Progress Report
Complexity of Data Structures

Gerth Stølting Brodal
BRICS[1]
Computer Science Department, University of Aarhus
Ny Munkegade, DK-8000 Aarhus C, Denmark
gerth@daimi.aau.dk

November 19, 1994

[1]**B**asic **R**esearch **i**n **C**omputer **S**cience, Centre of the Danish National Research Foundation

Abstract

This progress report presents the work accomplished by the author during part A of the Ph.D. programme at the University of Aarhus.

We consider the complexity of different data structures, and introduce the distinction between *query* and *restructuring* complexity of data structures. In the light of three different computational frameworks we argue that data structures should be designed to have minimal restructuring complexity.

The main result is the result of [3] where we show how to make bounded degree data structures partially persistent with worst case slowdown in $O(1)$. We also give a restricted result for the case of fully persistent data structures.

Reference [3] is appended at the end of the report.

Contents

Chapter 1

Introduction

In this progress report we consider the complexity of data structures. Many existing data structures are characterised by updates consisting of *cascades of local restructurings*. Typical local restructurings are copying or splitting of nodes, linking or cutting of trees or rotations in a tree. We are interested in how to avoid or control these cascades of local restructurings. In this report we review known approaches to improving the restructuring complexity of data structures. The results obtained by the author are presented in Chap. 3, Chap. 4, Chap. 5 and [3].

In the following we assume that data structures are described in the pointer machine model [26]. So our view of a data structure is a directed graph of bounded out-degree where each node contains a fixed amount of information. We also assume that a single operation on a data structure consists of one or more of the following three independent steps:

i) Perform a search in the data structure to locate a given position.

ii) Perform an update at the position found in *i*).

iii) Reorganise the data structure such that it satisfies some structural constraints.

As a concrete example consider an insertion into an (a, b)–tree [18]. Here *i*) is a search in the tree, *ii*) is the creation of a new leaf containing the element to be inserted and *iii*) is the splitting of a sequence of nodes so that all internal nodes (except possibly for the root) have degree between a and b.

Table 1.1 gives a short list of data structures that satisfies these assumptions.

In general the first step can often be done by an ordinary search or is trivial because the relevant position is already known. This is true for all data structures in Table 1.1. As an example step *i*) of insertions into binomial heaps and Fibonacci heaps will just be to do nothing because new elements are always inserted at the same place. The second step can often be done in time $O(1)$, as in all the given examples. What actually can take time, is to perform step *iii*) which can be regarded as maintaining the constraints on the structure. If we look at the data structures in Table 1.1 we see that restructurings are actually sequences of *local restructuring* transformations which can be performed independently of each other. Table 1.1 mentions which local restructurings are performed on the different data structures and how large the worst case times of performing step *ii*) and *iii*) are.

Data structure	Local restructuring	Step *ii) & iii)*
$(2,3)$–trees, B-trees, $(2,4)$–trees and (a,b)–trees [2, 18]	Splitting-, fusion- and sharing-steps	$\Theta(\log n)$
Red-black-trees [27]	Rotations	$\Theta(\log n)$
Dynamic fractional cascading [21]	Insertion/deletion of bridges	$\Theta(n)$
Partially persistent BID[1] [13]	Node copying	$\Theta(n)$
Fully persistent BID[1] [13]	Node splitting	$\Theta(n)$
Binomial heaps [29]	Linking binomial trees	$\Theta(\log n)$
Fibonacci heaps [15]	Linking/cutting trees	$\Theta(\log n)$
Catenable min-dequeues [4]	Path compression	$\Theta(n)$

Table 1.1: Local restructuring operations on different data structures

The above examples and the constraints that we have forced on the considered data structures give us that a data structure is characterised by two different kinds of complexities:

- The complexity of performing searches in the data structure.
- The complexity of maintaining the structural constraints on the data structure.

In the following, the *query complexity* of a data structure refers to the complexity of performing step *i*) and the *restructuring complexity* to the complexity of performing step *ii*) and *iii*).

Below we give three applications of data structures where it is important that we make a distinction between the query and the restructuring complexity. We present two parallel environments and one sequential environment.

In the first parallel environment we have a number of processes that share a data structure. Here it is often the restructuring steps that are expensive because they need to lock the data structure (or parts of the data structure) to be able to perform the updates safely. There is no problem when performing searches, because processes can read the same nodes of the data structure concurrently without blocking each other. So in this parallel environments it is important that the restructuring complexity is as small as possible.

