Basic Research in Computer Science

BRICS DS-97-1 G. S. Brodal: Worst Case Efficient Data Structures

Worst Case Efficient Data Structures

Gerth Stølting Brodal

BRICS Dissertation Series **DS-97-1**

ISSN 1396-7002 **January 1997**

See back inner page for a list of recent BRICS Dissertation Series publications. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `DS/97/1/`

Worst Case Efficient Data Structures

Gerth Stølting Brodal

Ph.D. Dissertation

Department of Computer Science
University of Aarhus
Denmark

Worst Case Efficient Data Structures

A Dissertation
Presented to the Faculty of Science
of the University of Aarhus
in Partial Fulfillment of the Requirements for the
Ph.D. Degree

by
Gerth Stølting Brodal
January 31, 1997

Abstract

We study the design of efficient data structures. In particular we focus on the design of data structures where each operation has a worst case efficient implementations. The concrete problems we consider are *partial persistence*, implementation of *priority queues*, and implementation of *dictionaries*.

The first problem we consider is how to make bounded in-degree and out-degree data structures partially persistent, *i.e.*, how to remember old versions of a data structure for later access. A *node copying* technique of Driscoll *et al.* supports update steps in amortized constant time and access steps in worst case constant time. The worst case time for an update step can be linear in the size of the structure. We show how to extend the technique of Driscoll *et al.* such that update steps can be performed in worst case constant time on the pointer machine model.

We present two new comparison based priority queue implementations, with the following properties. The first implementation supports the operations FindMin, Insert and Meld in worst case constant time and Delete and DeleteMin in worst case time $O(\log n)$. The priority queues can be implemented on the pointer machine and require linear space. The second implementation achieves the same worst case performance, but furthermore supports DecreaseKey in worst case constant time. The space requirement is again linear, but the implementation requires auxiliary arrays of size $O(\log n)$. Our bounds match the best known amortized bounds (achieved by respectively binomial queues and Fibonacci heaps). The data structures presented are the first achieving these worst case bounds, in particular supporting Meld in worst case constant time. We show that these time bounds are optimal for all implementations supporting Meld in worst case time $o(n)$. We also present a tradeoff between the update time and the query time of comparison based priority queue implementations. Finally we show that any randomized implementation with expected amortized cost t comparisons per Insert and Delete operation has expected cost at least $n/2^{O(t)}$ comparisons for FindMin.

Next we consider how to implement priority queues on parallel (comparison based) models. We present time and work optimal priority queues for the CREW PRAM, supporting FindMin, Insert, Meld, DeleteMin, Delete and DecreaseKey in constant time with $O(\log n)$ processors. Our implementation is the first supporting all of the listed operations in constant time. To be able to speed up Dijkstra's algorithm for the single-source shortest path problem we present a different parallel priority data structure. With this specialized data structure we give a parallel implementation of Dijkstra's algorithm which runs in $O(n)$ time and performs $O(m \log n)$ work on a CREW PRAM. This represents a logarithmic factor improvement for the running time compared with previous approaches.

We also consider priority queues on a RAM model which is stronger than the comparison model. The specific problem is the maintenance of a set of n integers in the range $0..2^w - 1$ under the operations Insert, Delete, FindMin, FindMax and Pred (predecessor query) on a unit cost RAM with word size w bits. The RAM operations used are addition, left and right bit shifts,

and bit-wise boolean operations. For any function $f(n)$ satisfying $\log\log n \leq f(n) \leq \sqrt{\log n}$, we present a data structure supporting FINDMIN and FINDMAX in worst case constant time, INSERT and DELETE in worst case $O(f(n))$ time, and PRED in worst case $O((\log n)/f(n))$ time. This represents the first priority queue implementation for a RAM which supports INSERT, DELETE and FINDMIN in worst case time $O(\log\log n)$ — previous bounds were only amortized. The data structure is also the first dictionary implementation for a RAM which supports PRED in worst case $O(\log n/\log\log n)$ time while having worst case $O(\log\log n)$ update time. Previous sublogarithmic dictionary implementations do not provide for updates that are significantly faster than queries. The best solutions known support both updates and queries in worst case time $O(\sqrt{\log n})$.

The last problem consider is the following dictionary problem over binary strings. Given a set of n binary strings of length m each, we want to answer d–queries, *i.e.*, given a binary query string of length m to report if there exists a string in the set within Hamming distance d of the query string. We present a data structure of size $O(nm)$ supporting 1–queries in time $O(m)$ and the reporting of all strings within Hamming distance 1 of the query string in time $O(m)$. The data structure can be constructed in time $O(nm)$. The implementation presented is the first achieving these optimal time bounds for the preprocessing of the dictionary and for 1–queries. The data structure can be extended to support the insertion of new strings in amortized time $O(m)$.

Acknowledgments

The four years which have past since I started my Ph.D. studies have been an exciting time where I have had the opportunity to meet a lot of wise people and to travel to various interesting places. One quite curious experience was that to meet Rolf Fagerberg from Odense, I had to travel to places like Kingston (Canada), Dagstuhl (Germany) and Reykjavik (Iceland).

I am deeply grateful to my advisor Erik Meineche Schmidt for his support throughout the years and for letting me pursue my own research interests, and who gave me the opportunity to participate in conferences right from the beginning of my Ph.D. studies.

Thanks goes to my co-authors Shiva Chaudhuri, Leszek Gąsieniec, Thore Husfeldt, Chris Okasaki, Jaikumar Radhakrishnan, Sven Skyum, Jesper Larsson Träff and Christos D. Zaroliagis.

I also would like to thank all the people at BRICS who throughout the years were willing/had the patience to listen to my problems and ideas: Lars Arge, Peter G. Binderup, Dany Breslauer, Devdatt Dubhashi, Gudmund S. Frandsen, Rune Bang Lyngsø, Peter Bro Miltersen, Christian Nørgaard Storm Pedersen, Saša Pekeč and Theis Rauhe. Also thanks to all the other people at DAIMI who made the years pass incredible fast, including all the people willing to share a pot of coffee from time to time, the always helpful secretaries, and the staff keeping the system running.

I am also deeply grateful to Kurt Mehlhorn at the Max-Planck-Institut für Informatik for letting me visit his group. During my nine months in Saarbrücken I met many people, all making it a pleasant time for me to be there. Except for those already mentioned thank goes to Susanne Albers, Phil Bradford, Gautam Das, Eddie and Jeannie Grove, Dimitios Gunopulos, Prosenjit Gupta, Thorben Hagerup, Andrea Rafael and Michel Smid for a great time in Saarbrücken.

Finally I would like to thank Arne Andersson, Ian Munro and Mogens Nielsen for being on my thesis evaluation committee.

The work in this thesis was financially supported by the Danish Research Academy, the Danish Natural Science Research Council (Grant No. 9400044), BRICS (Acronym for Basic Research in Computer Science, a Centre of the Danish National Research Foundation), the ESPRIT II Basic Research Actions Program of the EC under contract No. 7141 (project ALCOM II), and the ESPRIT Long Term Research Program of the EU under contract No. 20244 (project ALCOM-IT).

Contents

Chapter 1

Introduction

In this thesis we study the design of efficient data structures. In particular we focus on the design of data structures allowing each of the individual operations on the data structures to have worst case efficient implementations. The concrete problems we consider in this thesis are general techniques to make data structure *partially persistent*, the implementation of *priority queues*, and the implementation of *dictionaries*.

The thesis consists of an introduction summarizing the results achieved and a sequence of papers. In Sections 1.1–1.5 we summarize our contributions and their relation to previous work. Chapter 2 summarizes the role of a very general "tool" used throughout the thesis, the application of *invariants*.

Chapter 3 [16]: *Partially Persistent Data Structures of Bounded Degree with Constant Update Time.* In Nordic Journal of Computing, pages 238–255, volume 3(3), 1996.

Chapter 4 [20]: *The Randomized Complexity of Maintaining the Minimum*, with Shiva Chaudhuri and Jaikumar Radhakrishnan. In Nordic Journal of Computing, Selected Papers of the 5th Scandinavian Workshop on Algorithm Theory (SWAT'96), volume 3(4), pages 337-351, 1996.

Chapter 5 [15]: *Fast Meldable Priority Queues.* In Proc. 4th Workshop on Algorithms and Data Structures, LNCS volume 955, pages 282–290, 1995.

Chapter 6 [18]: *Worst-Case Efficient Priority Queues.* In Proc. 7th ACM-SIAM Symposium on Discrete Algorithms, pages 52–58, 1996.

Chapter 7 [17]: *Priority Queues on Parallel Machines.* In Proc. 5th Scandinavian Workshop on Algorithm Theory, LNCS volume 1097, pages 416–427, 1996.

Chapter 8 [23]: *A Parallel Priority Data Structure with Applications*, with Jesper Larsson Träff and Christos D. Zaroliagis. In Proc. 11th International Parallel Processing Symposium, pages 689-693, 1997.

Chapter 9 [19]: *Predecessor Queries in Dynamic Integer Sets.* In Proc. 14th Symposium on Theoretical Aspects of Computer Science, LNCS volume 1200, pages 21-32, 1997.

Chapter 10 [21]: *Approximate Dictionary Queries*, with Leszek Gąsieniec. In Proc. 7th Combinatorial Pattern Matching, LNCS volume 1075, pages 65–74, 1996.

1.1 Persistent data structures

A data structure is said to be *ephemeral* if updates to the data structure destroy the old version of the data structure, *i.e.*, the only version of the data structure remembered is the current version of the data structure. A data structure is said to be *persistent* if old versions of the data structure are remembered and can be accessed. Ordinary data structures are ephemeral. Persistent data

structures are used in a variety of areas, such as computational geometry, text and file editing, and high-level programming languages. For references see Driscoll *et al.* [44].

We distinguish three types of persistent data structures, depending on how new versions of the data structure are achieved:

- A *partially* persistent data structure allows all old versions to be accessed but only the most recent version to be modified.
- A *fully* persistent data structure allows all old versions both to be both accessed and modified.
- A *confluently* persistent data structure allows all old versions to be accessed and modified. In addition two versions can be *combined* to a new version. The semantics of combining two versions of the data structures is context dependent.

The differences between the three types of persistence are best illustrated by the corresponding *version graphs*. The version graph of a persistent data structure is a directed graph where each node corresponds to a version of the ephemeral data structure and an edge from a node representing version D_1 of the data structure to a node representing another version D_2 states that D_2 is obtained from D_1. The version graph of a partially persistent data structure is always a list, and the version graph of a fully persistent data structure is a rooted tree. The version graph of a confluently persistent data structure is a directed acyclic graph where all nodes have in-degree one or two, except for the node corresponding to the initial data structure.

The first general approaches to making data structures partially persistent were described by Overmars [86]. The first (straightforward) approach considered by Overmars is to explicitly store a copy of each of the old versions of the data structure. The drawbacks of this approach are obvious: It is very space inefficient and the time overhead to make the data structure persistent is linear in the size of the data structure for each update. The second approach is to store only the sequence of updates, and when accessing an old version of the data structure to rebuild that version from scratch. The third approach considered by Overmars is a hybrid version where in addition to the sequence of operations also every kth version of the data structure is stored (for k chosen appropriately).

Later Driscoll *et al.* [44] developed two efficient general techniques for making a large class of pointer based data structures both partially and fully persistent. The two techniques are denoted *node copying* and *node spitting*. The techniques require that the ephemeral data structures are pointer based, *i.e.*, the ephemeral data structures consist of records each containing a constant number of fields containing either atomic values or pointers to other records. In addition it is required that the ephemeral data structures satisfy the *bounded in-degree* assumption: There exists a constant d such that all records have in-degree bounded by d.

Both persistence techniques allow each update step to be performed in amortized constant time and amortized constant space, and support each access step in worst case constant time.

Our contribution in Chapter 3 is to extend the partial persistence technique of Driscoll *et al.* such that both update and access steps can be supported in worst case constant time, *i.e.*, we show how to eliminate the amortization from the node copying technique. Our main contribution is a data structure allowing us to avoid cascades of node copyings in the original node copying technique (see Chapter 3 for the details). We present our solution as a strategy for a two player pebble game on dynamic graphs. The reformulation of the partial persistence technique as a pebble game was first considered by Dietz and Raman [33, 92]; our solution solves an open problem about the pebble game introduced by Dietz and Raman [33].

By assuming the more powerful unit cost RAM model Dietz and Raman [33] have presented a different solution for the partial persistence problem achieving worst case constant time for access and update steps for any pointer based data structure with polylogarithmic bounded in-degree. Dietz and Raman also describe how to make data structures of constant bounded in-degree fully persistent with a worst case slow down of $O(\log\log n)$.

Dietz [31] has shown how to make arrays fully persistent on a unit cost RAM in expected amortized time $O(\log\log n)$ and constant space per operation. Because the memory of a RAM can be considered as an array the result of Dietz applies to all data structures on a unit cost RAM.

As mentioned several general techniques have been developed to convert ephemeral data structure into their partially or fully persistent counterparts, but for confluently persistence no such technique has been identified. The design of confluently persistent data structures is quite involved. An example is the development of confluently persistent catenable lists [25, 45, 67]. But by designing data structures such that they can be implemented in a purely functional language they automatically become confluently persistent [85]. Such data structures have been denoted *purely functional* data structures. Some recently developed purely functional data structures are: queues and deques [84], random access lists [83], catenable lists [67], priority queues [22] and catenable finger search trees [68]. A survey on the design of functional data structures can be found in the thesis of Okasaki [85].

It remains an interesting open problem if there exists a construction which can remove the amortization from the node splitting technique of Driscoll *et al.* [44] for making data structures fully persistent, *i.e.*, if there exists a data structure that can prevent cascades of node splittings in the node splitting technique. A related open problem involving node splitting is to give a pointer based data structure for avoiding cascades of node splittings when inserting elements into an (a,b)-tree [64]. A solution to this problem would imply the first pointer based implementation for constant update time finger search trees [60] and a first step towards removing the amortization from the node splitting technique of of Driscoll *et al.* [44] . On a RAM constant update time finger search trees have been given by Dietz and Raman [34]. Pointer based implementations of constant update time search trees have been given by Levcopoulos and Overmars [73] and Fleischer [48].

1.2 Comparison based priority queues

A major part of this thesis is devoted to the implementation of *priority queues*. In this section we consider implementations in three different types of computational models: sequential comparison based models, parallel comparison based models and RAM models allowing the manipulation of words. Two other types models are described in Sections 1.3 and 1.4.

A priority queue is a data structure storing a set of elements from a totally ordered universe and supporting a subset of the operations listed below.

MakeQueue creates and returns an empty priority queue.

FindMin(Q) returns the minimum element contained in priority queue Q.

Insert(Q,e) inserts an element e into priority queue Q.

Meld(Q_1,Q_2) melds priority queues Q_1 and Q_2 to a new priority queue and returns the resulting priority queue.

DecreaseKey(Q,e,e') replaces element e by e' in priority queue Q provided $e' \leq e$ and it is known where e is stored in Q.

DeleteMin(Q) deletes and returns the minimum element from priority queue Q.

DELETE(Q, e) deletes element e from priority queue Q provided it is known where e is stored in Q.

The minimum requirement is that the operations MAKEQUEUE, INSERT and DELETEMIN are supported. For sake of simplicity we do not distinguish between elements and their associated keys [76] (except for the data structure mentioned in Chapter 8).

1.2.1 Lower bounds

No unique optimal priority queue implementation exists, because any non constant operation can be made constant by increasing the cost of some of the other operations supported. The two extreme examples illustrating this fact are the following two trivial implementations:

- By storing each priority queue as a doubly linked list of the elements in an arbitrary order, the operations FINDMIN and DELETEMIN can be implemented in worst case time $O(n)$ and all other operations in worst case constant time.

- By storing each priority queue as a sorted doubly linked list of the elements, the operations INSERT, MELD and DECREASEKEY can be implemented in worst case time $O(n)$ and all other operations in worst case constant time.

In the following we list known lower bound relations between the worst case time of the individual operations in the comparison model.

Either INSERT or DELETEMIN requires $\Omega(\log n)$[1] comparisons because n elements can be sorted by performing MAKEQUEUE, n INSERT operations and n DELETEMIN operations, and comparison based sorting requires $\Omega(n \log n)$ comparisons. In Chapter 5 we present another reduction to sorting showing that if MELD is supported in worst case time $o(n)$, then DELETEMIN requires worst case time $\Omega(\log n)$.

But DELETEMIN consists of both a FINDMIN and a DELETE operation, and the above statements do not identify which of the two operations require time $\Omega(\log n)$. In Chapter 4 we show the following tradeoff answering this question: If INSERT and DELETE take worst case time $O(t(n))$, then FINDMIN requires worst case time $n/2^{O(t(n))}$. We give two proofs of the tradeoff; the first is an explicit adversary argument, and the second is a decision tree argument. The decision tree argument implies the stronger result that if the updates take expected amortized time $O(t(n))$ then the expected time for FINDMIN is at least $n/2^{O(t)}$.

The same tradeoff and proof holds if the assumption about INSERT is replaced by the assumption that a priority queue with n elements can be built in worst case time $O(n \cdot t(n))$ — the proof only uses the fact that a priority queue containing n elements can be build by n applications of INSERT in worst case time $O(n \cdot t(n))$. If MELD is supported in worst case time $o(n)$ then from Chapter 5 a priority queue with n elements can be build in time $o(n \log n)$. If in addition FINDMIN is supported in time $O(n^\epsilon)$ for a constant $\epsilon < 1$ then from the tradeoff of Chapter 4 it follows that DELETE must require time $\Omega(\log n)$.

This gives the following characterization of an "optimal" priority queue implementation.

Observation *If a priority queue implementation supports* MELD *in worst case time* $o(n)$ *and* FINDMIN *in worst case time* $O(n^\epsilon)$ *for a constant* $\epsilon < 1$, *then* DELETE *and* DELETEMIN *require worst case time* $\Omega(\log n)$.

[1] $f(n) \in \Omega(g(n))$ if and only if there exists an $\epsilon > 0$ such that $f(n) \geq \epsilon g(n)$ for infinitely many $n \in I\!N$.

The priority queue implementations we give in Chapters 5 and 6 are optimal in this sense, because they support DELETE and DELETEMIN in worst case time $O(\log n)$ and all other operations in worst case constant time.

1.2.2 Implementations

The first implementation of a priority queue was given by Williams in 1964 [109]. Williams' data structure, also known as a *heap*, represents a priority queue as a complete binary tree where each node stores one element and the elements satisfy heap order, *i.e.*, the element stored at a node is larger than or equal to the element stored at the node's parent. A heap can support all operations except MELD in worst case time $O(\log n)$, and can be stored efficiently in an array of size n without the use of pointers, *i.e.*, a heap is an example of an *implicit data structure* [82].

The properties of heaps have been the topic of comprehensive research since the introduction by Williams. That a heap containing n elements can be built in worst case time $O(n)$ was shown by Floyd [49]. In Section 1.3 we mention the bounds achieved by parallel heap construction algorithms. The worst case number of comparisons for INSERT and DELETEMIN has among others been considered by Gonnet and Munro [58] and Carlsson [26]. The average case behavior of heaps has been considered in [13, 39, 40, 91].

Many priority queue implementations have been developed which support all the listed operations in worst case or amortized time $O(\log n)$. Common to all the implementations is that they all are based on heap ordered trees. The constructions we give in Chapter 5 and 6 are no exception. The most prominent implementations are binomial queues [24, 108], heap ordered $(2,3)$-trees [1], self-adjusting heaps [99], pairing heaps [52], Fibonacci heaps [53] and relaxed heaps [43]. Further priority queue implementations can be found in [27, 46, 47, 63, 71, 97, 107].

The best amortized performance achieved by the data structures mentioned above is achieved by binomial queues and Fibonacci heaps. Binomial queues support all operations except DELETE, DELETEMIN and DECREASEKEY in amortized constant time, and DELETE, DELETEMIN and DECREASEKEY in amortized time $O(\log n)$ (DECREASEKEY implemented as DELETE followed by INSERT). Fibonacci heaps achieve the same time bounds as binomial queues except that DECREASEKEY is supported in amortized constant time too. The best worst case bounds are achieved by relaxed heaps. Relaxed heaps achieve worst case constant time for all operations except for DELETE, DELETEMIN and MELD which require worst case time $O(\log n)$.

The first nontrivial priority queue implementation supporting MELD in worst case time $o(\log n)$ was presented by Fagerberg [46]. The cost of achieving this sublogarithmic melding is that the time required for DELETEMIN increases to $\omega(\log n)$.

In Chapters 5 and 6 we present the first priority queue implementations that simultaneously support MELD in worst case constant time and DELETEMIN in worst case time $O(\log n)$. This is similar to the amortized time achieved by binomial queues and Fibonacci heaps, but in the worst case sense. From the tradeoff characterization it follows that the time bounds are the "best possible" for the individual operations. Table 1.1 summarizes the best known bounds for different priority queue implementations.

As mentioned both our constructions are based on heap ordered trees. To achieve constant time for the operations INSERT and MELD we adopt properties of redundant counter systems to control the linking of heap ordered subtrees. Previously van Leeuwen [107] and Carlsson *et al.* [27] have adopted similar ideas to construct priority queue implementations supporting INSERT in constant time. In Chapter 6 we achieve constant time for DECREASEKEY by combining ideas from redundant counter systems with the concept of heap order violations, *i.e.*, a subset of the tree nodes are not

	Amortized		Worst case		
	Vuillemin [108]	Fredman *et al.* [53]	Driscoll *et al.* [43]	Chapter 5	Chapter 6
MakeQueue	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
FindMin	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Insert	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Meld	$O(1)$	$O(1)$	$O(\log n)$	$O(1)$	$O(1)$
DecreaseKey	$O(\log n)$	$O(1)$	$O(1)$	$O(\log n)$	$O(1)$
Delete/DeleteMin	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$

Table 1.1: The best amortized and worst case time bounds for priority queue implementations.

required to satisfy heap order. Allowing heap order violations can be used to achieve constant time DecreaseKey operations as shown by Driscoll *et al.* [43]. The data structures in Chapters 5 and 6 require slightly different computational models. Whereas the data structure in Chapter 5 can be implemented on a pointer based machine model, the data structure in Chapter 6 requires arrays of size $O(\log n)$. It remains an open problem if the worst case bounds of Chapter 6 can be obtained without the use of arrays, *i.e.*, if a pointer based implementation exists which supports both Meld and DecreaseKey in worst case constant time and DeleteMin in worst case time $O(\log n)$.

The data structure of Chapter 5 has been reconsidered in a functional setting by Brodal and Okasaki [22]. In [22] a purely functional and therefore also confluently persistent priority queue implementation is presented supporting MakeQueue, FindMin, Insert and Meld in constant time and DeleteMin in time $O(\log n)$.

1.3 Parallel priority queues

Floyd showed in [49] how to build a heap sequentially in time $O(n)$. How to build heaps in parallel with optimal work $O(n)$ has been considered in a sequence of papers [32, 35, 70, 95]. On an EREW PRAM a work optimal algorithm achieving time $O(\log n)$ was given by Rao and Zhang [95] and on a CRCW PRAM a work optimal algorithm achieving time $O(\log \log n)$ was given by Dietz and Raman [35]. A work optimal randomized CRCW PRAM algorithm has been presented by Dietz and Raman [35]; it runs in time $O(\log \log \log n)$ with high probability. In the randomized parallel comparison tree model Dietz has presented an algorithm that with high probability takes time $O(\alpha(n))$[2] and does work $O(n)$ [32].

There exist two different avenues of research adopting parallelism to priority queues. The first is to speed up the individual priority queue operations by using $O(\log n)$ processors such that the individual operations require time $o(\log n)$. The second is to support the concurrent insertion/deletion of k elements by the following two operations, where k is assumed to be a constant.

MultiInsert$(Q, e_1, \ldots, e_k)$ Inserts elements $e_1, \ldots, e_k$ into priority queue Q.

MultiDelete(Q) Deletes and returns the k least elements contained in priority queue Q.

The first approach is appropriate for applications where the number of processors is small compared to the number of elements, say $O(\log n)$ processors. The second is appropriate when

[2] $\alpha(n)$ denotes an inverse of Ackerman's function.

there are a large number of processors available. An interesting application of parallel priority queues is to parallelize Branch-and-Bound algorithms [89, 96].

We first summarize the speed up of the individual priority queue operations. It is parallel computing folklore that a systolic processor array with $\Theta(n)$ processors can implement a priority queue supporting the operations INSERT and DELETEMIN in constant time, see Exercise 1.119 in [72]. The first approach using $o(n)$ processors is due to Biswas and Browne [11] and Rao and Kumar [96] who considered how to let $O(\log n)$ processors concurrently access a binary heap by pipelining INSERT and DELETEMIN operations. However, each operation still takes time $O(\log n)$ to finish. In [11, 96] the algorithms assume that the processors have shared memory. Later Ranade *et al.* [94] showed how to obtain the same result for the simplest network of processors namely a linear array of $O(\log n)$ processors.

Pinotti and Pucci [90] presented a non-pipelined EREW PRAM priority queue implementation that supports INSERT and DELETEMIN operations in time $O(\log \log n)$ with $O(\log n / \log \log n)$ processors. Chen and Hu [28] later gave an implementation which also supports MELD in time $O(\log \log n)$. Recently Pinotti *et al.* [88] achieved matching bounds and in addition supported the operations DELETE and DECREASEKEY in amortized time $O(\log \log n)$ on a CREW PRAM.

Our contribution (Chapter 7) is a new parallel priority queue implementation supporting all the operations considered in Section 1.2. Compared to the implementation of Pinotti and Pucci [90] our implementation is the first non-pipelined EREW PRAM priority queue implementation supporting INSERT and DELETEMIN in time $O(1)$ with $O(\log n)$ processors. By assuming the more powerful CREW PRAM we can also support the operations MELD, DELETE and DECREASEKEY in time $O(1)$ with $O(\log n)$ processors. A priority queue containing n elements can be build optimally on an EREW PRAM in time $O(\log n)$ with $O(n / \log n)$ processors. Because a straightforward implementation of MULTIINSERT is to perform BUILD on the elements to be inserted followed by a MELD operation we immediately get a CREW PRAM implementation supporting MULTIINSERT in time $O(\log k)$ with $O((\log n + k)/ \log k)$ processors. We also describe how our priority queues can be modified to allow operations to be performed via pipelining. As a result we get an implementation of priority queues on a processor array with $O(\log n)$ processors, supporting the operations MAKEQUEUE, INSERT, MELD, FINDMIN, DELETEMIN, DELETE and DECREASEKEY in constant time, which extends the result of [94].

The second avenue of research mentioned is to support the operations MULTIINSERT and MULTIDELETE. In [89] Pinotti and Pucci introduced the notion of *k-bandwidth* parallel priority queue implementations. The basic idea of the k-bandwidth technique is to modify a heap ordered priority queue implementation such that each node stores k elements instead of one and to require extended heap order among the elements, *i.e.*, the k elements stored at a node are required to be larger than or equal to the k elements stored at the parent of the node. Implementations of k-bandwidth-heaps and k-bandwidth-leftist-heaps for the CREW PRAM are contained in [89].

The k-bandwidth technique is central to all parallel priority queue implementations that support MULTIDELETE. Ranade *et al.* [94] show how to apply the k-bandwidth technique to achieve a parallel priority queue implementation for a d-dimensional array of processors, and Pinotti *et al.* [88] and Das *et al.* [30] give implementations for hypercubes.

Table 1.2 summarizes the performance of different implementations adopting parallelism to priority queues.

A classical application of priority queues is in Dijkstra's algorithm for the single-source shortest path problem on a graph with n vertices and m positive weighted edges [37]. By using Fibonacci heaps Dijkstra's (sequential) algorithm gets a running time of $O(m + n \log n)$ [53]. The essential

[3]The operations DELETE and DECREASEKEY require the CREW PRAM and require amortized time $O(\log \log n)$.

Model	[90] EREW	[88] EREW^3	[89] CREW	[28] EREW	[94] Array	Chapter 7 CREW
FindMin	1	$\log\log n$	1	1	1	1
Insert	$\log\log n$	$\log\log n$	–	–	1	1
DeleteMin	$\log\log n$	$\log\log n$	–	–	1	1
Meld	–	$\log\log n$	$\log\frac{n}{k}+\log\log k$	$\log\log\frac{n}{k}+\log k$	–	1
Delete	–	$\log\log n$	–	–	–	1
DecreaseKey	–	$\log\log n$	–	–	–	1
Build	$\log n$	–	$\frac{n}{k}\log k$	$\log\frac{n}{k}\log k$	–	$\log n$
MultiInsert	–	–	$\log\frac{n}{k}+\log k$	$\log\log\frac{n}{k}+\log k$	–	$\log k$
MultiDelete	–	–	$\log\frac{n}{k}+\log\log k$	$\log\log\frac{n}{k}+\log k$	–	–

Table 1.2: Performance of some parallel implementations of priority queues.

property of Dijkstra's algorithm is that it performs n iterations, each iteration consisting of a DeleteMin operation followed by a number of DecreaseKey operations. Driscoll *et al.* [43] have presented an EREW PRAM implementation of Dijkstra's algorithm running in time $O(n\log n)$ and doing work $O(m+n\log n)$. The question addressed by Driscoll *et al.* is how to parallelize the DecreaseKey operations involved in each of the iterations. In Chapter 8 we address the same problem by giving two implementations of Dijkstra's algorithm for a CREW PRAM running in time $O(n)$ and doing respectively work $O(n^2)$ and $O(m\log n)$. Our contribution is a parallel priority queue implementation which supports an arbitrary number of DecreaseKey operations in constant time provided that the elements involved in the DecreaseKey operations have been presorted.

1.4 RAM priority queues and dictionaries

Section 1.2 and 1.3 discussed the implementation of comparison based priority queues. In this section we consider how to implement priority queues and *dictionaries* on the more powerful RAM models. The details of our contribution are contained in Chapter 9.

As model we assume a unit cost RAM with a word size of w bits, allowing addition, arbitrary left and right bit shifts and bit-wise boolean operations on words to be performed in constant time. In addition we assume direct and indirect addressing, jumps and conditional statements. Miltersen [78] refers to this model as a *Practical RAM*. We assume the elements stored are integers in the range $0..2^w-1$, *i.e.*, each element fits within a machine word.

The operations we consider on a set of elements S are:

Insert(e) inserts element e into S,

Delete(e) deletes element e from S,

Pred(e) returns the largest element $\leq e$ in S, and

FindMin/FindMax returns the minimum/maximum element in S.

We denote a data structure supporting the operations Insert, Delete and FindMin a *RAM priority queue*. Notice that in contrast to Section 1.2 Delete does not require knowledge of the position of the element to be deleted. A data structure supporting Insert, Delete and Pred we denote a *RAM dictionary*. In the following we summarize existing RAM priority queue

and dictionary implementations – they all achieve strictly better bounds than comparison based implementations. For a survey on comparison based implementations we refer to Mehlhorn and Tsakalidis [76], but one representative example is (a,b)-trees [64] which support all the mentioned operations in time $O(\log n)$.

The first data structure showing that the $\Omega(\log n)$ lower bound for comparison based implementations does not hold for bounded universes is due to van Emde Boas *et al.* [104, 106]. The data structure of van Emde Boas *et al.* supports the operations INSERT, DELETE, PRED, FINDMIN and FINDMAX on a Practical RAM in worst case $O(\log w)$ time. For word size $\log^{O(1)} n$ this implies a time $O(\log\log n)$ implementation. Thorup [102] has presented a priority queue supporting INSERT and DELETEMIN in worst case time $O(\log\log n)$ independently of the word size w. Thorup notes that by tabulating the multiplicity of each of the inserted elements the construction supports DELETE in amortized $O(\log\log n)$ time by skipping extracted integers of multiplicity zero.

Andersson [5] has presented an implementation of a RAM dictionary supporting INSERT, DELETE and PRED in worst case $O(\sqrt{\log n})$ time and FINDMIN and FINDMAX in worst case constant time. Several data structures can achieve the same time bounds as Andersson [5], but they all require constant time multiplication [6, 54, 93]. Andersson [5] mentions that there exists a $\Omega(\log^{1/3-o(1)} n)$ lower bound for the dictionary problem on a practical RAM.

Our contribution in Chapter 9 is a data structure for a Practical RAM supporting FINDMIN and FINDMAX in worst case constant time, INSERT and DELETE in worst case $O(f(n))$ time, and PRED in worst case $O((\log n)/f(n))$ time where $f(n)$ is an arbitrary nondecreasing smooth function satisfying $\log\log n \leq f(n) \leq \sqrt{\log n}$.

If $f(n) = \log\log n$ we support the operations INSERT, DELETEMIN and DELETE in worst case time $O(\log\log n)$, *i.e.*, we achieve the result of Thorup but in the worst case sense. Furthermore we support PRED queries in worst case $O(\log n/\log\log n)$ time. If $f(n) = \sqrt{\log n}$, we achieve time bounds matching those of Andersson [5].

Our construction is obtained by combining the data structure of van Emde Boas *et al.* [104, 106] with packed search trees similar to those of Andersson [5], but where we add *buffers* of delayed insertions and deletions to the nodes of the packed search tree. The idea of adding buffers to a search tree is inspired by Arge [7] who designs efficient external memory data structures.

The data structure presented in Chapter 9 is the first allowing predecessor queries in time $O(\log n/\log\log n)$ while having update time $O(\log\log n)$, *i.e.*, updates are exponentially faster than PRED queries. It remains an open problem if INSERT and DELETE can be supported in time $O(\log\log n)$ while supporting PRED queries in time $O(\sqrt{\log n})$.

1.5 Approximate Dictionary Queries

In Chapter 10 we consider the following *approximate dictionary problem* on a unit cost RAM. Let W be a dictionary of n binary strings each of length m. We consider the problem of answering d–queries, *i.e.*, for a binary query string α of length m to decide if there is a string in W with at most Hamming distance d of α.

Minsky and Papert originally raised this problem in [80]. Recently a sequence of papers have considered how to solve this problem efficiently [41, 42, 59, 74, 112]. Manber and Wu [74] considered the application of approximate dictionary queries to password security and spelling correction of bibliographic files. Dolev *et al.* [41, 42] and Greene, Parnas and Yao [59] considered approximate dictionary queries for the case where d is large. The initial effort towards a theoretical study of the small d case was given by Yao and Yao in [112]. They present for the case $d = 1$ a data structure

supporting queries in time $O(m \log\log n)$ with space requirement $O(nm \log m)$. Their solution is described in the cell-probe model of Yao [111] with word size one.

For the general case where $d > 1$, d–queries can be answered in optimal space $O(nm)$ doing $\sum_{i=0}^{d} \binom{m}{i}$ exact dictionary queries for all the possible strings with Hamming distance at most d of the query string, where each exact dictionary query requires time $O(m)$ by using the data structure of Fredman, Komlos and Szemeredi [51]. On the other hand d–queries can be answered in time $O(m)$ when the size of the data structure can be $O(n \sum_{i=0}^{d} \binom{m}{i})$ by inserting all $n \sum_{i=0}^{d} \binom{m}{i}$ strings with Hamming distance at most d into a compressed trie.

It is unknown how the above mentioned data structure which supports 1–queries in time $O(m)$ can be constructed in time $O(nm)$. In Chapter 10 we present a standard unit cost RAM implementation which has optimal size $O(nm)$ and supports 1–queries in time $O(m)$ and which can be constructed in time $O(nm)$. Our data structure can be made semi-dynamic by supporting insertions of new binary strings in amortized time $O(m)$, when starting with an initial empty dictionary.

It remains an open problem if there exists a data structure having size $O(nm)$ which can answer d–queries in time $o(m^d)$ for $d \geq 2$.

Chapter 2

The role of invariants

One of the basic ideas used intensively to obtain the results of this thesis is the fundamental concept of *invariants*. We use invariants in the description of data structures, in lower bound/adversary arguments, in the analysis of expected running times of randomized algorithms, and to argue about the correctness of our algorithms. However, the most significant application of invariants has been in the *design* of the data structures presented. In the following we describe the role of invariants in our work, in particular the interaction between developing invariants and designing data structures.

In Chapters 3 and 4 we use invariants to describe explicit adversaries for respectively the pebble game of Dietz and Raman [33] and for algorithms maintaining the minimum of a set. In both cases the invariants capture the basic idea of the adversary, and the adversaries' moves are "dictated" by the invariants. In Chapter 4 we also use an invariant to reason about the expected running time of a randomized algorithm maintaining the minimum of a set. In fact, the random choices of the algorithm have been designed such that the invariant is satisfied.

In Chapters 5, 6 and 7 we use invariants to describe the needed extra properties of the (heap ordered) trees. The invariants describe how many sons of each rank a node can have, how many trees there are of each rank, and how many heap order violations there are. In Chapters 8 and 9 we similarly use invariants to bound the sizes of the involved buffers. In Chapter 8 the main application of invariants is to show that the involved buffers do not temporarily get empty, *i.e.*, to argue about a safety property used to prove the correctness of the algorithms.

Common to all the examples are that we developed the data structures hand in hand with the invariants. In most cases we actually first proposed a set of invariants, and then considered how to implement the required operations of the data structure under the constraints of the proposed invariants. If this failed, we modified the invariants and reconsidered the implementation of the operations. This cyclic process continued until feasible invariants were found. We believe that the main advantage of using invariants while designing data structures is that invariants make it quite easy to identify bottlenecks in the proposed constructions.

