Worst Case Efficient Data Structures for Priority Queues and Deques with Heap Order

A Project Report
Submitted in partial fulfilment of the requirements of the degree of

Bachelor of Technology
in
Computer Science and Engineering

by

Boyapati Chandra Sekhar

under the guidance of

Prof. C. Pandu Rangan

Department of Computer Science and Engineering
Indian Institute of Technology, Madras

May 1996

Certificate

This is to certify that the thesis entitled **Worst Case Efficient Data Structures for Priority Queues and Deques with Heap Order** is a bonafide record of the work done by **Boyapati Chandra Sekhar** in the Department of Computer Science and Engineering, Indian Institute of Technology, Madras, under my guidance and supervision, in the partial fulfilment of the requirements for the award of the degree of Bachelor of Technology in Computer Science and Engineering.

Place: Madras
Date:

C. Pandu Rangan
Professor
Department of Computer Science
and Engineering
I.I.T.Madras

Acknowledgements

I sincerely thank Prof. C. Pandu Rangan for introducing me to the field of data structures and algorithms and for inspiring me to excel in it. I am particularly indebted to him for being a constant source of my encouragement and for all the faith he expressed in my abilities. His enthusiastic help, friendly guidance and valuable advice have made him much more than a project guide.

Special thanks to Lallu for continuously motivating me to try harder, especially when new results were difficult to come by, and above all, for being a great friend. Thanks also to Ninja, Mixie, Pi and Preetham for maintaining a lively atmosphere in AI Lab and for giving me company through all those 'night-outs'.

I would also like to thank the rest of my classmates Dham, Belani, Kanta, Vishnu, ET, Pach, G2, Andy, Stash, Arsham, Nirma, Kudi, Kanna, SOS, Madhu, Sunku, Jayaram, Vsri, Suchitra and NK, whose friendship I shall always cherish. Thanks also my wing-mates Koshi, Nishat, JJ, Arnari, Zygote, Dud and Wasim for making my stay in IIT an enjoyable one. I would like to thank the AI Lab junta Kama, Ganesh, Janaki, Renuka and Bhoopal for their friendship.

I would also like thank all the faculty members of the Department of Computer Science and Engineering, IIT Madras, for their guidance and help throughout these four years.

Last but not the least, my family has always encouraged me to aim high and to pursue my dreams. I thank my parents and my brother Shyam for their continued love and support.

Abstract

An efficient *amortized* data structure is one that ensures that the average time per operation spent on processing any sequence of operations is small. Amortized data structures typically have very non-uniform response times, *i.e.*, individual operations can be occasionally and unpredictably slow, although the average time over a sequence is kept small by completing most of the other operations quickly. This makes amortized data structures unsuitable in many important contexts, such as real time systems, parallel programs, persistent data structures and interactive software. On the other hand, an efficient *worst case* data structure guarantees that every operation will be performed quickly.

The construction of worst case efficient data structures from amortized ones is a fundamental problem which is also of pragmatic interest. In this report, we have studied two different problems in data structures, namely, the implementation of priority queues and concatenable double ended queues with heap order. We have eliminated amortization from the existing data structures and have proposed new worst case efficient data structures for these problems.

Contents

List of Figures

Chapter 1

Introduction

1.1 Amortization: What, Why and Why Not?

In an amortized analysis, the time required to perform a sequence of data structure operations is averaged over all the operations performed. Amortized analysis can be used to show that the average cost of an operation is small, if one averages over a sequence of operations, even though a single operation might be expensive [19] [3].

1.1.1 An Example of Amortized Analysis

Consider the data type *counter*. A counter takes non-negative integer values. The two operations are *set to zero* and *increment*.

We realize counters as a sequence of binary digits. Then *set to zero* returns a string of zeros and *increment* has to add 1 to a number in binary representation. We implement *increment* by first incrementing the least significant digit by 1 and then calling a procedure *propagate carry* if the increased digit is 2. This procedure changes the digit 2 into 0, increases the digit to the left by 1, and calls itself recursively, if necessary.

Suppose we perform a sequence of one *set to zero* operation followed by n *increment* operations. The worst case cost of an increment operation is, of course, $\log n$. Hence, a naive analysis might lead to the conclusion that the total cost of the sequence of operations is $n \times O(\log n) = O(n \log n)$.

However, it turns out that the above is a very weak upper bound on the total cost. It can be easily shown that the total cost is bounded by $1 + 2n$. Thus,the average cost per operation is $O(1)$.

1.1.2 Applications Where Amortization is Inappropriate

Amortized efficiency is a weaker requirement for an algorithm to satisfy than worst case efficiency. Not surprisingly, there are problems whose amortized complexity is less than their worst case counterparts. However, in order for an algorithm to make full use of the leeway offered to it by the weaker constraint of amortized efficiency, it must exhibit large variations in response time, *i.e.*, occasionally, there must be some very expensive operations which are "paid for" by performing other operations quickly. Unfortunately, it is difficult to control when the expensive operations occur; they may choose to happen at extremely inconvenient moments. Therefore, there are many situations where amortized algorithms are inappropriate. We discuss a few of them below.

Real time systems In real time systems, the most valuable properties a system component can have are *predictability* and *speed*. Amortized algorithms are not predictable, unless we assume that every operation takes as long as any other operation in the sequence and allocate time accordingly. This can be an extremely wasteful assumption since there is normally a wide difference between the single operation and average complexity of an amortized data structure. For example, in the traditional mode of garbage collection in LISP, processing is temporarily halted in order to garbage collect. Since this is an infrequent operation, the cost of garbage collection is small if amortized over the operations performed since the last time the garbage collection routines were invoked. However, the occasional suspensions of processing could prove disastrous if employed in systems that need to react in real time. Wadler [7] gives us the compelling example of a tennis playing robot halting in mid-swing in order to garbage collect. Furthermore, it is clearly inefficient to assume that garbage will be collected after every operation.

Parallel programs Amortization also interacts poorly with parallelization. If several processors require shared access to an amortized data structure to perform a task, then it is possible that all the operations of one processor are performed slowly, causing this processor to finish well after others. Alternatively, if processors are performing operations in lock-step and need to synchronize frequently, they may have large amounts of idle time if some one processor executes operations that are occasionally and unpredictably expensive. For example, Fredman and Tarjan [11] invented an efficient amortized data structure called Fibonacci heaps and showed how it could be used to speed up Dijkstra's shortest path algorithm in the sequential setting. Driscoll *et al.* [10] observed that attempts to use Fibonacci heaps to get a similar improvement in efficiency for a parallel setting failed because of these reasons. Motivated by this, they developed the relaxed heaps.