Another parallel environment is the following (of Smid [25]). Instead of having only one data structure we maintain several copies of a data structure in a network of processes where all processes have their own memory. There is one *central structure* on which updates are performed and which is maintained by a special process. All other processes have copies of the central structure (or a restricted version of it) which are called the *client structures*. The processes are only allowed to perform queries on the client structures, updates have to be done via the central structure. An update is performed by the process having the central structure. After having updated the central structure the necessary information is broadcasted to all the processes. We are interested in the *client update time*, which is defined as the time a client needs to perform the corresponding update plus the

[1]BID $\equiv$ bounded in-degree data structures

Data structure			Approach
Original		Improved[2]	
Search trees	[2, 18, 27]	*unnamed* [14, 20]	Bucketing
Finger search trees	[17]	*unnamed* [9]	Bucketing + RAM model
Dynamic fractional cascading	[21]	*unnamed* [10]	Bucketing
Partially persistent BID	[13]	*unnamed* [22]	Bucketing + RAM model
		unnamed [3]	Regularity constraint
Fully persistent BID	[13]	*unnamed* [10]	Bucketing
Binomial heaps	[29]	*unnamed*[3] [6]	Regularity constraint
Fibonacci heaps	[15]	Relaxed heaps[3][12]	Structural relaxation
Catenable min-dequeues	[4]	*unnamed* [19]	Regularity constraint

Table 1.2: Improvement of restructuring complexities

number of bits in the information broadcasted [25]. For a given data structure there is a simple protocol that bounds the clients update time by the restructuring complexity of the data structure — the central process just has to broadcast the locations of the different modifications and the modifications done (this assumes that the index of a memory cell has size $O(1)$ and that the clients have random access to their data structures). A simple example of this strategy is to maintain a balanced binary search tree. If we let the central structure be a red-black tree [27], the client structures can be copies of the red-black tree without the colour information, because only the central structure needs this information for restructuring the tree. Because updates only involve a constant number of rotations the client update times will be worst case $O(1)$. The central update time is of course still $O(\log n)$. In Chap. 5 we give a similar result for bounded degree fully persistent data structures.

When returning to Table 1.1 we see that for four of the data structures the restructuring time can be as bad as $\Theta(n)$. But in the amortised sense (see Chap. 2) the restructuring complexities of all the four data structures are $O(1)$, so in off-line applications of the data structures the restructuring complexity does not affect the overall complexity. In on-line applications the worst case restructuring complexity is unsatisfying, especially because the worst case restructuring complexity is worse than the query complexity. The query complexity of the persistence techniques of [13] is worst case $O(1)$. If possible, we would like to have the restructuring complexity bounded by the query complexity. There can of course be a lower bound that makes this impossible, as for priority queues where at least one of the operations has to take time $\Omega(\log n)$.

A lot of work has been done in the past to improve the restructuring complexity of different existing data structures where the restructuring complexity dominates the query complexity. Table 1.2 summarises the approaches done on the data structures mentioned in Table 1.1. In Chap. 2 the different approaches will be considered in more detail.

In the following chapters we consider the different ideas used to improve the restructuring complexity of data structures. Often, the goal is to remove the amortisation from an existing data structure. In Chap. 2 we review existing approaches to remove the

[2]*unnamed* $\equiv$ the improved data structure is just an extension of the original data structure

[3]The improvements do not affect the delete operations

amortisation from data structures.

In Chap. 3 we present a counter which supports addtion/subtraction of an arbitrary power of two and test for zero in worst case constant time. This problem is of interest because a number of data structures use ideas that come from considering redundant counter representations.

In Chap. 4 and Chap. 5 we describe a new approach to improving the restructuring complexity of data structures. We consider a pebble game on graphs, that in [3] enables us to remove the amortisation from the restructuring complexity of the partial persistence technique of [13] with query complexity slowdown in $O(1)$. We also present a restricted result for the full persistence technique of [13].

Chapter 2

Elimination of Amortisation

As mentioned in Chap. 1 the restructuring complexities of the data structures we consider are characterised by being bad in the worst case sense, but often good (i.e. $O(1)$) in the amortised sense. In this chapter we briefly review the definitions of amortised complexity, give typical examples of data structures with good amortised performance and review known approaches to remove the amortisation from the restructuring complexity of different data structures.

2.1 Amortised Complexity

The concept of amortised complexity was introduced by Tarjan in [28] as an alternative to the worst case complexity measure. Tarjan defined amortised complexity as "to average the running time of operations in a sequence over the sequence" [28]. We shall view amortised complexity as the *banker view*, i.e. the computer is coin-operated, and coins are deposited in the data structure. One coin can pay for a fixed number of operations. At the beginning the data structure contains no coins. The amortised time of an operation is the number of coins we add to the data structure when we perform the operation. Unused coins can be used by latter operations to make the amortised cost less than the actual cost for these operation.

