operation

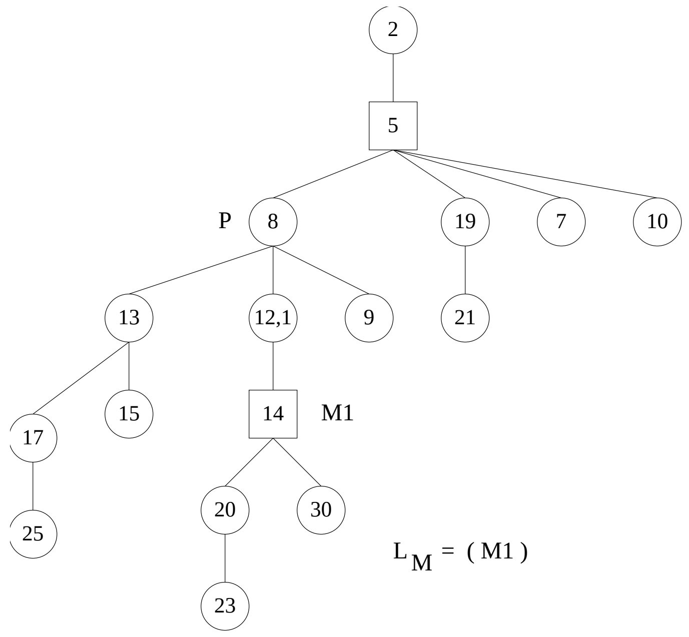


B After melding with a priority queue containing only 1, 7 and 10

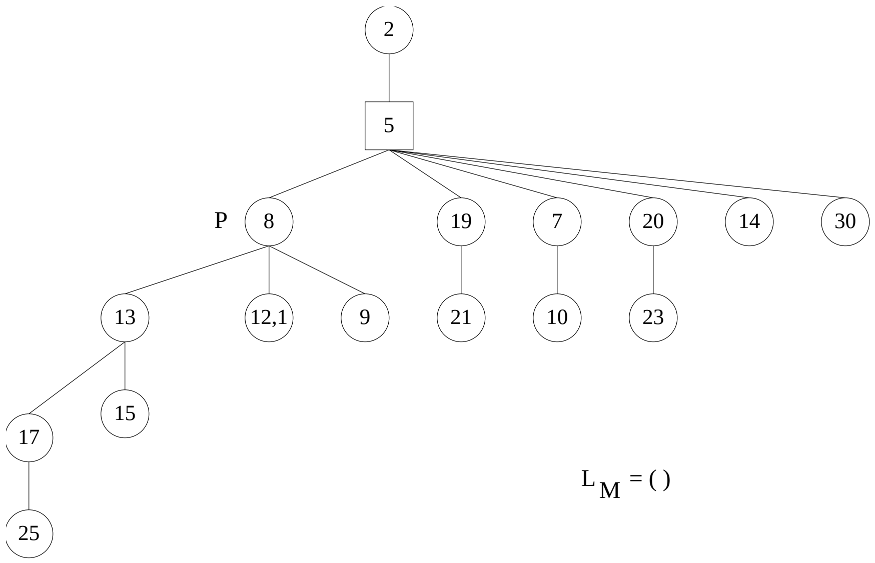


C Doing a delete min

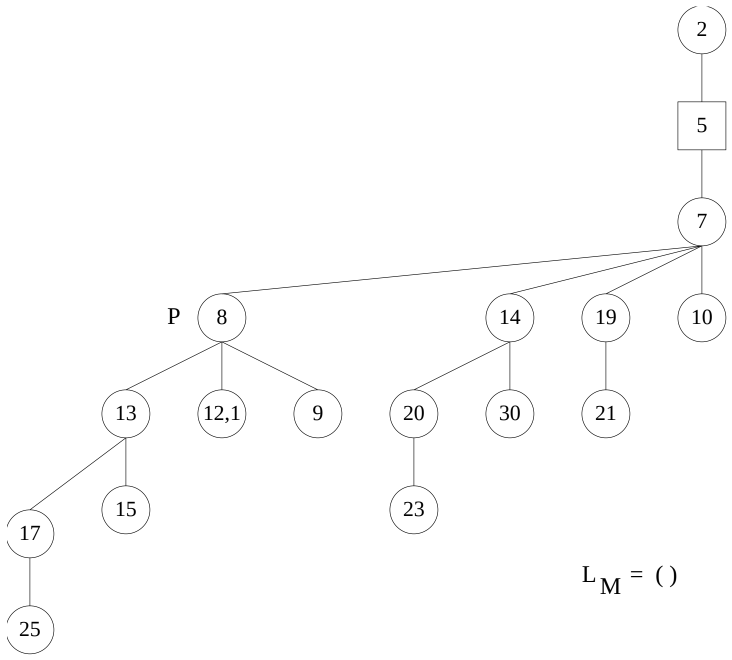
C.1 After Step 3c of delete min operation



C.2 After Step 5e of delete min operation



C.3 After the delete min operation



Errata

We later discovered, in August 1996, that the paper contains a bug. It turns out that the proof that out **delete min** operation takes $O(\log n)$ time in the worst case is incorrect.

The proof consisted of two parts. The first part part showed that all the algorithms presented in the paper preserve the data structure invariants described in Section 2.1. The second part of the proof showed that the data structure invariants imply that our **delete min** operation takes $O(\log n)$ time in the worst case. It turns out that Statement 4 in Lemma 1 is incorrect, which in turn invalidates the second part of the proof that the data structure invariants imply that our **delete min** operation takes $O(\log n)$ time in the worst case.

We can still prove that the the data structure invariants in Section 2.1 imply that our **delete min** operation takes $O(\log^2 n/\log\log n)$ time in the worst case. Thus, the corrected version of the theorem presented in the paper is as follows.

Theorem 1 *An implementation of priority queues on a RAM exists that supports the operations find minimum, insert, decrease key and meld, each in* $O(1)$ *worst case time and delete and delete min in* $O(\log^2 n/\log\log n)$ *worst case time.*

This would still have been a new result at the time this report was published. However, Gerth Stoling Brodal later published a paper [6] that describes a completely different data structure than ours that supports the **delete min** operation in $O(\log n)$ time in the worst case.

References

[6] Gerth Stoling Brodal. *"Worst-case efficient priority queues"*. Proc. **7th** Ann. ACM Symp. on Discrete Algorithms, 52-58 (1996)