To illustrate this development process we briefly review the process of developing the data structure presented in Chapter 6. The goal was to develop a priority queue implementation supporting Meld and DecreaseKey in worst case constant time and DeleteMin in worst case time $O(\log n)$. The only data structure supporting Meld in constant time was the data structure in Chapter 5 (which supports Insert and Meld in worst case constant time and Delete and DeleteMin in worst case time $O(\log n)$), and the only one supporting DecreaseKey in constant time was the data structure of Driscoll *et al.* [43]. It therefore seemed an appropriate approach to combine the essential properties of both data structures, *i.e.*, to represent a priority queue by one heap ordered tree and to allow heap order violations. But due to the constant time requirement for

MELD it was necessary to dispense with the $O(\log n)$ bound on the number of violations (in [43]) and to allow $\Theta(n)$ violations! What followed were numerous iterations alternating between modifying the invariants and realizing that the current invariants were too strong/weak to allow an implementation of the operations achieving the required time bounds. Several mutually dependent technical problems had to be solved, including: a) how to distribute the violations, such that at most $O(\log n)$ violations should be considered during a single DELETEMIN, b) how to MELD two priority queues, in particular how to MELD two sets of $\Theta(n)$ violations, c) how to guarantee a $O(\log n)$ bound on the maximum rank, and d) how to make transformations reducing the number of violations. The final invariants solving all these problems can be found in Chapter 6 (invariants S1–S5, O1–O5 and R1–R3).

In Chapter 4 we similarly use invariants to develop a randomized algorithm for the FINDANY problem (see Chapter 4 for a definition). The main idea of the algorithm is to maintain the current rank of an "arbitrarily" selected element from the set of elements maintained. The invariant is used to formalize the meaning of "arbitrarily", by requiring that the selected element must be uniformly picked from the set of elements maintained. The implementation of INSERT and DELETE are straightforward, when the random choices satisfy the invariant. While the expected time for INSERT now follows directly from the implementation of INSERT, the expected time for DELETE follows from the invariant because DELETE is only expensive when the picked element is also the element to be deleted.

In Chapter 8 we present parallel implementations of Dijkstra's algorithm for the single-source shortest path problem based on pipelined merging of adjacency lists. Crucial to the correctness of the algorithms is that the buffers in the pipelines between processors do not illegally temporarily get empty. For the sequential pipeline presented we give an invariant which guarantees that buffers never become empty. For the tree pipeline presented we give a different invariant guaranteeing that if the buffers temporarily get empty (which they can in the tree pipeline) it is safe to ignore them.

Finally, in Chapter 3 we use invariants to describe an adversary for a pebble game on cliques (see Chapter 3 for a definition of the game). The goal of the adversary is to force a node from the clique to have a lot of pebbles. After an initialization phase the adversary maintains a subset of the nodes as candidates for the node to store many pebbles. A lower bound for the number of pebbles on each of the candidate nodes is described by a set of invariants (involving as a parameter the number of steps performed after the initialization phase). The moves of the adversary follow immediately from the invariants. From the invariants it follows that after a number of rounds there exists a candidate node having the number of pebbles claimed.

Viewed in isolation, none of these applications of invariance techniques are new. However, we consider it to be quite noteworthy that the power of the invariance technique is so clearly demonstrated in connection with the development of (highly nontrivial) combinatorial algorithms.

Chapter 3

Partially Persistent Data Structures of Bounded Degree with Constant Update Time

Partially Persistent Data Structures of Bounded Degree with Constant Update Time*

Gerth Stølting Brodal
BRICS†, Department of Computer Science, University of Aarhus
Ny Munkegade, DK-8000 Århus C, Denmark
gerth@brics.dk

Abstract

The problem of making bounded in-degree and out-degree data structures partially persistent is considered. The node copying method of Driscoll *et al.* is extended so that updates can be performed in *worst case* constant time on the pointer machine model. Previously it was only known to be possible in amortized constant time.

The result is presented in terms of a new strategy for Dietz and Raman's dynamic two player pebble game on graphs.

It is shown how to implement the strategy and the upper bound on the required number of pebbles is improved from $2b+2d+O(\sqrt{b})$ to $d+2b$, where b is the bound of the in-degree and d the bound of the out-degree. We also give a lower bound that shows that the number of pebbles depends on the out-degree d.

Category: E.1, F.2.2
Keywords: data structures, partial persistence, pebble game, lower bounds

3.1 Introduction

This paper describes a method to make data structures *partially persistent*. A partially persistent data structure is a data structure in which old versions are remembered and can always be inspected. However, only the latest version of the data structure can be modified.

An interesting application of a partially persistent data structure is given in [98] where the planar point location problem is solved by an elegant application of partially persistent search trees. The method given in [98] can be generalised to make arbitrary bounded in-degree data structures partially persistent [44].

As in [44], the data structures we consider will be described in the pointer machine model, *i.e.*, they consist of records with a constant number of fields each containing a unit of data or a pointer to another record. The data structures can be viewed as graphs with bounded out-degree. In the following let d denote this bound.

The main assumption is that the data structures also have *bounded in-degree*. Let b denote this bound. Not all data structures satisfy this constraint — but they can be converted to do it: Replace

*This work was partially supported by the ESPRIT II Basic Research Actions Program of the EC under contract no. 7141 (project ALCOM II).
†BRICS (Basic Research in Computer Science), a Centre of the Danish National Research Foundation.

nodes by balanced binary trees, so that all original pointers that point to a node now instead point to the leafs in the tree substituted into the data structure instead of the node, and store the node's original information in the root of the tree. The assumption can now be satisfied by letting at most a constant number of pointers point to the same leaf. The drawback of this approach is that the time to access a node v is increased from $O(1)$ to $O(\log b_v)$ where b_v is the original in-degree of v.

The problem with the method presented in [44, 98] is that an update of the data structure takes *amortized* time $O(1)$, in the worst case it can be $O(n)$ where n is the size of the current version of the data structure.

In this paper we describe how to extend the method of [44, 98] so that an update can be done in *worst case* constant time. The main result of this paper is:

Theorem 1 *It is possible to implement partially persistent data structures with bounded in-degree (and out-degree) such that each update step and access step can be performed in* worst case *time* $O(1)$.

The problem can be restated as a dynamic two player pebble game on dynamic directed graphs, which was done by Dietz and Raman in [33]. In fact, it is this game we consider in this paper.

The central rules of the game are that player **I** can add a *pebble* to an arbitrary node and player **D** can remove all pebbles from a node provided he places a pebble on all of the node's predecessors. For further details refer to Section 3.3. The goal of the game is to find a strategy for player **D** that can guarantee that the number of pebbles on all nodes are bounded by a constant M. Dietz and Raman gave a strategy which achieved $M \leq 2b + 2d + O(\sqrt{b})$ — but they were not able to implement it efficiently which is necessary to remove the amortization from the original persistency result.

In this paper we improve the bound to $M = d + 2b$ by a simple modification of the original strategy. In the static case (where the graph does not change) we get $M = d + b$.

We also consider the case where the nodes have different bounds on their in- and out-degree. In this case we would like to have $M_v = f(b_v, d_v)$ where $f : N^2 \to N$ is a monotonically increasing function. Hence only nodes with a high in-degree should have many pebbles. We call strategies with this property for *locally adaptive*. In fact, the strategy mentioned above satisfies that $M_v = d_v + 2b_v$ in the dynamic game and $M_v = d_v + b_v$ in the static game.

By an *efficiently implementable strategy* we mean a strategy that can be implemented such that the move of player **D** can be performed in time $O(1)$ if player **D** knows where player **I** performed his move. In the following we call such strategies implementable.

The implementable strategies we give do not obtain such good bounds. Our first strategy obtains $M = 2bd + 1$, whereas the second is locally adaptive and obtains $M_v = 2b_v d_v + 2b_v - 1$.

The analysis of our strategies are all tight — we give examples which match the upper bounds. The two efficiently implementable strategies have simple implementations with small constant factors.

We also give lower bounds for the value of M which shows that M depends both on b and d for all strategies. More precisely we show that (we define $\log x = \max\{1, \log_2 x\}$):

$$M \geq \max\{b+1, \lfloor \alpha + \sqrt{2\alpha - 7/4} - 1/2 \rfloor, \left\lceil \frac{\log \frac{2}{3}d}{\log\log \frac{2}{3}d} - 1 \right\rceil\},$$

where $\alpha = \min\{b, d\}$.

The paper is organized as follows. In Section 3.2 we describe the method of [44, 98] and in Section 3.3 we define the dynamic graph game of [33]. In Section 3.4 we give the new game strategy

for player **D** which is implementable. The technical details which are necessary to implement the strategy are described in Section 3.5 and the strategy is analysed in Section 3.6. In Section 3.7 we give a locally adaptive strategy and in Section 3.8 we give a locally adaptive strategy which is implementable. Finally, the lower bound for M is given in Section 3.9.

3.2 The node copying method

In this section we briefly review the method of [44, 98]. For further details we refer to these articles. The purpose of this section is to motivate the game that is defined in Section 3.3, and to show that if we can find a strategy for this game and implement it efficiently, then we can also remove the amortization from the partially persistency method described below.

The *ephemeral data structure* is the underlying data structure we want to make partially persistent. In the following we assume that we have access to the ephemeral data structure through a finite number of entry pointers. For every update of the data structure we increase a version counter which contains the number of the current version.

When we update a node v we cannot destroy the old information in v because this would not enable us to find the old information again. The idea is now to add the new information to v together with the current version number. So if we later want to look at an old version of the information, we just compare the version numbers to find out which information was in the node at the time we are looking for. This is in very few words the idea behind the so called *fat node* method.

An alternative to the previous approach is the *node copying* method. This method allows at most a constant number (M) of additional pieces of information in each node (depending on the size of b). When the number of different copies of information in a node gets greater than M we make a copy of the node and the old node now becomes *dead* because new pointers to the node has to point to the newly created copy. In the new node we only store the information of the dead node which exists in the current version of the ephemeral data structure. We now have to update all the nodes in the current version of the data structure which have pointers to the node that has now become dead. These pointers should be updated to point to the newly created node instead — so we recursively add information to all the predecessors of the node that we have copied. The copied node does not contain any additional information.

3.3 The dynamic graph game

The game Dietz and Raman defined in [33] is played on a directed graph $G = (V, E)$ with bounded in-degree and out-degree. Let b be the bound of the in-degree and d the bound of the out-degree. W.l.o.g. we do not allow self-loops or multiple edges. To each node a number of pebbles is associated, denoted by P_v. The *dynamic graph game* is now a game where two players **I** and **D** alternate to move. The moves they can perform are:

Player **I**:

- **a)** add a pebble to an arbitrary node v of the graph or
- **b)** remove an existing edge (v, u) and create a new edge (v, w) without violating the in-degree constraint on w, and place a pebble on the node v.

Player **D**:

c) do nothing or

d) remove all pebbles from a node v and place a new pebble on all the predecessors of v. This is denoted by ZERO(v).

The goal of the game is to show that there exists a constant M and a strategy for player **D** such that, whatever player **I** does, the maximum number of pebbles on any node after the move of player **D** is bounded by M. In the static version of the game player **I** can only do moves of type a).

The relationship between partially persistent data structures and the pebble game defined is the following. The graph of the pebble game corresponds to the current version of an ephemeral data structure. A pebble corresponds to additional information stored in a node. A move of player **I** of type a) corresponds to updating a data field in the ephemeral data structure and a move of type b) corresponds to updating a pointer field in the ephemeral data structure. A ZERO operation performed by player **D** corresponds to the copying of a node in the node copying method. The pebbles placed on the predecessor nodes correspond to updating the incoming pointers of the corresponding node copied in the persistent data structure.

The existence of a strategy for player **D** was shown in [33], but the given strategy could not be implemented efficiently (*i.e.*, the node v in d) could not be located in time $O(1)$).

Theorem 2 (Dietz and Raman [33]) *A strategy for player* **D** *exists that achieves* $M = O(b+d)$.

3.4 The strategy

We now describe our new strategy for player **D**. We start with some definitions. We associate the following additional information with the graph G.

- Edges are either *black* or *white*. Nodes have at most one incoming white edge. There are no white cycles.
- Nodes are either *black* or *white*. Nodes are white if and only if they have an incoming white edge.

The definitions give in a natural way rise to a partition of the nodes into components: two nodes connected by a white edge belong to the same component. It is easily seen that a component is a rooted tree of white edges with a black root and all other nodes white. A single black node with no adjacent white edge is also a component. We call a component consisting of a single node a *simple component* and a component with more than one node a *non simple component*. See Figure 3.1 (on the left) for an example of a graph with two simple components and one non simple component.

To each node v we associate a queue Q_v containing the predecessors of v. The queue operations used in the following are:

- ADD(Q_u, v) adds v to the back of Q_u.
- DELETE(Q_u, v) removes v from Q_u.
- ROTATE(Q_u) moves the front element v of Q_u to the back of Q_u, and returns v.

The central operation in our strategy is now the following BREAK operation. The component containing v is denoted C_v.

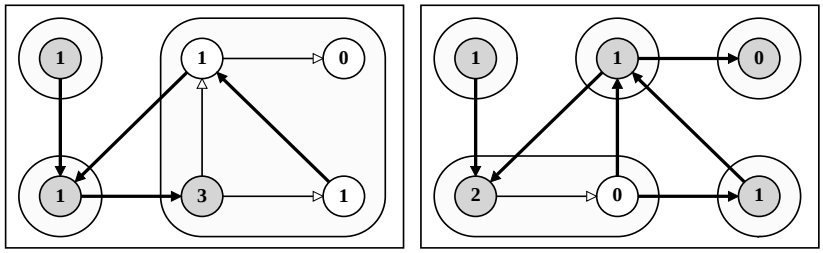

Figure 3.1: The effect of performing a BREAK operation. The numbers are the number of pebbles on the nodes.

procedure BREAK(C_v)
  $r \leftarrow$ the root of C_v
  color all nodes and edges in C_v black
  if $Q_r \neq \emptyset$ **then**
    color r and (ROTATE(Q_r), r) white
  endif
  ZERO(r)
end.

The effect of performing BREAK on a component is that the component is broken up into simple components and that the root of the original component is appended to the component of one of its predecessors (if any). An example of the application of the BREAK operation is shown in Figure 3.1.

A crucial property of BREAK is that all nodes in the component change color (except for the root when it does not have any predecessors, in this case we by definition say that the root changes its color twice).

Our strategy is now the following (for simplicity we give the moves of player **I** and the counter moves of player **D** as procedures).

procedure ADDPEBBLE(v)
  place a pebble on v
  BREAK(C_v)
end.

procedure MOVEEDGE($(v,u),(v,w)$)
  place a pebble on v
  if (v,u) is white **then**
    BREAK(C_v)
    DELETE(Q_u, v)
    replace (v,u) with (v,w) in E
    ADD(Q_w, v)
  else
    DELETE(Q_u, v)
    replace (v,u) with (v,w) in E
    ADD(Q_w, v)
    BREAK(C_v)
  endif
end.

In MoveEdge the place where we perform the Break operation depends on the color of the edge (v, u) being deleted. This is to guarantee that we only remove black edges from the graph (in order not to have to split components).

Observe that each time we apply AddPebble or MoveEdge to a node v we find the root of C_v and zero it. We also change the color of all nodes in C_v — in particular we change the color of v. Now, every time a black node becomes white it also becomes zeroed, so after two **I** moves have placed pebbles on v, v has been zeroed at least once. That the successors of a node v cannot be zeroed more than $O(1)$ times and therefore cannot place pebbles on v without v getting zeroed is shown in Section 3.6. The crucial property is the way in which Break colors nodes and edges white. The idea is that a successor u of v cannot be zeroed more than $O(1)$ times before the edge from (v, u) will become white. If (v, u) is white both v and u belong to the same component, and therefore u cannot change color without v changing color.

In Section 3.5 we show how to implement Break in worst case time $O(1)$ and in Section 3.6 we show that the approach achieves that $M = O(1)$.

3.5 The new data structure

The procedures in Section 3.4 can easily be implemented in worst case time $O(1)$ if we are able to perform the Break operation in constant time. The central idea is to represent the colors indirectly so that all white nodes and edges in a component points to the same variable. All the nodes and edges can now be made black by setting this variable to black.

A *component record* contains two fields. A color field and a pointer field. If the color field is white the pointer field will point to the root of the component.

To each node and edge is associated a pointer cr which points to a component record. We will now maintain the following invariant.

- The cr pointer of each black edge and each node forming a simple component will point to a component record where the color is black and the root pointer is the null pointer. Hence, there is a component record for each non simple component, but several black edges and nodes forming a simple component can share the same component record.
- For each non simple component there exist exactly one component record where the color is white and the root pointer points to the root of the component. All nodes and white edges in this component point to this component record.

An example of component records is shown in Figure 3.2. Notice that the color of an edge e is simply $e.cr.color$ so the test in MoveEdge is trivial to implement. The implementation of Break is now:

```
procedure BREAK(v)
    if v.cr.color = black then
        r ← v
    else
        r ← v.cr.root
        v.cr.color ← black
        v.cr.root ← −
    endif
    if r.Q ≠ ∅ then
        u ← ROTATE(r.Q)
```

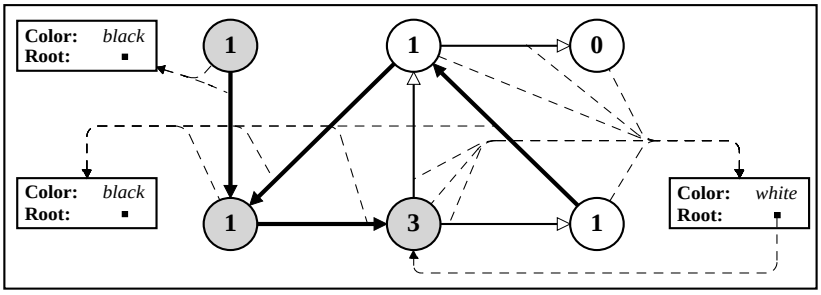

Figure 3.2: A graph with component records.

```
        if u.cr.color = black then
            u.cr ← new-component-record(white, u)
        endif
        r.cr ← (u, r).cr ← u.cr
    endif
    ZERO(r)
end.
```

From the discussion of the node copying method in Section 3.2 it should be clear that the above described data structure also applies to this method.

3.6 The analysis

Theorem 3 *The player* **D** *strategy given in Section 3.4 achieves* $M = 2bd + 1$.

Proof. A direct consequence of Lemmas 1 and 2. □

Lemma 1 *The player* **D** *strategy given in Section 3.4 achieves* $M \leq 2bd + 1$.

Proof. Let the first operation (either an ADDPEBBLE or MOVEEDGE operation) be performed at time 1, the next at time 2 and so on.

Assume that when the game starts all nodes are black and there are no pebbles on any node.

Fix an arbitrary node v at an arbitrary time t_{now}. Let t_{last} denote the last time before t_{now} when v was zeroed (if v has never been zeroed let t_{last} be 0). In the following we want to bound the number of pebbles placed on v in the interval $]t_{last}, t_{now}[$. In this interval v cannot change its color from black to white because this would zero v.

Assume without loss of generality that v is white at the end of time t_{last}, that at time $t_{break} \in]t_{last}, t_{now}[$ a BREAK(C_v) is performed and (therefore) at time t_{now} v is black (it is easy to see that all other cases are special cases of this case).

Note that the only time an ADDPEBBLE(v) or MOVEEDGE$((v,u),(v,w))$ operation can be performed is at time t_{break} because these operations force the color of v to change. Therefore, v's successors are the same in the interval $]t_{last}, t_{break}[$, and similarly for $]t_{break}, t_{now}[$.

We will handle each of the two intervals and the time t_{break} separately. Let us first consider the interval $]t_{last}, t_{break}[$. Let w be one of v's successors in this interval. w can be zeroed at most b times before it will be blocked by a white edge from v (w cannot change the color without changing the color of v), because after at most $b-1$ ZERO(w) operations, v will be the first element in Q_w.

So a successor of v can be zeroed at most bd times throughout the first interval which implies that at most bd pebbles can be placed on v during the first interval. For $]t_{break}, t_{now}[$ we can repeat the same argument so at most bd pebbles will be placed on b during this interval too.

We now just have to consider the operation at time t_{break}. The color of v changes so a BREAK(C_v) is performed. There are three possible reasons for that: a) An ADDPEBBLE(v) operation is performed, b) a MOVEEDGE $((v,u),(v,w))$ is performed or c) one of the operations is performed on a node different from v. In a) and b) we first add a pebble to v and then perform a BREAK(C_v) operation and in c) we first add a pebble to another node in C_v and then do BREAK(C_v). The BREAK operation can add at most one pebble to v when we perform a ZERO operation to the root of C_v (because we do not allow multiple edges) so at most two pebbles can be added to v at time t_{break}.

We have now shown that at time t_{now} the number of pebbles on v can be at most $2bd+2$. This is nearly the claimed result. To decrease this bound by one we have to analyse the effect of the operation performed at time t_{break} more carefully.

What we prove is that when two pebbles are placed on v at time t_{break} then at most $bd-1$ pebbles can be placed on v throughout $]t_{break}, t_{now}[$. This follows if we can prove that there exists a successor of v that cannot be zeroed more than $b-1$ times in the interval $]t_{break}, t_{now}[$.

In the following let r be the node that is zeroed at time t_{break}. We have the following cases to consider:

i) ADDPEBBLE(v) and BREAK(r) places a pebble on v. Now r and one of its incoming edges are white. So r can be zeroed at most $b-1$ times before (v,r) will become white and block further ZERO(r) operations.

ii) MOVEEDGE$((v,u),(v,w))$ and ZERO(r) places a pebble on v. Depending on the color of (v,u) we have two cases:

- **a)** (v,u) is white. Therefore u is white and $r \neq u$. Since we perform BREAK(r) before we modify the pointers we have that $r \neq w$. So as in i) r can be zeroed at most $b-1$ times throughout $]t_{break}, t_{now}[$.
- **b)** (v,u) is black. Since BREAK is the last operation we do, the successors of v will be the same until after t_{now}, so we can argue in the same way as i) and again get that r can be zeroed at most $b-1$ times throughout $]t_{break}, t_{now}[$.

We conclude that no node will ever have more than $2bd+1$ pebbles. □

Lemma 2 *The player* **D** *strategy given in Section 3.4 achieves* $M \geq 2bd+1$.

Proof. Let $G=(V,E)$ be the directed graph given by $V=\{r, v_1, \ldots, v_b, w_1, \ldots, w_d\}$ and $E = \{(r,v_b)\} \cup \{(v_i, w_j) | i \in \{1,\ldots,b\} \wedge j \in \{1,\ldots,d\}\}$. The graph is shown in Figure 3.3. Initially all nodes in V are black and all queues Q_{w_i} contain the nodes $(v_1,\ldots,v_b)$. We will now force the number of pebbles on v_b to become $2bd+1$.

First place one pebble on v_b — so that v_b becomes white. Then place $2b-1$ pebbles on each w_j. There will now be bd pebbles on v_b and all the edges (v_b, w_j) are white. Place one new pebble on v_b and place another $2b-1$ pebbles on each w_j. Now there will be $2bd+1$ pebbles on v_b. □

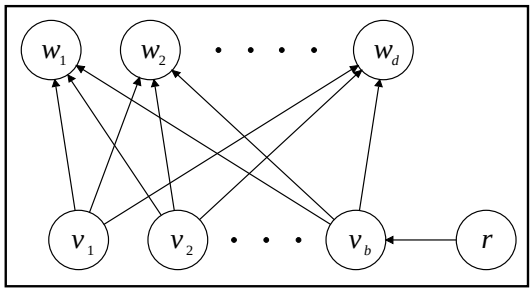

Figure 3.3: A graph which can force M to become $2bd + 1$.

3.7 A simple locally adaptive strategy

In this section we present a simple strategy that is adaptive to the local in- and out-degree bounds of the nodes. It improves the bound achieved in [33]. The main drawback is that the strategy cannot be implemented efficiently in the sense that the node to be zeroed cannot be found in constant time. In Section 3.8 we present an implementable strategy that is locally adaptive but does not achieve as good a bound on M.

Let d_v denote the bound of the out-degree of v and b_v the bound of the in-degree. Define M_v to be the best bound player **D** can guarantee on the number of pebbles on v. We would like to have that $M_v = f(b_v, d_v)$ for a monotonic function $f : N^2 \to N$.

The strategy is quite simple. To each node v we associate a queue Q_v containing the predecessors of v and a special element ZERO. Each time the ZERO element is rotated from the front of the queue the node is zeroed.

The simple adaptive strategy
  if the **I**-move deletes (v, u) and adds (v, w) **then**
    DELETE(Q_u, v)
    ADD(Q_w, v)
  endif
  while $(v' \leftarrow$ ROTATE$(Q_v)) \neq$ZERO **do** $v \leftarrow v'$ **od**
  ZERO(v)
end.

Notice that the strategy does not use the values of b_v and d_v explicitly. This gives the strategy the nice property that we can allow b_v and d_v to change dynamically.

The best bound Dietz and Raman could prove for their strategy was $M \leq 2b + 2d + O(\sqrt{b})$. The next theorem shows that the simple strategy above achieves a bound of $M_v = d_v + 2b_v$. If the graph is static the bound improves to $M_v = d_v + b_v$.

Theorem 4 *For the simple adaptive strategy we have that $M_v = d_v + 2b_v$. In the static case this improves to $M_v = d_v + b_v$.*

Proof. Each time we perform ADDPEBBLE(v) or MOVEEDGE$((v, u), (v, w))$ we rotate Q_v. It is possible to rotate Q_v at most b_v times without zeroing v. This implies that between two ZERO(v) operations at most b_v MOVEEDGE operations can be performed on outgoing edges of v. Therefore, v can have had at most $b_v + d_v$ different successors between two ZERO(v) operations. Between two zeroings of a successor w of v, Q_v must have been rotated because ROTATE(Q_w) returned v in the

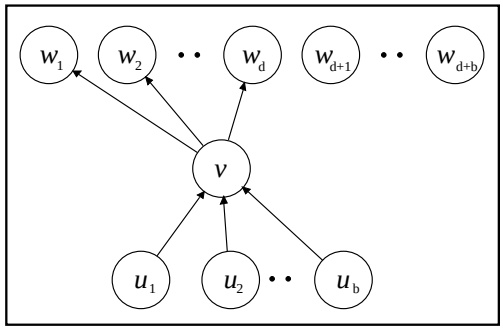

Figure 3.4: A graph which can force M to become $d_v + 2b_v$.

while-loop, this is because the ZERO element is moved to the back of Q_w when w is being zeroed. So except for the first zeroing of w all zeroings of w will be preceded by a rotation of Q_v.

For each operation performed on v we both place a pebble on v and rotate Q_v. So the bound on the number of rotations of Q_v gives the following bound on the number of pebbles that can be placed on v: $M_v \leq (d_v + b_v) + b_v$.

In the static case the number of different successors between two ZERO (v) operations is d_v so in the same way we get the bound $M_v \leq d_v + b_v$.

It is easy to construct an example that matches this upper bound. Let $G = (V, E)$ where

$$\begin{aligned} V &= \{v, u_1, \ldots, u_{b_v}, w_1, \ldots, w_{d_v}, w_{d_v+1}, \ldots, w_{d_v+b_v}\} \text{ and} \\ E &= \{(u_i, v) | i \in \{1, \ldots, b_v\}\} \cup \{(v, w_i) | i \in \{1, \ldots, d_v\}\}. \end{aligned}$$

The graph is shown in Figure 3.4.

At the beginning all nodes are black and the ZERO element is at the front of each queue. The sequence of operations which will force the number of pebbles on v to become $d_v + 2b_v$ is the following: ADDPEBBLE on $v, w_1, \ldots, w_{d_v}$, followed by MOVEEDGE$((v, w_{i-1+d_v}), (v, w_{i+d_v}))$ and ADDPEBBLE(w_{i+d_v}) for $i = 1, \ldots, b_v$. The matching example for the static case is constructed in a similar way. □

3.8 A locally adaptive data structure

We will now describe a strategy that is both implementable and locally adaptive. The data structure presented in Section 3.4 and Section 3.5 is *not* locally adaptive, because when redoing the analysis with local degree constraints we get the following bound for the static version of the game:

$$M_v = 1 + 2 \sum_{\{w | (v,w) \in E\}} b_w.$$

The solution to this problem is to incorporate a ZERO element into each of the queues Q_v as in Section 3.7 and then only zero a node when ROTATE returns this element. We now have the following BREAK operation:

procedure BREAK(C_v)
  $r \leftarrow$ the root of C_v
  color all nodes and edges in C_v black
  $w \leftarrow$ ROTATE(Q_r)

```
    if w =ZERO then
        ZERO(r)
        w ← ROTATE(Q_r)
    endif
    if w ≠ZERO then
        color r and (w, r) white
    endif
end.
```

The implementation is similar to the implementation of Section 3.5.

The next theorem shows that the number of pebbles on a node v with this strategy will be bounded by $M_v = 2b_v d_v + 2b_v - 1$, so only nodes with large in-degree (or out-degree) can have many pebbles.

Theorem 5 *The above strategy for player* **D** *achieves* $M_v = 2b_v d_v + 2b_v - 1$.

Proof. The proof follows the same lines as in the proof of Theorem 3. A node v can change its color at most $2b_v - 1$ times between two zeroings. We then have that the number of ADDPEBBLE and MOVEEDGE operations performed on v is at most $2b_v - 1$.

The time interval between two ZERO(v) operations is partitioned into $2b_v$ intervals and that v changes its color only on the boundary between two intervals. In each of the intervals each successor w of v can be zeroed at most once before it will be blocked by a white edge from v.

So when we restrict ourselves to the static case we have that each successor gets zeroed at most $2b_v$ times. Hence the successors of v can place at most $2b_v d_v$ pebbles on v.

Each ADDPEBBLE operation places a pebble on v, so for the static case, the total number of pebbles on v is bounded by $M_v = 2b_v d_v + 2b_v - 1$.

We now only have to show that a MOVEEDGE$((v,u),(v,w))$ operation does not affect this analysis. We have two cases to consider. If u has been zeroed in the last interval then u will either be blocked by a white edge from v or v appears before the ZERO element in Q_u and therefore none of the BREAK operations in MOVEEDGE can result in a ZERO(u). If u has not been zeroed then it is allowed to place a pebble on v in the MOVEEDGE operation. If the BREAK operation forces a ZERO(w) to place a pebble on v then w cannot place a pebble on v during the next time interval. So we can conclude that the analysis still holds.

The matching lower bound is given in the same way as in Theorem 4. □

3.9 A lower bound

In this section we will only consider the static game.

Raman states in [92] that "the dependence on d of M appears to be an artifact of the proof (for the strategy of [33])". Theorem 6 shows that it is not an artifact of the proof, but that the value of M always depends on the value of b and d.

It is shown in [44] that $M \leq b$ holds in the amortized sense, so in that game M does *not* depend of d.

Theorem 6 *For* $b \geq 1$ *and all player* **D** *strategies we have:*

$$M \geq \max\{b+1, \lfloor \alpha + \sqrt{2\alpha - 7/4} - 1/2 \rfloor, \left\lceil \frac{\log \frac{2}{3}d}{\log \log \frac{2}{3}d} - 1 \right\rceil\},$$

where $\alpha = \min\{b, d\}$.

Proof. Immediate consequence of Lemma 3 and 4 and Corollary 1. □

Lemma 3 *For* $b, d \geq 1$ *and all player* **D** *strategies we have* $M \geq b + 1$.

Proof. We will play the game on a convergent tree with l levels where each node has exactly b incoming edges. The player **I** strategy is simple, it just places the pebbles on the root of the tree.

The root has to be zeroed at least once for each group of $M + 1$ AddPebble operations. So at least a fraction $\frac{1}{M+1}$ of the time will be spent on zeroing the root. At most M pebbles can be placed on any internal node before the next Zero operation on that node, because we do not perform AddPebble on internal nodes. So a node on level 1 has to be zeroed at least once for every M Zero operation on the root. Zeroing a node at level 1 takes at least $\frac{1}{M(M+1)}$ of the time, and in general, zeroing a node at level i takes at least $\frac{1}{M^i(M+1)}$ of the time.

Because the number of nodes in each level of the tree increases by a factor b we now have the following constraint on M:

$$\sum_{i=0}^{l} \frac{b^i}{M^i(M+1)} = \frac{1}{M+1} \sum_{i=0}^{l} \left(\frac{b}{M}\right)^i \leq 1.$$

By letting $l \gg M$ we get the desired result $M \geq b + 1$. If $d = 1$, it follows from Theorem 4 that this bound is tight. □

Lemma 4 *For* $b, d \geq 1$ *and all player* **D** *strategies we have:*

$$M \geq \left\lceil \frac{\log \frac{2}{3}d}{\log \log \frac{2}{3}d} - 1 \right\rceil.$$

Proof. We will play the game on the following graph $G = (V, E)$ where $V = \{r, v_1, \ldots, v_d\}$ and $E = \{(r, v_1), \ldots, (r, v_d)\}$. The adversary strategy we will use for player **I** is to cyclically place pebbles on the subset of the v_i's which have not been zeroed yet. The idea is that for each cycle at least a certain fraction of the nodes will not be zeroed.

We start by considering how many nodes cannot be zeroed in one cycle. Let the number of nodes not zeroed at the beginning of the cycle be k. Each time one of the v_i's is zeroed a pebble is placed on r, so out of $M + 1$ zeroings at least one will be a Zero(r). So we have that at least $\lfloor \frac{k}{M+1} \rfloor$ of the nodes are still not zeroed at the end of the cycle. So after i cycles we have that the number of nodes not zeroed is at least (the number of floors is i):

$$\left\lfloor \cdots \left\lfloor \left\lfloor \frac{d}{M+1} \right\rfloor \frac{1}{M+1} \right\rfloor \cdots \frac{1}{M+1} \right\rfloor.$$

By the definition of M, we know that all nodes will be zeroed after $M + 1$ cycles, so we have the following equation (the number of floors is $M + 1$):

$$\left\lfloor \cdots \left\lfloor \left\lfloor \frac{d}{M+1} \right\rfloor \frac{1}{M+1} \right\rfloor \cdots \frac{1}{M+1} \right\rfloor = 0.$$

Lemma 3 gives us that $M \geq 2$. By induction on the number of floors is it easy to show that omitting the floors increases the result at most 3/2. Hence, we have

$$\frac{d}{(M+1)^{M+1}} \leq 3/2.$$

So the minimum solution of M for this inequality will be a lower bound for M. It is easy to see that this minimum solution has to be at least $\frac{\log \frac{2}{3}d}{\log\log \frac{2}{3}d} - 1$. $\square$

Lemma 5 *For all* **D** *strategies where* $b = d$ *we have:*

$$M \geq \lfloor b + \sqrt{2b - 7/4} - 1/2 \rfloor.$$

Proof. For $b = d = 0$ the lemma is trivial. The case $b = d = 1$ is true by Lemma 3. In the following we assume $b = d \geq 2$.

Again, the idea is to use player **I** as an adversary that forces the number of pebbles to become large on at least one node.

The graph we will play the game on is a clique of size $b + 1$. For all nodes u and v both (u, v) and (v, u) will be edges of the graph and all nodes will have in- and out-degree b. Each ZERO operation of player **D** will remove all pebbles from a node of the graph and place one pebble on all the other nodes.

At a time given $P_0, P_1, \ldots, P_b$ will denote the number of pebbles on each of the $b + 1$ nodes — in increasing order, so P_b will denote the number of pebbles on the node with the largest number of pebbles.

Let c_1, c_2 and c_3 denote constants characterising the adversary's strategy. The following invariants will hold from a certain moment of time to be defined later:

$$\begin{array}{ll}
I_1: & i \leq j \Rightarrow P_i \leq P_j, \\
I_2: & P_i \geq i, \\
I_3: & \begin{cases} P_{c_1+c_2-i} \geq c_1 + c_2 - 1 & \text{for } 1 \leq i \leq c_3, \\ P_{c_1+c_2-i} \geq c_1 + c_2 - 2 & \text{for } c_3 < i \leq c_2, \end{cases} \\
I_4: & 1 \leq c_3 \leq c_2 \quad \text{and} \quad c_1 + c_2 \leq b + 1.
\end{array}$$

I_1 is satisfied per definition. I_2 is not satisfied initially but after the first b ZERO's will be satisfied. This is easily seen. The nodes that have not been zeroed will have at least b pebbles and the nodes that have been zeroed can be ordered according to the last time they were zeroed. A node followed by i nodes in this order will have at least i pebbles because each of the following (at least) i zeroings will place a pebble on the node.

We can now satisfy I_3 and I_4 by setting $c_1 = c_2 = c_3 = 1$ so now we have that all the four invariants are satisfied after the first b ZERO operations.

Figure 3.5 illustrates the relationship between c_1, c_2 and c_3 and the number of pebbles on the nodes. The figure only shows the pebbles which are guaranteed to be on the nodes by the invariants. The idea is to build a *block* of nodes which all have the same number of pebbles. These nodes are shown as a dashed box in Figure 3.5. The moves of player **I** and **D** affect this box. A player **I** move will increase the block size whereas a player **D** move will push the block upwards. In the following we will show how large the block can be forced to be.

We will first consider an ADDPEBBLE operation. If $c_3 < c_2$ we know that on node $c_1 + c_2 - c_3 - 1$ (in the current ordering) there are at least $c_1 + c_2 - 2$ pebbles so by placing a pebble on the node $c_1 + c_2 - c_3 - 1$ we can increase c_3 by one and still satisfy the invariants $I_1, \ldots, I_4$. There are three cases to consider. If the node $c_1 + c_2 - c_3 - 1$ already has $c_1 + c_2 - 1$ pebbles we increase c_3 by one and try to place the pebble on another node. If $c_3 = c_2$ and $c_1 + c_2 < b + 1$ we can increase c_2 by one and set $c_3 = 1$ and then try to place the pebble on another node. If we have that $c_2 = c_3$ and

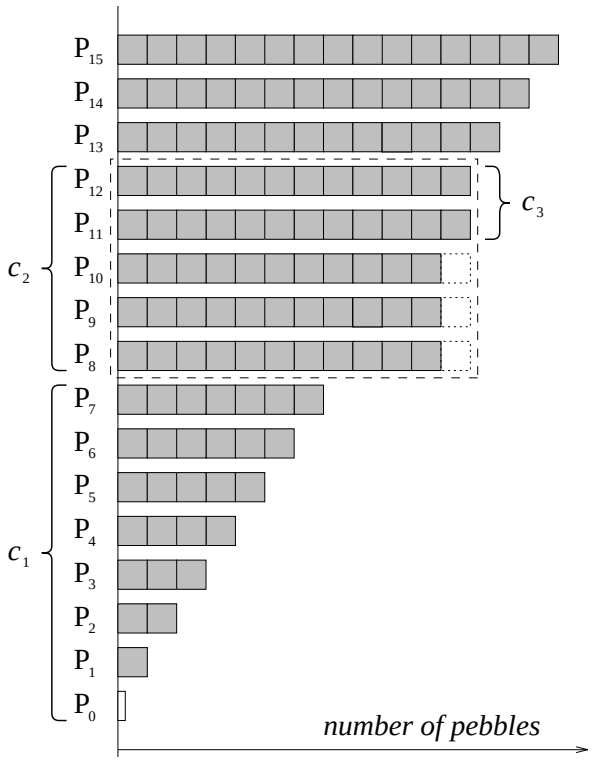

Figure 3.5: The adversary's strategy.