The power of amortised complexity is best illustrated by the idea of *splaying*. In [24] splay trees were introduced as an alternative to the well known variants of search trees with good worst case performance. Splay trees are characterised by being very simple, with no balancing information in the nodes, good amortised performance but bad worst case performance.

Examples of data structures where a simple coin deposit argument can give a good and very tight bound on the amortised performance are many, we just mention a list of references [13, 15, 16, 18, 21, 27, 29]. They all use the same idea to have small local *buffers* which ensure that potential cascades or local restructurings are prepayed in advance by implicitly placing coins in the buffers. An example is the cutting of trees in Fibonacci heaps [15] where it is allowed to cut off one son of a node without implying a cascaded cut. So in Fibonacci heaps each node has a buffer that allows one son to be cut off, and if a node has had a son cut off it contains coins to pay for a latter cut.

In the following sections we consider the approaches that have been developed to avoid cascades of local restructurings.

2.2 Global rebuilding

The simplest example of an application of the global rebuilding technique is min-dequeues. A min-dequeue is a double ended queue that supports insertion and deletion of elements at both ends of the queue, and which can return — but not delete — the current minimum element in the queue.

In [16] two different implementations are given. One with amortised $O(1)$ restructuring complexity and one with worst case $O(1)$. The amortised solution is based on two min-stacks whose concatenation is equal to the queue. When one of the min-stacks becomes empty two new min-stacks of about the same size are constructed. The conversion to a worst-case variant is done by the global rebuilding technique, where a new (more) "balanced" version is build incrementally by a process in the background. Here the measure of balance is the difference of the sizes of the two min-stacks. The main requirement to get global rebuilding to work appropriately is that the data structure does not degenerate faster than it can be rebuild.

In Sect. 2.4 we summerise the further work that has been done on min-dequeues the recent years.

2.3 Bucketing

Raman considered in his Ph.D. thesis several techniques to eliminate amortisation from different data structures [22]. The techniques are based on the following combinatorial *continuous zeroing game* (and various variations of the game). The game is played by two players **I** and **D** on n variables $x_1, \ldots, x_n$. Initially the variables are all zero. The two players-line alternate to perform the following moves:

Player I: Chooses n non negative real numbers $q_1, \ldots, q_n$ such that $\sum_{i=1}^{n} q_i = 1$, and sets $x_i \leftarrow x_i + q_i$ for $i = 1, \ldots, n$.

Player D: Chooses an integer $i \in \{1, \ldots, n\}$ and sets $x_i \leftarrow 0$.

The goal of the game is to give a strategy for player **D** that bounds the values of the x_i's as much as possible. Let M be a number such that $x_i \leq M$ for all i. We have the following theorem:

Theorem 1 (Dietz and Sleator [11]) *If player* **D** *picks i such that $x_i = \max_j\{x_j\}$, then will $M = \Theta(\log n)$ and this strategy is optimal.*

The upper bound is obtained by showing that $x_i \leq H_{n-1} + 1$ for all i, where H_k is the k'th hamonic number. The lower bound is obtained by letting player **I** uniformly increase all variables not zeroed. This leads to a situation where $M \geq H_n - 1$.

A variation of the zeroing game is the halving game where x_i is halved instead of being zeroed, $x_i \leftarrow x_i/2$. That this game also has $M = \Theta(\log n)$ is an easy consequence of Theorem 1, because playing the halving game on $x_1, \ldots, x_n$ corresponds to playing the zeroing game on $y_i = \max\{0, x_i - H_{n-1} - 2\}$.

The theorem is essential to the idea of *bucketing* [9, 11, 20, 22]. We will not go into the details of the technique but just sketch the main idea. A set of n elements is partitioned into buckets of polylogarithmic size and, at regular intervals, with frequency

f, the largest bucket of elements is split into two smaller buckets. By using the above theorem it is possible to show that the size of the largest bucket will be bounded by $O(f \log n)$, because the number of buckets is less than n and we have a variation of the halving game scaled up with a factor of f. In [9, 20] f is $O(\log n)$ so the buckets are of size $O(\log^2 n)$. The representation and implementation of the buckets, as well as the value of f, are specific to the given problem to which the bucketing technique is applied. The resulting data structures are *hybrid* or *two-level* data structures. The interval f between two zeroings or splittings can be used to perform a lazy update on the top level data structure as in the search trees of [20].