Persistent data structures Worst case algorithms are also useful in making data structure persistent [5] [4]. Persistent data structures support queries and updates on their older versions, in a "branching" model of time. Amortized data structures perform poorly when made persistent, as an expensive operation can be repeatedly requested.

Interactive software For algorithms that are to be used in interactive situations, the occasional stoppage or slowdown is undesirable from the user's point of view. Modern interactive LISP systems use on-the-fly incremental garbage collection (which is the worst case algorithm from our viewpoint) rather than the simpler traditional method (an amortized algorithm) [7] [6]. In some cases such slow downs are completely unacceptable. For example, in flight simulators, it is vital that the view changes smoothly in response to the pilot's movements.

1.2 Eliminating Amortization: Worst Case Efficient Data Structures

In worst case analysis, one derives worst case time bounds for each single operation.

While the notion of amortization is a natural and reasonable measure of efficiency in some settings, it proves inadequate in others. Apart from these considerations, the question of where amortization can be eliminated without a significant degradation in efficiency is interesting in its own right, and indeed this problem has been extensively studied in the context of language recognition by automata of various kinds, though the literature is much less extensive in the areas of data structures and algorithms.

1.3 Overview of the Report

We have studied two different problems in data structures. We have eliminated amortization from the existing data structures and proposed new worst case efficient data structures for these problems.

In Chapter 2, we have proposed a new data structure called relaxed Fibonacci heaps for implementing priority queues with worst case performance bounds. The new data structure has primarily been designed by relaxing some of the constraints in Fibonacci heaps, hence the name relaxed Fibonacci heaps.

In Chapter 3, we first proposed a new amortized data structure for implementing concatenable deques with heap order. We then modified it to eliminate amortization from all the operations except the meld operation.

Chapter 2

Priority Queues

2.1 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. [8] [10] [11] [12] [19]

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 [8] 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 [10] 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 time. The problem that was posed in [8] was to consider if it is possible to support both **decrease key** and **meld** simultaneously in constant worst case time.

In this chapter, 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.

Please see Errata at the end of the thesis.

	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 2.3, 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 2.4.1, we describe the relaxed Fibonacci heaps. In Section 2.5, we describe the various operations on relaxed Fibonacci heaps.

2.2 Preliminaries: Operations Supported

The operations that should be supported by priority queues are as follows:

MakeQueue Creates an empty priority queue.

FindMin(Q**)** Returns the minimum element contained in Q.

Insert(Q,e**)** Inserts element e into priority queue Q.

Meld(Q_1, Q_2**)** Melds the priority queues Q_1 and Q_2 to form one priority queue and returns the new priority queue.

DeleteMin(Q**)** Deletes the minimum element of Q and returns the minimum element.

Delete(Q, e**)** Deletes element e from Q provided that it is known where e is stored in Q (priority queues do not support the searching for an element).

DecreaseKey(Q, e, v**)** Decrease the value of element e in Q to v provided that it is known where e is stored in Q.

2.3 Some Results Regarding Binomial Trees

Consider the following problem. We are given a binomial tree B 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 2.1 *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 2.2 *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 $2^{d_0} \geq n_0$. □

2.4 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 [12] and Fibonacci heaps [11] where the representation is a forest of heap ordered trees.

2.4.1 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, $R'.degree \leq R_{k-1}.degree + 1$.
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) $R.lost = R_0.lost = R_1.lost = ... = R_{k-1}.lost = 0$.

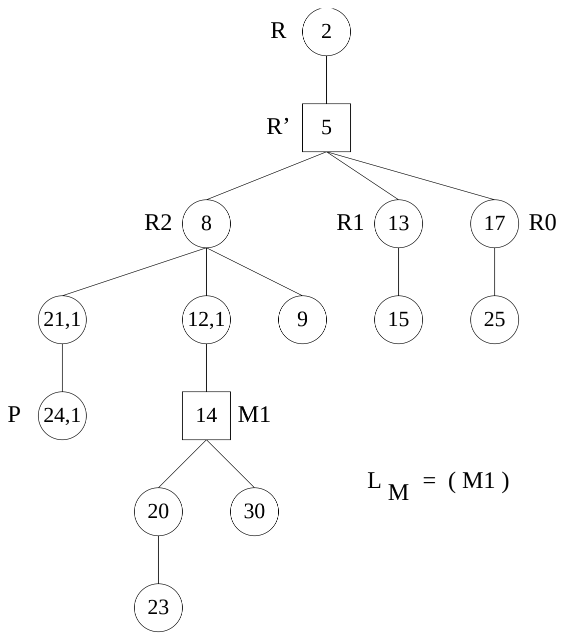

Figure 2.1: A relaxed Fibonacci heap

(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) $n_i + i \leq n$

Example: A Relaxed Fibonacci Heap

Figure 2.1 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 2.3 *The heap order implies that the minimum element is at the root.*

Lemma 2.4 *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 2.3, $N'_i.degree + N'_i.lost = i$. 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 2.2, this is more than n. Thus, T_N has more than n nodes, which is obviously not possible. □

Lemma 2.5 *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 $N \leq N.degree + 1 \leq 4d_0 + 4$, according to the previous lemma.

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

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, $M_i.degree \leq 4\lceil \log n_i \rceil + 4$. But according to property 4b, $n_i < n$. Thus the number of children of $N = M_i.degree \leq 4d_0 + 4$. □

Remarks

The restrictions imposed by property 3c are much weaker than those in fast meldable queues [8] or in relaxed heaps [10]. 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 chapter.

2.4.2 Representation

The representation is similar to that of Fibonacci heaps as described in [19]. 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 $N.lost > 1$, 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 $N.lost > 1$.

2.5 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.