$c_1 + c_2 = b + 1$ we just place the pebble on an arbitrary node — because the block has reached its maximum size.

Whenever player **D** does a ZERO operation we can easily maintain the invariant by just increasing c_1 by one — as long as $c_1 + c_2 < b + 1$. Here we have three cases to consider. Let i denote the number of the node that player **D** zeroes. We will only consider the case when $c_1 \leq i < c_1 + c_2$, the cases $0 \leq i < c_1$ and $c_1 + c_2 \leq i \leq b$ are treated in a similar way. The values of the Ps after the ZERO operation are: $P'_0 = 0, P'_1 = P_0 + 1, \ldots, P'_i = P_{i-1} + 1, P'_{i+1} = P_{i+1} + 1, \ldots, P'_b = P_b + 1$. So because I_2 and I_3 were satisfied before the ZERO operation it follows that when we increase c_1 by one the invariant will still be satisfied after the ZERO operation.

We will now see how large the value of c_2 can become before $c_1 + c_2 = b + 1$. We will allow the last move to be a player **I** move.

We let x denote the maximum value of c_2 when $c_1 + c_2 = b + 1$. At this point we have that $c_1 = b + 1 - x$. Initially we have that $c_1 = 1$. Each ZERO operation can increase c_1 by at most one so the maximum number of ADDPEBBLE operations we can perform is $1 + ((b + 1 - x) - 1) = b + 1 - x$.

It is easily seen that the worst case number of pebbles we have to add to bring c_2 up to x is $1 + \sum_{i=2}^{x-1} (i - 1)$ — because it is enough to have two pebbles in the last column of the block when we are finished.

So the size of $x \geq 0$ is now constrained by:

$$1 + \sum_{i=2}^{x-1} (i - 1) \quad \leq \quad b + 1 - x.$$

Hence, we have $x \geq \lfloor 1/2 + \sqrt{2b - 7/4} \rfloor$. Let $i \in \{0, 1, \ldots, x - 1\}$ denote the number of ZERO operations after the block has reached the top. By placing the pebbles on node $b - 1$ it is easy to

see that the following invariants will be satisfied (I_3 and I_4 will not be satisfied any longer):

$$\begin{aligned} I_5: &\quad P_b \geq b + i, \\ I_6: &\quad P_{b-j} \geq b + i - 1 \qquad \text{for } j = 1, \ldots, x - i - 1. \end{aligned}$$

So after the next $x - 1$ zeroings we see that $P_b \geq b + (x - 1)$ which gives the stated result. □

Corollary 1 *For all* **D** *strategies we have* $M \geq \lfloor \alpha + \sqrt{2\alpha - 7/4} - 1/2 \rfloor$ *where* $\alpha = min\{b, d\}$.

3.10 Conclusion

In the preceding sections we have shown that it is possible to implement partially persistent bounded in-degree (and out-degree) data structures where each access and update step can be done in worst case constant time. This improves the best previously known technique which used amortized constant time per update step.

It is a further consequence of our result that we can support the operation to delete the current version and go back to the previous version in constant time. We just have to store all our modifications of the data structure on a stack so that we can backtrack all our changes of the data structure.

3.11 Open problems

The following list states open problems concerning the dynamic two player game.

- Is it possible to show a general lower bound for M which shows how M depends on b and d?
- Do better (locally adaptive) strategies exist?
- Do implementable strategies for player **D** exist where $M \in O(b + d)$?

Acknowledgements

I want to thank Dany Breslauer, Thore Husfeldt and Lars A. Arge for encouraging discussions. Especially I want to thank Peter G. Binderup for the very encouraging discussions that lead to the proof of Lemma 5 and Erik M. Schmidt for comments on the presentation.

Chapter 4

The Randomized Complexity of Maintaining the Minimum

The Randomized Complexity of Maintaining the Minimum

Gerth Stølting Brodal*
BRICS†, Department of Computer Science
University of Aarhus
Ny Munkegade, 8000 Århus C, Denmark
gerth@brics.dk

Shiva Chaudhuri‡
Max–Planck–Institut für Informatik
Im Stadtwald, 66123 Saarbrücken, Germany
shiva@mpi-sb.mpg.de

Jaikumar Radhakrishnan
Tata Institute of Fundamental Research
Mumbai, India
jaikumar@tcs.tifr.res.in

Abstract

The complexity of maintaining a set under the operations INSERT, DELETE and FINDMIN is considered. In the comparison model it is shown that any randomized algorithm with expected amortized cost t comparisons per INSERT and DELETE has expected cost at least $n/(e2^{2t}) - 1$ comparisons for FINDMIN. If FINDMIN is replaced by a weaker operation, FINDANY, then it is shown that a randomized algorithm with constant expected cost per operation exists; in contrast, it is shown that no deterministic algorithm can have constant cost per operation. Finally, a deterministic algorithm with constant amortized cost per operation for an offline version of the problem is given.

Category: F.2.2
Keywords: set, updates, minimum queries, tradeoff

4.1 Introduction

We consider the complexity of maintaining a set S of elements from a totally ordered universe under the following operations:

INSERT(x): inserts the element x into S,

DELETE(x): removes from S the element x provided it is known where x is stored, and

FINDMIN: returns the minimum element in S without removing it.

We refer to this problem as the INSERT-DELETE-FINDMIN problem. We denote the size of S by n. The analysis is done in the comparison model, *i.e.*, the time required by the algorithm is the

*Supported by the Danish Natural Science Research Council (Grant No. 9400044). This research was done while visiting the Max-Planck Institut für Informatik, Saabrücken, Germany.
†Basic Research in Computer Science, a Centre of the Danish National Research Foundation.
‡This work was partially supported by the EU ESPRIT LTR project No. 20244 (ALCOM IT).

Table 4.1: Worst case asymptotic time bounds for different set implementations.

Implementation	INSERT	DELETE	FINDMIN
Doubly linked list	1	1	n
Heap [109]	$\log n$	$\log n$	1
Search tree [48, 73]	$\log n$	1	1
Priority queue [15, 27, 43]	1	$\log n$	1

number of comparisons it makes. The input is a sequence of operations, given to the algorithm in an online manner, that is, the algorithm must process the current operation before it receives the next operation in the sequence. The *worst case* time for an operation is the maximum, over all such operations in all sequences, of the time taken to process the operation. The *amortized* time of an operation is the maximum, over all sequences, of the total number of comparisons performed, while processing this type of operation in the sequence, divided by the length of the sequence.

Worst case asymptotic time bounds for some existing data structures supporting the above operations are listed in Table 4.1. The table suggests a tradeoff between the worst case times of the two update operations INSERT, DELETE and the query operation FINDMIN. We prove the following lower bound on this tradeoff: any randomized algorithm with expected amortized update time at most t requires expected time $(n/e2^t) - 1$ for FINDMIN. Thus, if the update operations have expected amortized constant cost, FINDMIN requires linear expected time. On the other hand if FINDMIN has constant expected time, then one of the update operations requires logarithmic expected amortized time. This shows that all the data structures in Table 4.1 are optimal in the sense of the tradeoff, and they cannot be improved even by considering amortized cost and allowing randomization.

For each n and t, the lower bound is tight. A simple data structure for the INSERT-DELETE-FINDMIN problem is the following. Assume INSERT and DELETE are allowed to make at most t comparisons. We represent a set by $\lceil n/2^t \rceil$ sorted lists. All lists except for the last contain exactly 2^t elements. The minimum of a set can be found among all the list minima by $\lceil n/2^t \rceil - 1$ comparisons. New elements are added to the last list, requiring at most t comparisons by a binary search. To perform DELETE we replace the element to be deleted by an arbitrary element from the last list. This also requires at most t comparisons.

The above lower bound shows that it is hard to maintain the minimum. Is it any easier to maintain the rank of *some* element, not necessarily the minimum? We consider a weaker problem called INSERT-DELETE-FINDANY, which is defined exactly as the previous problem, except that FINDMIN is replaced by the weaker operation FINDANY that returns an element in S and its rank. FINDANY is not constrained to return the same element each time it is invoked or to return the element with the same rank. The only condition is that the rank returned should be the rank of the element returned. We give a randomized algorithm for the INSERT-DELETE-FINDANY problem with constant expected time per operation. Thus, this problem is strictly easier than INSERT-DELETE-FINDMIN, when randomization is allowed. However, we show that for deterministic algorithms, the two problems are essentially equally hard. We show that any deterministic algorithm with amortized update time at most t requires $n/2^{4t+3} - 1$ comparisons for some FINDANY operation. This lower bound is proved using an explicit adversary argument, similar to the one used by Borodin, Guibas, Lynch and Yao [14]. The adversary strategy is simple, yet surprisingly powerful. The same strategy may be used to obtain the well known $\Omega(n \log n)$ lower bound for sorting. An

explicit adversary for sorting has previously been given by Atallah and Kosaraju [8].

The previous results show that maintaining any kind of rank information online is hard. However, if the sequence of instructions to be processed is known in advance, then one can do better. We give a deterministic algorithm for the offline INSERT-DELETE-FINDMIN problem which has an amortized cost per operation of at most three comparisons.

Our proofs use various averaging arguments which are used to derive general combinatorial properties of trees. These are presented in Section 4.2.2.

4.2 Preliminaries

4.2.1 Definitions and notation

For a rooted tree T, let $\textsc{leaves}(T)$ be the set of leaves of T. For a vertex, v in T, define $\textsc{deg}(v)$ to be the number of children of v. Define, for $\ell \in \textsc{leaves}(T)$, $\textsc{depth}(\ell)$ to be the distance of ℓ from the root and $\textsc{path}(\ell)$ to be the set of vertices on the path from the root to ℓ, not including ℓ.

For a random variable X, let support$[X]$ be the set of values that X assumes with non-zero probability. For any non-negative real valued function f, defined on support$[X]$, we define the arithmetic mean and geometric mean of f by

$$\begin{aligned} \mathop{\mathbf{E}}_{X}[f(X)] &= \sum_{x \in \text{support}[X]} \Pr[X = x] f(x), \quad \text{and} \\ \mathop{\mathbf{GM}}_{X}[f(X)] &= \prod_{x \in \text{support}[X]} f(x)^{\Pr[X=x]}. \end{aligned}$$

We will also use the notation $\mathbf{E}$ and $\mathbf{GM}$ to denote the arithmetic and geometric means of a set of values as follows: for a set R, and any non-negative real valued function f, defined on R, define

$$\mathop{\mathbf{E}}_{r \in R}[f(r)] = \frac{1}{|R|} \sum_{r \in R} f(r), \quad \text{and} \quad \mathop{\mathbf{GM}}_{r \in R}[f(r)] = \prod_{r \in R} f(r)^{1/|R|}.$$

It can be shown (see [62]) that the geometric mean is at most the arithmetic mean.

4.2.2 Some useful lemmas

Let T be the infinite complete binary tree. Suppose each element of $[n] = \{1, \ldots, n\}$ is assigned to a node of the tree (more than one element may be assigned to the same node). That is, we have a function $f : [n] \to V(T)$. For $v \in V(T)$, define $\textsc{wt}_f(v) = |\{i \in [n] : f(i) = v\}|$, $d_f = \mathbf{E}_{i \in [n]}[\textsc{depth}(f(i))]$, $D_f = \max\{\textsc{depth}(f(i)) : i \in [n]\}$ and $m_f = \max\{\textsc{wt}_f(v) : v \in V(T)\}$.

Lemma 6 *For every assignment $f : [n] \to V(T)$, the maximum number of elements on a path starting at the root of T is at least $n2^{-d_f}$.*

Proof. Let P be a random infinite path starting from the root. Then, for $i \in [n]$, $\Pr[f(i) \in P] = 2^{-\textsc{depth}(f(i))}$. Then the expected number of elements of $[n]$ assigned to P is

$$\begin{aligned} \sum_{i=1}^{n} 2^{-\textsc{depth}(f(i))} &= n \mathop{\mathbf{E}}_{i \in [n]}[2^{-\textsc{depth}(f(i))}] \geq n \mathop{\mathbf{GM}}_{i \in [n]}[2^{-\textsc{depth}(f(i))}] \\ &= n2^{-\mathbf{E}_{i \in [n]}[\textsc{depth}(f(i))]} = n2^{-d_f}. \end{aligned}$$

Since the maximum is at least the expected value, the lemma follows. □

Lemma 7 *For every assignment $f : [n] \to V(T)$, $m_f \geq n/2^{d_f+3}$.*

Proof. Let $H = \{h : m_h = m_f\}$. Let h be the assignment in H with minimum average depth d_h (the minimum exists). Let $m = m_h = m_f$, and $D = D_h$. We claim that

$$\text{WT}_h(v) = m, \text{ for each } v \in V(T) \text{ with } \text{DEPTH}(v) < D. \tag{4.1}$$

For suppose there is a vertex v with $\text{DEPTH}(v) < D$ and $\text{WT}(v) < m$ (*i.e.*, $\text{WT}(v) \leq m-1$). First, consider the case when some node w at depth D has m elements assigned to it. Consider the assignment h' given by

$$h'(i) \stackrel{\text{def}}{=} \begin{cases} w & \text{if } h(i) = v, \\ v & \text{if } h(i) = w, \\ h(i) & \text{otherwise.} \end{cases}$$

Then $h' \in H$ and $d_{h'} < d_h$, contradicting the choice of h. Next, suppose that every node at depth D has less than m elements assigned to it. Now, there exists $i \in [n]$ such that $\text{DEPTH}(h(i)) = D$. Let h' be the assignment that is identical to h everywhere except at i, and for i, $h'(i) = v$. Then, $h' \in H$ and $d_{h'} < d_h$, again contradicting the choice of h. Thus (4.1) holds.

The number of elements assigned to nodes at depth at most $D-1$ is $m(2^D - 1)$, and the average depth of these elements is

$$\frac{1}{m(2^D-1)} \sum_{i=0}^{D-1} mi2^i = \frac{(D-2)2^D + 2}{2^D - 1} \geq D - 2.$$

Since all other elements are at depth D, we have $d_h \geq D - 2$. The total number of nodes in the tree with depth at most D is $2^{D+1} - 1$. Hence, we have

$$m_f = m \geq \frac{n}{2^{D+1}-1} \geq \frac{n}{2^{d_h+3}-1} \geq \frac{n}{2^{d_f+3}-1}. \quad \square$$

For a rooted tree T, let $W_\ell = \prod_{v \in \text{PATH}(\ell)} \text{DEG}(v)$. Then, it can be shown by induction on the height of tree that $\sum_{\ell \in \text{LEAVES}(T)} 1/W_\ell = 1$. The following lemma is implicit in the work of McDiarmid [75].

Lemma 8 *For a rooted tree T with m leaves, $\underset{\ell \in \text{LEAVES}(T)}{\mathbf{GM}}[W_\ell] \geq m$.*

Proof. Since the geometric mean is at most the arithmetic mean, we have

$$\underset{\ell}{\mathbf{GM}}[\frac{1}{W_\ell}] \;\leq\; \underset{\ell}{\mathbf{E}}[\frac{1}{W_\ell}] = \frac{1}{m}\sum_{\ell} \frac{1}{W_\ell} \;=\; \frac{1}{m}.$$

Now,

$$\underset{\ell}{\mathbf{GM}}[W_\ell] \;=\; \frac{1}{\underset{\ell}{\mathbf{GM}}[1/W_\ell]} \;\geq\; m. \quad \square$$

4.3 Deterministic offline algorithm

We now consider an offline version of the INSERT-DELETE-FINDMIN problem. The sequence of operations to be performed is given in advance, however, the ordering of the set elements is unknown. The ith operation is performed at time i. We assume that an element is inserted and deleted at most once. If an element is inserted and deleted more than once, it can be treated as a distinct element each time it is inserted.

From the given operation sequence, the offline algorithm can compute, for each element x, the time, $t(x)$, at which x is deleted from the data structure ($t(x)$ is ∞ if x is never deleted).

The data structure maintained by the offline algorithm is a sorted (in increasing order) list $L = (x_1, \ldots, x_k)$ of the set elements that can become minimum elements in the data structure. The list satisfies that $t(x_i) < t(x_j)$ for $i < j$, because otherwise x_j could never become a minimum element.

FINDMIN returns the first element in L and DELETE(x) deletes x from L, if L contains x, *i.e.*, $x = x_1$. To process INSERT(x), the algorithm computes two values, ℓ and r, where $r = \min\{i : t(x_i) > t(x)\}$ and $\ell = \max\{i : x_i < x\}$. Notice that once x is in the data structure, none of $x_{\ell+1}, \ldots, x_{r-1}$ can ever be the minimum element. Hence, all these elements are deleted and x is inserted into the list between x_ℓ and x_r. No comparisons are required among the elements to find r, because r can be computed by a search for $t(x)$ in $(t(x_1), \ldots, t(x_k))$. Thus, INSERT(x) may be implemented as follows: starting at x_r, step backwards through the list, deleting elements until the first element smaller than x is encountered.

The number of comparisons for an insertion is two plus the number of elements deleted from L. By letting the potential of L be $|L|$ the amortized cost of INSERT is $|L'| - |L| + \#$ of elements removed during INSERT+2 which is at most 3 because the number of elements removed is at most $|L| - |L'| + 1$. DELETE only decreases the potential, and the initial potential is zero. It follows that

Theorem 7 *For the offline* INSERT-DELETE-FINDMIN *problem the amortized cost of* INSERT *is three comparisons. No comparisons are required for* DELETE *and* FINDMIN.

4.4 Deterministic lower bound for FINDANY

In this section we show that it is difficult for a deterministic algorithm to maintain any rank information at all. We prove

Theorem 8 *Let $\mathcal{A}$ be a deterministic algorithm for the* INSERT-DELETE-FINDANY *problem with amortized time at most $t = t(n)$ per update. Then, there exists an input for which $\mathcal{A}$ takes at least $n/2^{4t+3} - 1$ comparisons to process one* FINDANY.

The Adversary. We describe an adversary strategy for responding to the comparisons.

The adversary maintains an infinite binary tree and the elements currently in the data structure are distributed among the nodes of this tree. New elements inserted into the data structure are placed at the root. For $x \in S$ let $v(x)$ denote the node of the tree at which x is. The adversary maintains two invariants. For any distribution of the elements among the nodes of the infinite tree, define the *occupancy tree* to be the finite tree given by the union of the paths from every non-empty node to the root. The invariants are

(A) If neither of $v(x)$ or $v(y)$ is a descendant of the other then $x < y$ is consistent with the responses given so far if $v(x)$ appears before $v(y)$ in an inorder traversal of the occupancy tree, and

(B) If $v(x) = v(y)$ or $v(x)$ is a descendant of $v(y)$, the responses given so far yield no information on the order of x and y. More precisely, in this case, x and y are incomparable in the partial order induced on the elements by the responses so far.

The comparisons made by any algorithm can be classified into three types, and the adversary responds to each type of the comparison as described below. Let the elements compared be x and y.

- $v(x) = v(y)$: Then x is moved to the left child of $v(x)$ and y to the right child and the adversary answers $x < y$.
- $v(x)$ is a descendant of $v(y)$: Then y is moved to the unique child of $v(y)$ that is not an ancestor of $v(x)$. If this child is a left child then the adversary answers $y < x$ and if it is a right child then the adversary answers $x < y$.
- $v(x) \neq v(y)$ and neither is a descendant of the other: If $v(x)$ is visited before $v(y)$ in the inorder traversal of the occupancy tree, the adversary answers $x < y$ and otherwise the adversary answers $y < x$.

The key observation is that each comparison pushes two elements down one level each, in the worst case.

Maintaining ranks. We now give a proof of Theorem 8.

Consider the behavior of the algorithm when responses to its comparisons are given according to the adversary strategy above. Define the sequences $S_1 \ldots S_{n+1}$ as follows.

$$S_1 = \textsc{Insert}(a_1) \ldots \textsc{Insert}(a_n) \textsc{FindAny}.$$

Let b_1 be the element returned in response to the FindAny instruction in S_1. For $i = 2, 3, \ldots n$, define

$$S_i = \textsc{Insert}(a_1) \ldots \textsc{Insert}(a_n) \textsc{Delete}(b_1) \ldots \textsc{Delete}(b_{i-1}) \textsc{FindAny}$$

and let b_i be the element returned in response to the FindAny instruction in S_i. Finally, let

$$S_{n+1} = \textsc{Insert}(a_1) \ldots \textsc{Insert}(a_n) \textsc{Delete}(b_1) \ldots \textsc{Delete}(b_n).$$

For $1 \leq i \leq n$, b_i is well defined and for $1 \leq i < j \leq n$, $b_i \neq b_j$. The latter point follows from the fact that at the time b_i is returned by a FindAny, $b_1, \ldots, b_{i-1}$ have already been deleted from the data structure.

Let T be the infinite binary tree maintained by the adversary. Then the sequence S_{n+1} defines a function $f : [n] \to V(T)$, given by $f(i) = v$ if b_i is in node v just before the Delete(b_i) instruction during the processing of S_{n+1}. Since the amortized cost of an update is at most t, the total number of comparisons performed while processing S_{n+1} is at most $2tn$. A comparison pushes at most two elements down one level each. Then, writing d_i for the distance of $f(i)$ from the root, we have $\sum_{i=1}^{n} d_i \leq 4tn$. By Lemma 7 we know that there is a set $R \subseteq [n]$ with at least $n/2^{4t+3}$ elements and a vertex v of T such that for each $i \in R$, $f(b_i) = v$.

Let $j = \min R$. Then, while processing S_j, just before the FindAny instruction, each element b_i, $i \in R$ is in some node on the path from the root to $f(i) = v$. Since the element returned by the FindAny is b_j, it must be the case that after the comparisons for the FindAny are performed, b_j is the only element on the path from the root to the vertex in which b_j is. This is because invariant (B) implies that any other element that is on this path is incomparable with b_j. Hence, these comparisons move all the elements b_i, $i \in R \backslash j$, out of the path from the root to $f(j)$. A comparison can move at most one element out of this path, hence, the number of comparisons performed is at least $|R| - 1$, which proves the theorem.

4.4.1 Sorting

The same adversary can be used to give a lower bound for sorting. We note that this argument is fundamentally different from the usual information theoretic argument in that it gives an explicit adversary against which sorting is hard.

Consider an algorithm that sorts a set S, of n elements. The same adversary strategy is used to respond to comparisons. Then, invariant (B) implies that at the end of the algorithm, each element in the tree must be in a node by itself. Let the function $f : S \to V(T)$ indicate the node where each element is at the end of the algorithm, where T is the infinite binary tree maintained by the adversary. Then, f assigns at most one element to each path starting at the root of T. By Lemma 6 we have $1 \geq n2^{-d}$, where d is the average distance of an element from the root. It follows that the sum of the distances from the root to the elements in this tree is at least $n \log n$, and this is equal to the sum of the number of levels each element has been pushed down. Since each comparison contributes at most two to this sum, the number of comparisons made is at least $(n \log n)/2$.

4.5 Randomized algorithm for FindAny

We present a randomized algorithm supporting Insert, Delete and FindAny using, on an average, a constant number of comparisons per operation.

4.5.1 The algorithm

The algorithm maintains three variables: S, z and *rank*. S is the set of elements currently in the data structure, z is an element in S, and *rank* is the rank of z in S. Initially, S is the empty set, and z and *rank* are null. The algorithm responds to instructions as follows.

Insert(x): Set $S \leftarrow S \cup \{x\}$. With probability $1/|S|$ we set z to x and let *rank* be the rank of z in S, that is, one plus the number of elements in S smaller than z. In the other case, that is with probability $1 - 1/|S|$, we retain the old value of z; that is, we compare z and x and update *rank* if necessary. In particular, if the set was empty before the instruction, then z is assigned x and *rank* is set to 1.

Delete(x): Set $S \leftarrow S - \{x\}$. If S is empty then set z and *rank* to null and return.

Otherwise (*i.e.*, if $S \neq \emptyset$), if $x \equiv z$ then get the new value of z by picking an element of S randomly; set *rank* to be the rank of z in S. On the other hand, if x is different from z, then decrement *rank* by one if x was smaller than z.

FindAny: Return z and *rank*.

4.5.2 Analysis

Claim 9 *The expected number of comparisons made by the algorithm for a fixed instruction in any sequence of instructions is constant.*

Proof. FindAny takes no comparisons. Consider an Insert instruction. Suppose the number of elements in S just before the instruction was s. Then, the expected number of comparisons made by the algorithm is $s \cdot (1/(s+1)) + 1 \cdot (s/(s+1)) < 2$.

We now consider the expected number of comparisons performed for a Delete instruction. Fix a sequence of instructions. Let S_i and z_i be the values of S and z just before the ith instruction. Note

that S_i depends only on the sequence of instructions and not on the coin tosses of the algorithm; on the other hand, z_i might vary depending on the coin tosses of the algorithm. We first show that the following invariant holds for all i:

$$|S_i| \neq \emptyset \implies \Pr[z_i = x] = \frac{1}{|S_i|} \quad \text{for all } x \in S_i. \tag{4.2}$$

We use induction on i. For $i = 1$, S_i is empty and the claim holds trivially. Assume that the claim holds for $i = \ell$; we shall show that then it holds for $i = \ell + 1$. If the ℓth instruction is a FINDANY, then S and z are not disturbed and the claim continues to hold.

Suppose the ℓth instruction is an INSERT. For $x \in S_\ell$, we can have $z_{\ell+1} = x$ only if $z_\ell = x$ and we retain the old value of z after the INSERT instruction. The probability that we retain the old value of z is $|S_\ell|/(|S_\ell| + 1)$. Thus, using the induction hypothesis, we have for all $x \in S_\ell$

$$\Pr[z_{\ell+1} = x] = \Pr[z_\ell = x] \cdot \Pr[z_{\ell+1} = z_\ell] = \frac{1}{|S_\ell|} \cdot \frac{|S_\ell|}{|S_\ell| + 1} = \frac{1}{|S_\ell| + 1}.$$

Also, the newly inserted element is made $z_{\ell+1}$ with probability $\frac{1}{|S_\ell|+1}$. Since $|S_{\ell+1}| = |S_\ell| + 1$, (4.2) holds for $i = \ell + 1$.

Next, suppose the ℓth instruction is a DELETE(x). If the set becomes empty after this instruction, there is nothing to prove. Otherwise, for all $y \in S_{\ell+1}$,

$$\begin{aligned}
&\Pr[z_{\ell+1} = y] \\
&\quad = \Pr[z_\ell = x \ \&\ z_{\ell+1} = y] + \Pr[z_\ell \neq x \ \&\ z_{\ell+1} = y] \\
&\quad = \Pr[z_\ell = x] \cdot \Pr[z_{\ell+1} = y \mid z_\ell = x] + \Pr[z_\ell \neq x] \cdot \Pr[z_\ell = y \mid z_\ell \neq x].
\end{aligned}$$

By the induction hypothesis we have $\Pr[z_\ell = x] = 1/|S_\ell|$. Also, if $z_\ell = x$ then we pick $z_{\ell+1}$ randomly from $S_{\ell+1}$; hence $\Pr[z_{\ell+1} = y \mid z_\ell = x] = 1/|S_{\ell+1}|$. For the second term, by the induction hypothesis we have $\Pr[z_\ell \neq x] = 1 - 1/|S_\ell|$ and $\Pr[z_\ell = y \mid z_\ell \neq x] = 1/(|S_\ell| - 1) = 1/|S_{\ell+1}|$ (because $|S_{\ell+1}| = |S_\ell| - 1$). By substituting these, we obtain

$$\begin{aligned}
\Pr[z_{\ell+1} = y] &= \frac{1}{|S_\ell|} \cdot \frac{1}{|S_{\ell+1}|} + (1 - \frac{1}{|S_\ell|}) \cdot \frac{1}{|S_{\ell+1}|} \\
&= \frac{1}{|S_{\ell+1}|}.
\end{aligned}$$

Thus, (4.2) holds for $i = \ell + 1$. This completes the induction.

Now, suppose the ith instruction is DELETE(x). Then, the probability that $z_i = x$ is precisely $1/|S_i|$. Thus, the expected number of comparisons performed by the algorithm is

$$(|S_i| - 2) \cdot \frac{1}{|S_i|} < 1. \quad \square$$

4.6 Randomized lower bounds for FINDMIN

One may view the problem of maintaining the minimum as a game between two players: the algorithm and the adversary. The adversary gives instructions and supplies answers for the comparisons made by the algorithm. The objective of the algorithm is to respond to the instructions by making

as few comparisons as possible, whereas the objective of the adversary is to force the algorithm to use a large number of comparisons.

Similarly, if randomization is permitted while maintaining the minimum, one may consider the randomized variants of this game. We have two cases based on whether or not the adversary is adaptive. An adaptive adversary constructs the input as the game progresses; its actions depend on the moves the algorithm has made so far. On the other hand, a non-adaptive adversary fixes the instruction sequence and the ordering of the elements before the game begins. The input it constructs can depend on the algorithm's strategy but not on its coin toss sequence.

It can be shown that against the adaptive adversary randomization does not help. In fact, if there is a randomized strategy for the algorithm against an adaptive adversary then there is a deterministic strategy against the adversary. Thus, the complexity of maintaining the minimum in this case is the same as in the deterministic case. In this section, we show lower bounds with a non-adaptive adversary.

The input to the algorithm is specified by fixing a sequence of INSERT, DELETE and FINDMIN instructions, and an ordering for the set $\{a_1, a_2, \ldots, a_n\}$, based on which the comparisons of the algorithm are answered.

Distributions. We will use two distributions on inputs. For the first distribution, we construct a random input I by first picking a random permutation σ of $[n]$; we associate with σ the sequence of instructions

$$\textsc{Insert}(a_1), \ldots, \textsc{Insert}(a_n), \textsc{Delete}(a_{\sigma(1)}), \textsc{Delete}(a_{\sigma(2)}), \ldots, \textsc{Delete}(a_{\sigma(n)}), \tag{4.3}$$

and the ordering

$$a_{\sigma(1)} < a_{\sigma(2)} < \ldots < a_{\sigma(n)}. \tag{4.4}$$

For the second distribution, we construct the random input J by picking $i \in [n]$ at random and a random permutation σ of $[n]$; the instruction sequence associated with i and σ is

$$\textsc{Insert}(a_1), \ldots, \textsc{Insert}(a_n), \textsc{Delete}(a_{\sigma(1)}), \ldots, \textsc{Delete}(a_{\sigma(i-1)}), \textsc{FindMin}, \tag{4.5}$$

and the ordering is given, as before, by (4.4).

For an algorithm $\mathcal{A}$ and an input I, let $C_U(\mathcal{A}, I)$ be the number of comparisons made by the algorithm in response to the update instructions (INSERT and DELETE) in I; let $C_F(\mathcal{A}, I)$ be the number of comparisons made by the algorithm while responding to the FINDMIN instructions.

Theorem 10 *Let $\mathcal{A}$ be a deterministic algorithm for maintaining the minimum. Suppose*

$$\mathop{\mathbf{E}}_{I}[C_U(\mathcal{A}, I)] \leq tn. \tag{4.6}$$

Then

$$\mathop{\mathbf{GM}}_{J}[C_F(\mathcal{A}, J) + 1] \geq \frac{n}{e2^t}.$$

Before we discuss the proof of this result, we derive from it the lower bounds on the randomized and average case complexities of maintaining the minimum. Yao showed that a randomized algorithm can be viewed as a random variable assuming values in some set of deterministic algorithms according to some probability distribution over the set [110]. The randomized lower bound follows from this fact and Theorem 10.

Corollary 2 (Randomized complexity) *Let $\mathcal{R}$ be a randomized algorithm for* INSERT-DELETE-FINDMIN *with expected amortized time per update at most $t = t(n)$. Then the expected time for* FINDMIN *is at least $n/(e2^{2t}) - 1$.*

Proof. We view $\mathcal{R}$ as a random variable taking values in a set of deterministic algorithms with some distribution. For every deterministic algorithm $\mathcal{A}$ in this set, let

$$t(\mathcal{A}) \stackrel{\text{def}}{=} \mathop{\mathbf{E}}_{I}[C_U(\mathcal{A}, I)]/n.$$

Then by Theorem 10 we have $\mathop{\mathbf{GM}}_{J}[C_F(\mathcal{A}, J) + 1] \geq \left(\frac{n}{e}\right) \cdot 2^{-t(\mathcal{A})}$. Hence,

$$\mathop{\mathbf{GM}}_{\mathcal{R}}[\mathop{\mathbf{GM}}_{J}[C_F(\mathcal{R}, J) + 1] \;\geq\; \mathop{\mathbf{GM}}_{\mathcal{R}}[\left(\frac{n}{e}\right) \cdot 2^{-t(\mathcal{R})}] \;=\; \left(\frac{n}{e}\right) \cdot 2^{-\mathop{\mathbf{E}}_{\mathcal{R}}[t(\mathcal{R})]}.$$

Since the expected amortized time per update is at most t, we have $\mathbf{E}_{\mathcal{R}}[t(\mathcal{R})] \leq 2t$. Hence,

$$\mathop{\mathbf{E}}_{\mathcal{R},J}[C_F(\mathcal{R}, J)] + 1 = \mathop{\mathbf{E}}_{\mathcal{R},J}[C_F(\mathcal{R}, J) + 1] \geq \mathop{\mathbf{GM}}_{\mathcal{R},J}[C_F(\mathcal{R}, J) + 1] \geq \frac{n}{e2^{2t}}.$$

Thus, there exists an instance of J with instructions of the form (4.5), for which the expected number of comparisons performed by $\mathcal{R}$ in response to the last FINDMIN instruction is at least $n/(e2^{2t}) - 1$. □

The average case lower bound follows from the arithmetic-geometric mean inequality and Theorem 10.

Corollary 3 (Average case complexity) *Let $\mathcal{A}$ be a deterministic algorithm for* INSERT-DELETE-FINDMIN *with amortized time per update at most $t = t(n)$. Then the expected time to find the minimum for inputs with distribution J is at least $n/(e2^{2t}) - 1$.*

Proof. $\mathcal{A}$ takes amortized time at most t per update. Therefore,

$$\mathop{\mathbf{E}}_{I}[C_U(\mathcal{A}, I)] \leq 2tn.$$

Then, by Theorem 10 we have

$$\mathop{\mathbf{E}}_{J}[C_F(\mathcal{A}, J)] + 1 = \mathop{\mathbf{E}}_{J}[C_F(\mathcal{A}, J) + 1] \geq \mathop{\mathbf{GM}}_{J}[C_F(\mathcal{A}, J) + 1] \geq \frac{n}{e2^{2t}}. \quad \square$$

4.6.1 Proof of Theorem 10

The Decision Tree representation. Consider the set of sequences in support$[I]$. The actions of a deterministic algorithm on this set of sequences can be represented by a decision tree with *comparison* nodes and *deletion* nodes. (Normally a decision tree representing an algorithm would also have *insertion* nodes, but since, in support$[I]$, the elements are always inserted in the same order, we may omit them.) Each comparison node is labeled by a comparison of the form $a_i : a_j$, and has two children, corresponding to the two outcomes $a_i > a_j$ and $a_i \leq a_j$. Each deletion

node has a certain number of children and each edge, x, to a child, is labeled by some element a_x, denoting that element a_x is deleted by this delete instruction.

For a sequence corresponding to some permutation σ, the algorithm behaves as follows. The first instruction it must process is INSERT(a_1). The root of the tree is labeled by the first comparison that the algorithm makes in order to process this instruction. Depending on the outcome of this comparison, the algorithm makes one of two comparisons, and these label the two children of the root. Thus, the processing of the first instruction can be viewed as following a path down the tree. Depending on the outcomes of the comparisons made to process the first instruction, the algorithm is currently at some vertex in the tree, and this vertex is labeled by the first comparison that the algorithm makes in order to process the second instruction. In this way, the processing of all the insert instructions corresponds to following a path consisting of comparison nodes down the tree. When the last insert instruction has been processed, the algorithm is at a delete node corresponding to the first delete instruction. Depending on the sequence, some element, $a_{\sigma(1)}$ is deleted. The algorithm follows the edge labeled by $a_{\sigma(1)}$ and the next vertex is labeled by the first comparison that the algorithm makes in order to process the next delete instruction. In this manner, each sequence determines a path down the tree, terminating at a leaf.

We make two simple observations. First, since, in different sequences, the elements are deleted in different orders, each sequence reaches a distinct leaf of the tree. Hence the number of leaves is exactly $n!$. Second, consider the ordering information available to the algorithm when it reaches a delete node v. This information consists of the outcomes of all the comparisons on the comparison nodes on the path from the root to v. This information can be represented as a poset, P_v, on the elements not deleted yet. For every sequence that causes the algorithm to reach v, the algorithm has obtained only the information in P_v. If a sequence corresponding to some permutation σ takes the algorithm to the delete node v, where a_i is deleted, then a_i is a minimal element in P_v, since, in σ, a_i is the minimum among the remaining elements. Hence each of the elements labeling an edge from v to a child is a minimal element of P_v. If this DELETE instruction was replaced by a FINDMIN, then the comparisons done by the FINDMIN would have to find the minimum among these minimal elements. A comparison between any two poset elements can cause at most one of these minimal elements to become non-minimal. Hence, the FINDMIN instruction would cost the algorithm DEG$(v) - 1$ comparisons.