By appropriate representations of the buckets [22] they can in some situations be implemented on the RAM such that they can be manipulated in time $O(1)$. The idea is to put sets of size $O(\log n)$ into a constant number of words on the RAM and to store a number of incrementally built tables [1, 9].

Two implementations of search trees that use this lemma are the search trees of [14] and [20], where it is possible to perform updates in worst case time $O(1)$, when it is known where to insert/delete an element. Different representations of the buckets are used that do not need the RAM model.

2.4 Data Structural Bootstrapping

Recently a new approach has been taken to develop data structures — *data structural bootstrapping* [5]. Given an implementation of a data structure that has a limited repertoire of operations, a new implementation of the data structure can be constructed by recursively applying the old data structure and thereby extending the repertoire of supported operations. The new data structure is a tree where all nodes correspond to instances of the original data structure.

In [4] catenable heap ordered double ended queues (catenable min-dequeues) are constructed in this way by recursively using min-dequeues. The main operation is a pull operation which pulls a subtree towards the root of the tree of min-dequeues. In a complex analysis it is shown that the number of pulls is linear in the number of operations, which gives an amortised performance in $O(1)$.

Kosaraju has recently shown [19] that the amortised bound can be made worst case. Again the approach is data structural bootstrapping. But in contrast with the original construction the new construction uses two levels of bootstrapping. First a new implementation of min-dequeues is constructed. These are then extended to restricted catenable min-dequeues, where catenations of min-dequeues are performed lazily over the subsequent sequence of insertions and deletions, and finally catenable min-dequeues are constructed by using data structural bootstrapping. The interesting idea of the applied form of bootstrapping is that it is based on an explicit *regularity* constraint, which guarantees that nodes can always be removed in worst case time $O(1)$. The idea is to consider a left-to-right Euler walk of the tree of restricted min-dequeues and then to maintain the invariant that to the left of the i'th node in the Euler walk there is at least $\Omega(i)$ elements. More precisely that the potential to the left of the i'th node is $\Omega(i)$ where the potential is defined as the number of elements minus the heights of the restricted min-dequeues in the leafs and minus the missing amount of lazy melding which still remains to be performed on these restricted min-dequeues. The symmetric invariant is maintained on the right-to-left Euler walk of the tree.

2.5 Regularity Constraints and Structural Relaxing

As mentioned above, a regularity condition is used in [19] to achieve the worst case bound of $O(1)$ of the restructuring complexity of catenable min-dequeues. We give two other applications of this idea.

The first is the maintenance of a priority queue. A simple and elegant implementation is the binomial heap [29], that supports the operations INSERT and DELETE in time $O(\log n)$ (INSERT amortised $O(1)$) and FINDMIN in time $O(1)$. Binomial heaps were later extended to Fibonacci heaps [15], which also supports a DECREASEKEY in amortised time $O(1)$. As mentioned in Sect. 2.1 the amortised bound on DECREASEKEY is reached by relaxing the structural constraint on the data structure. The restructuring complexity of Fibonacci heaps was improved to worst case $O(1)$ for all operations, except for delete which costs $O(\log n)$, by relaxed heaps [12]. The general idea is to relax the heap order at $O(\log n)$ nodes in the Fibonacci heaps. This enables the data structure to be improved lazily with worst case work $O(1)$ per update. Another approach to improve the insertion time in binomial heaps to worst case $O(1)$ is taken in [6]. Here the idea is to relax the number of binomial trees of height h to be between zero and two. It is shown that by repeating the step of linking the two smallest binomial trees of the same height three times per insertion, the number of binomial trees of the same height is between zero and two and the number of heights where two binomial trees are of the same height is $O(\log^* n)$.

The second example of a data structure where a relaxation improves the restructuring complexity is the finger search trees of [17]. The basic structure used to represent a sorted list is an (a, b)–tree where a finger is a pointer to a leaf. To be able to insert/delete an element in the neighbourhood of a finger in time $O(\log d)$, where d is the distance between the finger and the node to insert, a regularity constraint is maintained on the path from the finger to the the root of the (a, b)–tree. The regularity constraint implies that it is sufficient to split only one node on the path per insertion. The idea is related to the idea of using redundant counters, a problem we consider in Chap. 3. The regularity condition is very simple. Each node on the path has at least two and at most six sons. Between two nodes with six sons on the path there will be at least one node with at most four sons, and between two nodes with two sons at least one node has at least four sons. The only secondary data structures needed on the path are two double linked lists of nodes, containing the nodes with respectively two and six sons.

An important property we will mention is that the explicitly stated regularity constraint only involves one path and therefore only works for a single finger. To our knowledge no corresponding regularity constraints exist for trees that would enable multiple fingers. Multiple fingers are also considered in [17] but the idea is just to maintain a path for each finger and the constants in the construction increase with the number of fingers to maintain.