2.5.1 Some Basic Operations

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

Algorithm 2.1 : 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 2.2 : 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 2.3 : 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 2.4 : 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 2.6 *All the operations described above take constant time in the representation described in Section 2.4.2.*

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. $\square$

Lemma 2.7 *Property 2b described in Section 2.4.1 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. $\square$

Lemma 2.8 *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. $\square$

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 2.9 *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, $W^+ = W^- - W_M + w_{P^+} \leq (n + w_{P_1} + w_{P_2}) - W_M + w_{P^+} = n + w_{P^+} - (W_M - w_{P_1} - w_{P_2}) \leq n + w_{P^+}$ $\square$

Lemma 2.10 *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, $W^+ = W^- - W_M + w_{P^+} \leq (n + 1 + w_{P^-}) - W_M + w_{P^+} = n + w_{P^+} - (W_M - w_{P^-} - 1) \leq n + w_{P^+}$. $\square$

2.5.2 The Find Min Operation

Algorithm 2.5 : FindMin(Q)
/* Return the minimum element in the relaxed Fibonacci heap Q */

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

Lemma 2.11 *The find min operation described above returns the minimum element in Q in constant time.*

2.5.3 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 2.6 : 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 2.12 *All the properties described in Section 2.4.1 are maintained under the insert operation.*

Proof: Since **insert** works by calling **add**, property 2b is preserved under the **insert** operation, according to Lemma 2.7. The rest of the properties are trivially preserved under **insert**. □

Lemma 2.13 *The insert operation takes constant time in the representation described in Section 2.4.2.*

Proof: Follows from Lemma 2.6. □

2.5.4 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 2.7 : 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. $Adjust(Q, P, P')$
```

Example: Decreasing the Value of the Element 24 to 19 in the Tree in Figure 2.1

This is shown in figures 2.2 and 2.3.

Lemma 2.14 *All the properties described in Section 2.4.1 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 2.7. Also, since **decrease key** works by calling **reinsert**, property 3b is preserved according to Lemma 2.8.

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 2.9, 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. □

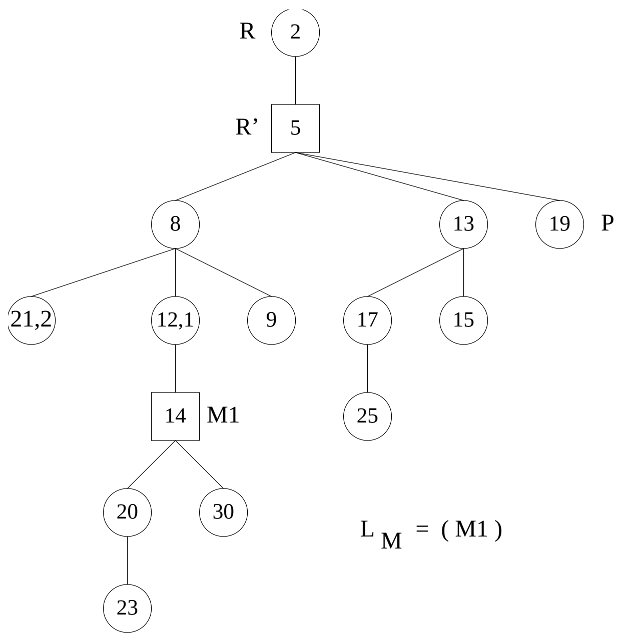

Figure 2.2: After Step 5 of decrease key operation

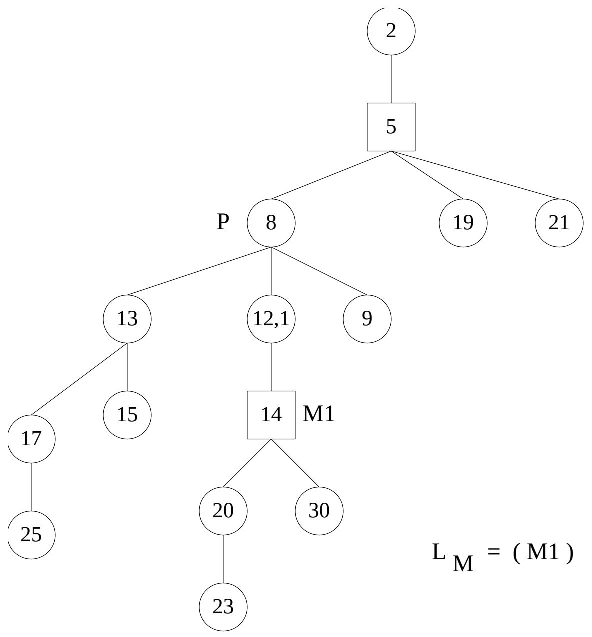

Figure 2.3: After the decrease key operation

Lemma 2.15 *The decrease key operation takes constant time in the representation described in Section 2.4.2.*

Proof: Follows from Lemma 2.6. □

2.5.5 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 2.8 : 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 $Q_1$
```

Example: After Melding with a Priority Queue containing only 1, 7 and 10

This is shown in figure 2.4.

Lemma 2.16 *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 $m_1 + 2 \leq i \leq m_1 + m_2 + 1$.

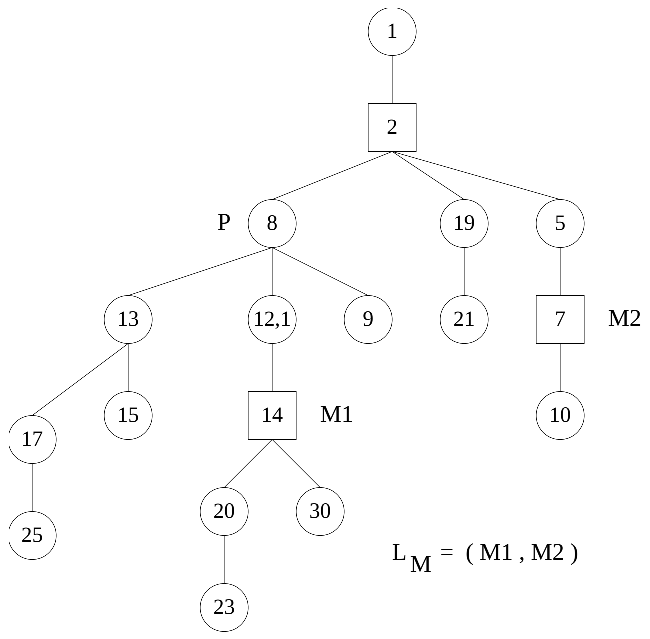

Figure 2.4: After the meld operation

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 2.17 *All the properties described in Section 2.4.1 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 2.7.