The proof. Let T be the decision tree corresponding to the deterministic algorithm $\mathcal{A}$. Set $m = n!$. For $\ell \in$ LEAVES(T), let D_ℓ be the set of delete nodes on the path from the root to ℓ, and C_ℓ be the set of comparison nodes on the path from the root to ℓ.

Each input specified by a permutation σ and a value $i \in [n]$, in support$[J]$ causes the algorithm to follow a path in T upto some delete node, v, where, instead of a DELETE, the sequence issues a FINDMIN instruction. As argued previously, the number of comparisons made to process this FINDMIN is at least DEG$(v) - 1$. There are exactly n delete nodes on any path from the root to a leaf and different inputs cause the algorithm to arrive at a different delete nodes. Hence

$$\mathop{\mathbf{GM}}_{J}[C_F(\mathcal{A}, J) + 1] \geq \prod_{\ell \in \text{LEAVES}(T)} \prod_{v \in D_\ell} (\text{DEG}(v))^{1/nm}. \tag{4.7}$$

Since T has m leaves, we have using Lemma 8 that

$$\begin{aligned} m &\leq \mathop{\mathbf{GM}}_{\ell \in \text{LEAVES}(T)} [\prod_{v \in \text{PATH}(\ell)} \text{DEG}(v)] \\ &= \mathop{\mathbf{GM}}_{\ell \in \text{LEAVES}(T)} [\prod_{v \in C_\ell} \text{DEG}(v)] \cdot \mathop{\mathbf{GM}}_{\ell \in \text{LEAVES}(T)} [\prod_{v \in D_\ell} \text{DEG}(v)]. \end{aligned} \tag{4.8}$$

Consider the first term on the right. Since every comparison node v has arity at most two, we have $\prod_{v \in C_\ell} \textsc{deg}(v) = 2^{|C_\ell|}$. Also, by the assumption (4.6) of our theorem,

$$\mathop{\mathbf{E}}_{\ell \in \textsc{leaves}(T)}[|C_\ell|] = \mathop{\mathbf{E}}_{I}[C_U(A, I)] \leq tn.$$

Thus

$$\mathop{\mathbf{GM}}_{\ell \in \textsc{leaves}(T)}[\prod_{v \in C_\ell} \textsc{deg}(v)] \;\leq\; \mathop{\mathbf{GM}}_{\ell \in \textsc{leaves}(T)}[2^{|C_\ell|}] \;\leq\; 2^{\mathbf{E}_\ell[|C_\ell|]} \;\leq\; 2^{tn}.$$

From this and (4.8), we have

$$\mathop{\mathbf{GM}}_{\ell \in \textsc{leaves}(T)}[\prod_{v \in D_\ell} \textsc{deg}(v)] \geq m 2^{-tn}.$$

Then using (4.7) and the inequality $n! \geq (n/e)^n$, we get

$$\begin{aligned} \mathop{\mathbf{GM}}_{J}[C_F(\mathcal{A}, J) + 1] \;&\geq\; \prod_{\ell \in \textsc{leaves}(T)} \prod_{v \in D_\ell} (\textsc{deg}(v))^{1/nm} \\ &=\; (\mathop{\mathbf{GM}}_{\ell \in \textsc{leaves}(T)}[\prod_{v \in D_\ell} \textsc{deg}(v)])^{1/n} \;\geq\; \frac{n}{e 2^t}. \quad \square \end{aligned}$$

Remark. One may also consider the problem of maintaining the minimum when the algorithm is allowed to use an operator that enables it to compute the minimum of some m values in one step. The case $m = 2$ corresponds to the binary comparisons model. Since an m-ary minimum operation can be simulated by $m - 1$ binary minimum operations, the above proof yields a lower bound of

$$\frac{1}{m-1}\left[\frac{n}{e 2^{2t(m-1)}} - 1\right]$$

on the cost of FindMin, if the amortized cost of Insert and Delete is at most t. However, by modifying our proof one can improve this lower bound to

$$\frac{1}{m-1}\left[\frac{n}{e m^{2t}} - 1\right].$$

Acknowledgment.

We thank the referee for his suggestions.

Chapter 5

Fast Meldable Priority Queues

Fast Meldable Priority Queues*

Gerth Stølting Brodal
BRICS†, Department of Computer Science, University of Aarhus
Ny Munkegade, DK-8000 Århus C, Denmark
gerth@brics.dk

Abstract

We present priority queues that support the operations FindMin, Insert, MakeQueue and Meld in worst case time $O(1)$ and Delete and DeleteMin in worst case time $O(\log n)$. They can be implemented on the pointer machine and require linear space. The time bounds are optimal for all implementations where Meld takes worst case time $o(n)$.

To our knowledge this is the first priority queue implementation that supports Meld in worst case constant time and DeleteMin in logarithmic time.

Category: E.1
Keywords: priority queues, meld, worst case complexity

Introduction

We consider the problem of implementing meldable priority queues. The operations that should be supported are:

MakeQueue Creates a new empty priority queue.

FindMin(Q) Returns the minimum element contained in priority queue 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 one priority queue and returns the new priority queue.

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

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

The implementation of priority queues is a classical problem in data structures. A few references are [43, 47, 52, 53, 63, 108, 109].

In the amortized sense, [101], the best performance is achieved by binomial heaps [108]. They support Delete and DeleteMin in amortized time $O(\log n)$ and all other operations in amortized constant time. If we want to perform Insert in worst case constant time a few efficient data

*This work was partially supported by the ESPRIT II Basic Research Actions Program of the EC under contract no. 7141 (project ALCOM II) and by the Danish Natural Science Research Council (Grant No. 9400044).
†BRICS (Basic Research in Computer Science), a Centre of the Danish National Research Foundation.

structures exist. The priority queue of van Leeuwen [107], the implicit priority queues of Carlsson *et al.* [27] and the relaxed heaps of Driscoll *et al.* [43], but neither of these support MELD efficiently. However, the last two do support MAKEQUEUE, FINDMIN and INSERT in worst case constant time and DELETE and DELETEMIN in worst case time $O(\log n)$.

Our implementation beats the above by supporting MAKEQUEUE, FINDMIN, INSERT and MELD in worst case time $O(1)$ and DELETE and DELETEMIN in worst case time $O(\log n)$. The computational model is the pointer machine and the space requirement is linear in the number of elements contained in the priority queues.

We assume that the priority queues contain elements from a totally ordered universe. The only allowed operation on the elements is the comparisons of two elements. We assume that comparisons can be performed in worst case constant time. For simplicity we assume that all priority queues are nonempty. For a given operation we let n denote the size of the priority queue of maximum size involved in the operation.

In Section 5.1 we describe the data structure and in Section 5.2 we show how to implement the operations. In Section 5.3 we show that our construction is optimal. Section 5.4 contains some final remarks.

5.1 The Data Structure

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 [108] and Fibonacci heaps [53] where the representation is a forest of heap ordered trees.

With each node we associate a rank and we partition the children of a node into two types, type I and type II. The heap ordered tree must satisfy the following structural constraints.

a) A node has at most one child of type I. This child may be of arbitrary rank.

b) The children of type II of a node of rank r have all rank less than r.

c) For a fixed node of rank r, let n_i denote the number of children of type II that have rank i. We maintain the regularity constraint that

$$\begin{aligned} &i) && \forall i : (0 \leq i < r \Rightarrow 1 \leq n_i \leq 3), \\ &ii) && \forall i, j : (i < j \wedge n_i = n_j = 3 \Rightarrow \exists k : i < k < j \wedge n_k = 1), \\ &iii) && \forall i : (n_i = 3 \Rightarrow \exists k : k < i \wedge n_k = 1). \end{aligned}$$

d) The root has rank zero.

The heap order implies that the minimum element is at the root. Properties a), b) and c) bound the degree of a node by three times the rank of the node plus one. The size of the subtree rooted at a node is controlled by property c). Lemma 9 shows that the size is at least exponential in the rank. The last two properties are essential to achieve MELD in worst case constant time. The regularity constraint c) is a variation of the regularity constraint that Guibas *et al.* [60] used in their construction of finger search trees. The idea is that between two ranks where three children have equal rank there is a rank of which there only is one child. Figure 5.1 shows a heap ordered tree that satisfies the requirements a) to d) (the elements contained in the tree are omitted).

Lemma 9 *Any subtree rooted at a node of rank r has size $\geq 2^r$.*

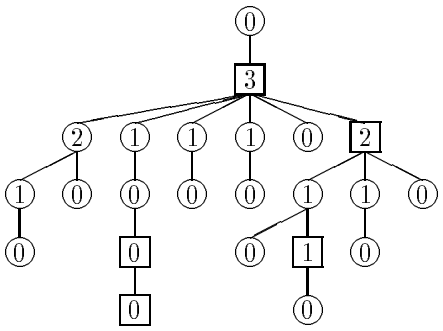

Figure 5.1: A heap ordered tree satisfying the properties a) to d). A box denotes a child of type I, a circle denotes a child of type II, and the numbers are the ranks of the nodes.

Proof. The proof is a simple induction in the structure of the tree. By c.*i*) leaves have rank zero and the lemma is true. For a node of rank r property c.*i*) implies that the node has at least one child of each rank less than r. By induction we get that the size is at least $1 + \sum_{i=0}^{r-1} 2^i = 2^r$. □

Corollary 4 *The only child of the root of a tree containing n elements has rank at most $\lfloor \log(n-1) \rfloor$.*

We now describe the details of how to represent a heap ordered tree. A child of type I is always the rightmost child. The children of type II appear in increasing rank order from right to left. See Figure 5.1 and Figure 5.2 for examples.

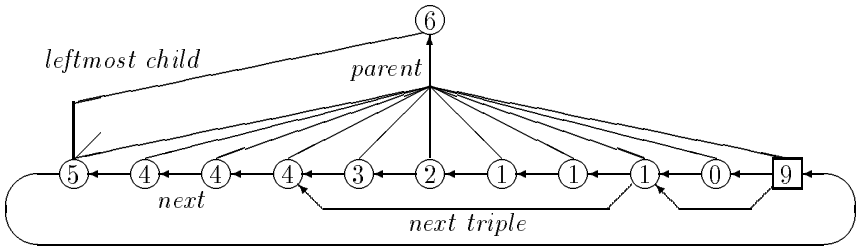

Figure 5.2: The arrangement of the children of a node.

A node consists of the following seven fields: 1) the element associated with the node, 2) the rank of the node, 3) the type of the node, 4) a pointer to the parent node, 5) a pointer to the leftmost child and 6) a pointer to the next sibling to the left. The next sibling pointer of the leftmost child points to the rightmost child in the list. This enables the access to the rightmost child of a node in constant time too. Field 7) is used to maintain a single linked list of triples of children of type II that have equal rank (see Figure 5.2). The nodes appear in increasing rank order. We only maintain these pointers for the rightmost child and for the rightmost child in a triple of children of equal rank. Figure 5.2 shows an example of how the children of a node are arranged.

In the next section we describe how to implement the operations. There are two essential transformations. The first transformation is to add a child of rank r to a node of rank r. Because we have a pointer to the leftmost child of a node (that has rank $r-1$ when $r > 0$) this can be done in constant time. Notice that this transformation cannot create three children of equal rank. The

second transformation is to find the smallest rank i where three children have equal rank. Two of the children are replaced by a child of rank $i+1$. Because we maintain a single linked list of triples of nodes of equal rank we can also do this in constant time.

5.2 Operations

In this section we describe how to implement the different operations. The basic operation we use is to link two nodes of equal rank r. This is done by comparing the elements associated with the two nodes and making the node with the largest element a child of the other node. By increasing the rank of the node with the smallest element to $r+1$ the properties a) to d) are satisfied. The operation is illustrated in Figure 5.3. This is similar to the linking of trees in binomial heaps and Fibonacci heaps [108, 53].

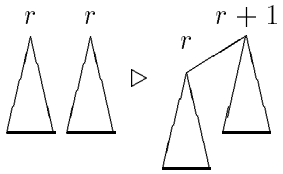

Figure 5.3: The linking of two nodes of equal rank.

We now describe how to implement the operations.

- MakeQueue is trivial. We just return the NULL pointer.
- FindMin(Q) returns the element located at the root of the tree representing Q.
- Insert(Q, e) is equal to Meld Q with a priority queue only consisting of a rank zero node containing e.
- Meld(Q_1, Q_2) can be implemented in two steps. In the first we insert one of the heap ordered trees into the other heap ordered tree. This can violate property c) at one node because the node gets one additional child of rank zero. In the second step we reestablish property c) at the node. Figure 5.4 shows an example of the first step.

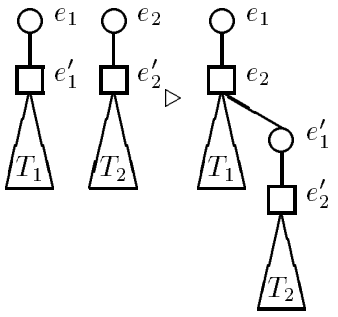

Figure 5.4: The first step of a Meld operation (the case $e_1 \leq e_2 < e_1' \leq e_2'$).

Let e_1 and e_2 denote the roots of the trees representing Q_1 and Q_2 and let e_1' and e_2' denote the only children of e_1 and e_2. Assume w.l.o.g. that e_1 is the smallest element. If $e_2 \geq e_1'$ we let e_2 become a rank zero child of e_1', otherwise $e_2 < e_1'$. If $e_2' < e_1'$ we can interchange the subtrees rooted at e_2' and e_1', so w.l.o.g. we assume $e_1 \leq e_2 < e_1' \leq e_2'$. In this case we make e_2 a rank zero child of e_1' and swap the elements e_1' and e_2 (see Figure 5.4). We have assumed that the sizes of Q_1 and Q_2 are at least two, but the other cases are just simplified cases of the general case.

The only invariants that can be violated now are the invariants b) and c) at the child of the root because it has got one additional rank zero child. Let v denote the child of the root. If v had rank zero we can satisfy the invariants by setting the rank of v to one. Otherwise only c) can be violated at v. Let n_i denote the number of children of v that have rank i. By linking two nodes of rank i where i is the smallest rank where $n_i = 3$ it is easy to verify that c) can be reestablished. The linking reduces n_i by two and increments n_{i+1} by one.

If we let $(n_{r-1}, \ldots, n_0)$ be a string in $\{1,2,3\}^*$ the following table shows that c) is reestablished after the above described transformations. We let x denote a string in $\{1,2,3\}^*$ and y_i strings in $\{1,2\}^*$. The table shows all the possible cases. Recall that c) states that between every two $n_i = 3$ there is at least one $n_i = 1$. The different cases are also considered in [60].

$$
\begin{aligned}
y_1 1 &\rhd y_1 2 \\
y_2 13 y_1 1 &\rhd y_2 21 y_1 2 \\
y_2 23 y_1 1 &\rhd y_2 31 y_1 2 \\
x 3 y_2 13 y_1 1 &\rhd x 3 y_2 21 y_1 2 \\
x 3 y_3 1 y_2 23 y_1 1 &\rhd x 3 y_3 1 y_2 31 y_1 2 \\
y_1 12 &\rhd y_1 21 \\
y_1 22 &\rhd y_1 31 \\
x 3 y_1 12 &\rhd x 3 y_1 21 \\
x 3 y_2 1 y_1 22 &\rhd x 3 y_2 1 y_1 31
\end{aligned}
$$

After the linking only b) can be violated at v because a child of rank r has been created. This problem can be solved by increasing the rank of v by one.

Because of the given representation MELD can be performed in worst case time $O(1)$.

- DELETEMIN(Q) removes the root e_1 of the tree representing Q. The problem is that now property d) can be violated because the new root e_2 can have arbitrary rank. This problem is solved by the following transformations.

  First we remove the root e_2. This element later on becomes the new root of rank zero. At most $O(\log n)$ trees can be created by removing the root. Among these trees the root that contains the minimum element e_3 is found and removed. This again creates at most $O(\log n)$ trees. We now find the root e_4 of maximum rank among all the trees and replaces it by the element e_3. A rank zero node containing e_4 is created.

  The tree of maximum rank and with root e_3 is made the only child of e_2. All other trees are made children of the node containing e_3. Notice that all the new children of e_3 have rank less than the rank of e_3. By iterated linking of children of equal rank where there are three children with equal rank, we can guarantee that $n_i \in \{1,2\}$ for all i less than the rank of e_3. Possibly, we have to increase the rank of e_3.

  Finally, we return the element e_1.

Because the number of trees is at most $O(\log n)$ DELETEMIN can be performed in worst case time $O(\log n)$. Figure 5.5 illustrates how DELETEMIN is performed.

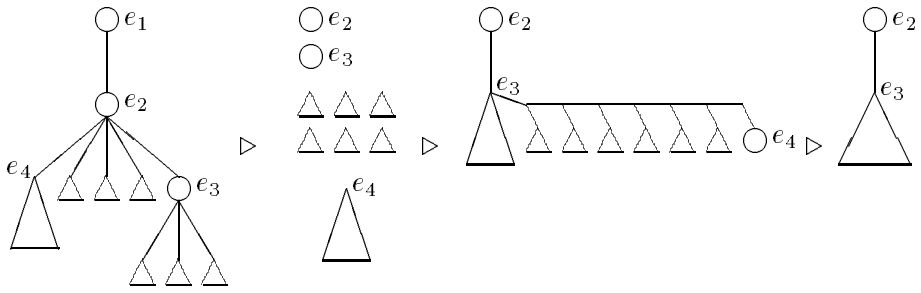

Figure 5.5: The implementation of DELETEMIN.

- DELETE(Q, e) can be implemented similar to DELETEMIN. If e is the root we just perform DELETEMIN. Otherwise we start by bubbling e upwards in the tree. We replace e with its parent until the parent of e has rank less than or equal to the rank of e. Now, e is the arbitrarily ranked child of its parent. This allows us to replace e with an arbitrary ranked node, provided that the heap order is still satisfied. Because the rank of e increases for each bubble step, and the rank of a node is bounded by $\lfloor \log(n-1) \rfloor$, this can be performed in time $O(\log n)$.

  We can now replace e with the meld of the children of e as described in the implementation of DELETEMIN. This again can be performed in worst case time $O(\log n)$.

To summarize, we have the theorem:

Theorem 11 *There exists an implementation of priority queues supporting* DELETE *and* DELETEMIN *in worst case time* $O(\log n)$ *and* MAKEQUEUE, FINDMIN, INSERT *and* MELD *in worst case time* $O(1)$. *The implementation requires linear space and can be implemented on the pointer machine.*

5.3 Optimality

The following theorem shows that if MELD is required to be nontrivial, *i.e.*, to take worst case sublinear time, then DELETEMIN must take worst case logarithmic time. This shows that the construction described in the previous sections is optimal among all implementations where MELD takes sublinear time.

If MELD is allowed to take linear time it is possible to support DELETEMIN in worst case constant time by using the finger search trees of Dietz and Raman [34]. By using their data structure MAKEQUEUE, FINDMIN, DELETEMIN, DELETE can be supported in worst case time $O(1)$, INSERT in worst case time $O(\log n)$ and MELD in worst case time $O(n)$.

Theorem 12 *If* MELD *can be performed in worst case time* $o(n)$ *then* DELETEMIN *cannot be performed in worst case time* $o(\log n)$.

Proof. The proof is by contradiction. Assume MELD takes worst case time $o(n)$ and DELETEMIN takes worst cast time $o(\log n)$. We show that this implies a contradiction with the $\Omega(n \log n)$ lower bound on comparison based sorting.

Assume we have n elements that we want to sort. Assume w.l.o.g. that n is a power of 2, $n = 2^k$. We can sort the elements by the following list of priority queue operations. First, create n priority queues each containing one of the n elements (each creation takes worst case time $O(1)$). Then join the n priority queues to one priority queue by $n-1$ MELD operations. The MELD operations are done bottom-up by always melding two priority queues of smallest size. Finally, perform n DELETEMIN operations. The elements are now sorted.

The total time for this sequence of operations is:

$$n\mathrm{T}_{\textsc{MakeQueue}} + \sum_{i=0}^{k-1} 2^{k-1-i}\mathrm{T}_{\textsc{Meld}}(2^i) + \sum_{i=1}^{n} \mathrm{T}_{\textsc{DeleteMin}}(i) = o(n \log n).$$

This contradicts the lower bound on comparison based sorting. □

5.4 Conclusion

We have presented an implementation of meldable priority queues where MELD takes worst case time $O(1)$ and DELETEMIN worst case time $O(\log n)$.

Another interesting operation to consider is DECREASEKEY. Our data structure supports DECREASEKEY in worst case time $O(\log n)$, because DECREASEKEY can be implemented in terms of a DELETE operation followed by an INSERT operation. Relaxed heaps [43] support DECREASEKEY in worst case time $O(1)$ but do not support MELD. But it is easy to see that relaxed heaps can be extended to support MELD in worst case time $O(\log n)$. The problem to consider is if it is possible to support both DECREASEKEY and MELD simultaneously in worst case constant time.

As a simple consequence of our construction we get a new implementation of meldable double ended priority queues, which is a data type that allows both FINDMIN/FINDMAX and DELETEMIN/ DELETEMAX [9, 38]. For each queue we just have to maintain two heap ordered trees as described in Section 5.1. One tree ordered with respect to minimum and the other with respect to maximum. If we let both trees contain all elements and the elements know their positions in both trees we get the following corollary.

Corollary 5 *An implementation of meldable double ended priority queues exists that supports* MAKEQUEUE, FINDMIN, FINDMAX, INSERT *and* MELD *in worst case time* $O(1)$ *and* DELETEMIN, DELETEMAX, DELETE, DECREASEKEY *and* INCREASEKEY *in worst case time* $O(\log n)$.

Chapter 6

Worst Case Efficient Priority Queues

Worst Case Efficient Priority Queues*

Gerth Stølting Brodal
BRICS†, Department of Computer Science, University of Aarhus
Ny Munkegade, DK-8000 Århus C, Denmark
gerth@brics.dk

Abstract

An implementation of priority queues is presented that supports the operations MakeQueue, FindMin, Insert, Meld and DecreaseKey in worst case time $O(1)$ and DeleteMin and Delete in worst case time $O(\log n)$. The space requirement is linear. The data structure presented is the first achieving this worst case performance.

Category: E.1
Keywords: priority queues, meld, decrease key, worst case complexity

6.1 Introduction

We consider the problem of implementing priority queues which are efficient in the worst case sense. The operations we want to support are the following commonly needed priority queue operations [76].

MakeQueue creates and returns an empty priority queue.

FindMin(Q) returns the minimum element contained in priority queue Q.

Insert(Q, e) inserts an element e into priority queue Q.

Meld(Q_1, Q_2) melds priority queues Q_1 and Q_2 to a new priority queue and returns the resulting priority queue.

DecreaseKey(Q, e, e') replaces element e by e' in priority queue Q provided $e' \leq e$ and it is known where e is stored in Q.

DeleteMin(Q) deletes and returns the minimum element from priority queue Q.

Delete(Q, e) deletes element e from priority queue Q provided it is known where e is stored in Q.

The construction of priority queues is a classical topic in data structures [15, 27, 43, 46, 49, 52, 53, 71, 97, 107, 108, 109]. A historical overview of implementations can be found in [76]. There are many applications of priority queues. Two of the most prominent examples are sorting problems and network optimization problems [100].

*This work was partially supported by the ESPRIT II Basic Research Actions Program of the EC under contract no. 7141 (project ALCOM II) and by the Danish Natural Science Research Council (Grant No. 9400044).
†BRICS (Basic Research in Computer Science), a Centre of the Danish National Research Foundation.

	Amortized	Worst case		
	Fredman *et al.* [53]	Driscoll *et al.* [43]	Brodal [15]	New result
MakeQueue	$O(1)$	$O(1)$	$O(1)$	$O(1)$
FindMin	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Insert	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Meld	$O(1)$	$O(\log n)$	$O(1)$	$O(1)$
DecreaseKey	$O(1)$	$O(1)$	$O(\log n)$	$O(1)$
Delete/DeleteMin	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$

Table 6.1: Time bounds for the previously best priority queue implementations.

In the amortized sense, [101], the best performance for these operations is achieved by Fibonacci heaps [53]. They achieve amortized constant time for all operations except for the two delete operations which require amortized time $O(\log n)$. The data structure we present achieves matching worst case time bounds for all operations. Previously, this was only achieved for various strict subsets of the listed operations [15, 27, 43, 107]. For example the relaxed heaps of Driscoll *et al.* [43] and the priority queues in [15] achieve the above time bounds in the worst case sense except that in [43] Meld requires worst case time $\Theta(\log n)$ and in [15] DecreaseKey requires worst case time $\Theta(\log n)$. Refer to Table 6.1. If we ignore the Delete operation our results are optimal in the following sense. A lower bound for DeleteMin in the comparison model is proved in [15] where it is proved that if Meld can be performed in time $o(n)$ then DeleteMin *cannot* be performed in time $o(\log n)$.

The data structure presented in this paper originates from the same ideas as the relaxed heaps of Driscoll *et al.* [43]. In [43] the data structure is based on heap ordered trees where $\Theta(\log n)$ nodes may violate heap order. We extend this to allow $\Theta(n)$ heap order violations which is a necessary condition to be able to support Meld in worst case constant time and if we allow a nonconstant number of violations.

In Section 6.2 we describe the data structure representing a priority queue. In Section 6.3 we describe a special data structure needed internally in the priority queue construction. In Section 6.4 we show how to implement the priority queue operations. In Section 6.5 we summarize the required implementation details. Finally some concluding remarks on our construction are given in Section 6.6.

6.2 The Data Structure

In this section we describe the components of the data structure representing a priority queue. A lot of technical constraints are involved in the construction. Primary these are consequences of the transformations to be described in Section 6.3 and Section 6.4.3. In Section 6.5 we summarize the required parts of the construction described in the following sections.

The basic idea is to represent a priority queue by two trees T_1 and T_2 where all nodes contain one element and have a nonnegative integer rank assigned. Intuitively the rank of a node is the logarithm of the size of the subtree rooted at the node. The details of the rank assignment achieving this follow below.

The children of a node are stored in a doubly linked list in increasing rank order from right to left. Each node has also a pointer to its leftmost child and a pointer to its parent.

The notation we use is the following. We make no distinction between a node and the element

it contains. We let $x, y, \ldots$ denote nodes, $p(x)$ the parent of x, $r(x)$ the rank of x, $n_i(x)$ the number of children of rank i that x has and t_i the root of T_i. Nodes which are larger than their parents are called *good* nodes — good because they satisfy heap order. Nodes which are not good are called *violating* nodes.

The idea is to let t_1 be the minimum element and to lazy merge the two trees T_1 and T_2 such that T_2 becomes empty. Since t_1 is the minimum element we can support FINDMIN in worst case constant time and the lazy merging of the two trees corresponds intuitively to performing MELD incrementally over the next sequence of operations. The merging of the two trees is done by incrementally increasing the rank of t_1 by moving the children of t_2 to t_1 such that T_2 becomes empty and t_1 becomes the node of maximum rank. The actual details of implementing MELD follow in Section 6.4.5.

As mentioned before we have some restrictions on the trees forcing the rank of a node to be related to the size of the subtree rooted at the node. For this purpose we maintain the invariants S1–S5 below for any node x.

S1 : If x is a leaf, then $r(x) = 0$,

S2 : $r(x) < r(p(x))$,

S3 : if $r(x) > 0$, then $n_{r(x)-1}(x) \geq 2$,

S4 : $n_i(x) \in \{0, 2, 3, \ldots, 7\}$,

S5 : $T_2 = \emptyset$ or $r(t_1) \leq r(t_2)$.

The first two invariants just say that leaves have rank zero and that the ranks of the nodes strictly increase towards the root. Invariant S3 says that a node of rank k has at least two children of rank $k-1$. This guarantees that the size of the subtree rooted at a node is at least exponential in the rank of the node (by induction it follows from S1 and S3 that the subtree rooted at node x has size at least $2^{r(x)+1} - 1$). Invariant S4 bounds the number of children of a node that have the same rank within a constant. This implies the crucial fact that all nodes have rank and degree $O(\log n)$. Finally S5 says that either T_2 is empty or its root has rank larger than or equal to the rank of the root of T_1.

Notice that in S4 we do not allow a node to have only a single child of a given rank. This is because this allows us to cut off the leftmost children of a node such that the node can get a new rank assigned where S3 is still satisfied. This property is essential to the transformations to be described in Section 6.4.3. The requirement $n_i(x) \leq 7$ in S4 is a consequence of the construction described in Section 6.3.

After having described the conditions of how nodes are assigned ranks and how this forces structure on the trees we now turn to consider how to handle the violating nodes — which together with the two roots could be potential minimum elements. To keep track of the violating nodes we associate to each node x two subsets $V(x)$ and $W(x)$ of nodes larger than x from the trees T_1 and T_2. That is the nodes in $V(x)$ and $W(x)$ are good with respect to x. We do not require that if $y \in V(x) \cup W(x)$ that x and y belong to the same T_i tree. But we require that a node y belongs to at most one V or one W set. Also we do not require that if $y \in V(x) \cup W(x)$ then $r(y) \leq r(x)$.

The V sets and the W sets are all stored as doubly linked lists. Violations added to a V set are always added to the front of the list. Violations added to a W set are always added in such a way that violations of the same rank are adjacent. So if we have to add a violation to $W(x)$ and there is already a node in $W(x)$ of the same rank, then we insert the new node adjacent to this node. Otherwise we just insert the new node at the front of $W(x)$.

We implement the $V(x)$ and $W(x)$ sets by letting each tree node x have four additional pointers: One to the first node in $V(x)$, one to the first node in $W(x)$, and two to the next and previous node in the violation list that x belongs to — provided x is contained in a violation list. Each time we add a node to a violation set we always first remove the node from the set it possibly belonged to.

Intuitively $V(x)$'s purpose is to contain violating nodes of large rank. Whereas $W(x)$'s purpose is to contain violating nodes of small rank. If a new violating node is created which has large rank, *i.e.*, $r(x) \geq r(t_1)$, we add the violation to $V(t_1)$, otherwise we add the violation to $W(t_1)$. To be able to add a node to $W(t_1)$ at the correct position we need to know if a node already exists in $W(t_1)$ of the same rank. In case there is we need to know such an element. For this purpose we maintain an extendible array[1] of size $r(t_1)$ of pointers to nodes in $W(t_1)$ of each possible rank. If no node exists of a given rank in $W(t_1)$ the corresponding entry in the array is NULL.

The structure on the V and W sets is enforced by the following invariants O1–O5. We let $w_i(x)$ denote the number of nodes in $W(x)$ of rank i.

O1 : $t_1 = \min T_1 \cup T_2$,

O2 : if $y \in V(x) \cup W(x)$, then $y \geq x$,

O3 : if $y < p(y)$, then an $x \neq y$ exists such that $y \in V(x) \cup W(x)$,

O4 : $w_i(x) \leq 6$,

O5 : if $V(x) = (y_{|V(x)|}, \ldots, y_2, y_1)$, then

$$r(y_i) \geq \lfloor (i-1)/\alpha \rfloor \text{ for } i = 1, 2, \ldots, |V(x)|$$

where α is a constant.

O1 guarantees that the minimum element contained in a priority queue always is the root of T_1. O2 says that the elements are heap ordered with respect to membership of the V and W sets. O3 says that all violating nodes belong to a V or W set. Because all nodes have rank $O(\log n)$ invariants O4 and O5 imply that the sizes of all V and W sets are $O(\log n)$. Notice that if we remove an element from a V or W set, then the invariants O4 and O5 cannot become violated.

That invariants O4 and O5 are stated quite differently is because the V and W sets have very different roles in the construction. Recall that the V sets take care of large violations, *i.e.*, violations that have rank larger than $r(t_1)$ when they are created. The constant α is the number of large violations that can be created between two increases in the rank of t_1.

For the roots t_1 and t_2 we strengthen the invariants such that R1–R3 also should be satisfied.

R1 : $n_i(t_j) \in \{2, 3, \ldots, 7\}$ for $i = 0, 1, \ldots, r(t_j) - 1$,

R2 : $|V(t_1)| \leq \alpha r(t_1)$,

R3 : if $y \in W(t_1)$ then $r(y) < r(t_1)$.

Invariant R1 guarantees that there are at least two children of each rank at both roots. This property is important for the transformations to be described in Section 6.4.2 and Section 6.4.3. Invariant R2 together with invariant O5 guarantee that if we can increase the rank of t_1 by one we can create α new large violations and add them to $V(t_1)$ without violating invariant O5. Invariant R3 says that all violations in $W(t_1)$ have to be small.

[1] An extendible array is an array of which the length can be increased by one in worst case constant time. It is folklore that extendible arrays can be obtained from ordinary arrays by array doubling and incremental copying. In the rest of this paper all arrays are extendible arrays.

The maintenance of R1 and O4 turns out to be nontrivial but they can all be maintained by applying the same idea. To unify this idea we introduce the concept of a *guide* to be described in Section 6.3.

The main idea behind the construction is the following captured by the DECREASEKEY operation. The details follow in Section 6.4. Each time we perform a DECREASEKEY operation we just add the new violating node to one of the sets $V(t_1)$ or $W(t_1)$. To avoid having too many violations stored at the root of T_1 we incrementally do two different kinds of transformations. The first transformation moves the children of t_2 to t_1 such that the rank of t_1 increases. The second transformation reduces the number of violations in $W(t_1)$ by replacing two violations of rank k by at most one violation of rank $k+1$. These transformations are performed to reestablish invariants R2 and O4.

6.3 Guides

In this section we describe the *guide* data structure that helps us maintaining the invariants R1 and O4 on $n_i(t_1), n_i(t_2)$ and $w_i(t_1)$. The relationship between the abstract sequences of variables in this section and the children and the violations stored at the roots are explained in Section 6.4.

The problem can informally be described as follows. Assume we have to maintain a sequence of integer variables $x_k, x_{k-1}, \ldots, x_1$ (all sequences in this section goes from right to left) and we want to satisfy the invariant that all $x_i \leq T$ for some threshold T. On the sequence we can only perform REDUCE(i) operations which decrease x_i by at least two and increase x_{i+1} by at most one. The x_is can be forced to increase and decrease by one, but for each change in an x_i we are allowed to do $O(1)$ REDUCE operations to prevent any x_i from exceeding T. The guide's job is to tell us which operations to perform.

This problem also arises implicitly in [15, 27, 60, 67]. But the solution presented in [60] requires time $\Theta(k)$ to find which REDUCE operations to perform whereas the problems in the other papers are simpler because only x_1 can be forced to increase and decrease. The data structure we present can find which operations to perform in worst case time $O(1)$ for the general problem.

To make the guide's knowledge about the x_is as small as possible we reveal to the guide another sequence $x'_k, \ldots, x'_1$ such that $x_i \leq x'_i \in \{T-2, T-1, T\}$ (this choice is a consequence of the construction we describe below). As long as all $x_i \leq x'_i$ we do not require help from the guide. First when an $x_i = x'_i$ is forced to become x_i+1 we require help from the guide. In the following we assume w.l.o.g. that the threshold T is two such that $x'_i \in \{0,1,2\}$ and that REDUCE(i) decreases x'_i by two and increases x'_{i+1} by one.

The data structure maintained by the guide partitions the sequence $x'_k, \ldots, x'_1$ into *blocks* of consecutive x'_is of the form $2,1,1,\ldots,1,0$ where the number of ones is allowed to be zero. The guide maintains the invariant that all x'_is not belonging to a block of the above type have value either zero or one. An example of a sequence satisfying this is the following where blocks are shown by underlining the subsequences.

$$1,\underline{2,1,1,0},1,1,\underline{2,0},\underline{2,0},1,0,\underline{2,1,0}.$$

The guide stores the values of the variables x'_i in one array and uses another array to handle the blocks. The second array contains pointers to memory cells which contain the index of an x_i or the value $-$. All variables in the same block point to the same cell and this cell contains the index of the leftmost variable in the block. Variables not belonging to a block point to a cell containing $-$. A data structure for the previous example is illustrated in Figure 6.1. Notice that several variables can share a memory cell containing $-$. This data structure has two very important properties:

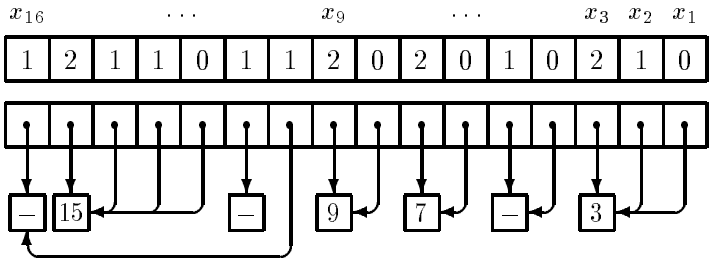

Figure 6.1: The guide data structure.

1. Given a variable we can in worst case time $O(1)$ find the leftmost variable in the block, and
2. we can in worst case time $O(1)$ destroy a given block, *i.e.*, let all nodes in the block belong to no block, by simply assigning $-$ to the block's memory cell.

When an x_i' is forced to increase the guide can in worst case time $O(1)$ decide which REDUCE operations to perform. We only show how to handle one nontrivial case, all other cases are similar. Assume that there are two blocks of variables adjacent to each other and that the leftmost $x_i' = 1$ in the rightmost block has to be increased. Then the following transformations have to be performed:

$$
\begin{array}{lll}
 & \underline{2,1,1,0},\underline{2,1,1,1,0} & \\
\triangleright & \underline{2,1,1,0},\underline{2,2,1,1,0} & \text{increment } x_i', \\
\triangleright & \underline{2,1,1,1},\underline{0,2,1,1,0} & \text{REDUCE}, \\
\triangleright & \underline{2,1,1,1},\underline{1,0,1,1,0} & \text{REDUCE}, \\
\triangleright & \underline{2,1,1,1,1,0},1,1,0 & \text{reestablish blocks.}
\end{array}
$$

To reestablish the blocks the two pointers of the new variables in the leftmost block are set to point to the leftmost block's memory cell and the rightmost block's memory cell is assigned the value $-$.