2.6 Summary

The ideas to improve the restructuring complexity of data structures that have been used in the past can be summarised by the following list:

- Global rebuilding of the data structure.
- Partition the elements into buckets.
- Place small arrays into a single word on the RAM model.
- Relaxation of the structure constraints.
- Regularity conditions on the structure of the data structure.

Chapter 3

A Counter

The result of this chapter is a new counter where it is possible to increase and decrease an arbitrary "bit". More precisely we implement a data type over Z that supports the following three operations.

- Add(i) adds 2^i to the counter,
- Sub(i) subtracts 2^i from the counter,
- Zero tests if the counter is equal to zero.

We assume that the length (i.e. the number of bits in the binary representation) of the counter is bounded by N. Our result is:

Theorem 2 *A data structure exists that implements* Add(i), Sub(i) *and* Zero *in worst case time* $O(1)$.

The motivation to study this problem is that a number of data structures involve ideas that come from considering redundant counter representations [6, 17]. Our counter shows how to avoid the predecessor problem on a subset of the set $\{1, \ldots, N\}$, which seemed fundamental to the redundant counter that Guibas *et al.* used in the construction of finger search trees [17]. We have sketched the redundant counter of [17] in Sect. 2.5 in terms of a regularity constraint on a path in a tree. In terms of a redundant counter, the idea is to have a redundant counter representation where the digits are allowed to be between -2 and 2. Between two digits of value 2 there will at least be one digit that is less than or equal to zero, and similarly for -2. The main problem with this counter is that when adding or subtracting 2^i for an arbitrary i, the implementation needs to find the nearest digit that is 2 or -2.

The solution we sketch is based on the power of shared indirect pointers.

The usual binary representation of a number n is a string of digits $(a_{\lfloor \log n \rfloor}, \ldots, a_0)$ such that $a_i \in \{0, 1\}$ and $n = \sum_{i=0}^{\lfloor \log n \rfloor} a_i 2^i$. In our redundant representation we allow $a_i \in \{-2, -1, 0, 1, 2\}$. When using this representation n can have many different representations: $(1,0,1), (2,-1,-1)$ and $(1,1,-1)$ all represent the value 5_{10}.

When performing Add(i) (Sub(i)) the first step is to increment (decrement) a_i. To guarantee that the digits are in the given bounds we have to do some additional value preserving transformations on the digits. The transformations that we allow are: $(a_{j+1}, a_j) \leftarrow (a_{j+1} + 1, a_j - 2)$ and $(a_{j+1}, a_j) \leftarrow (a_{j+1} - 1, a_j + 2)$. Compared to binomial heaps this corresponds to joining two binomial trees of height i to obtain a new tree of height $i+1$ or splitting a tree of height $i+1$ into two trees of height i. When performing an Add or Sub operation we only want to do $O(1)$ transformations of this type. For simplicity we allow that temporary while performing the transformations the values of the a_is to become outside the given bounds.

The main idea in our data structure is the following observation about a sequence of 1s in the representation of a value (in the following we underline such sequences):

$$(\underline{1,1,1,1,1}) \equiv (1,\underline{0,0,0,0,-1}) \equiv (\underline{2,0,0,0,-1}).$$

A similar observation holds for a sequence of -1s. We will call maximal sequences of 1's 1–blocks (respectively -1–blocks). For each index we will maintain information about which block it is contained in. In the following we describe how to do that.

The situation we want to avoid is the following where two 1–blocks get joined because of an Add operation (the italic digits indicate the position where we perform Add):

$$(\underline{1,1,1},\mathit{0},\underline{1,1},0,0) \;\triangleright (\underline{1,1,1,\mathit{1},1,1},0,0).$$

Instead we want the result of the transformation to be $(\underline{1},0,0,\underline{-1},\underline{\mathit{1},1,1},0,0)$, where we use one of the above identities on the leftmost block in the original counter. Because a 1–block can be arbitrary long we can not do this transformation in time $O(1)$. This can be avoided if we instead do the transformation in advance by using the other identity mentioned. The above mentioned example will instead become the following transformation:

$$(\underline{2,0,-1},\mathit{0},\underline{2,-1},0,0) \;\triangleright (\underline{1},0,0,\underline{-1},\underline{\mathit{2},0,-1},0,0).$$

The only non trivial information we need to store for each digit is whether it belongs to a ±1–block and, if so, the leftmost and rightmost digit in the block (remember that $(\underline{2,0,0,0,-1})$ is a 1–block). This problem we solve in a way that resembles the partial persistence problem [3] by using the power of shared indirect pointers. Each digit contains a pointer to a *block-record* that contains pointers to both the extremes of a ±1–block. If these pointers are NULL the digits do not belong to a block. It is now clear how to append a new ±1 to an existing block — we just have to set the block pointer of the new digit to the block pointer of its neighbour digit and update one of the extreme pointers in the block-record.