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 2.9, 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 2.18 *The meld operation takes constant time in the representation described in Section 2.4.2.*

Proof: Follows from Lemma 2.6. □

2.5.6 The Delete Min Operation

Algorithm 2.9 : 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
```

Example: Doing a Delete Min

This is shown in figures 2.5, 2.6 and 2.7.

Lemma 2.19 *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. □

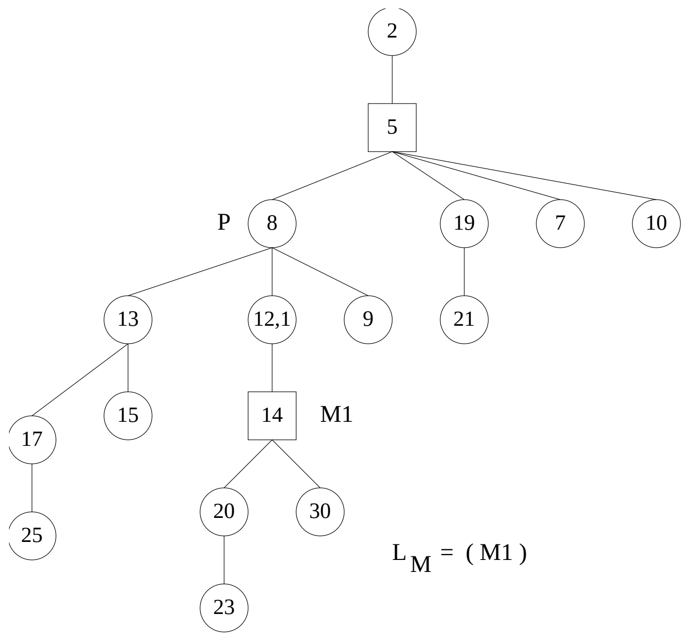

Figure 2.5: After Step 3c of delete min operation

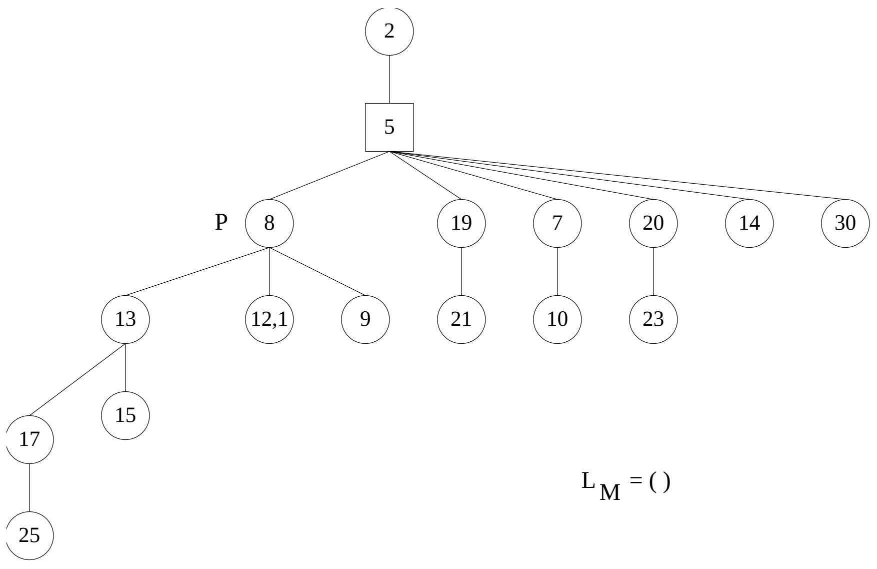

Figure 2.6: After Step 5e of delete min operation

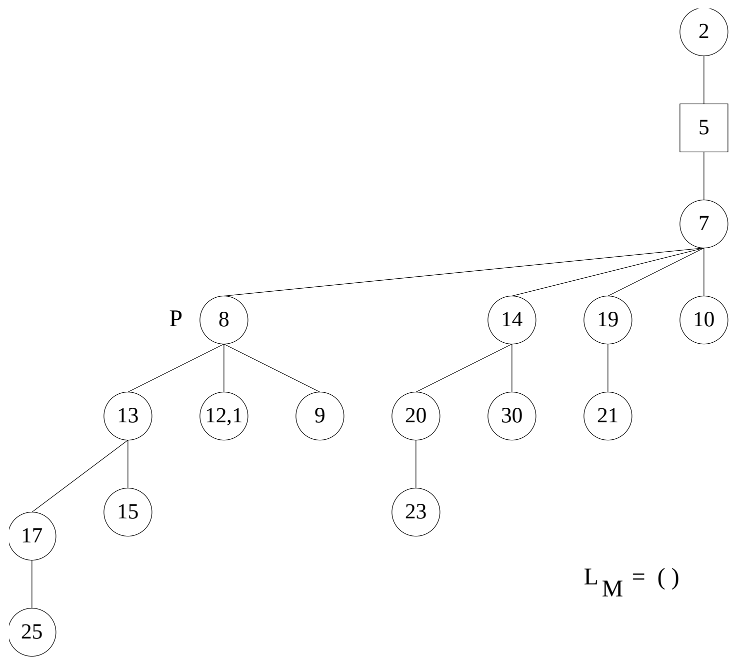

Figure 2.7: After the delete min operation

Lemma 2.20 *All the properties described in Section 2.4.1 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 2.9, 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. $\square$

Lemma 2.21 *It is easy to see that in the representation described in Section 2.4.2, the delete min operation takes $O(R'.degree + R''.degree + M'.degree + M.degree) = O(\log n)$ time, according to lemma 2.5.*

2.5.7 The Delete Operation

Algorithm 2.10 : 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 2.22 *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.*

2.6 Conclusion

We can summarize the discussion in this chapter by the following theorem.

Theorem 2.1 *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.

Chapter 3

Deques with Heap Order

3.1 Introduction

A *deque with heap order* is a deque (double-ended queue) such that each item has a real-valued key and the operation of finding an item of minimum key is allowed as well as the usual deque operations. Queues with heap order have applications in paging and river routing [15] [17] [16].

Gajeuska and Tarjan [15] have described a data structure that supports in $O(1)$ worst case time the usual deque operations *push, pop, inject, eject* and the *find minimum* operation. They also mentioned that Kosaraju's methods [18] extend to support $O(1)$ time concatenation of deques **when the number of deques is fixed.** However, for the case of a variable number of deques, they left the same as an open problem. In [14], Buchsbaum, Rajamani and Tarjan partially answered this question by describing a data structure that supports all the operations mentioned above in $O(1)$ amortized time using the technique of bootstrapping and path compression.

In this chapter we first present a simple data structure that also supports all the above mentioned operations in $O(1)$ amortized time even when the number of deques is arbitrarily large, without using the technique of bootstrapping or path compression. Also, this data structure is flexible enough to support operations like given the actual position of a key, *insert a key next to it, or delete it, or change its value* in $O(\log n)$ worst case time, while the data structure in [15] or [14] takes $O(n)$ worst case time for each of these operations. Finally, we present an improved version of our data structure that supports *push, pop, inject* and *eject* in $O(1)$ worst case time without changing the complexity of the other operations.

3.2 The Amortized $O(1)$ Time Data Structure

3.2.1 Structural Properties

Our data structure shares the following structural properties of 2-4 trees:

1. Every internal node has 2 or 3 or 4 sons. We will say that a node satisfies the *degree property* if it satisfies this condition.
2. All the leaves are at the same level.

3.2.2 Organization of the Leaves

Every data item is present in some leaf and every leaf contains one data item. The tree is so arranged that if we go through all the leaves of the tree from left to right, we get the order in which the data items are present in the deque, with the item in the left-most leaf corresponding to the head of the sequence and the item in the right-most leaf corresponding to the tail.

3.2.3 Organization of the Internal Nodes

3.2.3.1 Classification of the Internal Nodes

For convenience of discussion, we will first classify the internal nodes of our tree into three categories:

1. The *root.* (R in figure 3.1)
2. The *exterior nodes,* which are the internal nodes in the right-most and left-most paths other than the root. (N_1, N_6, N_7 and N_9 in figure 3.1)
3. The *interior nodes,* which are the rest of the internal nodes. (N_2, N_3, N_4, N_5 and N_8 in figure 3.1)

3.2.3.2 Graphical Representation

Consider the graph representation of this tree. We will define a direction[1] for every edge of this graph. We will say that a node M *points* to a node N if there is a directed edge from M to N. We define that —

[1]These directions are just for our conceptual understanding and have nothing to do with the way the nodes are hooked up in the tree.

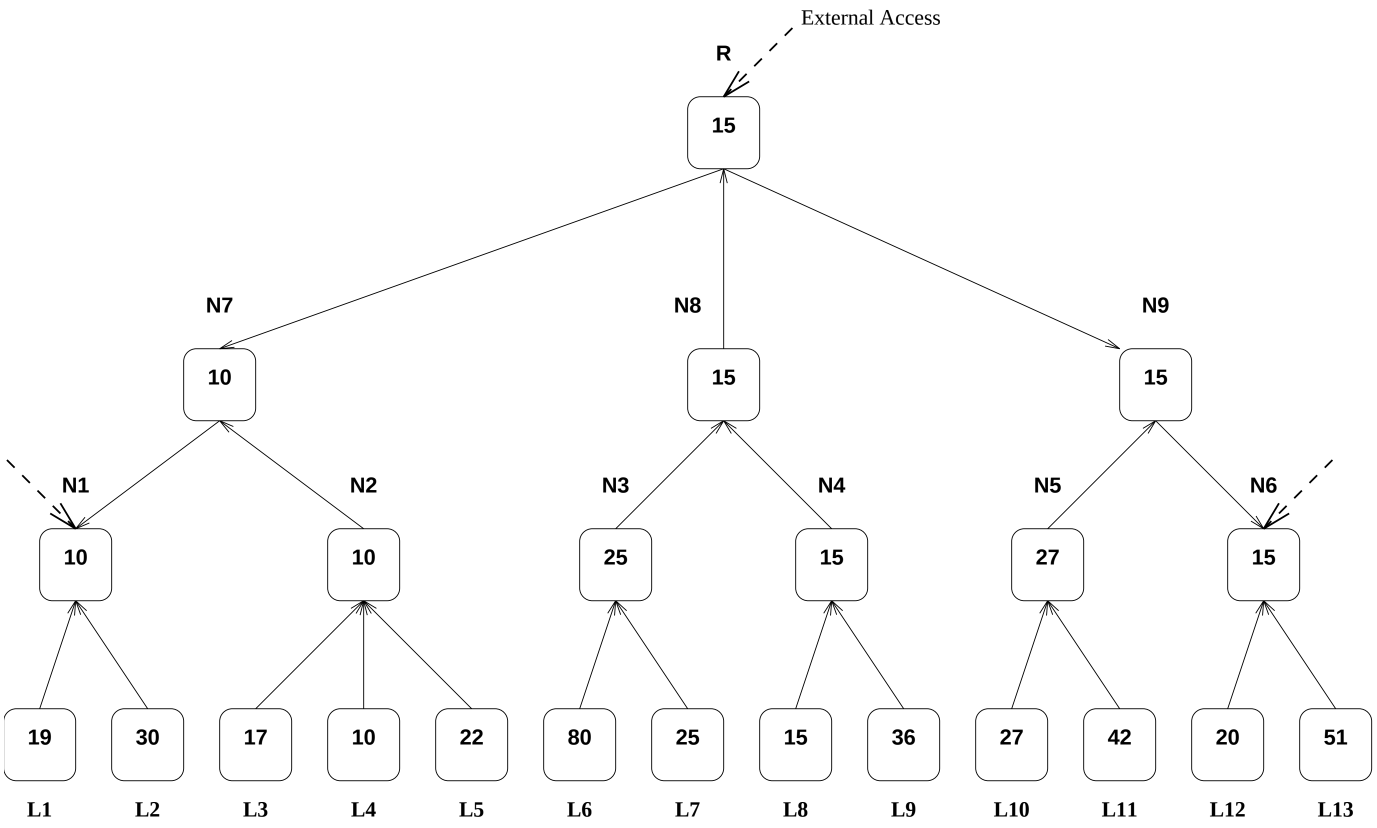

Figure 3.1: A deque with heap order

1. Every leaf *points* to its parent.
2. Every interior node *points* to its parent.
3. The root *points* to its left-most and right-most sons, if they are not leaves.
4. Every exterior node, except the left-most one and the right-most one, *points* to its exterior son.