In the case described above only two REDUCE operations were required and these were performed on x_j's where $j \geq i$. This is true for all cases.

We conclude this section with two remarks on the construction. By using extendible arrays the sequence of variables can be extended by a new x_{k+1} equal to zero or one in worst case time $O(1)$. If we add a reference counter to each memory cell we can reuse the memory cells such that the total number of needed memory cells is at most k.

6.4 Operations

In this section we describe how to implement the different priority queue operations. We begin by describing some transformations on the trees which are essential to the operations to be implemented.

6.4.1 Linking and delinking trees

The fundamental operation on the trees is the linking of trees. Assume that we have three nodes x_1, x_2 and x_3 of equal rank and none of them is a root t_i. By doing two comparisons we can find

the minimum. Assume w.l.o.g. that x_1 is the minimum. We can now make x_2 and x_3 the leftmost children of x_1 and increase the rank of x_1 by one. Neither x_2 or x_3 become violating nodes and x_1 still satisfies all the invariants S1–S5 and O1–O5.

The delinking of a tree rooted at node x is a little bit more tricky. If x has exactly two or three children of rank $r(x)-1$, then these two or three children can be cut off and x gets the rank of the largest ranked child plus one. From S4 it follows that x still satisfies S3 and it follows that S1–S5 and O1–O5 are still satisfied. In the case where x has at least four children of rank $r(x)-1$ two of these children are simply cut off. Because x still has at least two children of rank $r(x)-1$ the invariants are still satisfied.

It follows that the delinking of a tree of rank k always results in two or three trees of rank $k-1$ and one additional tree of rank at most k (the tree can be of any rank between zero and k).

6.4.2 Maintaining the children of a root

We now describe how to add children below a root and how to cut off children at a root while keeping R1 satisfied. For this purpose we require four guides, two for each of the roots t_1 and t_2. We only sketch the situation at t_1 because the construction for t_2 is analogous.

To have constant time access to the children of t_1 we maintain an extendible array of pointers that for each rank $i = 0, \ldots, r(t_1)-1$ has a pointer to a child of t_1 of rank i. Because of R1 such children are guaranteed to exist. This enables us to link and delink children of rank i in worst case time $O(1)$ for an arbitrary i. One guide takes care of that $n_i(t_1) \leq 7$ and the other of that $n_i(t_1) \geq 2$ for $i = 0, \ldots, r(t_1)-3$ (to maintain a lower bound on a sequence of variables is equivalent to maintaining an upper bound on the negated sequence). The children of t_1 of rank $r(t_1)-1$ and $r(t_1)-2$ are treated separately in a straight forward way such that there always are between 2 and 7 children of these ranks. This is necessary because of the dependency between the guide maintaining the upper bound on $n_i(t_1)$ and the guide maintaining the lower bound on $n_i(t_1)$. The "marked" variables that we reveal to the guide that maintains the upper bound on $n_i(t_1)$ have values $\{5,6,7\}$ and to the guide that maintains the lower bound have values $\{4,3,2\}$.

If we add a new child at t_1 of rank i we tell the guide maintaining the upper bound that $n_i(t_1)$ is forced to increase by one (this assumes $i < r(t_1)-2$). Then the guide then tells us where to do at most two REDUCE operations. The REDUCE(i) operation in this context corresponds to the linking of three trees of rank i. This decreases $n_i(t_1)$ by three and increases $n_{i+1}(t_1)$ by one. We only do the linking when $n_i(t_1) = 7$ so that the guide maintaining the lower bound on $n_i(t_1)$ will be unaffected (this implies a minor change in the guide). If this results in too many children of rank $r(t_1)-2$ or $r(t_1)-1$ we link some of these children and possibly increase the rank of t_1. If the rank of t_1 increases we also have to increase the domain of the two guides.

To cut off a child is similar, but now the REDUCE operation corresponds to the delinking of a tree. The additional tree from the delinking transformation that can have various ranks is treated separately after the delinking. We just add it below t_1 as described above.

At t_2 the situation is nearly the same. The major difference is that because we knew that t_1 was the smallest element the linking and delinking of children of t_1 would not create new violations. This is not true at t_2. The linking of children never creates new violations but the delinking of children at t_2 can create three new violations. We will see in Section 6.4.4 that it turns out that we only cut off children of t_2 which have rank larger than $r(t_1)$. The tree "left over" by a delinking is made a child of t_1 if it has rank less than $r(t_1)$. Otherwise it is made a child of t_2. The new violations which have rank larger than $r(t_1)$ are added to $V(t_1)$. To satisfy O5 and R2 we just have to guarantee that the rank of t_1 will be increased and that α in R2 and O5 is chosen large enough.

6.4.3 Violation reducing transformations

We now describe the most essential transformation on the trees. The transformation reduces the number of *potential violations* $\bigcup_{y \in T_1 \cup T_2} V(y) \cup W(y)$ in the tree by at least one.

Assume we have two potential violations x_1 and x_2 of equal rank $k < r(t_1)$ which are not roots or children of a root. First we check that both x_1 and x_2 are violating nodes. If one of the nodes already is a good node we remove it from the corresponding violation set. Otherwise we proceed as described below.

Because of S4 we know that both x_1 and x_2 have at least one brother. If x_1 and x_2 are not brothers assume w.l.o.g. that $p(x_1) \leq p(x_2)$ and swap the subtrees rooted at x_1 and at a brother of x_2. The number of violations can only decrease by doing this swap. We can now w.l.o.g. assume that x_1 and x_2 are brothers and both children of node y.

If x_1 has more than one brother of rank k we just cut off x_1 and make it a good child of t_1 as described in Section 6.4.2. Because x_1 had at least two brothers of rank k, S4 is still satisfied at y.

In case x_1 and x_2 are the only brothers of rank k and $r(y) > k + 1$ we just cut off both x_1 and x_2 and make them new good children of t_1 as described in Section 6.4.2. Because of invariant S4 we are forced to cut off both children.

The only case that remains to be considered is when x_1 and x_2 are the only children of rank k and that $r(y) = k + 1$. In this case we cut off x_1, x_2 and y. The new rank of y is uniquely given by one plus the rank of its new leftmost child. We replace y by a child of t_1 of rank $k + 1$ which can be cut off as described in Section 6.4.2. If y was a child of t_1 we only cut off y. If the replacement for y becomes a violating node of rank $k + 1$ we add it to $W(t_1)$. Finally, x_1, x_2 and y are made good children of t_1 as described in Section 6.4.2.

Above it is important that the node y is replaced by is not an ancestor of y, because if y was replaced by such a node a cycle among the parent pointers would be created. Invariant S2 guarantees that this cannot happen.

6.4.4 Avoiding too many violations

We now describe how to avoid too many violations. The only violation sets we add violations to are $V(t_1)$ and $W(t_1)$. Violations of rank larger than $r(t_1)$ are added to $V(t_1)$ and otherwise they are added to $W(t_1)$. The violations in $W(t_1)$ are controlled by a guide. This guide guarantees that $w_i(t_1) \leq 6$. We maintained a single array so we could access the violating nodes in $W(t_1)$ by their rank.

If we add a violation to $W(t_1)$ the guide tells us for which ranks we should do violation reducing transformations as described in the previous section. We only do the transformation if there are exactly six violations of the given rank and that there is at least two violating nodes which are not children of t_2. If there are more than four violations that are children of t_2 we cut the additional violations off and links them below t_1. This makes these nodes good and does not affect the guides maintaining the children at t_2.

For each priority queue operation that is performed we increase the rank of t_1 by at least one by moving a constant number of children from t_2 to t_1 — provided $T_2 \neq \emptyset$. By increasing the rank of t_1 by one we can afford creating α new violations of rank larger than $r(t_1)$ by invariant O5 and we can just add the violations to the list $V(t_1)$. If $T_2 \neq \emptyset$ and $r(t_2) \leq r(t_1) + 2$ we just cut of the largest children of t_2 and link them below t_1 and finally add t_2 below t_1. This will satisfy the invariants. Otherwise we cut off a child of t_2 of rank $r(t_1) + 2$ and delink this child and add the resulting trees below t_1 such that the rank of t_1 increases by at least one. By choosing α large enough the invariants will become reestablished.

If T_2 is empty we cannot increase the rank t_1, but this also implies that t_1 is the node of maximum rank so no large violations can be created and R2 cannot become violated.

6.4.5 Priority queue operations

In the following we describe how to implement the different priority queue operations such that the invariants from Section 6.2 are maintained.

- MakeQueue is trivial. We return a pair of empty trees.
- FindMin(Q) returns t_1.
- Insert(Q, e) is a special case of Meld where Q_2 is a priority queue only containing one element.
- Meld(Q_1, Q_2) involves at most four trees; two for each queue. The tree having the new minimum element as root becomes the new T_1 tree. This tree was either the T_1 tree of Q_1 or of Q_2. If this tree is the tree of maximum rank we just add the other trees below this tree as described previously. In this case no violating node is created so no transformation is done on the violating nodes.

  Otherwise the tree of maximum rank becomes the new T_2 tree and the remaining trees are added below this node as described in Section 6.4.2, possibly delinking the new children once if they have the same rank as t_2. The violations created by this are treated as described in Section 6.4.4. The guides and arrays used at the old roots that now are linked below the new t_2 node we just discard.
- DecreaseKey(Q, e, e') replaces the element of e by e' ($e' \leq e$). If e' is less than t_1 we swap the elements in the two nodes. If e' is a good node we stop, otherwise we proceed as described in Section 6.4.4 to avoid having too many violations stored at t_1.
- DeleteMin(Q) is allowed to take worst case time $O(\log n)$. First T_2 is made empty by moving all children of T_2 to T_1 and making the root t_2 a rank zero child of t_1. Then t_1 is deleted. This gives us at most $O(\log n)$ independent trees. The minimum element is then found by looking at the sets V and W of the old root of T_1 and all the roots of the independent trees. If the minimum element is not a root we swap it with one of the independent trees of equal rank. This at most creates one new violation. By making the independent trees children of the new minimum element and performing $O(\log n)$ linking and delinking operations on these children we can reestablish S1–S5 and R1 and R3. By merging the V and W sets at the root to one set and merging the old minimum element's V and W sets with the set we get one new set of violations of size $O(\log n)$. Possibly we also have to add the single violation created by the swapping. By doing at most $O(\log n)$ violation reducing transformations as described previously we can reduce the set to contain at most one violation of each rank. We make the resulting set the new W set of the new root and let the corresponding V set be empty. This implies that O1–O5 and R2 are being reestablished. The guides involved are initiated according to the new situation at the root of T_1.
- Delete(Q, e). If we let $-\infty$ denote the smallest possible element, then Delete can be implemented as DecreaseKey($Q, e, -\infty$) followed by DeleteMin(Q).

6.5 Implementation details

In this section we summarize the required details of our new data structure.

Each node we represent by a record having the following fields.

- The element associated with the node,
- the rank of the node,
- pointers to the node's left and right brothers,
- a pointer to the parent node,
- a pointer to the leftmost child,
- pointers to the first node in the node's V and W sets, and
- pointers to the next and the previous node in the violation list that the node belongs to. The first node in a violation list $V(x)$ or $W(x)$ has its previous violation pointer pointing to x.

In addition to the above nodes we maintain the following three extendible arrays:

- An array of pointers to children of t_1 of rank $i = 0, \ldots, r(t_1) - 1$,
- a similar array for t_2, and
- an array of pointers to nodes in $W(t_1)$ of rank $i = 0, \ldots, r(t_1) - 1$ (if no node in $W(t_1)$ exist of a given rank we let the corresponding pointer be NULL).

Finally we have five guides: Three to maintain the upper bounds on $n_i(t_1), n_i(t_2)$ and $w_i(t_1)$ and two to maintain the lower bounds on $n_i(t_1)$ and $n_i(t_2)$.

6.6 Conclusion

From the construction presented in the previous sections we conclude that:

Theorem 13 *An implementation of priority queues exists that supports the operations* MakeQueue, FindMin, Insert, Meld *and* DecreaseKey *in worst case time* $O(1)$ *and* DeleteMin *and* Delete *in worst case time* $O(\log n)$. *The space required is linear in the number of elements contained in the priority queues.*

The data structure presented is quite complicated. An important issue for further work is to simplify the construction to make it applicable in practice. It would also be interesting to see if it is possible to remove the requirement for arrays from the construction.

Acknowledgement

The author thanks Rolf Fagerberg for the long discussions that lead to the results presented in the paper.

Chapter 7

Priority Queues on Parallel Machines

Priority Queues on Parallel Machines

Gerth Stølting Brodal*
BRICS†, Department of Computer Science, University of Aarhus
Ny Munkegade, DK-8000 Århus C, Denmark
gerth@brics.dk

Abstract

We present time and work optimal priority queues for the CREW PRAM, supporting FindMin in constant time with one processor and MakeQueue, Insert, Meld, DeleteMin, Delete and DecreaseKey in constant time with $O(\log n)$ processors. A priority queue can be build in time $O(\log n)$ with $O(n/\log n)$ processors and k elements can be inserted into a priority queue in time $O(\log k)$ with $O((\log n + k)/\log k)$ processors. With a slowdown of $O(\log\log n)$ in time the priority queues adopt to the EREW PRAM by only increasing the required work by a constant factor. A pipelined version of the priority queues adopt to a processor array of size $O(\log n)$, supporting the operations MakeQueue, Insert, Meld, FindMin, DeleteMin, Delete and DecreaseKey in constant time.

Category: E.1, F.1.2
Keywords: priority queues, meld, PRAM, worst case complexity

7.1 Introduction

The construction of priority queues is a classical topic in data structures. Some references are [15, 18, 43, 49, 52, 53, 108, 109]. A historical overview of implementations can be found in [76]. Recently several papers have also considered how to implement priority queues on parallel machines [28, 32, 35, 70, 89, 90, 94, 95]. In this paper we focus on how to achieve optimal speedup for the individual priority queue operations known from the sequential setting [90, 94]. The operations we support are all the commonly needed priority queue operations from the sequential setting [76] and the parallel insertion of several elements at the same time [28, 89].

MakeQueue Creates and returns a new empty priority queue.

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

Meld(Q_1, Q_2) Melds priority queues Q_1 and Q_2. The resulting priority queue is stored in Q_1.

FindMin(Q) Returns the minimum element in priority queue Q.

DeleteMin(Q) Deletes and returns the minimum element in priority queue Q.

Delete(Q, e) Deletes element e from priority queue Q provided a pointer to e is given.

*Supported by the Danish Natural Science Research Council (Grant No. 9400044). Partially supported by the ESPRIT Long Term Research Program of the EU under contract #20244 (ALCOM-IT). This research was done while visiting the Max-Planck Institut für Informatik, Saarbrücken, Germany.
†BRICS (Basic Research in Computer Science), a Centre of the Danish National Research Foundation.

DECREASEKEY(Q, e, e') Replaces element e by e' in priority queue Q provided $e' \leq e$ and a pointer to e is given.

BUILD$(e_1, \ldots, e_n)$ Creates a new priority queue containing elements $e_1, \ldots, e_n$.

MULTIINSERT$(Q, e_1, \ldots, e_k)$ Inserts elements $e_1, \ldots, e_k$ into priority queue Q.

We assume that elements are taken from a totally ordered universe and that the only operation allowed on elements is the comparison of two elements that can be done in constant time. Throughout this paper n denotes the maximum allowed number of elements in a priority queue. We assume w.l.o.g. that n is of the form 2^k. This guarantees that $\log n$ is an integer.[1] Our main result is:

Theorem 14 *On a CREW PRAM priority queues exist supporting* FINDMIN *in constant time with one processor, and* MAKEQUEUE, INSERT, MELD, DELETEMIN, DELETE *and* DECREASEKEY *in constant time with $O(\log n)$ processors.* BUILD *is supported in time $O(\log n)$ with $O(n/\log n)$ processors and* MULTIINSERT *in time $O(\log k)$ with $O((\log n + k)/\log k)$ processors.*

Table 7.1 lists the performance of different implementations adopting parallelism to priority queues. Several papers consider how to build heaps [49] optimally in parallel [32, 35, 70, 95]. On an EREW PRAM an optimal construction time of $O(\log n)$ is achieved in [95] and on a CRCW PRAM an optimal construction time of $O(\log \log n)$ is achieved in [35].

An immediate consequence of the CREW PRAM priority queues we present is that on an EREW PRAM we achieve the bounds stated in Corollary 6, because the only bottleneck in the construction requiring concurrent read is the broadcasting of information of constant size, that on an $O(\log n/\log \log n)$ processor EREW PRAM requires time $O(\log \log n)$. The bounds we achieve matches those of [28] for k equal one and those of [88]. See Table 7.1.

Corollary 6 *On an EREW PRAM priority queues exist supporting* FINDMIN *in constant time with one processor, and supporting* MAKEQUEUE, INSERT, MELD, DELETEMIN, DELETE *and* DECREASEKEY *in time $O(\log \log n)$ with $O(\log n/\log \log n)$ processors. With $O(n/\log n)$ processors* BUILD *can be performed in time $O(\log n)$ and with $O((k+\log n)/(\log k + \log \log n))$ processors* MULTIINSERT *can be performed in time $O(\log k + \log \log n)$.*

That a systolic processor array with $\Theta(n)$ processors can implement a priority queue supporting the operations INSERT and DELETEMIN in constant time is parallel computing folklore, see Exercise 1.119 in [72]. Recently Ranade *et al.* [94] showed how to achieve the same bounds on a processor array with only $O(\log n)$ processors. In Section 7.5 we describe how the priority queues can be modified to allow operations to be performed via pipelining. As a result we get an implementation of priority queues on a processor array with $O(\log n)$ processors, supporting the operations MAKEQUEUE, INSERT, MELD, FINDMIN, DELETEMIN, DELETE and DECREASEKEY in constant time. This extends the result of [94].

The priority queues we present in this paper do not support the operation MULTIDELETE, that deletes the k smallest elements from a priority queue (where k is fixed [28, 89]). However, a possible solution is to apply the k-bandwidth idea used in [28, 89], by letting each node contain k elements instead of one. If we apply the idea to the data structure in Section 7.2 we get the time bounds in Theorem 15, improving upon the bounds achieved in [89], see Table 7.1. We omit the details and refer the reader to [89].

[1] All logarithms in this paper are to the base two.

Model	[90] EREW	[88] EREW[2]	[89] CREW	[28] EREW	[94] Array	This paper CREW
FindMin	1	$\log\log n$	1	1	1	1
Insert	$\log\log n$	$\log\log n$	–	–	1	1
DeleteMin	$\log\log n$	$\log\log n$	–	–	1	1
Meld	–	$\log\log n$	$\log\frac{n}{k}+\log\log k$	$\log\log\frac{n}{k}+\log k$	–	1
Delete	–	$\log\log n$	–	–	–	1
DecreaseKey	–	$\log\log n$	–	–	–	1
Build	$\log n$	–	$\frac{n}{k}\log k$	$\log\frac{n}{k}\log k$	–	$\log n$
MultiInsert	–	–	$\log\frac{n}{k}+\log k$	$\log\log\frac{n}{k}+\log k$	–	$\log k$
MultiDelete	–	–	$\log\frac{n}{k}+\log\log k$	$\log\log\frac{n}{k}+\log k$	–	–

Table 7.1: Performance of different parallel implementations of priority queues.

Theorem 15 *On a CREW PRAM priority queues exist supporting* MultiInsert *in time* $O(\log k)$, MultiDelete *and* Meld *in time* $O(\log\log k)$, *and* Build *in time* $O(\log k+\log\frac{n}{k}\log\log k)$.

7.2 Meldable priority queues

In this section we describe how to implement the priority queue operations MakeQueue, Insert, Meld, FindMin and DeleteMin in constant time on a CREW PRAM with $O(\log n)$ processors. In Section 7.3 we describe how to extend the repertoire of priority queue operations to include Delete and DecreaseKey.

The priority queues in this section are based on heap ordered binomial trees [108]. Throughout this paper we assume a one to one mapping between tree nodes and priority queue elements.

Binomial trees are defined as follows. A binomial tree of *rank* zero is a single node. A binomial tree of rank $r>0$ is achieved from two binomial trees of rank $r-1$ by making one of the roots a child of the other root. It follows by induction that a binomial tree of rank r contains exactly 2^r nodes and that a node of rank r has exactly one child of each of the ranks $0,\ldots,r-1$. Throughout this section a tree denotes a heap ordered binomial tree.

A priority queue is represented by a forest of binomial trees. In the following we let the largest ranked tree be of rank $r(Q)$, we let $n_i(Q)$ denote the number of trees of rank i and we let $n_{\max}(Q)$ denote the value $\max_{0\le i\le r(Q)} n_i(Q)$. We require that a priority queue satisfies the constraints:

$\mathbf{A_1}$: $n_i(Q)\in\{1,2,3\}$ for $i=0,\ldots,r(Q)$, and

$\mathbf{A_2}$: the minimum root of rank i is smaller than all roots of rank larger than i.

It follows from A_2 that the minimum root of rank zero is the minimum element.

A priority queue is stored as follows. Each node v in a priority queue is represented by a record consisting of:

e : the element associated to v,

r : the rank of v, and

L : a linked list of the children of v in decreasing rank order.

[2]The operations Delete and DecreaseKey require the CREW PRAM and require amortized time $O(\log\log n)$.

```
Proc ParLink(Q)
  for p := 0 to log n − 1 pardo
    if n_p(Q) ≥ 3 then
      Link two trees from Q.L[p] \ min(Q.L[p]) and
      add the resulting tree to Q.L[p + 1]

Proc ParUnlink(Q)
  for p := 1 to log n pardo
    if n_p(Q) ≥ 1 then
      Unlink min(Q.L[p]) and add the resulting two trees to Q.L[p − 1]
```

Figure 7.1: Parallel linking and unlinking binomial trees.

```
Proc FindMin(Q)
  return min(Q.L[0])

Proc Insert(Q, e)
  Q.L[0] := Q.L[0] ∪ {new-node(e)}
  ParLink(Q)

Proc Meld(Q_1, Q_2)
  for p := 0 to log n pardo
    Q_1.L[p] := Q_1.L[p] ∪ Q_2.L[p]
  do 3 times ParLink(Q_1)

Proc MakeQueue
  Q :=new-queue
  for p := 0 to log n pardo Q.L[p] := ∅
  return Q

Proc DeleteMin(Q)
  e := min(Q.L[0])
  Q.L[0] := Q.L[0] \ {e}
  ParUnlink(Q)
  ParLink(Q)
  return e
```

Figure 7.2: CREW PRAM priority queue operations.

For each priority queue Q an array $Q.L$ is maintained of size $1 + \log n$ of pointers to linked lists of roots of equal rank. By A_1, $|Q.L[i]| \leq 3$ for all i. Notice that the chosen representation for storing the children of a node allows two nodes of equal rank to be linked in constant time by one processor. The required space for a priority queue is $O(n)$.

Two essential procedures used by our algorithms are the procedures ParLink and ParUnlink in Figure 7.1. In parallel ParLink for each rank i links two trees of rank i to one tree of rank $i+1$, if possible. By requiring that the trees of rank i that are linked together are different from $\min(Q.L[i])$, A_2 does not become violated. Let $n'_i(Q)$ denote the value of $n_i(Q)$ after performing ParLink. If $n_i(Q) \geq 3$ before performing ParLink then $n'_i(Q) \leq n_i(Q) - 2 + 1$, because processor i removes two trees of rank i and processor $i-1$ adds at most one tree of rank i. Otherwise $n'_i(Q) \leq n_i(Q) + 1$. This implies that $n'_{\max}(Q) \leq \max\{3, n_{\max}(Q) - 1\}$. The equality states that if the maximum number of trees of equal rank is larger than three, then an application of ParLink decreases this value by at least one. The procedure ParUnlink unlinks the minima of all $Q.L[i]$. All $n_i(Q)$ at most increase by one except for $n_0(Q)$ that can increase by two. Notice that the new minimum of $Q.L[i]$ is less than or equal to the old minimum of $Q.L[i+1]$. This implies that if A_2 is satisfied before performing ParUnlink then A_2 is also satisfied after the unlinking. Notice that ParLink and ParUnlink can be performed on an EREW PRAM with $O(\log n)$ processors in constant time if all processors know Q.

The priority queue operations can now be implemented as:

MakeQueue The list $Q.L$ is allocated and in parallel all $Q.L[i]$ are assigned the empty set.

Insert(Q, e) A new tree of rank zero containing e is created and added to $Q.L[0]$. To avoid $n_{\max}(Q) > 3$, ParLink(Q) is performed once.

MELD(Q_1, Q_2) First $Q_2.L$ is merged into $Q_1.L$ by letting processor p set $Q_1.L[p]$ to $Q_1.L[p] \cup Q_2.L[p]$. The resulting forest satisfies $n_{\max}(Q_1) \leq 6$. Performing PARLINK(Q_1) three times reestablishes A_1.

FINDMIN(Q) The minimum element in priority queue Q is $\min(Q.L[0])$.

DELETEMIN(Q) First the minimum element $\min(Q.L[0])$ is removed. Performing PARUNLINK once guarantees that A_2 is satisfied, especially that the new minimum element is contained in $Q.L[0]$, because the new minimum element was either already contained in $Q.L[0]$ or it was the minimum element in $Q.L[1]$. Finally PARLINK performed once reestablishes A_1.

A pseudo code implementation for a CREW PRAM based on the previous discussion is shown in Figure 7.2. Notice that the only part of the code requiring concurrent read is to "broadcast" the values of Q, Q_1 and Q_2 to all the processors. Otherwise the code only requires an EREW PRAM. From the fact that PARLINK and PARUNLINK can be performed in constant time with $O(\log n)$ processors we get:

Theorem 16 *On a CREW PRAM priority queues exist supporting* FINDMIN *in constant time with one processor, and* MAKEQUEUE, INSERT, MELD *and* DELETEMIN *in constant time with* $O(\log n)$ *processors.*

7.3 Priority queues with deletions

In this section we extend the repertoire of supported priority queue operations to include DELETE and DECREASEKEY. Notice that DECREASEKEY(Q, e, e') can be implemented as DELETE(Q, e) followed by INSERT(Q, e').

The priority queues in this section are based on heap ordered trees defined as follows. A rank zero tree is a single node. A rank r tree is a tree where the root has exactly five children of each of the ranks $0, 1, \ldots, r-1$. A tree of rank r can be created by linking six trees of rank $r-1$ by making the five larger roots children of the smallest root.

The efficiency we achieve for DELETE and DECREASEKEY is due to the concept of *holes*. A hole of rank r in a tree is a location in the tree where a child of rank r is missing.

We represent a priority queue by a forest of trees with holes. Let $r(Q), n_i(Q)$ and $n_{\max}(Q)$ be defined as in Section 7.2. We require that:

B_1 : $n_i(Q) \in \{1, 2, \ldots, 7\}$, for $i = 1, \ldots, r(Q)$,

B_2 : the minimum root of rank i is smaller than all roots of rank larger than i,

B_3 : at most two holes have equal rank.

Temporary while performing MELD we allow the number of holes of equal rank to be at most four. The requirement that a node of rank r has five children of each of the ranks $0, \ldots, r-1$ implies that at least one child of each rank is not replaced by a hole. This implies that the subtree rooted at a node has at least size 2^r and therefore the largest possible rank is at most $\log n$.

A priority queue is stored as follows. Each node v of a tree is represented by a record consisting of:

e : the element associated to v,

r : the rank of v,

f : a pointer to the parent of v, and

L : an array of size $\log n$ of pointers to linked lists of children of equal rank.

For each priority queue Q two arrays $Q.L$ and $Q.H$ are maintained of size $1+\log n$. $Q.L$ contains pointers to linked lists of trees of equal rank and $Q.H$ contains pointers to linked lists of "holes" of equal rank. More precisely $Q.H[i]$ is a linked list of nodes such that for each missing child of rank i of node v, v appears once in $Q.H[i]$. By B_1 and B_3, $|Q.L[i]| \leq 7$ and $|Q.H[i]| \leq 2$ for all i. Notice that the space required is $O(n \log n)$. By using worst case constant time extendible arrays to store the required arrays such that $|v.L| = v.r$, the space requirement can be reduced to $O(n)$. For simplicity we in the following assume that $|v.L| = \log n$ for all v.

The procedures ParLink and ParUnlink have to be modified such that linking and unlinking involves six trees and such that ParUnlink catches holes to be removed from $Q.H$. ParLink now satisfies $n'_{\max}(Q) \leq \max\{7, n_{\max}(Q) - 5\}$, and ParUnlink $n'_i(Q) \leq n_i(Q) + 5$ for $i > 0$ and $n'_0(Q) \leq n_0(Q) + 6$.

We now describe a procedure FixHoles that reduces the number of holes similar to how ParLink reduces the number of trees. The procedure is constructed such that processor p takes care of holes of rank p. The work done by processor p is the following. If $|Q.H[p]| < 2$ the processor does nothing. Otherwise it considers two holes in $Q.H[p]$. Recall that all holes have at least one real tree node of rank p as a brother. If the two holes have different parents, we swap one of the holes with a brother of the other hole. This makes both holes have the same parent f. By choosing the largest node among the two holes' brothers as the swap node we are guaranteed to satisfy heap order after the swap.

There are now two cases to consider. The first case is when the two holes have a brother b of rank $p+1$. Notice that b has at least three children of rank p because we allowed at most four holes of rank p. We can now cut off b and all children of b of rank p. By assigning b the rank p we only create one hole of rank $p+1$. We can now eliminate the two original holes by replacing them with two previous children of b. At most four trees remain to be added to $Q.L[p]$. The second case is when f has rank $p+1$. Assume first that $f \neq \min(Q.L[p+1])$. In this case the subtree rooted at f can be cut off without violating B_2. This creates a new hole of rank $p+1$. We can now cut off all children of f that have rank p and assign f the rank p. This eliminates the two holes. At most four trees now need to be added to $Q.L[p]$. Finally there is the case where $f = \min(Q.L[p+1])$. By performing ParUnlink and ParLink once the two holes disappear. To compensate for the created new trees we finally perform ParLink once.

The priority queue operations can now be implemented as follows.

MakeQueue Allocate a new priority queue Q and assign the empty set to all $Q.L[i]$ and $Q.H[i]$.

Insert(Q, e) Create a tree of rank zero containing e and add this tree to $Q.L[0]$. Perform ParLink(Q) once to reestablish B_1. Notice that Insert does not affect the number of holes in Q.

Meld(Q_1, Q_2) Merge $Q_2.L$ into $Q_1.L$, and $Q_2.H$ into $Q_1.H$. We now have $|Q_1.L| \leq 14$ and $|Q_1.H[i]| \leq 4$ for all i. That B_2 is satisfied follows from that Q_1 and Q_2 satisfied B_2. Performing ParLink(Q_1) twice followed by FixHoles(Q_2) twice reestablishes B_1 and B_3.

FindMin(Q) Return $\min(Q.L[0])$.

DeleteMin(Q) First perform FindMin and then perform Delete on the found minimum.

Delete(Q, e) Let v be the node containing e. Remove the subtree with root v. If this creates a hole then add the hole to $Q.H$. Merge $v.L$ into $Q.L$ and remove all appearances of v from

```
Proc MakeQueue
  Q :=new-queue
  for p := 0 to log n pardo
    Q.L[p], Q.H[p] := ∅
  return Q

Proc FindMin(Q)
  return min(Q.L[0])

Proc Insert(Q, e)
  Q.L[0] := Q.L[0] ∪ {new-node(e)}
  ParLink(Q)

Proc Meld(Q_1, Q_2)
  for p := 0 to log n pardo
    Q_1.L[p] := Q_1.L[p] ∪ Q_2.L[p]
    Q_1.H[p] := Q_1.H[p] ∪ Q_2.H[p]
  do 2 times ParLink(Q_1)
  do 2 times FixHoles(Q_1)

Proc DecreaseKey(Q, e, e')
  Delete(Q, e)
  Insert(Q, e')

Proc DeleteMin(Q)
  e := FindMin(Q)
  Delete(Q, e)
  return e

Proc Delete(Q, e)
  v := the node containing e
  if v.f ≠ null then
    Q.H[v.r] := Q.H[v.r] ∪ {v.f}
    v.f.L[v.r] := v.f.L[v.r] \ {v}
  for p := 0 to log n pardo
    for u ∈ v.L[p] do u.f := null
    Q.L[p] := Q.L[p] ∪ v.L[p]
    Q.H[p] := Q.H[p] \ {v}
  for p := 0 to log n pardo
    if n_p(Q) ≥ 1 and p > v.r then
      Q.H[p − 1] := Q.H[p − 1] \ min(Q.L[p])
      Unlink min(Q.L[p]) and
      add the resulting trees to Q.L[p − 1]
  do 2 times ParLink(Q)
  FixHoles(Q)
```

Figure 7.3: CREW PRAM priority queue operations.

$Q.H$. Notice that only for $i = v.r$, $\min(Q.L[i])$ can change and this only happens if e was $\min(Q.L[i])$. Unlinking $\min(Q.L[i])$ for $i = v.r+1, \ldots, r(Q)$ reestablishes B_2. Finally perform ParLink twice to reestablish B_1 and FixHoles once to reestablish B_3.

DecreaseKey(Q, e, e') Perform Delete(Q, e) followed by Insert(Q, e').

A pseudo code implementation for a CREW PRAM based on the previous discussion is shown in Figure 7.3. Notice that the only part of the code that requires concurrent read is the "broadcasting" of the parameters of the procedures and $v.r$ in Delete. The rest of the code does very local computing, in fact processor p only accesses entries p and $p \pm 1$ of arrays, and that these local computations can be done in constant time with $O(\log n)$ processors on an EREW PRAM.

Theorem 17 *On a CREW PRAM priority queues exist supporting* FindMin *in constant time with one processor, and* MakeQueue, Insert, Meld, DeleteMin, Delete *and* DecreaseKey *in constant time with* $O(\log n)$ *processors.*

7.4 Building priority queues

In this section we describe how to perform Build$(e_1, \ldots, e_n)$ for the priority queues in Section 7.3. Because our priority queues can report a minimum element in constant time and that there is lower bound of $\Omega(\log n)$ for finding the minimum of a set of elements on a CREW PRAM [66] we have an $\Omega(\log n)$ lower bound on the construction time on a CREW PRAM. We now give a matching upper bound on an EREW PRAM.

First a collection of trees is constructed satisfying B_1 and B_3 but not B_2. We partition the elements into $\lfloor (n-1)/6 \rfloor$ blocks of size six. In parallel we now construct a rank one tree from

each block. The remaining 1–6 elements are stored in $Q.L[0]$. The same block partitioning and linking is now done for the rank one trees. The remaining rank one trees are stored in $Q.L[1]$. This process continues until no tree remains. There are at most $O(\log n)$ iterations because each iteration reduces the number of trees by a factor six. The resulting forest satisfies B_1 and B_3. It is easy to see that the above construction can be done in time $O(\log n)$ with $O(n/\log n)$ processors on an EREW PRAM.

To establish B_2 we $\log n$ times perform ParUnlink followed by ParLink. By induction it follows that in the ith iteration all $Q.L[j]$ where $j \geq \log n - i$ satisfy B_2. This finishes the construction of the priority queue. The last step of the construction requires time $O(\log n)$ with $O(\log n)$ processors. We conclude that:

Theorem 18 *On an EREW PRAM a priority queue containing n elements can be constructed optimally with $O(n/\log n)$ processors in time $O(\log n)$.*

Because $\textsc{Meld}(Q, \textsc{Build}(e_1, \ldots, e_k))$ implements the priority queue operation MultiInsert $(Q, e_1, \ldots, e_k)$ we have the corollary below. Notice that k does not have to be fixed as in [28, 89].

Corollary 7 *On a CREW PRAM* MultiInsert *can be performed in time $O(\log k)$ with $O((\log n + k)/\log k)$ processors.*

7.5 Pipelined priority queue operations

The priority queues in Section 7.2, 7.3 and 7.4 require the CREW PRAM to achieve constant time per operation. In this section we address how to perform priority queue operations in a pipelined fashion. As a consequence we get an implementation of priority queues on a processor array of size $O(\log n)$ supporting priority queue operations in constant time. On a processor array we assume that all requests are entered at processor zero and that output is generated at processor zero too [94].

The basic idea is to represent a priority queue by a forest of heap ordered binomial trees as in Section 7.2, and to perform the operations sequentially in a loop that does constant work for each rank in increasing rank order. This approach then allows us to pipeline the operations. We require that a forest of binomial trees representing a priority queue satisfies:

$\mathbf{C_1}$: $n_i(Q) \in \{1, 2\}$, for $i = 1, \ldots, r(Q)$,

$\mathbf{C_2}$: the minimum root of rank i is smaller than all roots of rank larger than i.

Notice that C_1 is stronger than A_1 in Section 7.2. Sequential implementations of the priority queue operations are shown in Figure 7.4. We assume a similar representation as in Section 7.3. The pseudo code uses the following two procedures similar to those used in Section 7.2.

Link(Q, i) Links two trees from $Q.L[i] \setminus \min(Q.L[i])$ to one tree of rank $i+1$ that is added to $Q.L[i+1]$, provided $i \geq 0$ and $|Q.L[i]| \geq 3$.

Unlink(Q, i) Unlinks the tree $\min(Q.L[i])$ and adds the resulting two trees to $Q.L[i-1]$, provided $i \geq 1$ and $|Q.L[i]| \geq 1$.