There are a number of cases to consider when performing Add and Sub. We mention only two cases of Add, which capture the central idea of the construction:

$$(\underline{1},0,\underline{2,0,0,0,-1},\mathit{0},\underline{2,0,1}) \;\triangleright (\underline{2,-1},0,0,0,0,\underline{-1},\underline{\mathit{2},0,0,1}),$$

$$(\underline{2,-1},\underline{-2,\mathit{0},1},\underline{2,0,-1}) \;\triangleright (\underline{1},0,0,\underline{\mathit{2},0,0,0,1}).$$

What happens is that we destroy a ±1–block by performing $O(1)$ transformations at the ends of the block, exploiting that we have an alternative representation of the block and that we have access to where the ends of the block are. The transformations at the ends of the block either expand or shrink the neighbouring blocks or create new blocks containing only one digit. By doing these block transformations in an appropriate way we can avoid joining two neighbouring ±1–blocks with the same digits when we perform Add and Sub as illustrated by the first example.

We skip further details of the implementation; the above discussion should give a clear idea of the implementation. We just have to maintain the invariant that we do not have two adjacent blocks with the same digits.

The ZERO test is very simple. We just count how many digits are different from zero. The counter is zero if and only if all digits are zero. This is because the least significant non zero digit is always a ± 1 (strings like $(1, -2)$ are not possible, because we maintain ± 1–blocks).

The counter presented here is a good example of the power of indirect pointers combined with structural relaxation[1].

We conclude this chapter by mentioning two problems where the ideas perhaps can be used. It is data structural problems, where there are gaps between the best known worst case bounds and the amortised bounds on the update complexities.

- Is it possible to construct heaps, such that INSERT, FINDMIN and MERGE can be done in worst case time $O(1)$ and DELETE in time $O(\log n)$? Binomial heaps [29] do it in the amortised sense.

  By using the ideas of the described counter we can modify binomial heaps such that they can support the insertion of a binomial tree of arbitrary height in worst case constant time and that the number of binomial trees of every height at most is a constant.

- Does an extension of (a, b)–trees [18] exist, where it is enough to do one split operation per insertion, and the place to split can be found in worst case time $O(1)$ (on the pointer machine model)? This is the simplest version of the splitting game of Chap. 4.

  This problem can also be viewed as the problem to maintain finger search trees with an arbitrary number of fingers. It was this problem that originally motivated the study of the counter. If we only have to maintain $O(1)$ fingers we can just replace the counter in Guibas *et al.* [17] by our counter.

[1] Here the redundancy of the representation

Chapter 4

Games on Graphs

In Chap. 2 we presented a general technique to remove the amortisation from the restructuring complexity of data structures. One draw back of this idea is that it introduces the idea of bucketing that does not necessarily have natural relation to the original problem. An example is the construction of finger search trees in [9] where it becomes necessary to use the power of the RAM model to put small arrays into a single word. Another example is the removing of the amortisation from the update steps of the full persistency technique of [13]. The original persistency technique could perform query steps with slowdown in $O(1)$ whereas by using the bucketing technique the slowdown is in $O(\log\log n)$ [10].

In the following we take a completely different approach, that is based on the local properties of the data structure. The problem with techniques based on the combinatorial lemma in Chap. 2 is that it does not use the topology of the data structure at all, so the fact that restructurings are local is neglected completely.

If we consider data structures to be graphs we can use the notion of on-line two player games on graphs where the moves of the first player correspond to performing step *ii*) of updates and the moves of the second player correspond to the local restructurings on a data structure. The goal of the games is to find strategies for the second player which:

- guarantee that some structural constraints on the graphs are satisfied,
- do as few local restructurings as possible,
- helps locating the position to perform the restructurings as fast as possible.

If we are interested in maintaining multiple versions of a data structure in a parallel environment [25] the last item is not as important as the first two, because it is the number of restructurings that is expensive.

Two different games will be considered. The first game is strongly related to the partial persistence technique [13] and the second game is related to the full persistence technique [13], finger search trees [14] and dynamic fractional cascading [21].