For example, these directions are explicitly shown in figure 3.1. The graph thus defined above is obviously a directed acyclic graph.

3.2.3.3 Contents of the Internal Nodes

For an internal node N, let S_N be the set of nodes that *point* to N. In every internal node N of the tree, we will store (a pointer to) an item as follows:

$$\begin{aligned} item(N) &= \min\{item(M) \mid M \in S_N\},\ if\ S_N\ is\ not\ empty, \\ &= \textbf{infinity},\ if\ S_N\ is\ empty. \end{aligned}$$

Observe that S_N can be empty only when N is the root and has only two exterior sons. We will say that an internal node satisfies the *content property* if it satisfies the above equation.

3.2.4 External Access

Our tree itself is accessible from outside *via* three pointers, one for the *root*, and one each for the *left-most and right-most exterior nodes.*
For example, the tree in figure 3.1 is accessed through pointers to N_1, N_6 and R.

3.3 The Algorithms and their Analysis

We shall present algorithms for the non-degenerate cases only, that is, when the tree has more then three internal nodes. The degenerate cases can be handled individually, since the number of such cases is finite.

3.3.1 The Potential Function

Corresponding to each configuration of our data structure, we define a potential[2] Φ as follows:

$$\begin{aligned} \Phi &= \Theta(\#\ \textit{nodes with } 2 \textit{ sons} + 2 \times \#\ \textit{nodes with } 4 \textit{ sons}) \\ &+ \Theta(\textit{height of the tree}) \end{aligned}$$

The amortized cost of an operation is defined as the sum of *its actual cost* and *the increase in potential of the data structure due to the operation.*
Since the potential is initially zero and is always positive, the total amortized cost after n operations is an upper bound on the total actual cost.

3.3.2 Find Minimum

Since the only nodes in the graph with out-degree zero are the left-most and the right-most exterior nodes, at least one of them has to contain the minimum item. Hence, it takes at most one comparison to find the minimum. Thus the time complexity of this operation is $O(1)$, in the worst case.

3.3.3 Push, Pop, Inject, Eject

From the left-right symmetry in the properties of our tree, we know that both push and inject are identical operations, and so are pop and eject. Hence, we will discuss the operations of insertion and deletion from the right end of the deque only.

For convenience of discussion, we will be using the following function:

Algorithm 3.1 : Update N, where N is an internal node

1. Update the item stored in N such that it satisfies the *content property*, as discussed in Section 3.2.3.3.

The structural adjustments performed on our tree are exactly identical to those performed on a 2-4 tree following every insertion or deletion. However, with every node split, or node merge, or borrowing of a child by a node from a brother, we will also do a constant time updating of internal nodes as discussed below:

1. When an exterior node N is split into an interior node N_1 and an exterior node N_2, then update N_1, P and N_2, in that order, where P is the parent of N_1 and N_2.

[2]Note that our potential function is only a slight modification of the potential function of the 2-4 trees, to accommodate the analysis of the *concatenate* operation.

2. When an exterior node is merged with its brother to form a node N, then update P and N, in that order, where P is the parent of N.
3. When an exterior node N_2 borrows a son from a brother N_1, then update N_1 and P, in that order, where P is the parent of N_1 and N_2.

It is easy to see the correctness of the following lemma from the above discussion.

Lemma 3.1 *If every internal node of the tree satisfies the* content property *before an internal node N splits or merges or borrows a son, then we can restore the* content property *of every internal node by doing a constant time updating of internal nodes (as discussed above) after the split or merge or borrowing, provided that neither N nor its parent is the root.*

We will now present the algorithm for insertion and deletion from the right end.

Algorithm 3.2 : Insert (or delete) an item at the right end

1. Insert the data item as the right-most leaf L of the tree.
   (Or, delete the right-most leaf L of the tree.)
2. Perform structural adjustments on the tree just as is done in 2-4 trees, accompanied with the updating of internal nodes as discussed above.
3. If no structural adjustments were required in the above step, then let N be the right-most exterior node. Else let N be the parent of the last node that was split, in case of insertion. (In case of deletion, let N be the parent of the last node that was merged, or parent of the last node that borrowed a son. If the last merged node happens to be the root, then let N be the root itself).
   If N is the root then
   (a) Update all the internal nodes in the left-most path, starting from the root till the left-most exterior node.
   (b) Update all the internal nodes in the right-most path, starting from the root till the right-most exterior node.

3.3.3.1 Time Complexity Analysis