Each of the priority queue operations can be viewed as running in steps $i = 0, \ldots, \log n$. Step i only accesses, creates and destroys nodes of rank i and $i+1$. Notice that requirement C_1 implies

```
Proc MakeQueue
  Q :=new-queue
  for p := 0 to log n do Q.L[p] := ∅
  return Q

Proc FindMin(Q)
  return min(Q.L[0])

Proc Insert(Q, e)
  Q.L[0] := Q.L[0] ∪ {new-node(e)}
  for i := 0 to log n do Link(Q, i)

Proc Meld(Q1, Q2)
  for i := 0 to log n do
    Q1.L[i] := Q1.L[i] ∪ Q2.L[i]
    do 2 times Link(Q1, i)

Proc DecreaseKey(Q, e, e')
  Delete(Q, e)
  Insert(Q, e')

Proc DeleteMin(Q)
  e := FindMin(Q)
  Delete(Q, e)
  return e

Proc Delete(Q, e)
  v := the node containing e
  for i := 0 to v.r − 1 do
    Move v.L[i] to Q.L[i]
    Link(Q, i)
  r, f := v.r, v.f
  Remove node v
  while f ≠ null do
    if f.r = r + 1 then
      f.r := f.r − 1
      Move f to Q.L[r] and
      f := f.f
    else
      Unlink f.L[r + 1] and add
      one tree to f.L[r] and
      one tree to Q.L[r]
    Link(Q, i)
    r := r + 1
  for i := r to log n do
    Unlink(Q, i + 1)
    do 2 times Link(Q, i)
```

Figure 7.4: A sequential implementation allowing pipelining.

that Meld only has to perform Link two times for each rank, whereas the implementation of Meld in Figure 7.2 has to do the corresponding linking three times. Otherwise the only interesting procedure is Delete. Procedure Delete proceeds in three phases. First all children of the node to be removed are cut off and moved to $Q.L$. In the second phase the hole created is eliminated by moving it up thru the tree by unlinking the brother node of the hole's current position or unlinking the parent node of the hole. Finally the third phase reestablishes C_2 in case phase two removed $\min(Q.L[i])$ for some i. This phase is similar to the last for loop in the implementation of Delete in Figure 7.3.

The pseudo code given in Figure 7.4 assumes the same representation for nodes as in Section 7.3. To implement the priority queues on a processors array a representation is required that is distributed among the processors. The canonical distribution is to let processor p store nodes of rank p.

The representation we distribute is the following. Assume that the children of a node are ordered from right-to-left in increasing rank order (this allows us to talk about the leftmost and rightmost children of a node). A node v is represented by a record with the fields:

e : the element associated to v,

r : the rank of v,

left, right : pointers to the left and right brothers of v,

leftmost-child : a pointer to the leftmost child of v,

f : a pointer to the parent of v, if v is the leftmost child. Otherwise null.

The array $Q.L$ is replaced by linked lists. Finally an array *rightmost-child* is maintained that for each node stores a pointer to the rank zero child of the node or to the node itself if it has rank zero. Notice that this representation only has pointers between nodes with rank difference at most one.

It is straightforward to modify the code given in Figure 7.4 to this new representation. The only essential difference is when performing DELETE. The first rank zero child of v to be moved to $Q.L$ is found by using the array *rightmost-child*. The succeeding children are found by using the *left* pointers.

On a processor array we let processor p store all nodes of rank p. In addition processor p stores $Q.L[p]$ for all priority queues Q. The array *rightmost-child* is stored at processor zero. The "locations" that DELETE and DECREASEKEY refer to are now not the nodes but the corresponding entries in the *rightmost-child* array.

With the above described representation step i of an operation only involves information stored at processors $\{i-1, i, i+1, i+2\}$ (processor $i-1$ and $i+2$ because back pointers have to be updated in the involved linked lists) that can be accessed in constant time. This immediately allows us to pipeline the operations, such that we for each new operation perform exactly four steps of each of the previous operations. Notice that no latency is involved in performing the queries: The answer to a FINDMIN query is known immediately.

Theorem 19 *On a processor array of size $O(\log n)$ each of the operations* MAKEQUEUE, INSERT, MELD, FINDMIN, DELETEMIN, DELETE *and* DECREASEKEY *can be supported in constant time.*

Chapter 8

A Parallel Priority Data Structure with Applications

A parallel priority data structure with applications*

Gerth Stølting Brodal†
BRICS‡, Department of Computer Science, University of Aarhus
Ny Munkegade, DK-8000 Århus C, Denmark
gerth@brics.dk

Jesper Larsson Träff Christos D. Zaroliagis
Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany
{traff,zaro}@mpi-sb.mpg.de

Abstract

We present a parallel priority data structure that improves the running time of certain algorithms for problems that lack a fast and work efficient parallel solution. As a main application, we give a parallel implementation of Dijkstra's algorithm for the single-source shortest path problem which runs in $O(n)$ time while performing $O(m \log n)$ work on a CREW PRAM. This is a logarithmic factor improvement for the running time compared with previous approaches. The main feature of our data structure is that the operations needed in each iteration of Dijkstra's algorithm can be supported in $O(1)$ time.

Keywords: Parallel algorithms, network optimization, graph algorithms, data structures.

8.1 Introduction

Developing work efficient parallel algorithms for graph and network optimization problems continues to be an important area of research in parallel computing. Despite much effort a number of basic problems have tenaciously resisted a very fast (*i.e.*, NC) parallel solution that is simultaneously work efficient. A notorious example is the single-source shortest path problem.

The best sequential algorithm for the single-source shortest path problem on directed graphs with non-negative real valued edge weights is Dijkstra's algorithm [37]. For a given digraph $G = (V, E)$ the algorithm iteratively steps through the set of vertices, in each iteration fixing the distance of a vertex for which a shortest path has been found, while maintaining in the process, for each of the remaining vertices, a tentative distance from the source. For an n-vertex, m-edge digraph the algorithm can be implemented to run in $O(m + n \log n)$ operations by using efficient priority queues like Fibonacci heaps [53] for maintaining tentative distances, or other priority queue implementations supporting deletion of the minimum key element in amortized or worst case logarithmic time, and decrease key in amortized or worst case constant time [18, 43, 63].

*This work was partially supported by the EU ESPRIT LTR Project No. 20244 (ALCOM-IT), and by the DFG project SFB 124-D6 (VLSI Entwurfsmethoden und Parallelität).

†Supported by the Danish Natural Science Research Council (Grant No. 9400044).

‡BRICS (Basic Research in Computer Science), a Centre of the Danish National Research Foundation.

The single-source shortest path problem is in NC (by virtue of the all-pairs shortest path problem being in NC), and thus a fast parallel algorithm exists, but for general digraphs no *work efficient* algorithm in NC is known. The best NC algorithm runs in $O(\log^2 n)$ time and performs $O(n^3(\log\log n/\log n)^{1/3})$ work on an EREW PRAM [61]. Moreover, work efficient algorithms which are (at least) sublinearly fast are also not known for general digraphs.

Dijkstra's algorithm is highly sequential, and can probably not be used as a basis for a fast (NC) parallel algorithm. However, it is easy to give a parallel implementation of the algorithm that runs in $O(n \log n)$ time [87]. The idea is to perform the distance updates within each iteration in parallel by associating a local priority queue with each processor. The vertex of minimum distance for the next iteration is determined (in parallel) as the minimum of the minima in the local priority queues. For this parallelization it is important that the priority queue operations have worst case running time, and therefore the original Fibonacci heap cannot be used to implement the local queues. This was first observed in [43] where a new data structure, called relaxed heaps, was developed to overcome this problem. Using relaxed heaps, an $O(n \log n)$ time and $O(m+n \log n)$ work(-optimal) parallel implementation of Dijkstra's algorithm is obtained. This seems to be the currently fastest work efficient parallel algorithm for the single-source shortest path problem. The parallel time spent in each iteration of the above implementation of Dijkstra's algorithm is determined by the (processor local) priority queue operations of finding a vertex of minimum distance and deleting an arbitrary vertex, plus the time to find and broadcast a global minimum among the local minima. Either or both of the priority queue operations take $O(\log n)$ time, as does the parallel minimum computation; for the latter $\Omega(\log n)$ time is required, even on a CREW PRAM. Hence, the approach with processor local priority queues does not seem to make it possible to improve the running time beyond $O(n \log n)$ without resorting to a more powerful PRAM model. This was considered in [87] where two faster (but not work efficient) implementations of Dijkstra's algorithm were given on a CRCW PRAM: the first (resp. second) algorithm runs in $O(n \log\log n)$ (resp. $O(n)$) time, and performs $O(n^2)$ (resp. $O(n^{2+\epsilon})$, $\forall 0 < \epsilon < 1$) work.

An alternative approach would be to use a parallel global priority queue supporting some form of multi-decrease key operation. Unfortunately, no known parallel priority queues support such an operation; they only support a multi-delete operation which assumes that the k elements to be deleted are the k elements with smallest priority in the priority queue (see *e.g.*, [17, 28, 35, 88, 89, 90, 94]). A different idea is required to improve upon the running time.

We present a parallel priority data structure that speeds up the parallel implementation of Dijkstra's algorithm, by supporting the operations required at each iteration in $O(1)$ time. Using this data structure we give an alternative implementation of Dijkstra's algorithm that runs in $O(n)$ time and performs $O(m \log n)$ work on a CREW PRAM. More specifically, by sorting the adjacency lists after weight it is possible in constant time both to determine a vertex of minimum distance, as well as to add in parallel any number of new vertices and/or update the distance of vertices maintained by the priority data structure. It should also be mentioned that the PRAM implementation of the data structure requires concurrent read only for broadcasting constant size information to all processors in constant time.

The idea of the parallel priority data structure is to perform a *pipelined merging* of keys. We illustrate the idea by first giving a simple implementation using a linear pipeline, which requires $O(n^2+m \log n)$ work (Section 8.2). We then show how the pipeline can be dynamically restructured in a tree like fashion and how to schedule the available processors over the tree such that only $O(m \log n)$ operations are required (Section 8.3). Further applications are discussed in Section 8.4.

8.2 A parallel priority data structure

In this section we introduce our new parallel priority data structure, and show how to use it to give an alternative, parallel implementation of Dijkstra's algorithm. Let $G = (V, E)$ be an n-vertex, m-edge directed graph with edge weights $c : E \to \mathbb{R}_0^+$, represented as a collection of adjacency lists. For a set $S \subseteq V$ of vertices, define $?(S)$ to be the neighbors of the vertices in S, excluding vertices in S, *i.e.*, $?(S) = \{w \in V \setminus S | \exists v \in S, (v, w) \in E\}$. We associate with each vertex $v \in S$ a (fixed) real valued label Δ_v. For a vertex $w \in ?(S)$, define the *distance from S to w* as $\text{dist}(S, w) = \min_{u \in S}\{\Delta_u + c(u, w)\}$. The distance has the property that $\text{dist}(S \cup \{v\}, w) = \min\{\text{dist}(S, w), \Delta_v + c(v, w)\}$. We define the *vertex closest to S* to be the vertex $z \in ?(S)$ that attains the minimum $\min_{w \in \Gamma(S)}\{\text{dist}(S, w)\}$ (with ties broken arbitrarily).

Assume that a processor P_v is associated with each vertex $v \in V$ of G. Among the processors associated with vertices in S at any given instant one will be designated as the *master processor*. Our data structure supports the following four operations:

- INIT: initializes the priority data structure.
- EJECT(S): deletes the vertex v of $?(S)$ that is closest to S, and returns the pair (v, D_v) to the master processor, where $D_v = \text{dist}(S, v)$.
- EXTEND(S, v, Δ, P_v): adds a vertex v associated with processor P_v to S, and assigns it label Δ. Processor P_v becomes the new master processor.
- EMPTY(S): returns **true** to the master processor of S if $?(S) = \emptyset$.

Performing $|?(S)|$ successive EJECT-operations on a set S ejects the vertices in $?(S)$ in non-decreasing order of closeness, and leaves the priority data structure empty. Each vertex of $?(S)$ is ejected once. Note also that there is no operation to change the labels associated with vertices in S.

These operations suffice for an alternative, parallel implementation of Dijkstra's algorithm. Let $s \in V$ be a distinguished *source vertex*. The algorithm computes for each vertex $v \in V$ the length of a shortest path from s to v, where the length of a path is the sum of the weights of the edges on the path. Dijkstra's algorithm maintains a set S of vertices for which a shortest path have been found, in each iteration adding one more vertex to S. Each vertex $w \in V \setminus S$ has a tentative distance which is equal to $\text{dist}(S, w)$ as defined above. Hence, instead of the usual priority queue with DELETEMIN to select the vertex closest to S, and DECREASEKEY operations to update tentative distances for the vertices in $V \setminus S$, we use the priority data structure above to determine in each iteration a vertex closest to the current set S of correct vertices. The EXTEND-operation replaces the updating of tentative distances. Let P_v be the processor associated with vertex v.

Our main result in this section is that the New-Parallel-Dijkstra algorithm runs in linear time in parallel.

Theorem 20 *Dijkstra's algorithm can be implemented to run in $O(n)$ time and $O(n^2 + m \log n)$ work using $O(n + m)$ space on a CREW PRAM.*

The proof of Theorem 20 is based on the following lemma. The space bound of Theorem 20 is discussed at the end of this section.

Lemma 10 *Operation* INIT *takes $O(m \log n)$ work and $O(\log n)$ time. After initialization, each* EJECT(S)*-operation takes constant time using $|S|$ processors, and each* EXTEND(S, v, Δ, P_v)*-operation takes constant time using $|S| + \deg_{\text{in}}(v)$ processors, where $\deg_{\text{in}}(v)$ is the in-degree of v. The* EMPTY(S)*-operation takes constant time per processor. The space required per processor is $O(n)$.*

Algorithm New-Parallel-Dijkstra
/* Initialization */
INIT; $d(s) \leftarrow 0$; $S \leftarrow \emptyset$;
EXTEND$(S, s, d(s), P_s)$;
/* Main loop */
while $\neg$EMPTY(S) **do**
    $(v, D_v) \leftarrow$ EJECT(S); /* instead of DELETEMIN */
    $d(v) \leftarrow D_v$;
    EXTEND$(S, v, d(v), P_v)$;
    /* replaces the update step */
od

Figure 8.1: The $O(n)$ time parallel Dijkstra algorithm.

The remainder of this section will be devoted to provide a proof for Lemma 10.

In the INIT-operation the adjacency lists of G are sorted in non-decreasing order after edge weight, *i.e.*, on the adjacency list of v vertex w_1 appears before w_2 if $c(v, w_1) \leq c(v, w_2)$ (with ties broken arbitrarily). The adjacency lists are assumed to be implemented as doubly linked lists, such that any vertex w on v's adjacency list can be removed in constant time. For each vertex v we also associate an array of vertices u_j to which v is adjacent, *i.e.*, vertices u_j for which $(u_j, v) \in E$. In the array of v we store for each such u_j a pointer to the position of v in the adjacency list of u_j. This enables us to delete all occurrences of v in adjacency lists of such vertices $u_j \in \overline{S} = V \setminus S$ in constant time. Sorting of the adjacency lists takes $O(\log n)$ time and $O(m \log n)$ work [29]. Constructing links and building the required arrays can then be done in constant time using $O(m)$ operations. This completes the description of the INIT operation.

The processors associated with vertices in S at any given instant are organized in a linear pipeline. Let v_i be the ith vertex added to S, let S_i denote S after the ith EXTEND(S, v_i, Δ_i, P_i)-operation where Δ_i is the label to be associated with v_i, and let P_i be the processor assigned to v_i (in the implementation of Dijkstra's algorithm the label Δ_i to be associated with vertex v_i was $d(v)$). Let finally L_i be the sorted, doubly linked adjacency list of v_i. Processor P_i which was assigned at the ith EXTEND-operation receives input from P_{i-1}, and, after the $(i+1)$th EXTEND-operation, will send output to P_{i+1}. The last processor assigned to S will be the master processor, and the output from this processor will be the result of the next EJECT-operation, *i.e.*, the vertex closest to S. The pipeline for $i = 4$ is shown in Figure 8.2. The input queue Q_1 of processor P_1 is empty and not shown.

Assume now that EJECT(S_{i-1}) can be performed in constant time by the processors assigned to the vertices in S_{i-1}, and returns to the master processor of S_{i-1} the vertex in $?(S_{i-1})$ that is closest to S_{i-1}. We show how to maintain this property after an EXTEND-operation; more specifically, that the vertex v ejected by EJECT(S_i), immediately after EXTEND$(S_{i-1}, v_i, \Delta_i, P_i)$, is produced in constant time, is indeed the vertex closest to S_i, and that each vertex in $?(S_i)$ is ejected exactly once.

Performing an EJECT(S_{i-1}) returns the vertex u closest to S_{i-1} with value $D_u = \text{dist}(S_{i-1}, u)$. Either this vertex, or the vertex closest to v_i is the vertex to be ejected from S_i. Let w be the first vertex on the sorted adjacency list L_i. If $\Delta_i + c(v_i, w) \leq D_u$, then the result of EJECT(S_i) is w with value $D_w = \Delta_i + c(v_i, w)$; otherwise, the result is u with value D_u. In the first case, w

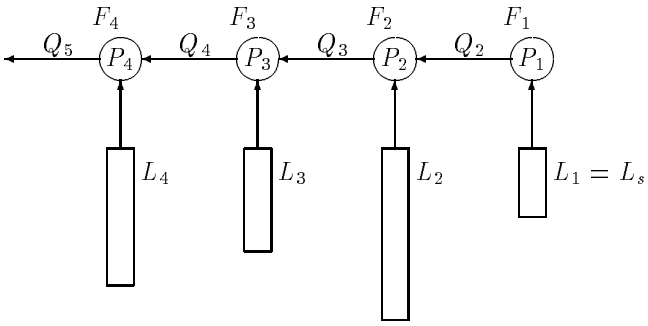

Figure 8.2: The linear processor pipeline with the associated data structures.

Function EJECT(S)
for all $v_i \in S$ **do in parallel**
    /* processor P_i is associated with vertex v_i */
    $(v', D') \leftarrow$ HEAD(Q_i);
    $v'' \leftarrow$ HEAD(L_i); $D'' \leftarrow c(v_i, v'') + \Delta_i$;
    if $D'' < D'$ **then** $(v', D') \leftarrow (v'', D'')$ **fi**;
    remove v' from L_i and Q_i if present;
    insert v' into F_i;
    if $v' \notin F_{i+1}$ **then** append (v', D') to Q_{i+1} **fi**
od;
if P_i is the master processor **return** HEAD(Q_{i+1})

Figure 8.3: The EJECT-operation.

is ejected and simply removed from L_i, but the ejected vertex of S_{i-1} must be saved for a later EJECT-operation. For this purpose we associate an *input queue* Q_i with each P_i which stores the vertices ejected from S_{i-1} by processor P_{i-1}. The EJECT-operation of P_i thus consists in selecting the smaller value from either the input queue Q_i or the adjacency list L_i of v_i. In other words, P_i performs one merging step of the two ordered lists Q_i and L_i. In case P_i exhausts its own adjacency list L_i, it always ejects from Q_i. It must be shown that Q_i never gets empty, unless all vertices of $?(S_{i-1})$ have been ejected, in which case processor P_i may terminate. The EMPTY(S_i) thus has to return **true** when both adjacency list L_i and input queue Q_i of the master processor are empty.

In order to ensure that a vertex output by P_i is never output at a later EJECT-operation (*i.e.*, inserted into Q_{i+1} with different priorities), we associate a set F_i of *forbidden* vertices with each P_i. Each F_i set is implemented as a Boolean array (*i.e.*, $F_i[w] =$ **true** if and only if w has been ejected from L_i). When a vertex w is removed from L_i and ejected, w is put into F_i and removed from Q_i (if it is there). A vertex ejected from S_{i-1} is only put into the input queue Q_i of P_i if it is *not* in the forbidden set F_i of P_i. In the case where a vertex u at the head of Q_i (previously ejected from S_{i-1}) "wins" at P_i and is ejected, it is removed from L_i (in case u is adjacent to v_i), and is made forbidden for P_i by putting it into F_i. In order to be able to remove vertices from Q_i in constant time, each P_i has an array of pointers into Q_i, which is updated whenever P_{i-1} outputs a vertex into Q_i. The EJECT-operation is shown in Figure 8.3.

Function EXTEND(S, v, Δ, P)
connect the master processor of S to P;
make P the (new) master processor;
$(u, D') \leftarrow$ EJECT(S);
append (u, D') to the input queue Q of P;
$\Delta_v \leftarrow \Delta$; $S \leftarrow S \cup \{v\}$;
remove v from $\overline{S}$ using pointers constructed by INIT

Figure 8.4: The EXTEND-operation.

An EXTEND$(S_{i-1}, v_i, \Delta_i, P_i)$-operation must first perform an EJECT(S_{i-1}) in order to get an element into the input queue Q_i of P_i. Since we must prevent that a vertex already in S is ever ejected (as $?(S)$ excludes S), once a vertex is appended to S it must be removed from the adjacency lists of all vertices in $\overline{S}$. This can be done in parallel in constant time using the array of pointers constructed by the INIT-operation (since v occurs at most once in any adjacency list), if concurrent read is allowed: a pointer to the array of vertices u_j to which v is adjacent must be made available to all processors. In parallel they remove v from the adjacency lists of the u_j's, which takes constant time using $\deg_{\text{in}}(v)$ processors, $\deg_{\text{in}}(v)$ being the *in-degree* of v. The required concurrent read is of the restricted sort of broadcasting the same constant size information to all processors. The EXTEND-operation is shown in Figure 8.4.

We now show that each input queue $Q_i \neq \emptyset$ unless there are no more vertices to be ejected from $?(S_{i-1})$. We argue that it always holds that $|Q_i| > |F_i \setminus F_{i-1}| \geq 0$, and that Q_i always contains different vertices. The last claim follows by induction. Assume namely that EJECT(S_{i-1}) produces each vertex in $?(S_{i-1})$ once. Whenever a vertex from $?(S_{i-1})$ is ejected by P_i it is removed from L_i, and hence by induction can never occur again; on the other hand, whenever a vertex is ejected from L_i it is removed from Q_i and also put into the forbidden set F_i which prevents it from entering Q_i at any future EJECT-operation on S_{i-1}. After each new EXTEND$(S_{i-1}, v_i, \Delta_i, P_i)$ the queue of P_i is not empty since the EXTEND-operation first performs an EJECT(S_{i-1}). It remains to show that as long as there are still vertices in $?(S_{i-1})$ the invariant that $|Q_i| > |F_i \setminus F_{i-1}|$ holds by all subsequent EJECT-operations. Consider the work of P_i at some EJECT-operation. Either $|F_i \setminus F_{i-1}|$ is increased by one, or $|Q_i|$ is decreased by one, but not both since in the case where P_i outputs a vertex from Q_i this vertex has been put into F_{i-1} at some previous operation, and in the case where P_i outputs a vertex from L_i which was also in Q_i, this vertex has again been put into F_{i-1} at some previous operation. In both cases $|F_i \setminus F_{i-1}|$ does not change when P_i puts the vertex into F_i. The operation of P_i therefore maintains $|Q_i| \geq |F_i \setminus F_{i-1}|$; inequality is reestablished by considering the work of P_{i-1} which either increases $|Q_i|$ or, in the case where P_{i-1} is not allowed to put its ejected vertex u into Q_i (because $u \in F_i$), decreases $|F_i \setminus F_{i-1}|$ (because u is inserted into F_{i-1}).

The $O(n^2)$ space due to the forbidden sets and the arrays of pointers into the input queues can actually be reduced to $O(n+m)$. The idea is to maintain for each vertex $v \in V$ a doubly linked list of occurrences of v in the priority data structure in such a way that processor P_{i-1} can still determine in constant time whether $v \in F_i$, and such that P_i can in constant time remove v from Q_i when it is ejected from L_i. We do this as follows. We maintain an array of size n of pointers, which for each each $v \in V$ points to the occurrence of v in the priority data structure which is closest to the master processor (*i.e.*, the highest index i for which either $v \in L_i$ or $v \in Q_i$). Each v can occur in either an L_j list of some processor P_j or an input queue $Q_{j'}$ of some other processor

$P_{j'}$, where $j' > j$; furthermore a vertex v in $Q_{j'}$ is marked as forbidden for all processors $P_j, \ldots, P_{j'}$, assuming that v was at some instant ejected from the adjacency list L_j of P_j. Each occurrence of v has information recording whether it is (still) in a list L_j or whether it has been moved to some queue $Q_{j'}$, in which case it is also recorded that this occurrence of v is forbidden from processor P_j. If v occurs in some $Q_{j'}$, the position in the queue is also kept such that v can eventually be removed from $Q_{j'}$ in constant time. Obviously, this information takes only constant space. For each occurrence of v there is a pointer to the next occurrence of v closer to the master processor (unless this v is the closest occurrence), and a pointer to the previous occurrence (unless this v is the first occurrence).

We can now implement the EJECT-operation for processor P_j as follows. This processor must take the minimum vertex from either Q_j or L_j, say v, and eject it to Q_{j+1}, unless it is forbidden by P_{j+1}. To check this, P_j looks at the next occurrence of v. If this is an occurrence in some queue $Q_{j'}$, then P_j can immediately see if v was forbidden from P_{j+1} (and onwards), in which case v is not ejected; instead, the next occurrence of v is marked as forbidden from the processor P_j, from which v originated, if v was taken from L_j, or some earlier processor if v was taken from Q_j. This occurrence of v is then removed from the doubly linked list of v's occurrences. If v is not forbidden for processor P_{j+1}, v is put into Q_{j+1}, marked as forbidden from P_j and its position in Q_{j+1} is recorded. If v was taken from L_j, it has to be checked if an occurrence of v is also in the input queue Q_j of P_j, in which case it has to be removed from Q_j. This can be done by looking at the previous occurrence of v, which records where this previous occurrence can be found. In case an occurrence of v was present and is removed from Q_j, this occurrence is unlinked from the occurrence list, updating the Q_{j+1} occurrence of v as forbidden from some previous processor. If v was instead taken from Q_j, a possible occurrence of v in L_j must be taken out of the occurrence list.

The implementation of the EXTEND-operation is simple. Upon a call EXTEND(S, v, Δ, P) all vertices adjacent to v just have to be linked up in their respective occurrence lists, which can be done in constant time in parallel using the aforementioned array of pointers to the closest occurrence. We have thus dispensed with all of the linear-sized arrays used in the previous implementation at the expense of only one array of size $O(n)$.

This concludes the proof of Lemma 10, Theorem 20 and the basic implementation of the priority data structure.

Note that the n^2 term in the work comes from the fact that once a processor is assigned to the data structure it resides there until the end of the computation, even if its own adjacency list gets empty. In order to reduce the work we need a way of removing processors whose adjacency list has become empty.

8.3 A dynamic tree pipeline

We now describe how to decrease the amount of work required by the algorithm in Section 8.2. Before doing so, we first make an observation about the merging part of the algorithm. The work done by processor P_i is intuitively to output vertices by incrementally merging its adjacency list L_i with the incoming stream Q_i of vertices output by processor P_{i-1}. Processor P_i terminates when it has nothing left to merge. An alternative bound on the real work done by this algorithm is then the sum of the distance each vertex v from an adjacency list L_i travels, where the distance is the number of processors that output v. Because each vertex v from L_i can at most be output by a prefix of the processors $P_i, P_{i+1}, \ldots, P_n$, the distance v travels is at most $n - i + 1$. This gives a total bound on the work done by the processors of $O(mn)$. That the real work can actually be

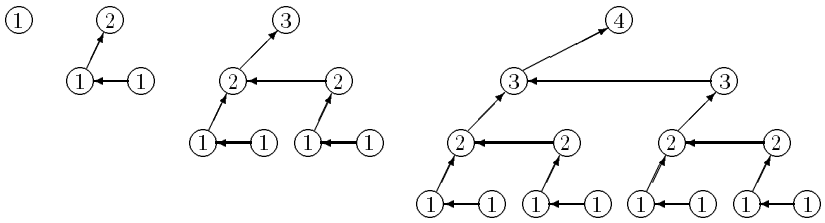

Figure 8.5: The tree arrangement of processors. Numbers denote processor ranks.

bounded by $O(n^2)$ is due to the fact that vertices get annihilated by forbidden sets.

Using this view of the work done by the algorithm during merging, we now describe a variation of the data structure that basically bounds the distance a vertex can travel by $O(\log n)$, *i.e.*, bounds the work by $O(m \log n)$. The main idea is to replace the sequential pipeline of processors by a binary tree pipeline of processors of height $O(\log n)$. What we prove is:

Theorem 21 *Dijkstra's algorithm can be implemented to run in $O(n)$ time and $O(m \log n)$ work on a CREW PRAM.*

We first describe how to arrange the processors in a tree and how to dynamically change this tree while adding new processors for each EXTEND-operation. We then describe how the work done by the processors can be bounded by $O(m \log n)$ and finally how to perform the required processor scheduling.

8.3.1 Tree structured processor connections

To arrange the processors in a tree we modify slightly the information stored at each processor. Each processor P_i still maintains an adjacency list L_i and a set of forbidden vertices F_i. The output of processor P_i is still inserted into an input queue of a processor P_j, but P_i can now receive input from two processors instead of one.

The basic organization of the processor connections are perfect binary trees. Each node corresponds to a processor and the unique outgoing edge of a node corresponds to the output queue of the node (and an input queue to the successor node). The rank of a node is the height of the node in the perfect binary tree and the rank of a tree is the rank of the root. The nodes are connected such that the incoming edges of a node v come from the left child of v and the sibling of v. Figure 8.5 shows trees of size 1, 3, 7 and 15 (processor local information is omitted). A tree of rank $r+1$ can be constructed from two trees of rank r plus a single node, by connecting the two roots with the new node. By induction a tree of rank r has size $2^r - 1$.

The processors are organized in a sequence of trees of rank $r_k, r_{k-1} \ldots, r_1$, where the ith root is connected to the $i+1$st root (see Figure 8.6). We maintain the invariant that

$$r_k \leq r_{k-1} < r_{k-2} < \cdots < r_2 < r_1. \tag{8.1}$$

When performing an EXTEND-operation a new processor is initialized. If $r_k < r_{k-1}$ the new processor is inserted as a new rank one tree at the front of the list of trees (as in the sequential pipeline case). That (8.1) is satisfied follows from $1 \leq r_k < r_{k-1} < \cdots < r_1$. If $r_k = r_{k-1}$ we link the kth and $k-1$st tree with the new node to form a tree of rank $1 + r_{k-1}$. That (8.1) is satisfied follows from $1 + r_{k-1} \leq r_{k-2} < r_{k-3} < \cdots < r_1$. Figure 8.6 illustrates the relinking for the case

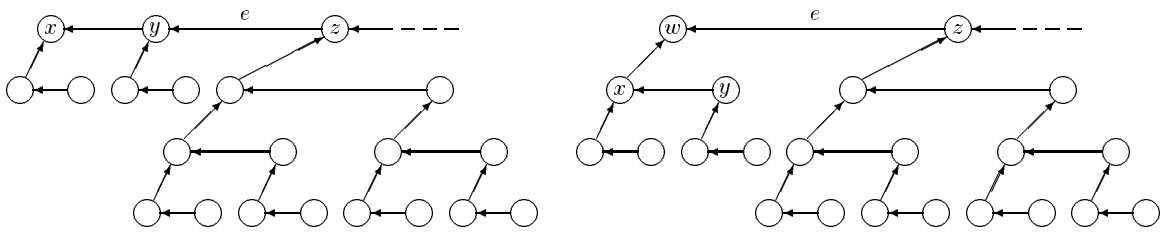

Figure 8.6: How to restructure the tree when performing EXTEND.

where $r_k = r_{k-1} = 2$ and $r_{k-2} = 4$. Note that the only restructuring required is to make e an incoming edge of the new node w.

The described approach for relinking has recently been applied in a different context to construct purely functional random-access lists [83]. In [83] it is proved that a sequence of trees satisfying (8.1) is unique for a given number of nodes.

8.3.2 A work efficient implementation

In the following we let the output queue of processor P_i be denoted $Q_{\text{out}(i)}$. Compared to the sequential pipeline, processor P_i now only outputs a subset of $?\,(S_i)$ due to the dynamic relinking. For the tree pipeline we basically only have to prove that all non-terminated processors have the next vertex to output in one of its input queues. Let P_j be a processor connected to a processor P_i, $i > j$, by the queue $Q_{\text{out}(j)}$. Let J_j denote the set of vertices ejected between the creation of P_j and P_i (excluding calls to EJECT internal to EXTEND). Our main invariant is

$$|(F_i \cup J_j) \setminus F_j| < |F_j \setminus (F_i \cup J_j)|. \tag{8.2}$$

The important observation is that J_j are the vertices that can be output by P_j but are illegal as input to P_i, because they already have been ejected prior to the creation of P_i. To guarantee that $Q_{\text{out}(j)}$ does not contain any illegal input to P_i we maintain the invariant

$$|Q_{\text{out}(j)} \cap (F_i \cup J_j)| = \emptyset. \tag{8.3}$$

We now describe how to implement the EJECT-operation such that the invariants (8.2) and (8.3) remain satisfied. The implementation is basically the same as for the sequential pipeline. Processor P_j first selects the vertex v with least priority in L_j and the input queues of P_j in constant time. Then all occurrences of v is removed from L_j and the input queues of P_j, and v is added to F_j. If $Q_{\text{out}(j)}$ is an input queue of P_i and $v \notin F_i \cup J_j$, then v is inserted in $Q_{\text{out}(j)}$. That (8.2) and (8.3) are satisfied after an EJECT-operation follows by the same arguments as for the sequential pipeline.

Invariant (8.2) allows $Q_{\text{out}(j)}$ to be empty throughout an EJECT-operation (without P_j being terminated) because $F_j \setminus (F_i \cup J_j) \neq \emptyset$ implies that there exists a vertex v that has been output by P_j that neither has been ejected from the data structure before P_i was created nor has been output by P_i (yet). Because $Q_{\text{out}(j)}$ is assumed to be empty it is easily seen that v can only be stored in an output queue of a processor in the subtree rooted at P_i due to how the dynamic relinking is performed, *i.e.*, v appears in a $Q_{\text{out}(k)}$, $k \in \{j+1, \ldots, i-1\}$. It follows that v has to be output by P_i (perhaps with a smaller priority because v gets annihilated by an appearance of v with less priority) before the next vertex to be output by P_j can be output by P_i. This means that P_i can

safely skip to consider input from the empty input queue $Q_{\text{out}(j)}$, even if $Q_{\text{out}(j)}$ later can become nonempty. Note that (8.2) guarantees that a queue between P_{i-1} and P_i always is nonempty.

We now describe how to implement the EXTEND-operation. The implementation is as for the sequential pipeline case, except for the dynamic relinking of a connection (edge e in Figure 8.6). Assume that P_i is the newly created processor. That $Q_{\text{out}(i-1)}$ satisfies (8.2) and (8.3) follows from the fact that $J_{i-1} \subset F_{i-1}$ ($|J_{i-1}| + 1 = |F_{i-1}|$) and $F_i = \emptyset$. What remains to be shown is how to satisfy the invariants for the node P_j when $Q_{\text{out}(j)}$, $j < i$, is relinked to become an input queue of P_i.

When $Q_{\text{out}(j)}$ is relinked, P_j has totally output $|J_j| + i - j$ vertices ($|J_j|$ for EJECT-operations and $i - j$ for EXTEND-operation). Because $F_i = \emptyset$ and $i > j$ it follows that (8.2) is satisfied after the relinking. To guarantee that (8.3) is satisfied we *just* have to perform the following updating of $Q_{\text{out}(j)}$

$$Q_{\text{out}(j)} \leftarrow Q_{\text{out}(j)} \setminus J_j.$$

Since $Q_{\text{out}(j)}$ and J_j can be arbitrary sets it seems hard to do this updating in constant time without some kind of precomputation. Note that the only connections that can be relinked is the connections between the tree roots. The approach we take is that for each EJECT-operation, we mark the ejected vertex v as "dirty" in all the output queues $Q_{\text{out}(j)}$ where P_j is a root. Whenever a queue $Q_{\text{out}(j)}$ is relinked we just need to be able to delete all vertices marked dirty from $Q_{\text{out}(j)}$ in constant time. When inserting a new vertex into a queue $Q_{\text{out}(j)}$ it can easily be checked if it is dirty or not.

A reasonably simple solution to the above problem is the following. Note that each time $Q_{\text{out}(j)}$ is relinked it is connected to a node having rank one higher, *i.e.*, we can use this rank to count the number of delete-dirty operations or as a time stamp t. We represent a queue as a sequence of vertices where each vertex v has two time stamped links to vertices in each direction from v. The link with the highest time stamp $\leq t$ is the current link in a direction. A link with time stamp $t + 1$ is a link that will become active when a delete-dirty command is performed. We ommit the implementation details of the marking procedure.[1]

That the real work done by the processors is $O(m \log n)$ follows from the following argument. Vertices can at most travel a distance of $2 \log n$ in a tree (in the sense mentioned in the beginning of this section) before they reach the root of the tree. The problem is that the root processors move vertices to lower ranked vertices, but the total distance to travel increases at most by $2 \log n$ for each EJECT-operation, because the increase in total distance to travel along the root path results in a telescoping sum that is bounded by $2 \log n$. Because there are $2n$ calls to EJECT by Dijkstra's algorithm (n internal to EXTEND), we conclude that the actual merging work is bounded by $O(2m \log n + 4n \log n)$, *i.e.*, $O(m \log n)$.

8.3.3 Processor scheduling

What remains is to divide the $O(m \log n)$ work among the available processors. Assuming that $O(\frac{m \log n}{n})$ processors are available, the idea is to simulate the tree structured pipeline for $O(\log n)$ time steps, after which we stop the simulation and in $O(\log n)$ time eliminate the (simulated) terminated processors, and reschedule. By this scheme a terminated processor is kept alive for only $O(\log n)$ time steps, and hence no superfluous work is done. In total the simulation takes linear time.

[1] As described here the marking of dirty vertices requires concurrent read to know the ejected vertex, but by pipelining the dirty marking process along the tree roots, concurrent read can be avoided in this part of the construction.

8.4 Further applications

The improved single-source shortest path algorithm immediately gives rise to corresponding improvements in algorithms in which the single-source shortest path problem occurs as a subproblem. We mention here the assignment problem, the minimum cost flow problem, (for definitions see [3]), and the single-source shortest path problem in planar digraphs.