4.1 Zeroing Game

The first game is played on a directed graph $G = (V, E)$ of bounded out-degree d and bounded in-degree b. On each node of the graph we place a number of *pebbles*. The

Player **I**:

a) adds a pebble to an arbitrary node v of the graph or

b) removes an existing edge (v, u) and creates a new edge (v, w) without violating the degree constraints, and places a pebble on the node v.

Player **D**:

c) does nothing or

d) picks a node v and removes all pebbles from v and places a new pebble on all the predecessors of v.

Figure 4.1: Zeroing game.

two players **I** and **D** alternate to do one of the moves described in Fig. 4.1. We call this game the *zeroing game* on graphs. The goal of the game is to find a strategy for player **D** that can guarantee that the number of pebbles on all nodes is bounded by a constant M. The game was defined by Dietz and Raman in [10] to capture the essential problem of removing the amortisation from the partial persistence technique of [13]. They gave a strategy for player **D**, that achieved $M \leq 2b+2d+O(\sqrt{b})$. But they could not implement their strategy efficiently (they could not find the appropriate node in constant time).

In [3] we give a new strategy which improves the upper bound on the number of pebbles to $M = b+2d$. But we are not able to implement this efficiently either. However we can implement another strategy that achieves a reasonable bound. The result is the following:

Theorem 3 (Brodal [3]) *There exists a strategy for player* **D** *that achieves a bound on the number of pebbles of* $M = 2bd+1$. *Furthermore the strategy can be implemented to find the node where to remove the pebbles in time* $O(1)$ *on the pointer machine model.*

The important consequence of this theorem is that we can make data structures of bounded degree partially persistent with worst case slowdown $O(1)$, see Chap. 5 and [3]. The hidden constants in the secondary data structure we give are very small, we only need one indirect pointer per node to find the node where to remove the pebbles.

We have also considered a lower bound for the game. By fixing three different graphs we get the following lower bound on M. We have not been able to obtain the detailed dependence of M on b and d.

Theorem 4 (Brodal [3]) *For all strategies of player* **D** *where* $b, d \geq 1$

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

4.2 Splitting game

The second game we consider is the on-line splitting game on graphs that we define below. The main difference to the zeroing game is that the graphs involved in the splitting game

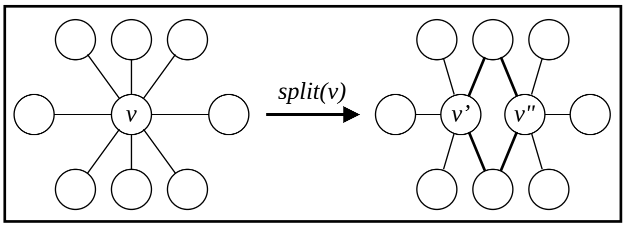

Figure 4.2: The effect of performing Split on a node v. Notice that the degree of two adjacent nodes increase by one.

Player **I**: Selects two nodes $u, v \in V$ and inserts the edge $\{u, v\}$ into the graph.

Player **S**: A number of times selects a node v and performs Split(v). The number of times is between 0 and s, where $s \in N$ is a constant.

Figure 4.3: Splitting game.

are much more dynamic and that there is no explicitly stated degree constraint on the nodes.

The game is motivated by data structures where splittings are involved. In Chap. 5 we mention a list of such data structures and show the technical details in the relationship between the data structures and the splitting game. Again the main motivation for using the terminology of games is to avoid the tedious details of the related data structures. The game is defined as simply as possible to concentrate on the main problem — *cascades of splittings* in a graph.

The game is played on an undirected graph $G = (V, E)$ without self loops and multiple edges. We assume that initially $E = \emptyset$ and $|V|$ can be infinite. The last assumption allows us to get new nodes for free. The number of nodes connected to v is denoted by $d(v)$. Let $k, \tau \in N$ and $\alpha \in]0, 1[$ be constants and Split be a function that can split a node $v \in V$ into two nodes v', v'' — provided that $d(v) \geq \tau$. Let $G' = (V', E')$ be the graph after a Split(v) has been performed. The function Split shall guarantee that the edges adjacent to $v \in V$ and to $v', v'' \in V'$ are related by the following equations. All other edges remain unchanged.

$$\begin{gathered}\{u, v\} \in E \Leftrightarrow \{u, v'\} \in E' \vee \{u, v''\} \in E', \\ |\{u | \{u, v'\} \in E' \wedge \{u, v''\} \in E'\}| \leq k, \\ d(v'), d(v'') \geq \lfloor \alpha d(v) \rfloor.\end{gathered}$$

The constant α is a measure on how evenly the edges connected to v are distributed between v' and v'', and k is a measure on how many new edges can be introduced when v is split. Figure 4.2 shows the effect of performing Split on a node v. We see that in the example $k \geq 2$.