Let there be k node splits or node merges in Step 2. Since the first step takes $O(1)$ time and the rest of the steps take $O(k)$ time each, the actual cost of the operation is $O(k)$.

But, the corresponding decrease in potential is $\Theta(k)$, just as in case of 2-4 trees. Thus, choosing appropriate values for the constants in the potential function, the amortized costs of each of these operations turn out to be $O(1)$.

However, the worst case cost is $O(\log n)$.

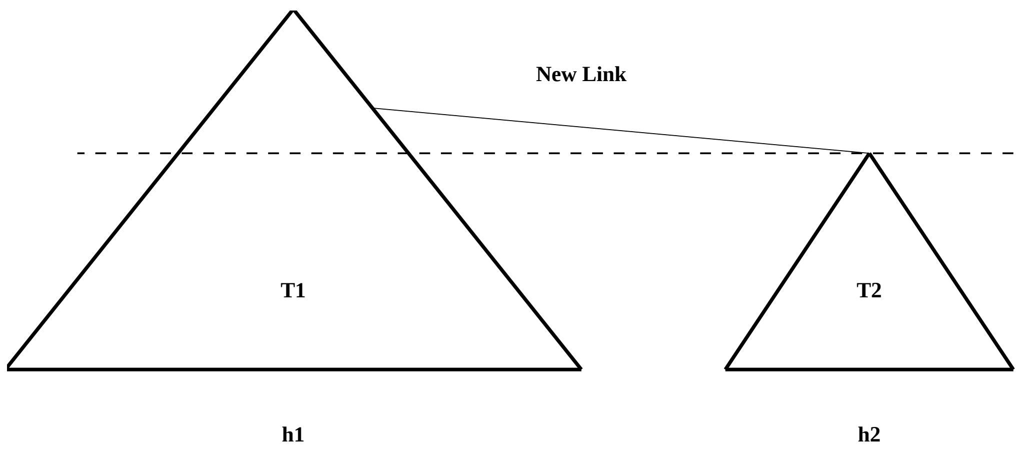

Figure 3.2: Concatenating when $h_1 \neq h_2$

3.3.4 Concatenate Two Deques

Let us assume, we have to concatenate two deques D_1 and D_2, with the head of D_2 following the tail of D_1 in the final deque. Let T_1 and T_2 be the corresponding trees of heights h_1 and h_2 and with roots R_1 and R_2 respectively, where T_1 is to be attached to the left of T_2. There might be two possibilities:

3.3.4.1 Case 1 : $h_1 \neq h_2$

Without loss of generality, say $h_1 > h_2$. In this case, concatenation can also be treated as an insertion, not at the leaf level, but at a higher level such that the leaves in the final tree are all at the same level. However, since the nodes in the left-most path of T_2 and some of the nodes in the right-most path of T_1 are converted from exterior nodes to interior nodes, certain nodes in the new tree must be updated. Also, the tree has to be adjusted, to satisfy the structural properties.

Algorithm 3.3 : Concatenate T_2 to the right of T_1, where $h_1 > h_2$

1. Insert the root R_2 of T_2 as the son of a node M in the right-most path of T_1 such that the leaves in the final tree are all of the same level.
2. Update all the internal nodes in the left-most path of T_2, starting from the left-most exterior node till the root.
3. Update all the internal nodes in the right-most path of T_1 starting from the right-most exterior node till the node M.
4. If M is the root of T_1 then update all the internal nodes in the left-most path of T_1, starting from the root till the left-most exterior node.

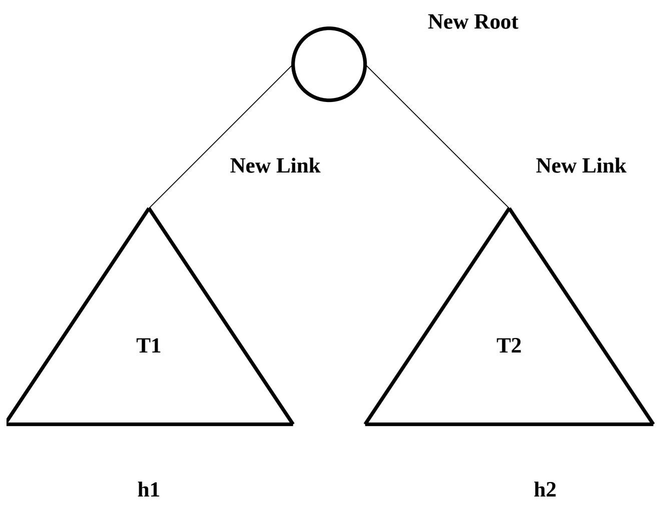

Figure 3.3: Concatenating when $h_1 = h_2$

5. If M violates the degree property, then execute Algorithm 3.2 from Step 2 onwards to structurally adjust the new tree.
6. Update all the internal nodes in the right-most path of the new tree starting from the node R_2 till the right-most exterior node.

3.3.4.2 Case 2: $h_1 = h_2$

In this case, we create a new root R with a dummy item such that $value(R) = \infty$, and make the roots of T_1 and T_2 sons of R. As in the previous case, we also have to update the right-most and left-most paths of T_1 and T_2.

Algorithm 3.4 : Concatenate T_2 to the right of T_1, where $h_1 = h_2$

1. Create a new root R, with $value(R) = \infty$.
2. Make the roots R_1 and R_2 of T_1 and T_2 sons of R.
3. Update all the internal nodes in the right-most path of T_1, starting from the right-most exterior node till the root R.
4. Update all the internal nodes in the left-most path of T_2, starting from the left-most exterior node till the root R.
5. Update all the internal nodes in the left-most path of T_1, starting from the root R till the left-most exterior node.

6. Update all the internal nodes in the right-most path of T_2, starting from the root R till the right-most exterior node.

3.3.4.3 Time Complexity Analysis

Let k be the number of node splits, if any, for structurally adjusting the tree. Then, the actual cost of the operation is obviously $O(h_2) + O(k)$.

But, the corresponding decrease in potential is $\Theta(h_2) + \Theta(k)$, since in place of two trees of heights h_1 and h_2, we finally have only one tree of height h_1 (or $h_1 + 1$). Thus the amortized cost of *concatenate* operation turns out to be $O(1)$.

However, the worst case cost is $O(\log n)$.

Theorem 3.1 *Deques with heap order can be implemented to support each of the operations push, pop, inject, eject and concatenate two deques in $O(1)$ amortized case time and the operation find minimum in $O(1)$ worst case time, even when the number of deques is arbitrarily large.*

3.4 Improved Data Structure to support Push, Pop, Inject and Eject in $O(1)$ Worst Case Time

Since the operations *push, pop, inject,* and *eject* cause insertions or deletions only at the right or left end of the tree, we can improve the efficiency of the data structure by performing the necessary structural adjustments and updating of internal nodes lazily over the next few operations.

To effect this, we will allow the exterior nodes to temporarily have at least one and at most five sons. We will also allow one internal node each in the right-most and left-most paths to violate the *content property.*

Algorithm 3.5 : Insert (or delete) an item at the right (or left) end

1. Do the insertion (or deletion) at the right (or left) end of the tree.
2. Structurally adjust the lowest node N in the right-most (or left-most) path that violates the *degree property.*
3. // N_L and N_R are the two nodes that may violate the content property.
   // Initially, N_L and N_R are respectively the left-most and right-most leaves.
   If the parent of node N is the root then