The minimum cost flow problem (which is P-complete [57]) can be solved by $O(m \log n)$ calls to Dijkstra's algorithm (see *e.g.* [3, Section 10.7]). Using our implementation, we obtain a parallel algorithm that runs in $O(nm \log n)$ time and performs $O(m^2 \log^2 n)$ work. The assignment problem can be solved by n calls to Dijkstra's algorithm (see *e.g.* [3, Section 12.4]). Using our implementation, we obtain a parallel algorithm that runs in $O(n^2)$ time and performs $O(nm \log n)$ work. The assignment problem is not known to be in NC, but an RNC algorithm exists for the special case of unary weights [81, 69], and a weakly polynomial CRCW PRAM algorithm exists that that runs in $O(n^{2/3} \log^2 n \log(nC))$ time and performs $O(n^{11/3} \log^2 n \log(nC))$ work for the case of integer weights in the range $[-C, C]$ [56]. Our bounds are strongly polynomial and speed up the best previous ones [43] by a logarithmic factor.

Greater parallelism for the single-source shortest path problem in the case of planar digraphs can be achieved by plugging our implementation of Dijkstra's algorithm into the algorithm of [103] resulting in an algorithm which runs $O(n^{2\epsilon} + n^{1-\epsilon})$ time and performs $O(n^{1+\epsilon})$ work on a CREW PRAM. With respect to work, this gives the best (deterministic) parallel algorithm known for the single-source shortest path problem in planar digraphs that runs in sublinear time.

Chapter 9

Predecessor Queries in Dynamic Integer Sets

Predecessor Queries in Dynamic Integer Sets

Gerth Stølting Brodal*
BRICS†, Department of Computer Science, University of Aarhus
Ny Munkegade, DK-8000 Århus C, Denmark
gerth@brics.dk

Abstract

We consider the problem of maintaining a set of n integers in the range $0..2^w - 1$ under the operations of insertion, deletion, predecessor queries, minimum queries and maximum queries on a unit cost RAM with word size w bits. Let $f(n)$ be an arbitrary nondecreasing smooth function satisfying $\log\log n \leq f(n) \leq \sqrt{\log n}$. A data structure is presented supporting insertions and deletions in worst case $O(f(n))$ time, predecessor queries in worst case $O((\log n)/f(n))$ time and minimum and maximum queries in worst case constant time. The required space is $O(n2^{\epsilon w})$ for an arbitrary constant $\epsilon > 0$. The RAM operations used are addition, arbitrary left and right bit shifts and bit-wise boolean operations. The data structure is the first supporting predecessor queries in worst case $O(\log n/\log\log n)$ time while having worst case $O(\log\log n)$ update time.

Category: E.1, F.2.2
Keywords: searching, dictionaries, priority queues, RAM model, worst case complexity

9.1 Introduction

We consider the problem of maintaining a set S of size n under the operations:

INSERT(e) inserts element e into S,

DELETE(e) deletes element e from S,

PRED(e) returns the largest element $\leq e$ in S, and

FINDMIN/FINDMAX returns the minimum/maximum element in S.

In the comparison model INSERT, DELETE and PRED can be supported in worst case $O(\log n)$ time and FINDMIN and FINDMAX in worst case constant time by a balanced search tree, say an (a,b)-tree [64]. For the comparison model a tradeoff between the operations has been shown by Brodal *et al.* [20]. The tradeoff shown in [20] is that if INSERT and DELETE take worst case $O(t(n))$ time then FINDMIN (and FINDMAX) requires at least worst case $n/2^{O(t(n))}$ time. Because predecessor queries can be used to answer member queries, minimum queries and maximum queries, PRED requires worst case $\max\{\Omega(\log n), n/2^{O(t(n))}\}$ time. For the sake of completeness we mention that matching upper bounds can be achieved by a $(2,4)$-tree of depth at most $t(n)$ where each leaf stores $\Theta(n/2^{t(n)})$ elements, provided DELETE takes a pointer to the element to be deleted.

*Supported by the Danish Natural Science Research Council (Grant No. 9400044). Partially supported by the ESPRIT Long Term Research Program of the EU under contract #20244 (ALCOM-IT).
†BRICS (Basic Research in Computer Science), a Centre of the Danish National Research Foundation.

In the following we consider the problem on a unit cost RAM with word size w bits allowing addition, arbitrary left and right bit shifts and bit-wise boolean operations on words in constant time. Miltersen [78] refers to this model as a *Practical RAM*. We assume the elements are integers in the range $0..2^w - 1$. A tradeoff similar to the one for the comparison model [20] is not known for a Practical RAM.

A data structure of van Emde Boas *et al.* [104, 106] supports the operations INSERT, DELETE, PRED, FINDMIN and FINDMAX on a Practical RAM in worst case $O(\log w)$ time. For word size $\log^{O(1)} n$ this implies an $O(\log \log n)$ time implementation.

Thorup [102] recently presented a priority queue supporting INSERT and DELETEMIN in worst case $O(\log \log n)$ time independently of the word size w. Thorup notes that by tabulating the multiplicity of each of the inserted elements the construction supports DELETE in amortized $O(\log \log n)$ time by skipping extracted integers of multiplicity zero. The data structure of Thorup does not support predecessor queries but Thorup mentions that an $\Omega(\log^{1/3-o(1)} n)$ lower bound for PRED can be extracted from [77, 79]. The space requirement of Thorup's data structure is $O(n2^{\epsilon w})$ (if the time bounds are amortized the space requirement is $O(n + 2^{\epsilon w})$).

Andersson [5] has presented a Practical RAM implementation supporting insertions, deletions and predecessor queries in worst case $O(\sqrt{\log n})$ time and minimum and maximum queries in worst case constant time. The space requirement of Andersson's data structure is $O(n + 2^{\epsilon w})$. Several data structures can achieve the same time bounds as Andersson [5], but they all require constant time multiplication [6, 54, 93].

The main result of this paper is Theorem 22 stated below. The theorem requires the notion of *smooth* functions. Overmars [86] defines a nondecreasing function f to be smooth if and only if $f(O(n)) = O(f(n))$.

Theorem 22 *Let $f(n)$ be a nondecreasing smooth function satisfying $\log \log n \leq f(n) \leq \sqrt{\log n}$. On a Practical RAM a data structure exists supporting* INSERT *and* DELETE *in worst case $O(f(n))$ time,* PRED *in worst case $O((\log n)/f(n))$ time and* FINDMIN *and* FINDMAX *in worst case constant time, where n is the number of integers stored. The space required is $O(n2^{\epsilon w})$ for any constant $\epsilon > 0$.*

If $f(n) = \log \log n$ we achieve the result of Thorup but in the worst case sense, *i.e.*, we can support INSERT, DELETEMIN and DELETE in worst case $O(\log \log n)$ time. We can support PRED queries in worst case $O(\log n / \log \log n)$ time. The data structure is the first allowing predecessor queries in $O(\log n / \log \log n)$ time while having $O(\log \log n)$ update time. If $f(n) = \sqrt{\log n}$, we achieve time bounds matching those of Andersson [5].

The basic idea of our construction is to apply the data structure of van Emde Boas *et al.* [104, 106] for $O(f(n))$ levels and then switch to a packed search tree of height $O(\log n / f(n))$. This is very similar to the data structure of Andersson [5]. But where Andersson uses $O(\log n / f(n))$ time to update his packed B-tree, we only need $O(f(n))$ time. The idea we apply to achieve this speedup is to add *buffers* of delayed insertions and deletions to the search tree, such that we can work on several insertions concurrently by using the word parallelism of the Practical RAM. The idea of adding buffers to a search tree has in the context of designing I/O efficient data structures been applied by Arge [7].

Throughout this paper we w.l.o.g. assume DELETE only deletes integers actually contained in the set and INSERT never inserts an already inserted integer. This can be satisfied by tabulating the multiplicity of each inserted integer.

In the description of our data structure we in the following assume n is a constant such that the current number of integers in the set is $\Theta(n)$. This can be satisfied by using the general dynamization technique described by Overmars [86], which requires $f(n)$ to be smooth. In Section 9.2 if we write

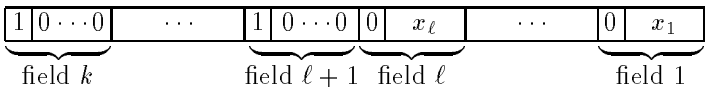

Figure 9.1: The structure of a list of maximum capacity k, containing integers $x_1, \ldots, x_\ell$.

$\log^5 n \leq k$, we actually mean that k is a function of n, but because we assume n to be a constant k is also assumed to be a constant.

In Section 9.2 we describe our packed search trees with buffers. In Section 9.3 we describe how to perform queries in a packed search tree and in Section 9.4 how to update a packed search tree. In Section 9.5 we combine the packed search trees with a range reduction based on the data structure of van Emde Boas *et al.* [104, 106] to achieve the result stated in Theorem 22. Section 9.6 contains some concluding remarks and lists some open problems.

9.2 Packed search trees with buffers

In this and the following two sections we describe how to maintain a set of integers of w/k bits each, for k satisfying $\log^5 n \leq k \leq w/\log n$. The bounds we achieve are:

Lemma 11 *Let k satisfy $\log^5 n \leq k \leq w/\log n$. If the integers to be stored are of w/k bits each then on a Practical RAM* Insert *and* Delete *can be supported in worst case $O(\log k)$ time,* Pred *in worst case $O(\log k + \log n/\log k)$ time and* FindMin *and* FindMax *in worst case constant time. The space required is $O(n)$.*

The basic idea is to store $O(k)$ integers in each word and to use the word parallelism of the Practical RAM to work on $O(k)$ integers in parallel in constant time. In the following we w.l.o.g. assume that we can apply Practical RAM operations to a list of $O(k)$ integers stored in $O(1)$ words in worst cast constant time. Together with each integer we store a *test bit*, as in [4, 5, 102]. An integer together with the associated test bit is denoted a *field*. Figure 9.1 illustrates the structure of a list of maximum capacity k containing $\ell \leq k$ integers $x_1, \ldots, x_\ell$. A field containing the integer x_i has a test bit equal to zero. The remaining $k - \ell$ empty fields store the integer zero and a test bit equal to one.

Essential to the data structure to be described is the following lemma due to Albers and Hagerup [4].

Lemma 12 (Albers and Hagerup) *On a Practical RAM two sorted lists each of at most $O(k)$ integers stored in $O(1)$ words can be merged into a single sorted list stored in $O(1)$ words in $O(\log k)$ time.*

Albers and Hagerup's proof of Lemma 12 is a description of how to implement the bitonic merging algorithm of Batcher [10] in a constant number of words on the Practical RAM. The algorithm of Albers and Hagerup does not handle partial full lists as defined (all test bits are assumed to be zero), but it is straightforward to modify their algorithm to do so, by considering an integer's test bit as the integer's most significant bit. A related lemma we need for our construction is the following:

Lemma 13 *Let k satisfy $k \leq w/\log n$. Let A and B be two sorted and repetition free lists each of at most $O(k)$ integers stored in $O(1)$ words on a Practical RAM. Then the sorted list $A \setminus B$ can be computed and stored in $O(1)$ words in $O(\log k)$ time.*

Proof. Let C be the list consisting of A merged with B twice. By Lemma 12 the merging can be done in worst case $O(\log k)$ time. By removing all integers appearing at least twice from C we get $A \setminus B$. In the following we outline how to eliminate these repetitions from C. Tedious implementation details are omitted.

First a mask is constructed corresponding to the integers only appearing once in C. This can be done in worst case constant time by performing the comparisons between neighbor integers in C by subtraction like the mask construction described in [4]. The integers appearing only once in C are compressed to form a single list as follows. First a prefix sum computation is performed to calculate how many fields each integer has to be shifted to the right. This can be done in $O(\log k)$ time by using the constructed mask. Notice that each of the calculated values is an integer in the range $0, \ldots, |A|+2|B|$, implying that each field is required to contain at least $O(\log k)$ bits. Finally we perform $O(\log k)$ iterations where we in the i'th iteration move all integers x_j, 2^i fields to the right if the binary representation of the number of fields x_j has to be shifted has the i'th bit set. A similar approach has been applied in [4] to reverse a list of integers. □

The main component of our data structure is a search tree T where all leaves have equal depth and all internal nodes have degree at least one and at most $\Delta \leq k/\log^4 n$. Each leaf v stores a sorted list I_v of between $k/2$ and k integers. With each internal node v of degree $d(v)$ we store $d(v) - 1$ keys to guide searches. The $d(v)$ pointers to the children of v can be packed into a single word because they require at most $d(v) \log n \leq w$ bits, provided that the number of nodes is less than n.

This part of the data structure is quite similar to the packed B-tree described by Andersson [5]. To achieve faster update times for INSERT and DELETE than Andersson, we add buffers of delayed INSERT and DELETE operations to each internal node of the tree.

With each internal node v we maintain a buffer I_v containing a sorted list of integers to be inserted into the leaves of the subtree T_v rooted at v, and a buffer D_v containing a sorted list of integers to be deleted from T_v. We maintain the invariants that I_v and D_v are disjoint and repetition free, and that

$$\max\{|I_v|, |D_v|\} < \Delta \log n. \tag{9.1}$$

The set S_v of integers stored in a subtree T_v can recursively be defined as

$$S_v = \begin{cases} I_v & \text{if } v \text{ is a leaf,} \\ I_v \cup ((\bigcup_{w \text{ a child of } v} S_w) \setminus D_v) & \text{otherwise.} \end{cases} \tag{9.2}$$

Finally we maintain two nonempty global buffers of integers L and R each of size $O(k)$ to be able to answer minimum and maximum queries in constant time. The integers in L are less than all other integers stored, and the integers in R are greater than all other integers stored.

Let h denote the height of T. In Section 9.4 we show how to guarantee that $h = O(\log n/\log k)$, implying that the number of nodes is $O(hn/k) = O(n)$.

9.3 Queries in packed search trees

By explicitly remembering the minimum integer in L and the maximum integer in R it is trivial to implement FINDMIN and FINDMAX in worst case constant time. A PRED(e) query can be answered

as follows. If $e \leq \max(L)$ then the predecessor of e is contained in L and can be found in worst case $O(\log k)$ time by standard techniques. If $\min(R) \leq e$ then the predecessor of e is contained in R. Otherwise we have to search for the predecessor of e in T.

We first perform a search for e in the search tree T. The implementation of the search for e in T is identical to how Andersson searches in a packed B-tree [5]. We refer to [5] for details. Let λ be the leaf reached and $w_1, \ldots, w_{h-1}$ be the internal nodes on the path from the root to λ. Define $w_h = \lambda$. Because we have introduced buffers at each internal node of T the predecessor of e does not necessarily have to be stored in I_λ but can also be contained in one of the insert buffers I_{w_i}. An integer $a \in I_{w_i}$ can only be a predecessor of e if it has not been deleted by a delayed delete operation, *i.e.*, $a \notin D_{w_j}$ for $1 \leq j < i$. It seems necessary to *flush* all buffers I_{w_i} and D_{w_i} for integers which should be inserted in or deleted from I_λ to be able to find the predecessor of e. If dom_λ denotes the interval of integers spanned by the leaf λ, the buffers I_{w_i} and D_{w_i} can be flushed for elements in dom_λ by the following sequence of operations:

$$\begin{aligned} I_{w_{i+1}} &\leftarrow I_{w_{i+1}} \setminus (D_{w_i} \cap \mathrm{dom}_\lambda) \cup (I_{w_i} \cap \mathrm{dom}_\lambda) \setminus D_{w_{i+1}}, \\ D_{w_{i+1}} &\leftarrow D_{w_{i+1}} \setminus (I_{w_i} \cap \mathrm{dom}_\lambda) \cup (D_{w_i} \cap \mathrm{dom}_\lambda) \setminus I_{w_{i+1}}, \\ I_{w_i} &\leftarrow I_{w_i} \setminus \mathrm{dom}_\lambda, \\ D_{w_i} &\leftarrow D_{w_i} \setminus \mathrm{dom}_\lambda. \end{aligned}$$

Let $\hat{I}_\lambda$ denote the value of I_λ after flushing all buffers I_{w_i} and D_{w_i} for integers in the range dom_λ. From (9.2) it follows that $\hat{I}_\lambda$ can also be computed directly by the expression

$$\hat{I}_\lambda = \mathrm{dom}_\lambda \cap (((\cdots((I_\lambda \setminus D_{w_{h-1}}) \cup I_{w_{h-1}}) \cdots) \setminus D_{w_1}) \cup I_{w_1}). \tag{9.3}$$

Based on Lemmas 12 and 13 we can compute this expression in $O(h \log k)$ time. This is unfortunately $O(\log n)$ for the tree height $h = \log n / \log k$. In the following we outline how to find the predecessor of e in $\hat{I}_\lambda$ without actually computing $\hat{I}_\lambda$ in $O(\log k + \log n / \log k)$ time.

Let I'_{w_i} be $I_{w_i} \cap \mathrm{dom}_\lambda \cap]\infty, e]$ for $i = 1, \ldots, h$. An alternative expression to compute the predecessor of e in $\hat{I}_\lambda$ is

$$\max \bigcup_{i=1,\ldots,h} (I'_{w_i} \setminus \bigcup_{j=1,\ldots,i-1} D_{w_j}). \tag{9.4}$$

Because $|\bigcup_{j=1,\ldots,h-1} D_{w_j}| < \Delta \log^2 n$ we can w.l.o.g. assume $|I'_{w_h}| \leq \Delta \log^2 n$ in (9.4) by restricting our attention to the $\Delta \log^2 n$ largest integers in I'_{w_h}, *i.e.*, all sets involved in (9.4) have size at most $\Delta \log^2 n$. The steps we perform to compute (9.4) are the following. All implementation details are omitted.

- First all buffers I_{w_i} and D_{w_i} for $i < h$ are inserted into a single word $\mathcal{W}$ where the contents of $\mathcal{W}$ is considered as $2h - 2$ independent lists each of maximum capacity $\Delta \log^2 n$. This can be done in $O(h) = O(\log n / \log k)$ time.
- Using the word parallelism of the Practical RAM we now for all I_{w_i} compute I'_{w_i}. This can be done in $O(\log k)$ time if $\min(\mathrm{dom}_\lambda)$ is known. The integer $\min(\mathrm{dom}_\lambda)$ can be computed in the search phase determining the leaf λ. $\mathcal{W}$ now contains I'_{w_i} and D_{w_i} for $i < h$.
- The value of I'_{w_h} is computed (satisfying $|I'_{w_h}| \leq \Delta \log^2 n$) and appended to $\mathcal{W}$. This can be done in $O(\log k)$ time. The contents of $\mathcal{W}$ is now

$$I'_{w_h} D_{w_{h-1}} I'_{w_{h-1}} \cdots D_{w_1} I'_{w_1}.$$

$\mathcal{W}_I$	I'_{w_h}	$\cdots$	I'_{w_h}	I'_{w_h}	$\cdots$	I'_{w_1}	$\cdots$	I'_{w_1}	I'_{w_1}
$\mathcal{W}_D$	$D_{w_{h-1}}$	$\cdots$	D_{w_2}	D_{w_1}	$\cdots$	$D_{w_{h-1}}$	$\cdots$	D_{w_2}	D_{w_1}
$\mathcal{W}_M$	$M_{h,h-1}$	$\cdots$	$M_{h,2}$	$M_{h,1}$	$\cdots$	$M_{1,h-1}$	$\cdots$	$M_{1,2}$	$M_{1,1}$

Figure 9.2: The structure of the words $\mathcal{W}_I$, $\mathcal{W}_D$ and $\mathcal{W}_M$.

- Let $\mathcal{W}_I = (I'_{w_h})^{h-1} \cdots (I'_{w_1})^{h-1}$ and $\mathcal{W}_D = (D_{w_{h-1}} \cdots D_{w_1})^h$. See Figure 9.2. The number of fields required in each word is $h(h-1)\Delta \log^2 n \leq \Delta \log^4 n \leq k$. The two words can be constructed from $\mathcal{W}$ in $O(\log k)$ time.
- From $\mathcal{W}_I$ and $\mathcal{W}_D$ we now construct $h(h-1)$ masks $M_{i,j}$ such that $M_{i,j}$ is a mask for the fields of I'_{w_i} which are not contained in D_{w_j}. See Figure 9.2. The construction of a mask $M_{i,j}$ from the two list I'_{w_i} and D_{w_j} is very similar to the proof of Lemma 13 and can be done as follows in $O(\log k)$ time.

  First I is merged with D twice (we omit the subscripts while outlining the mask construction). Let C be the resulting list. From C construct in constant time a mask C' that contains ones in the fields in which C stores an integer only appearing once in C and zero in all other fields. By removing all fields from C having *exactly* one identical neighbor we can recover I from C. By removing the corresponding fields from C' we get the required mask M. As an example assume $I = (7,5,4,3,1)$ and $D = (6,5,2)$. Then $C = (7,\underline{6},\underline{6},\underline{5},5,\underline{5},4,3,\underline{2},\underline{2},1)$, $C' = (1,\underline{0},\underline{0},\underline{0},0,\underline{0},1,1,\underline{0},\underline{0},1)$ and $M = (1,0,1,1,1)$ where underlined fields are the fields in C having exactly one identical neighbor.
- We now compute masks $M_i = \bigwedge_{j=1,\ldots,i-1} M_{i,j}$ for all i. By applying M_i to I'_{w_i} we get $I'_{w_i} \setminus \bigcup_{j=1,\ldots,i-1} D_{w_j}$. This can be done in $O(\log k)$ time from $\mathcal{W}_M$ and $\mathcal{W}_I$.
- Finally we in $O(\log k)$ time compute (9.4) as the maximum over all the integers in the sets computed in the previous step. Notice that it can easily be checked if e has a predecessor in $\hat{I}_\lambda$ by checking if all the sets computed in the previous step are empty.

We conclude that the predecessor of e in $\hat{I}_\lambda$ can be found in $O(\log k + h) = O(\log k + \log n / \log k)$ time.

If e does not have a predecessor in $\hat{I}_\lambda$ there are two cases to consider. The first is if there exists a leaf $\bar{\lambda}$ to the left of λ. Then the predecessor of e is the largest integer in $\hat{I}_{\bar{\lambda}}$. Notice that $\hat{I}_{\bar{\lambda}}$ is nonempty because $|\bigcup_{j=1,\ldots,h-1} D_{\bar{w}_j}| < |I_{\bar{\lambda}}|$. If λ is the leftmost leaf the predecessor of e is the largest integer in L. We conclude that PRED queries can be answered in worst case $O(\log k + \log n / \log k)$ time on a Practical RAM.

9.4 Updating packed search trees

In the following we describe how to perform INSERT and DELETE updates. We first give a solution achieving the claimed time bounds in the amortized sense. The amortized solution is then converted into a worst case solution by standard techniques.

We first consider INSERT(e). If $e < \max(L)$ we insert e into L in $\log k$ time, remove the maximum from L such that $|L|$ remains unchanged, and let e become the removed integer. If $\min(R) < e$ we insert e in R, remove the minimum from R, and let e become the removed integer.

Let r denote the root of T. If $e \in D_r$, remove e from D_r in worst case $O(\log k)$ time, *i.e.*, INSERT(e) cancels a delayed DELETE(e) operation. Otherwise insert e into I_r.

If $|I_r| < \Delta \log n$ this concludes the INSERT operation. Otherwise there must exist a child w of r such that $\log n$ integers can be moved from I_r to the subtree rooted at w. The child w and the $\log n$ integers X to be moved can be found by a binary search using the search keys stored at r in worst case $O(\log k)$ time. We omit the details of the binary search in I_r. We first remove the set of integers X from I_r such that $|I_r| < \Delta \log n$. We next remove all integers in $X \cap D_w$ from X and from D_w in $O(\log k)$ time by Lemma 13, *i.e.*, we let delayed deletions be cancel out by delayed insertions. The remaining integers in X are merged into I_w in $O(\log k)$ time. Notice that I_w and D_w are disjoint after the merging and that if w is an internal node then $|I_w| < (\Delta + 1) \log n$.

If $|I_w| \geq \Delta \log n$ and w is not a leaf we recursively apply the above to I_w. If w is a leaf and $|I_w| \leq k$ we are done. The only problem remaining is if w is a leaf and $k < |I_w| \leq k + \log n \leq 2k$. In this case we split the leaf w into two leaves each containing between $k/2$ and k integers, and update the search keys and child pointers stored at the parent of w. If the parent p of w now has $\Delta + 1$ children we split p into two nodes of degree $\geq \Delta/2$ while distributing the buffers I_p and D_p among the two nodes w.r.t. the new search key. The details of how to split a node is described in [5]. If the parent of p gets degree $\Delta + 1$ we recursively split the parent of p.

The implementation of inserting e in T takes worst case $O(h \log k)$ time. Because the number of leaves is $O(n)$ and that T is similar to a B-tree if we only consider insertions we get that the height of T is $h = O(\log n / \log \Delta) = O(\log n / \log(k / \log^4 n)) = O(\log n / \log k)$ because $k \geq \log^5 n$. It follows that the worst case insertion time in T is $O(\log n)$. But because we remove $\log n$ integers from I_r every time $|I_r| = \Delta \log n$ we spend at most worst case $O(\log n)$ time once for every $\log n$ insertion. All other insertions require worst case $O(\log k)$ time. We conclude that the amortized insertion time is $O(\log k)$.

We now describe how to implement DELETE(e) in amortized $O(\log k)$ time. If e is contained in L we remove e from L. If L is nonempty after having removed e we are done. If L becomes empty we proceed as follows. Let λ be the leftmost leaf of T. The basic idea is to let L become $\hat{I}_\lambda$. We do this as follows. First we flush all buffers along the leftmost path in the tree for integers contained in dom_λ. Based on (9.3) this can be done in $O(h \log k)$ time. We can now assume $(I_w \cup D_w) \cap \mathrm{dom}_\lambda = \emptyset$ for all nodes w on the leftmost path and that $I_\lambda = \hat{I}_\lambda$. We can now assign L the set I_λ and remove the leaf λ. If the parent p of λ gets degree zero we recursively remove p. Notice that if p gets degree zero then I_p and D_p are both empty. Because the total size of the of insertion and deletion buffers on the leftmost path is bounded by $h\Delta \log n \leq k / \log^2 n$ it follows that $\log n \leq k/2 - k/\log^2 n \leq |L| \leq k + k/\log^2 n$. It follows that L cannot become empty throughout the next $\log n$ DELETE operations. The case $e \in R$ is handled symmetrically by letting λ be the rightmost leaf.

If $e \notin L \cup R$ we insert e in D_r provided $e \notin I_r$. If $e \in I_r$ we remove e from I_r in $O(\log k)$ time and are done. If $|D_r| \geq \Delta \log n$ we can move $\log n$ integers X from D_r to a child w of r. If w is an internal node we first remove $X \cap I_w$ from X and I_w, *i.e.*, delayed insertions cancels delayed insertions, and then inserts the remaining elements in X into D_w. If $|D_w| \geq \Delta \log n$ we recursively move $\log n$ integers from D_w to a child of w. If w is a leaf λ we just remove the integers X from I_λ. If $|I_\lambda| \geq k/2$ we are done. Otherwise let $\bar{\lambda}$ denote the leaf to the right or left of λ (If $\bar{\lambda}$ does not exist the set only contains $O(k)$ integers and the problem is easy to handle. In the following we w.l.o.g. assume $\bar{\lambda}$ exists). We first flush all buffers on the paths from the root r to λ and $\bar{\lambda}$ such that the buffers do not contain elements from $\mathrm{dom}_\lambda \cup \mathrm{dom}_{\bar{\lambda}}$. This can be done in $O(h \log n)$ time as previously described. From

$$k/2 + k/2 - \log n - 2h\Delta \log n \leq |I_\lambda \cup I_{\bar{\lambda}}| \leq k/2 + k - 1 + 2h\Delta \log n$$

it follows that $k/2 \leq |I_\lambda \cup I_{\bar{\lambda}}| \leq 2k$. There are two cases to consider. If $|\lambda + \bar{\lambda}| \geq k$ we redistribute I_λ and $I_{\bar{\lambda}}$ such that they both have size at least $k/2$ and at most k. Because all buffers on the path from λ ($\bar{\lambda}$) to the root intersect empty with $\text{dom}_\lambda \cup \text{dom}_{\bar{\lambda}}$ we in addition only need to update the search key stored at the nearest common ancestor of λ and $\bar{\lambda}$ in T which separates dom_λ and $\text{dom}_{\bar{\lambda}}$. This can be done in $O(h + \log k)$ time. The second case is if $|\lambda + \bar{\lambda}| < k$. We then move the integers in I_λ to $I_{\bar{\lambda}}$ and remove the leaf λ as described previously. The total worst case time for a deletion becomes $O(h \log k) = O(\log n)$. But again the amortized time is $O(\log k)$ because L and R become empty for at most every $\log n$'th DELETE operation, and because D_r becomes full for at most every $\log n$'th DELETE operation.

In the previous description of DELETE we assumed the height of T is $h = O(\log n / \log k)$. We argued that this was true if only INSERT operations were performed because then our search tree is similar to a B-tree. It is easy to see that if only $O(n)$ leaves have been remove, then the height of T is still $h = O(\log n / \log k)$. One way to see this is by assuming that all removed nodes still resist in T. Then T has at most $O(n)$ leaves and each internal node has degree at least $\Delta/2$, which implies the claimed height. By rebuilding T completely such that all internal nodes have degree $\Theta(\Delta)$ for every n'th DELETE operation we can guarantee that at most n leaves have been removed since T was rebuild the last time. The rebuilding of T can easily be done in $O(n \log k)$ time implying that the amortized time for DELETE only increases by $O(\log k)$.

We conclude that INSERT and DELETE can be implemented in amortized $O(\log k)$ time. The space required is $O(n)$ because each node can be stored in $O(1)$ words.

To convert the amortized time bounds into worst case time bounds we apply the standard technique of incrementally performing a worst case expensive operation over the following sequence of operations by moving the expensive operation into a shadow process that is executed in a quasi-parallel fashion with the main algorithm. The rebuilding of T when $O(n)$ DELETE operations have been performed can be handled by the general dynamization technique of Overmars [86] in worst case $O(\log k)$ time per operation. For details refer to [86]. What remains to be described is how to handle the cases when L or R becomes empty and when I_r or D_r becomes full. The basic idea is to handle these cases by simply avoiding them. Below we outline the necessary changes to the amortized solution.

The idea is to allow I_r and D_r to have size $\Delta \log n + O(\log n)$ and to divide the sequence of INSERT and DELETE operations into phases of $\log n/4$ operations. In each phase we perform one of the transformations below to T incrementally over the $\log n/4$ operations of the phase by performing worst case $O(1)$ work per INSERT or DELETE operation. We cyclic choose which transformation to perform, such that for each $\log n$'th operation each transformation has been performed at least once. Each of the transformations can be implemented in worst case $O(\log n)$ time as described in the amortized solution.

- If $|L| < k$ at the start of the phase and λ denotes the leftmost leaf of T we incrementally merge L with $\hat{I}_\lambda$ and remove the leaf λ. It follows that L always has size at least $k - O(\log n) > 0$.
- The second transformation similarly guarantees that $|R| > 0$ by merging R with $\hat{I}_\lambda$ where λ is rightmost leaf of T if $|R| < k$.
- If $|I_r| \geq \Delta \log n$ at the start of the phase we incrementally remove $\log n$ integers from I_r. It follows that the size of I_r is bounded by $\Delta \log n + O(\log n) = O(k)$.
- The last transformation similarly guarantees that the size of D_r is bounded by $\Delta \log n + O(\log n)$ by removing $\log n$ integers from D_r if $|D_r| \geq \Delta \log n$.

This finishes our description of how to achieve the bounds stated in Lemma 11.

9.5 Range reduction

To prove Theorem 22 we combine Lemma 11 with a range reduction based on a data structure of van Emde Boas *et al.* [104, 106]. This is similar to the data structure of Andersson [5], and for details we refer to [5]. We w.l.o.g. assume $w \geq 2^{f(n)} \log n$.

The idea is to use the topmost $f(n)$ levels of the data structure of van Emde Boas *et al.* and then switch to our packed search trees. If $f(n) \geq 5 \log\log n$ the integers we need to store are of $w/2^{f(n)} \leq w/\log^5 n$ bits each and Lemma 11 applies for $k = 2^{f(n)}$. By explicitly remembering the minimum and maximum integer stored FINDMIN and FINDMAX are trivial to support in worst case constant time. The remaining time bounds follow from Lemma 11. The space bound of $O(n2^{\epsilon w})$ follows from storing the arrays at each of the $O(n)$ nodes in the data structure of van Emde Boas *et al.* as a trie of degree $2^{\epsilon w}$.

9.6 Conclusion

We have presented the first data structure for a Practical RAM allowing the update operations INSERT and DELETE in worst case $O(\log\log n)$ time while answering PRED queries in worst case $O(\log n/\log\log n)$ time. An interesting open problem is if it is possible to support INSERT and DELETE in worst case $O(\log\log n)$ time and PRED in worst case $O(\sqrt{\log n})$ time. The general open problem is to find a tradeoff between the update time and the time for predecessor queries on a Practical RAM.

Acknowledgments

The author thanks Theis Rauhe, Thore Husfeldt and Peter Bro Miltersen for encouraging discussions, and the referees for comments.

Chapter 10

Approximate Dictionary Queries

Approximate Dictionary Queries

Gerth Stølting Brodal*
BRICS†, Department of Computer Science
University of Aarhus
Ny Munkegade, 8000 Århus C, Denmark
gerth@brics.dk

Leszek Gąsieniec‡
Max–Planck–Institut für Informatik
Im Stadtwald, 66123 Saarbrücken, Germany
leszek@mpi-sb.mpg.de

Abstract

Given a set of n binary strings of length m each. We consider the problem of answering d–queries. Given a binary query string α of length m, a d–query is to report if there exists a string in the set within Hamming distance d of α.

We present a data structure of size $O(nm)$ supporting 1–queries in time $O(m)$ and the reporting of all strings within Hamming distance 1 of α in time $O(m)$. The data structure can be constructed in time $O(nm)$. A slightly modified version of the data structure supports the insertion of new strings in amortized time $O(m)$.

Category: F.2.2
Keywords: dictionaries, approximate queries, RAM model

10.1 Introduction

Let $W = \{w_1, \ldots, w_n\}$ be a set of n binary strings of length m each, *i.e.*, $w_i \in \{0,1\}^m$. The set W is called the *dictionary*. We are interested in answering d–queries, *i.e.*, for any query string $\alpha \in \{0,1\}^m$ to decide if there is a string w_i in W with at most Hamming distance d of α.

Minsky and Papert originally raised this problem in [80]. Recently a sequence of papers have considered how to solve this problem efficiently [41, 42, 59, 74, 112]. Manber and Wu [74] considered the application of approximate dictionary queries to password security and spelling correction of bibliographic files. Their method is based on Bloom filters [12] and uses hashing techniques. Dolev *et al.* [41, 42] and Greene, Parnas and Yao [59] considered approximate dictionary queries for the case where d is large.

The initial effort towards a theoretical study of the small d case was given by Yao and Yao in [112]. They present for the case $d = 1$ a data structure supporting queries in time $O(m \log\log n)$ with space requirement $O(nm \log m)$. Their solution was described in the cell-probe model of Yao [111] with word size equal to 1. In this paper we adopt the standard unit cost RAM model [105].

For the general case where $d > 1$, d–queries can be answered in optimal space $O(nm)$ doing $\sum_{i=0}^{d} \binom{m}{i}$ exact queries each requiring time $O(m)$ by using the data structure of Fredman, Komlos

*Supported by the Danish Natural Science Research Council (Grant No. 9400044). Partially supported by the ESPRIT Long Term Research Program of the EU under contract #20244 (ALCOM-IT). This research was done while visiting the Max-Planck Institut für Informatik, Saabrücken, Germany.

†Basic Research in Computer Science, a Centre of the Danish National Research Foundation.

‡On leave from Institute of Informatics, Warsaw University, ul. Banacha 2, 02–097, Warszawa, Poland. WWW: http://zaa.mimuw.edu.pl/~lechu/lechu.html.

and Szemeredi [51]. On the other hand d–queries can be answered in time $O(m)$ when the size of the data structure can be $O(n\sum_{i=0}^{d}\binom{m}{i})$. We present the corresponding data structure of size $O(nm)$ for the 1–query case.

We present a simple data structure based on tries [2, 50] which has optimal size $O(nm)$ and supports 1–queries in time $O(m)$. Unfortunately, we do not know how to construct the data structure in time $O(nm)$ and we leave this as an open problem. However, we give a more involved data structure of size $O(nm)$, based on two tries, supporting 1–queries in time $O(m)$ and which can be constructed in time $O(nm)$. Both data structures support the reporting of all strings with Hamming distance at most one of the query string α in time $O(m)$. For general d both data structures support d–queries in time $O(m\sum_{i=0}^{d-1}\binom{m}{i})$. The second data structure can be made semi-dynamic in terms of allowing insertions in amortized time $O(m)$, when starting with an initially empty dictionary. Both data structures work as well for larger alphabets $|\Sigma| > 2$, when the query time is slowed down by a $\log|\Sigma|$ factor.

The paper is organized as follows. In Section 10.2 we give a simple $O(nm)$ size data structure supporting 1–queries in time $O(m)$. In Section 10.3 we present an $O(nm)$ size data structure constructible in time $O(nm)$ which also supports 1–queries in time $O(m)$. In Section 10.4 we present a semi-dynamic version of the second data structure allowing insertions. Finally in Section 10.5 we give concluding remarks and mention open problems.

10.2 A trie based data structure

We assume that all strings considered are over a binary alphabet $\Sigma = \{0, 1\}$. We let $|w|$ denote the length of w, $w[i]$ denote the ith symbol of w and w^R denote w reversed. The strings in the dictionary W are called dictionary strings. We let $dist_H(u, v)$ denote the Hamming distance between the two strings u and v.

The basic component of our data structure is a trie [50]. A trie, also called a digital search tree, is a tree representation of a set of strings. In a trie all edges are labeled by symbols such that every string corresponds to a path in the trie. A trie is a prefix tree, *i.e.*, two strings have a common path from the root as long as they have the same prefix. Since we consider strings over a binary alphabet the maximum degree of a trie is at most two.