   (a) Structurally adjust the other exterior son of the root, if it is violates the *degree property.*

   (b) Structurally adjust the root, if it is violates the *degree property.*
   (c) Set $N_L \leftarrow$ left son of root and $N_R \leftarrow$ right son of root.

4. If N_L and N_R are not leaves then
   (a) Update N_L and N_R
   (b) Set $N_L \leftarrow$ left son of N_L and $N_R \leftarrow$ right son of N_R.

As an illustration of the above algorithm, consider a tree in which the right-most exterior node N_1 and some of its immediate ancestors have four sons each. Let N_{i+1} represent the parent of N_i. Let there be a series of insertions from the right end into this tree. The following table shows the number of sons of some of the right exterior nodes of the tree after each insertion. Note how we do not immediately split all the ancestors but instead 'lazily' split only the lowest node with every insertion.

	N_1	N_2	N_3	N_4	N_5
Initial	4	4	4	4	4
Insert	5	4	4	4	4
Split	3	5	4	4	4
Insert	4	5	4	4	4
Split	4	3	5	4	4
Insert	5	3	5	4	4
Split	3	4	5	4	4
Insert	4	4	5	4	4
Split	4	4	3	5	4

We will demonstrate the correctness of the above algorithm through the following lemmas:

Lemma 3.2 *Every exterior node will have at least 1 son and at most 5 sons after every operation.*

To prove this, we can think of the above algorithm as follows. After every insertion or deletion at the right end, the right-most exterior node gets a *chance* to adjust itself. If required, it uses the *chance* to adjust itself. Else it passes the *chance* to its parent, which in turn either uses the *chance* or passes it upwards, and so on. To prove the lemma, it suffices to prove that *every node in the right-most path and the left-most path gets at least one* chance *to adjust itself between two consecutive insertion or deletion operations on it.*

Let the above statement be true for an exterior node N_h at a height h in the tree. We will then prove the statement for its parent N_{h+1}.

Observe that an insertion into N_{h+1} corresponds to splitting of N_h and a deletion from N_{h+1} corresponds to merging of N_h with its brother. But after every node split or node merge, N_h will have exactly 3 sons. Hence, before the next node split or node merge of N_h, there have to be at least two insertions or deletions on N_h. Between these two insertions or deletions, N_h will get a *chance* to adjust itself which it does not require since it will have at least 2 sons and at most 4 sons. Hence, it will pass this *chance* to N_{h+1} which can thus adjust itself between every two operations on it. And since the above statement the trivially true for the right-most and left-most exterior nodes, it follows that it is true for all exterior nodes.

Lemma 3.3 *The lowest exterior node that violates the* degree property *in the right-most or left-most path can be located in constant time by maintaining for each of the two exterior paths, a stack of all the exterior nodes in that path that violate the* degree property*, with the lowest such node at the top of the stack. This ensures that the above algorithm takes $O(1)$ worst case time.*

Lemma 3.4 *Once the parent of node N in the above algorithm becomes the root, it takes only h operations for N_L and N_R to become leaves while it takes $O(2^h)$ operations for the root to be disturbed again, where h is the height of the tree. This ensures that at any time there are at most two internal nodes (namely N_L and N_R) that violate the* content property*, and hence, find minimum can still be implemented in $O(1)$ worst case time.*

Lemma 3.5 *Concatenate operation can still be implemented in $O(1)$ amortized time with the following algorithm.*

Algorithm 3.6 : Concatenate two deques T_1 and T_2

1. Structurally adjust both the trees so that all the internal nodes satisfy the *degree property.* Also, update the exterior nodes in both the trees, starting from N_R or N_L to right-most or left-most exterior nodes.
2. Execute the algorithm discussed in Section 3.3.4.

Since total work done in m operations by the algorithms for the improved data structure is no more than the total work done by the algorithms for the amortized data structure, the same amortized $O(1)$ bound holds here also.

Theorem 3.2 *Deques with heap order can be implemented to support each of the operations push, pop, inject, eject and find minimum in $O(1)$ worst case time and the operation concatenate two deques in $O(1)$ amortized time, even when the number of deques is arbitrarily large. Moreover the operations like given the actual position of a key,* insert a key next to it, or delete it, or change its value *can also be implemented in $O(\log n)$ worst case time with the same data structure.*

3.5 Conclusion

We have presented a simple data structure that supports the operations *find minimum, push, pop, inject* and *eject* in $O(1)$ worst case time and the *meld* operation in $O(1)$ amortized time. Also, this data structure is flexible enough to support operations like given the actual position of a key, *insert a key next to it, or delete it, or change its value* in $O(\log n)$ worst case time.

However, we later found out that Kosaraju [13] has also proposed a similar data structure with identical complexity bounds. He used the red-black trees to develop his data structure while we used the 2-4 trees. He even went on to further modify the data structure to achieve an $O(1)$ worst case time complexity for the **meld** operation also, thus solving the problem completely.

Chapter 4

Conclusions and Directions for Future Research

4.1 Summary of the Results

In this report, we have considered the problem of modifying several amortized data structures to achieve worst case time bounds.

In Chapter 2, we have presented relaxed Fibonacci heaps as a new data structure 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).

In Chapter 3, we have presented a simple data structure that supports the operations *find minimum, push, pop, inject* and *eject* in $O(1)$ worst case time and the *meld* operation in $O(1)$ amortized time. Also, this data structure is flexible enough to support operations like given the actual position of a key, *insert a key next to it, or delete it, or change its value* in $O(\log n)$ worst case time.

4.2 Open Problems

4.2.1 The General Problem of Eliminating Amortization

In [3], Rajeev Raman has described a unified framework for obtaining new worst case algorithms from existing amortized algorithms. He described two player combinatorial games so that winning strategies translate fairly readily into worst case data structures for some problems.

It would be very interesting to see how useful his techniques prove to be in eliminating amortization from data structures other than the ones he considered.

It would also be quite challenging to develop other unified approaches to solve the problem of eliminating amortization.

4.2.2 Use of Randomization for Eliminating Amortization

Though we do not have a concrete example yet, we conjecture that randomization may prove to be a very useful tool in eliminating amortization. It would be interesting to demonstrate the power of randomization in eliminating amortization in convincing terms.

Bibliography

[1] Chandrasekhar Boyapati and C. Pandu Rangan. *"Relaxed Fibonacci heaps: An alternative to Fibonacci heaps with worst case rather than amortized time bounds"*. IIT Madras Technical Report TR-TCS-95-07 (1995)

[2] Chandrasekhar Boyapati and C. Pandu Rangan. *"On $O(1)$ concatenation of deques with heap order"*. IIT Madras Technical Report TR-TCS-95-05 (1995)

[3] Rajeev Raman. *"Eliminating amortization: On data structures with guaranteed response time"*. PhD thesis, University of Rochester (1992)

[4] J. R. Driscoll, N. Sarnak, D. D. Sleator and R. E. Tarjan. *"Making data structures persistent"*. Journal of Computer and System Sciences **38** 86-124 (1989)

[5] N. Sarnak. *"Persistent data structures"*. PhD thesis, New York University (1986)

[6] H. Baker. *"List processing in real time on a serial computer"*. CACM **21** 280-294 (1978)

[7] P. Wadler. *"Analysis of an algorithm for real-time garbage collection"*. CACM **19** 491-500 (1976)

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

[9] Gerth Stoling Brodal. *"Priority Queues on Parallel Machines"*. To appear in SWAT (1996)

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

[11] 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)

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

[13] S. R. Kosaraju. *"An optimal RAM implementation of concatenable min double-ended queues"*. Proc. **5th** Ann. ACM Symp. on Discrete Algorithms, 195-203 (1994)

[14] Adam L. Buchsbaum, Rajamani Sundar and Robert E. Tarjan. *"Data structural bootstrapping, linear path compression and concatenable heap ordered double ended queues"*. Proc. **33rd** Ann. Symp. on Foundations of Computer Science, 40-49 (1992)

[15] Hanai Gajeuska and Robert E. Tarjan. *"Deque with heap order"*. IPL **22**, 197-200 (1986)

[16] G. Diehr and B. Faaland. *"Optimal pagination of B-trees with variable length items"*. Comm. ACM **27**, 241-247 (1984)

[17] R. Cole and A. Siegel. *"River routing every which way, but loose"*. Proc. **25th** Ann. IEEE Symp. on Foundations of Computer Science, 65-73 (1984)

[18] S. R. Kosaraju. *"Real-time simulation of concatenatable double-ended queues by double-ended queues"*. Proc. **11th** Ann. ACM Symp. on Theory of Computing, 346-351 (1979)

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

Errata

We later discovered, in August 1996, that our result on priority queues 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 Chapter 2 preserve the data structure invariants described in Section 2.4.1. 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 2.1 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 2.4.1 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 Chapter 2 is as follows.

Theorem 2.1 *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 our technical report on this was published. However, Gerth Stoling Brodal later published a paper [20] that describes a completely different data structure than ours that supports the **delete min** operation in $O(\log n)$ time in the worst case.

Bibliography

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