Assume that all strings $w_i \in W$ are stored in a 2-dimensional array $\mathcal{A}_W$ of size $n \times m$, *i.e.*, of n rows and m columns, such that the ith string is stored in the ith row of the array $\mathcal{A}_W$. Notice that $\mathcal{A}_W[i, j]$ is the jth symbol w_i. For every string $w_i \in W$ we define a set of *associated* strings $A_i = \{v \in \{0, 1\}^m | dist_H(v, w_i) = 1\}$, where $|A_i| = m$, for $i = 1, \ldots, n$. The main data structure is a trie T containing all strings $w_i \in W$ and all strings from A_i, for all $i = 1, \ldots, n$, *i.e.*, every path from the root to a leaf in the trie represents one of the strings. The leaves of T are labeled by indices of dictionary strings such that a leaf representing a string s and labeled by index i satisfies that $s = w_i$ or $s \in A_i$.

Given a query string α an 1–query can be answered as follows. The 1–query is answered positively if there is an exact match, *i.e.*, $\alpha = w_i \in W$, or $\alpha \in A_j$, for some $1 \leq j \leq n$. Thus the 1–query is answered positively if and only if there is a leaf in the trie T representing the query string α. This can be checked in time $O(m)$ by a top-down traverse in T. If the leaf exists then the index stored at the leaf is an index of a matched dictionary string.

Notice that T has at most $O(nm)$ leaves because it contains at most $O(nm)$ different strings. Thus T has at most $O(nm)$ internal vertices with degree greater than one. If we compress all chains in T into single edges we get a compressed trie T' of size $O(nm)$. Edges which correspond to compressed chains are labeled by proper intervals of rows in the array $\mathcal{A}_W$. If a compressed

chain is a substring of a string in the a A_j then the information about the corresponding substring of w_j is extended by the position of the changed bit. Since every entry in $\mathcal{A}_W$ can be accessed in constant time every 1–query can still be answered in time $O(m)$.

A slight modification of the trie T' allows all dictionary strings which match the query string α to be reported. At every leaf s representing a string u in T' instead of one index we store all indices i of dictionary strings satisfying $s = w_i$ or $s \in A_i$. Notice that the total size of the trie is still $O(nm)$ since every index i, for $i = 1, \ldots, n$, is stored at exactly $m + 1$ leaves. The reporting algorithm first finds the leaf representing the query string α and then reports all indices stored at that leaf. There are at most $m + 1$ reported string thus the reporting algorithm works in time $O(m)$. Thus the following theorem holds.

Theorem 23 *There exists a data structure of size $O(nm)$ which supports the reporting of all matched dictionary strings to an 1–query in time $O(m)$.*

The data structure above is quite simple, occupies optimally space $O(nm)$ and allows 1–queries to be answered optimally in time $O(m)$. But we do not know how to construct it in time $O(nm)$. The straight forward approach gives a construction time of $O(nm^2)$ (this is the total size of the strings in W and the associated strings from all A_i sets).

In the next section we give another data structure of size $O(nm)$, supporting 1–queries in time $O(m)$ and constructible in optimal time $O(nm)$.

10.3 A double-trie data structure

In the following we assume that all strings in W are enumerated according to their lexicographical order. We can satisfy this assumption by sorting the strings in W, for example, by radix sort in time $O(nm)$. Let $I = \{1, \ldots, n\}$ denote the set of the indices of the enumerated strings from W. We denote a set of consecutive indices (consecutive integers) an interval.

The new data structure is composed of two tries. The trie T_W contains the set of stings W whereas the trie $T_{\overline{W}}$ contains all strings from the set $\overline{W}$, where $\overline{W} = \{w_i^R | w_i \in W\}$.

Since T_W is a prefix trie every path from the root to a vertex u represents a prefix p_u of a string $w_i \in W$. Denote by W_u the set $\{w_i \in W | w_i \text{ has prefix } p_u\}$. Since strings in W are enumerated according to their lexicographical order those indices form an interval I_u, *i.e.*, $w_i \in W_u$ if and only if $i \in I_u$. Notice that an interval of a vertex in the trie T_W is the concatenation of the intervals of its children. For each vertex u in T_W we compute the corresponding interval I_u, storing at u the first and last index of I_u.

Similarly every path from the root to a vertex v in $T_{\overline{W}}$ represents a reversed suffix s_v^R of a string $w_j \in W$. Denote by W^v the set $\{w_i \in W | w_i \text{ has suffix } s_v\}$ and by $S_v \subseteq I$ the set of indices of strings in W^v. We organize the indices of every set S_v in sorted lists L_v (in increasing order). At the root r of the trie $T_{\overline{W}}$ the list L_r is supported by a search tree maintaining the indices of all the dictionary strings. For an index in a list L_v the neighbor with the smaller value is called left neighbor and the one with greater value is called right neighbor. If a vertex x is the only child of vertex $v \in T_{\overline{W}}$ then S_x and S_v are identical. If vertex $v \in T_{\overline{W}}$ has two children x and y (there are at most two children since $T_{\overline{W}}$ is a binary trie) the sets S_x and S_y form a partition of the set S_v. Since indices in the set S_v are not consecutive (S_v is usually not an interval) we use additional links to keep fast connection between the set S_v and its partition into S_x and S_y. Each element e in the list L_v has one additional link to the closest element in the list L_x, *i.e.*, to the smallest element e_r in the list L_x such that $e \leq e_r$ or the greatest element e_l in the list L_x such that $e \geq e_l$. Moreover

in case vertex v has two children, element e has also one additional link to the analogously defined element $e_l \in L_y$ or $e_r \in L_y$.

Lemma 14 *The tries T_W and $T_{\overline{W}}$ can be stored in $O(nm)$ space and they can be constructed in time $O(nm)$.*

Proof. The trie T_W has at most $O(nm)$ edges and vertices, *i.e.*, the number of symbols in all strings in W. Every vertex $u \in T_W$ keeps only information about the two ends of its interval $I_u = [l..r]$. For all $u \in T_W$ both indices l and r can be easily computed by a postorder traversal of T_W in time $O(nm)$.

The number of vertices in $T_{\overline{W}}$ is similarly bounded by $O(nm)$. Moreover, for any level $i = 1, \ldots, m$ in $T_{\overline{W}}$, the sum $\sum |S_v|$ over all vertices v at this level is exactly n since the sets of indices stored at the children forms a partition of the set kept by their parent. Since $T_{\overline{W}}$ has exactly m levels and every index in an L_v list has at most two additional links the size of $T_{\overline{W}}$ does not exceed $O(nm)$ too. The L_v lists are constructed by a postorder traversal of $T_{\overline{W}}$. A leaf representing the string w_i^R has $L_v = (i)$ and the L_v list of an internal vertex of $T_{\overline{W}}$ can be constructed by merging the corresponding disjoint lists at its children. The additional links are created along with the merging. Thus the trie $T_{\overline{W}}$ can be constructed in time $O(nm)$. □

Answering queries

In this section we show how to answer 1–queries in time $O(m)$ assuming that both tries T_W and $T_{\overline{W}}$ are already constructed. We present a sequence of three 1–query algorithms all based on the double-trie structure. The first algorithm QUERY1 outlines how to use the presented data structure to answer 1–queries. The second algorithm QUERY2 reports the index of a matched dictionary string. The third algorithm QUERY3 reports all matched dictionary strings.

Let $pref_\alpha$ be the longest prefix of the string α that is also a prefix of a string in W. The prefix $pref_\alpha$ is represented by a path from the root to a vertex $\overline{u}$ in the trie T_W, *i.e.*, $p_\alpha = p_{\overline{u}}$ but for the only child x of vertex $\overline{u}$ the string p_x is not a prefix of α. We call the vertex $\overline{u}$ the *kernel vertex* for the string α and the path from the root of T_W to the kernel vertex $\overline{u}$ the *leading path* in T_W. The interval $I_\alpha = I_{\overline{u}}$ associated with the kernel vertex $\overline{u}$ is called the *kernel interval* for the string α and the smallest element $\mu_\alpha \in I_\alpha$ is called the *key* for the query string α. Notice that the key $\mu_\alpha \in I_w$, for every vertex w on the leading path in T_W.

Similarly in the trie $T_{\overline{W}}$ we define the *kernel set* $S_{\hat{v}}$ which is associated with the vertex $\hat{v}$, where $\hat{v}$ corresponds to the longest prefix of the string α^R in $T_{\overline{W}}$. The vertex $\hat{v}$ is called a kernel vertex for the string α^R, and the path from the root of $T_{\overline{W}}$ to $\hat{v}$ is called the leading path in $T_{\overline{W}}$.

The general idea of the algorithm is as follows. If the query string α has an exact match in the set W, then there is a leaf in T_W which represents the query string α. The proper leaf can be found in time $O(m)$ by a top-down traverse of T_W, starting from its root.

If the query string α has no exact match in W but it has a match within distance one, we know that there is a string $w_i \in W$ which has a factorization $\pi_\alpha b \tau_\alpha$, satisfying:

(1) π_α is a prefix of α of length l_α,

(2) τ_α is a suffix of α of length r_α,

(3) $b \neq \alpha[l_\alpha + 1]$ and

(4) $l_\alpha + r_\alpha + 1 = m$.

```
ALGORITHM QUERY1
begin
u := ū — the kernel vertex in T_W.
Find on the leading path in T_W̄ vertex v such that (u, v) is a feasible pair.
while vertex v exists do
        if I_u ∩ S_v ≠ ∅ then return "There is a match"
        u :=PARENT(u)
        v :=CHILD-ON-LEADING-PATH(v)
od
return "No match"
end.
```

Notice that prefix π_α must be represented by a vertex u in the leading path in T_W and suffix τ_α must be represented by a vertex v in the leading path of $T_{\overline{W}}$. We call such a pair (u, v) a *feasible* pair. To find the string w_i within distance 1 of the query string α we have to search all feasible pairs (u, v). Every feasible pair (u, v) for which $I_u \cap S_v \neq \emptyset$, represents *at least one* string within distance 1 of the query string α. The algorithm QUERY1 generates consecutive feasible pairs (u, v) starting with $u = \overline{u}$, the kernel vertex in T_W. The algorithm QUERY1 stops with a positive answer just after the first pair (u, v) with $I_u \cap S_v \neq \emptyset$ is found. It stops with a negative answer if all feasible pairs (u, v) have $I_u \cap S_v = \emptyset$.

Notice that the steps before the while loop in the algorithm QUERY1 can be performed in time $O(m)$. The algorithm looks for the kernel vertex in T_W going from the root along the leading path (representing the prefix $pref_\alpha$) as long as possible. The last reached vertex u is the kernel vertex $\overline{u}$. Then the corresponding vertex v on the leading path in $T_{\overline{W}}$ is found, if such a vertex exists. Recall that a pair (u, v) must be a feasible pair. At this point the following problem arises. How to perform the test $I_u \cap S_v \neq \emptyset$ efficiently?

Recall that the smallest index μ_α in the kernel interval I_α is called the key for the query string α and recall also that the key $\mu_\alpha \in I_w$, for every vertex w in the leading path in the trie T_W. During the first test $I_u \cap S_v \neq \emptyset$ the position of the key μ_α in S_v is found in time $\log |S_v| \leq \log n \leq m$ (since W only contains binary strings we have $\log n \leq m$). Let $I_u = [l..r]$, a be the left ($a \leq \mu_\alpha$) and b the right ($b > \mu_\alpha$) neighbors of μ_α in the set S_v. Now the test $I_u \cap S_v \neq \emptyset$ can be stated as:

$$I_u \cap S_v \neq \emptyset \equiv l \leq a \ \vee \ b \leq r.$$

If the above test is positive the algorithm QUERY2 reports the proper index among a and b and stops. Otherwise, in the next round of the while loop the new neighbors a and b of the key μ_α in the new list L_v are computed in constant time by using the additional links between the elements of the old and new list L_v.

Theorem 24 *1–queries to a dictionary W of n strings of length m can be answered in time $O(m)$ and space $O(nm)$.*

Proof. The initial steps of the algorithm (preceding the while loop) are performed in time $O(m + \log n) = O(m)$. The feasible pair (u, v) (if such exists) is simply found in time $O(m)$. Then the algorithm finds in time $O(\log n)$ the neighbors of μ_α in the list L_r which is held at the root of $T_{\overline{W}}$.

ALGORITHM QUERY2
begin
$u := \overline{u}$ — the kernel vertex in T_W.
Find on the leading path in $T_{\overline{W}}$ vertex v such that (u, v) is a feasible pair.
Find the neighbors a and b of the key μ_α in S_v.
while vertex v exists **do**
    if $l \leq a$ **then return** "String a is matched"
    if $b \leq r$ **then return** "String b is matched"
    $u :=$PARENT(u); Set l and r according to the new interval I_u.
    $v :=$CHILD-ON-LEADING-PATH(v)
    Find new neighbors of μ_α, a and b, in the new list L_v.
od
return "No match"
end.

This is possible since the list L_r is supported by a search tree. Now the algorithm traverses the leading path in $T_{\overline{W}}$ recovering at each level neighbors of μ_α in constant time using the additional links. There are at most m iterations of the while loop since there is exactly m levels in both tries T_W and $T_{\overline{W}}$. Every iteration of the while loop is done in constant time since both neighbors a and b of the key μ_α in the new more sparse set S_v are found in constant time. Thus the total running time of the algorithm is $O(m)$. □

We explain now how to modify the algorithm QUERY2 to an algorithm reporting all matches to a query string. The main idea of the new algorithm is as follows. At any iteration of the while loop instead of looking only for the left and the right neighbor of the key index μ_α the algorithm QUERY3 searches one by one all indices to the left and right of μ_α which belong to the list L_v and to the interval I_u. To avoid multiple reporting of the same index the algorithm searches only that part of the new interval I_u which is an extension of the previous one. The variables a and b store the leftmost and the rightmost searched indices in the list L_v.

Theorem 25 *There exists a data structure of size $O(nm)$ and constructible in time $O(nm)$ which supports the reporting of all matched dictionary strings to a 1–query in time $O(m)$.*

Proof. The algorithm QUERY3 works in time $O(m + \#matched)$, where $\#matched$ is the number of all reported strings. Since there is at most $m + 1$ reported strings (one exact matching and at most m matches with one error) the total time of the reporting algorithm is $O(m)$. □

10.4 A semi-dynamic data structure

In this section we describe how the data structure presented in Section 10.3 can be made semi-dynamic such that new binary strings can be inserted into W in amortized time $O(m)$. In the following w' denotes a string to be inserted into W.

The data structure described in Section 10.3 requires that the strings w_i are lexicographically sorted and that each string has assigned its rank with respect to the lexicographical ordering of the strings. If we want to add w' to W we can use T_W to locate the position of w' in the sorted list of

```
ALGORITHM QUERY3
begin
u := ū — the kernel vertex in T_W.
Find on the leading path in T_W̄ vertex v such that (u, v) is a feasible pair.
Find the neighbors a and b of the key μ_α in S_v.
while vertex v exists do
        while l ≤ a do
                report "String a is matched"
                a := left neighbor of a in L_v.
        od
        while b ≤ r do
                report "String b is matched"
                b := right neighbor of b in L_v.
        od
        u :=PARENT(u); Set l and r according to new I_u.
        v :=CHILD-ON-LEADING-PATH(v)
        Find a or the left neighbor of a in the new list L_v.
        Find b or the right neighbor of b in the new list L_v.
od
end.
```

w_is in time $O(m)$. If we continue to maintain the ranks explicitly assigned to the strings we have to reassign new ranks to all strings larger than w'. This would require time $\Omega(n)$. To avoid this problem, observe that the indices are used to store the endpoints of the intervals I_u and to store the sets S_v, and that the only operation performed on the indices is the comparison of two indices to decide if one string is lexicographically less than another string in constant time.

Essentially what we need to know is if given the *handles* of two strings from W, which one of the two strings is the lexicographically smallest. A solution to this problem was given by Dietz and Sleator [36]. They presented a data structure that allows a new element to be inserted into a linked list in constant time if the new element's position is known, and that can answer order queries in constant time.

By applying the data structure of Dietz and Sleator to maintain the ordering between the strings, an insertion can now be implemented as follows. First insert w' into T_W. This requires time $O(m)$. The position of w' in T_W also determines its location in the lexicographically order implying that the data structure of Dietz and Sleator can be updated too. By traversing the path from the new leaf representing w' in T_W to the root of T_W, the endpoints of the intervals I_u can be updated in time $O(m)$.

The insertion of w'^R into $T_{\overline{W}}$ without updating the associated fields can be done in time $O(m)$. Analogously to the query algorithm in Section 10.3, the positions in the S_v sets along the insertion path of w' in $T_{\overline{W}}$ where to insert the handle of w' can be found in time $O(m)$.

The problem remaining is to update the additional links between the elements in the L_v lists. For this purpose we change our representation to the following. Let v be a node with children x and y. In the following we only consider how to handle the links between L_v and L_x. The links between L_v and L_y are handled analogously. For each element $e \in L_v \cap L_x$ we maintain a pointer

from the position of e in L_v to the position of e in L_x. For each element $e \in L_v \setminus L_x$ the pointer is null. Let $e \in L_v$. We can now find the closest element to e in L_x by finding the closest element in L_v that has a non null pointer. We denote such an element to be marked. For this purpose we use the FIND-SPLIT-ADD data structure of Imai and Asano [65], an extension of a data structure by Gabow and Tarjan [55]. The data structure supports the following operations: Given a pointer to an element in a list, to find the closest marked element (FIND); to mark an unmarked list element (SPLIT); and to insert a new unmarked element into the list adjacent to an element in the list (ADD). The operations SPLIT and ADD can be performed in amortized constant time and FIND in worst case constant time on a RAM. Going from e in L_v to e's closest neighbor in L_x can still be performed in worst case constant time, because this only requires one FIND operation to be performed. When a new element e is added to L_v we just perform ADD once, and in case e is added to L_x too we also perform SPLIT on e. This requires amortized constant time. Totally we can therefore update all the links between the L_v lists in amortized time $O(m)$ when inserting a new string into the dictionary.

Theorem 26 *There exists a data structure which supports the reporting of all matched dictionary strings to a 1–query in worst case time $O(m)$ and that allows new dictionary strings to be inserted in amortized time $O(m)$.*

10.5 Conclusion

We have presented a data structure for the approximate dictionary query problem that can be constructed in time $O(nm)$, stored in $O(nm)$ space and that can answer 1–queries in time $O(m)$. We have also shown that the data structure can be made semi-dynamic by allowing insertions in amortized time $O(m)$, when we start with an initially empty dictionary. For the general d case the presented data structure allows d–queries to be answered in time $O(m \sum_{i=0}^{d-1} \binom{m}{i})$ by asking 1–queries for all strings within Hamming distance $d-1$ of the query string α. This improves the query time of a naïve algorithm by a factor of m. We leave as an open problem if the above query time for the general d case can be improved when the size of the data structure is $O(nm)$. For example, is there any $o(m^2)$ 2–query algorithm?

Another interesting problem which is related to the approximate query problem and the approximate string matching problem can be stated as follows. Given a binary string t of length n, is it possible to create a data structure for t of size $O(n)$ which allows 1–queries, *i.e.*, queries for occurrences of a query string with at most one mismatch, in time $O(m)$, where m is the size of the query string? By creating a compressed suffix tree of size $O(n)$ for the string, 1–queries can be answered in time $O(m^2)$ by an exhaustive search.

Acknowledgment

The authors thank Dany Breslauer for pointing out the relation to the FIND-SPLIT-ADD problem.

Bibliography

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design And Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.

[2] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, MA, 1983.

[3] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows*. Prentice-Hall, 1993.

[4] Susanne Albers and Torben Hagerup. Improved parallel integer sorting without concurrent writing. In *Proc. 3rd ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 463–472, 1992.

[5] Arne Andersson. Sublogarithmic searching without multiplications. In *Proc. 36th Ann. Symp. on Foundations of Computer Science (FOCS)*, pages 655–663, 1995.

[6] Arne Andersson. Faster deterministic sorting and searching in linear space. In *Proc. 37th Ann. Symp. on Foundations of Computer Science (FOCS)*, pages 135–141, 1996.

[7] Lars Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *Proc. 4th Workshop on Algorithms and Data Structures (WADS)*, volume 955 of *Lecture Notes in Computer Science*, pages 334–345. Springer Verlag, Berlin, 1995.

[8] Mikhail J. Atallah and S. Rao Kosaraju. An adversary-based lower bound for sorting. *Information Processing Letters*, 13:55–57, 1981.

[9] Michael D. Atkinson, Jörg-Rüdiger Sack, Nicola Santoro, and Thomas Strothotte. Min-max heaps and generalized priority queues. *Communications of the ACM*, 29(10):996–1000, 1986.

[10] Kenneth E. Batcher. Sorting networks and their applications. In *Proc. AFIPS Spring Joint Computer Conference, 32*, pages 307–314, 1968.

[11] Jit Biswas and James C. Browne. Simultaneous update of priority structures. In *Int. Conference on Parallel Processing*, pages 124–131, 1987.

[12] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13:422–426, 1970.

[13] Béla Bollobás and Istvan Simon. Repeated random insertion into a priority queue. *Journal of Algorithms*, 6:466–477, 1985.

[14] Allan Borodin, Leonidas J. Guibas, Nancy A. Lynch, and Andrew C. Yao. Efficient searching using partial ordering. *Information Processing Letters*, 12:71–75, 1981.

[15] Gerth Stølting Brodal. Fast meldable priority queues. In *Proc. 4th Workshop on Algorithms and Data Structures (WADS)*, volume 955 of *Lecture Notes in Computer Science*, pages 282–290. Springer Verlag, Berlin, 1995.

[16] Gerth Stølting Brodal. Partially persistent data structures of bounded degree with constant update time. *Nordic Journal of Computing*, 3(3):238–255, 1996.

[17] Gerth Stølting Brodal. Priority queues on parallel machines. In *Proc. 5th Scandinavian Workshop on Algorithm Theory (SWAT)*, volume 1097 of *Lecture Notes in Computer Science*, pages 416–427. Springer Verlag, Berlin, 1996.

[18] Gerth Stølting Brodal. Worst-case efficient priority queues. In *Proc. 7th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 52–58, 1996.

[19] Gerth Stølting Brodal. Predecessor queries in dynamic integer sets. In *Proc. 14th Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 1200 of *Lecture Notes in Computer Science*, pages 21–32. Springer Verlag, Berlin, 1997.

[20] Gerth Stølting Brodal, Shiva Chaudhuri, and Jaikumar Radhakrishnan. The randomized complexity of maintaining the minimum. *Nordic Journal of Computing, Selected Papers of the 5th Scandinavian Workshop on Algorithm Theory (SWAT'96)*, 3(4):337–351, 1996.

[21] Gerth Stølting Brodal and Leszek Gąsieniec. Approximate dictionary queries. In *Proc. 7th Combinatorial Pattern Matching (CPM)*, volume 1075 of *Lecture Notes in Computer Science*, pages 65–74. Springer Verlag, Berlin, 1996.

[22] Gerth Stølting Brodal and Chris Okasaki. Optimal purely functional priority queues. *Journal of Functional Programming*, December 1996.

[23] Gerth Stølting Brodal, Jesper Larsson Träff, and Christos D. Zaroliagis. A parallel priority data structure with applications. In *Proc. 11th Int. Parallel Processing Symposium (IPPS)*, pages 689–693, 1997.

[24] Mark R. Brown. Implementation and analysis of binomial queue algorithms. *SIAM Journal of Computing*, 7:298–319, 1978.

[25] Adam L. Buchsbaum and Robert Endre Tarjan. Confluently persistent deques via data-structural bootstrapping. In *Proc. 4th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 155–164, 1993.

[26] Svante Carlsson. *Heaps*. PhD thesis, Dept. of Computer Science, Lund University, Lund, Sweden, 1986.

[27] Svante Carlsson, Patricio V. Poblete, and J. Ian Munro. An implicit binomial queue with constant insertion time. In *Proc. 1st Scandinavian Workshop on Algorithm Theory (SWAT)*, volume 318 of *Lecture Notes in Computer Science*, pages 1–13. Springer Verlag, Berlin, 1988.

[28] Danny Z. Chen and Xiaobo Hu. Fast and efficient operations on parallel priority queues (preliminary version). In *Algorithms and Computation: 5th International Symposium, ISAAC '93*, volume 834 of *Lecture Notes in Computer Science*, pages 279–287. Springer Verlag, Berlin, 1994.

[29] Richard Cole. Parallel merge sort. *SIAM Journal of Computing*, 17(4):130–145, 1988.

[30] Sajal K. Das, Maria C. Pinotti, and Falguni Sarkar. Optimal and load balanced mapping of parallel priority queues in hypercubes. To appear in IEEE Transactions on Parallel and Distributed Systems.

[31] Paul F. Dietz. Fully persistent arrays. In *Proc. 1st Workshop on Algorithms and Data Structures (WADS)*, volume 382 of *Lecture Notes in Computer Science*, pages 67–74. Springer Verlag, Berlin, 1989.

[32] Paul F. Dietz. Heap construction in the parallel comparison tree model. In *Proc. 3rd Scandinavian Workshop on Algorithm Theory (SWAT)*, volume 621 of *Lecture Notes in Computer Science*, pages 140–150. Springer Verlag, Berlin, 1992.

[33] Paul F. Dietz and Rajeev Raman. Persistence, amortization and randomization. In *Proc. 2nd ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 78–88, 1991.

[34] Paul F. Dietz and Rajeev Raman. A constant update time finger search tree. *Information Processing Letters*, 52:147–154, 1994.

[35] Paul F. Dietz and Rajeev Raman. Very fast optimal parallel algorithms for heap construction. In *Proc. 6th Symposium on Parallel and Distributed Processing*, pages 514–521, 1994.

[36] Paul F. Dietz and Daniel D. Sleator. Two algorithms for maintaining order in a list. In *Proc. 19th Ann. ACM Symp. on Theory of Computing (STOC)*, pages 365–372, 1987.

[37] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[38] Yuzheng Ding and Mark Allen Weiss. The relaxed min-max heap. *Acta Informatica*, 30:215–231, 1993.

[39] Ernst E. Doberkat. Deleting the root of a heap. *Acta Informatica*, 17:245–265, 1982.

[40] Ernst E. Doberkat. An average case analysis of Floyd's algorithm to compute heaps. *Information and Control*, 61:114–131, 1984.

[41] Danny Dolev, Yuval Harari, Nathan Linial, Noam Nisan, and Michael Parnas. Neighborhood preserving hashing and approximate queries. In *Proc. 5th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 251–259, 1994.

[42] Danny Dolev, Yuval Harari, and Michael Parnas. Finding the neighborhood of a query in a dictionary. In *Proc. 2nd Israel Symposium on Theory of Computing and Systems*, pages 33–42, 1993.

[43] James R. Driscoll, Harold N. Gabow, Ruth Shrairman, and Robert Endre Tarjan. Relaxed heaps: An alternative to fibonacci heaps with applications to parallel computation. *Communications of the ACM*, 31(11):1343–1354, 1988.

[44] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert Endre Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38:86–124, 1989.

[45] James R. Driscoll, Daniel D. K. Sleator, and Robert Endre Tarjan. Fully persistent lists with catenation. In *Proc. 2nd ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 89–99, 1991.

[46] Rolf Fagerberg. A generalization of binomial queues. *Information Processing Letters*, 57:109–114, 1996.

[47] Michael J. Fischer and Michael S. Paterson. Fishspear: A priority queue algorithm. *Journal of the ACM*, 41(1):3–30, 1994.

[48] Rudolf Fleischer. A simple balanced search tree with $O(1)$ worst-case update time. In *Algorithms and Computation: 4th International Symposium, ISAAC '93*, volume 762 of *Lecture Notes in Computer Science*, pages 138–146. Springer Verlag, Berlin, 1993.

[49] Robert W. Floyd. Algorithm 245: Treesort3. *Communications of the ACM*, 7(12):701, 1964.

[50] Edward Fredkin. Trie memory. *Communications of the ACM*, 3:490–499, 1962.

[51] Michael L. Fredman, Janós Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, 1984.

[52] Michael L. Fredman, Robert Sedgewick, Daniel D. Sleator, and Robert Endre Tarjan. The pairing heap: A new form of self–adjusting heap. *Algorithmica*, 1:111–129, 1986.

[53] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[54] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47:424–436, 1993.

[55] Harold N. Gabow and Robert Endre Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30:209–221, 1985.

[56] Andrew V. Goldberg, Serge A. Plotkin, and Pravin M. Vaidya. Sublinear-time parallel algorithms for matching and related problems. *Journal of Algorithms*, 14(2):180–213, 1993.

[57] Leslie M. Goldschlager, Ralph A. Shaw, and John Staples. The maximum flow problem is LOGSPACE complete for P. *Theoretical Computer Science*, 21:105–111, 1982.

[58] Gaston H. Gonnet and J. Ian Munro. Heaps on heaps. *SIAM Journal of Computing*, 15(4):964–971, 1986.

[59] Dan Greene, Michal Parnas, and Frances Yao. Multi-index hashing for information retrieval. In *Proc. 35th Ann. Symp. on Foundations of Computer Science (FOCS)*, pages 722–731, 1994.

[60] Leo J. Guibas, Edward M. McCreight, Michael F. Plass, and Janet R. Roberts. A new representation for linear lists. In *Proc. 9th Ann. ACM Symp. on Theory of Computing (STOC)*, pages 49–60, 1977.

[61] Y. Han, Victor Pan, and John H. Reif. Algorithms for computing all pair shortest paths in directed graphs. In *Proc. 4th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 353–362, 1992.

[62] Godfrey H. Hardy, John E. Littlewood, and György Pólya. *Inequalities*. Cambridge University Press, Cambridge, 1952.

[63] Peter Høyer. A general technique for implementation of efficient priority queues. In *Proc. 3rd Israel Symposium on Theory of Computing and Systems*, pages 57–66, 1995.

[64] Scott Huddleston and Kurt Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982.

[65] Hiroshi Imai and Taka Asano. Dynamic orthogonal segment intersection search. *Journal of Algorithms*, 8:1–18, 1987.

[66] Joseph JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[67] Haim Kaplan and Robert Endre Tarjan. Persistent lists with catenation via recursive slowdown. In *Proc. 27th Ann. ACM Symp. on Theory of Computing (STOC)*, pages 93–102, 1995.

[68] Haim Kaplan and Robert Endre Tarjan. Purely functional representations of catenable sorted lists. In *Proc. 28th Ann. ACM Symp. on Theory of Computing (STOC)*, pages 202–211, 1996.

[69] Richard M. Karp, Eli Upfal, and Avi Wigderson. Constructing a maximum matching is in random NC. *Combinatorica*, 6(1):35–38, 1986.

[70] Chan M. Khoong. Optimal parallel construction of heaps. *Information Processing Letters*, 48:159–161, 1993.

[71] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, Reading, MA, 1973.

[72] Frank Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann, 1992.

[73] Christos Levcopoulos and Mark H. Overmars. A balanced search tree with $O(1)$ worst-case update time. *Acta Informatica*, 26:269–277, 1988.

[74] Udi Manber and Sun Wu. An algorithm for approximate membership checking with application to password security. *Information Processing Letters*, 50:191–197, 1994.

[75] Colin McDiarmid. Average-case lower bounds for searching. *SIAM Journal of Computing*, 17(5):1044–1060, 1988.

[76] Kurt Mehlhorn and Athanasios K. Tsakalidis. Data structures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*. MIT Press/Elsevier, 1990.

[77] Peter Bro Miltersen. Lower bounds for Union-Split-Find related problems on random access machines. In *Proc. 26th Ann. ACM Symp. on Theory of Computing (STOC)*, pages 625–634, 1994.

[78] Peter Bro Miltersen. Lower bounds for static dictionaries on RAMs with bit operations but no multiplications. In *Proc. 23rd Int. Colloquium on Automata, Languages and Programming (ICALP)*, volume 1099 of *Lecture Notes in Computer Science*, pages 442–453. Springer Verlag, Berlin, 1996.

[79] Peter Bro Miltersen, Noam Nisan, Shmuel Safra, and Avi Wigderson. On data structures and asymmetric communication complexity. In *Proc. 27th Ann. ACM Symp. on Theory of Computing (STOC)*, pages 103–111, 1995.

[80] Marvin Minsky and Seymour Papert. *Perceptrons*. MIT Press, Cambridge, Mass., 1969.

[81] Ketan Mulmuley, Umesh V. Vazirani, and Vijay V. Vazirani. Matching is as easy as matrix inversion. *Combinatorica*, 7(1):105–113, 1987.

[82] J. Ian Munro and Hendra Suwanda. Implicit data structures for fast search and update. *Journal of Computer and System Sciences*, 21:236–250, 1980.

[83] Chris Okasaki. Purely functional random-access lists. In *Functional Programming Languages and Computer Architecutre*, pages 86–95, 1995.

[84] Chris Okasaki. Simple and efficient purely functional queues and deques. *Journal of Functional Programming*, October 1995.

[85] Chris Okasaki. *Purely Functional Data Structures*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1996. Tech report CMU-CS-96-177.

[86] Mark H. Overmars. *The Design of Dynamic Data Structures*, volume 156 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, 1983.

[87] Richard C. Paige and Clyde P. Kruskal. Parallel algorithms for shortest path problems. In *Int. Conference on Parallel Processing*, pages 14–20, 1985.

[88] Maria C. Pinotti, Sajal K. Das, and Vincenzo A. Crupi. Parallel and distributed meldable priority queues based on binomial heaps. In *Int. Conference on Parallel Processing*, 1996.

[89] Maria C. Pinotti and Geppino Pucci. Parallel priority queues. *Information Processing Letters*, 40:33–40, 1991.

[90] Maria C. Pinotti and Geppino Pucci. Parallel algorithms for priority queue operations. *Theoretical Computer Science*, 148(1):171–180, 1995.

[91] Thomas Porter and Istvan Simon. Random insertion into a priority queue structure. *IEEE Transactions on Software Engineering*, 1(3):292–298, 1975.

[92] Rajeev Raman. *Eliminating Amortization: On Data Structures with Guaranteed Response Time*. PhD thesis, University of Rochester, New York, 1992. Computer Science Dept., U. Rochester, tech report TR-439.

[93] Rajeev Raman. Priority queues: Small, monotone and trans-dichotomous. In *ESA '96, Algorithms*, volume 1136 of *Lecture Notes in Computer Science*, pages 121–137. Springer Verlag, Berlin, 1996.

[94] Abhiram Ranade, Szu-Tsung Cheng, Etienne Deprit, Jeff Jones, and Sun-Inn Shih. Parallelism and locality in priority queues. In *Proc. 6th Symposium on Parallel and Distributed Processing*, pages 490–496, 1994.

[95] Nageswara S. V. Rao and Weixiong Zhang. Building heaps in parallel. *Information Processing Letters*, 37:355–358, 1991.

[96] V. Nageshwara Rao and Vipin Kumar. Concurrent access of priority queues. *IEEE Transactions on Computers*, 37(12):1657–1665, 1988.

[97] Jörg-Rüdiger Sack and Thomas Strothotte. An algorithm for merging heaps. *Acta Informatica*, 22:171–186, 1985.

[98] Neil Sarnak and Robert Endre Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29:669–679, 1986.

[99] Daniel Dominic Sleator and Robert Endre Tarjan. Self adjusting heaps. *SIAM Journal of Computing*, 15(1):52–68, 1986.

[100] Robert Endre Tarjan. *Data Structures and Network Algortihms.* Society for Industrial and Applied Matehmatics, Philadelphia, Pennsylvania, 1983.

[101] Robert Endre Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6:306–318, 1985.

[102] Mikkel Thorup. On RAM priority queues. In *Proc. 7th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 59–67, 1996.

[103] Jesper Larsson Träff and Christos D. Zaroliagis. Simple parallel algorithm for the single-source shortest path problem on planar digraphs. In *A Parallel Algorithms for Irregularly Structured Problems (IRREGULAR'96)*, volume 1117 of *Lecture Notes in Computer Science*, pages 183–194. Springer Verlag, Berlin, 1996.

[104] Peter van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6:80–82, 1977.

[105] Peter van Emde Boas. Machine models and simulations. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity.* MIT Press/Elsevier, 1990.

[106] Peter van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

[107] Jan van Leeuwen. The composition of fast priority queues. Technical Report RUU-CS-78-5, Department of Computer Science, University of Utrecht, 1978.

[108] Jean Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–315, 1978.

[109] John William Joseph Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, 1964.

[110] A. C-C. Yao. Probabilistic computations: Towards a unified measure of complexity. In *Proc. 17th Ann. Symp. on Foundations of Computer Science (FOCS)*, pages 222–227, 1977.

[111] Andrew C. Yao. Should tables be sorted? *Journal of the ACM*, 28(3):615–628, 1981.

[112] Andrew C. Yao and Frances F. Yao. Dictionary look-up with small errors. In *Proc. 6th Combinatorial Pattern Matching (CPM)*, volume 937 of *Lecture Notes in Computer Science*, pages 388–394. Springer Verlag, Berlin, 1995.

Recent BRICS Dissertation Series Publications

DS-97-1 **Gerth Stølting Brodal.** ***Worst Case Efficient Data Structures*****. January 1997. Ph.D. thesis. x+121 pp.**

DS-96-4 **Torben Braüner.** ***An Axiomatic Approach to Adequacy*****. November 1996. Ph.D. thesis. 168 pp.**

DS-96-3 **Lars Arge.** ***Efficient External-Memory Data Structures and Applications*****. August 1996. Ph.D. thesis. xii+169 pp.**

DS-96-2 **Allan Cheng.** ***Reasoning About Concurrent Computational Systems*****. August 1996. Ph.D. thesis. xiv+229 pp.**

DS-96-1 **Urban Engberg.** ***Reasoning in the Temporal Logic of Actions — The design and implementation of an interactive computer system*****. August 1996. Ph.D. thesis. xvi+222 pp.**