The game is played by the two players **I** and **S**. Player **I** inserts edges into the graph and player **S** splits the nodes of the graph. More precisely the two players alternate to perform the moves in Fig. 4.3.

The goal of the game is to find a strategy for player **S** which guarantees, that whatever player **I** and the function Split do, the degree of all nodes will be bounded by a constant M. This bound will of course depend on α, k, τ and s.

Results on The Splitting Game

We have been able to give a strategy for the game that can be implemented to find the node to split in amortised time $O(1)$, but we have not found a data structure that can find the node to split in worst case time $O(1)$ — this is one of the problems we want to consider in future work.

We first give a strategy for player **S** to find the nodes on which to perform Split. The strategy is quite similar to the strategy that we give in [3].

The idea is to start controlled searches at the places where edge insertions are being performed. Where the searches end we perform Split operations. How to perform searches is described in the following.

To each node v we associate a queue Q_v (or any other set data structure where it is possible to delete an arbitrary element), which contains a subset of the nodes that is connected to v by an edge. We let a node $u \in Q_v$ be represented by the edge $\{u, v\}$ in Q_v, so that if the edge $\{u, v\}$ is moved to $\{u', v\}$ during a Split operation u indirectly is replaced by u' in Q_v.

The following procedure describes the central idea of the strategy of **S**. The procedure describes how to find a node in the graph to perform Split on.

```
procedure $pass(v)$
    while $Q_v \neq \emptyset$ do $v \leftarrow pop(Q_v)$ od
    if $d(v) \geq \tau$ then
        $(v', v'') \leftarrow \textsc{Split}(v)$
        $Q_{v'} \leftarrow Q_{v''} \leftarrow \{u | \{u, v'\} \in E \wedge \{u, v''\} \in E\}$
    fi
end.
```

When player **I** inserts the edge $\{u, v\}$ the strategy of player **S** is very simple. We assume that $s \geq 2$, but as Theorem 7 shows this is no restriction because s is always ≥ 2 whenever M is bounded by a constant.

```
The strategy of player S
    $pass(u)$
    $pass(v)$.
```

We now analyse this strategy. First we show that the amortised complexity of implementing this strategy is $O(1)$.

Theorem 5 *The strategy of player* **S** *can be implemented to perform the moves in amortised time* $O(1)$.

Proof: The amortised analysis is based on the simple coin deposit invariant that each node contains as many coins as there are elements in the node's queue. Each coin can pay for one iteration of the **while**–loop in the procedure *pass*, that for each iteration removes a node from one of the queues. Each time we split a node we have to add at most $2k$ coins to the data structure so the move of player **S** at most have to add $4k$ coins to the data structure. It is clear that the rest of the operations can be done in constant time. So the moves of player **S** can be implemented to run in amortised time $O(1)$. $\square$

Theorem 6 shows that the strategy achieves that the value of M is bounded by a constant. If we let $\alpha = 1/2$ and $\tau = 0$ we get that M is bounded by $27k + 15$. We have no reason to believe that this is an optimal bound but we only want to show that $M = O(1)$.

Theorem 6 *The given strategy for player* **S** *of the splitting game achieves*

$$M \leq \max\{(12k + 6)/\alpha, 2\tau\} + 3k + 3.$$

Proof: The analysis of the strategy is based on the following function, defined on the nodes in V:

$$P(v) = d(v) + |\{u|\{u, v\} \in E \wedge v \notin Q_u\}| + |Q_v|.$$

If we can show that for all v, $P(v)$ will be bounded by a constant we immediately get a bound on M.

We will first consider how large the value of $P(v)$ can become. Fix an arbitrary node v — where $P(v) \geq 2\tau + k$, so that $d(v) \geq \tau$. Let n_0 denote the size of $P(v)$. We will now find a bound on how large $P(v)$ is when v is split into two nodes v' and v'' — let n_1 denote this bound.

When v is created $|Q_v| \leq k$. Each time player **I** adds an edge to v we see that $|Q_v|$ decreases by at least one, because player **S** calls *pass*(v) — so at most $k + 1$ new edges adjacent to v can be added by player **I** before v will split. Hence, the edges added by player **I** increase $P(v)$ by at most $2(k + 1)$.

Now, we will consider how the procedure *pass* affects the value of $P(v)$. There are two possible reasons why $P(v)$ can increase. The first is when $d(v)$ increases. This happens when a neighbour, u, of v is split into two nodes u', u'' such that $\{u', v\}, \{u'', v\} \in E$. But in this case $v \notin Q_u$ and $v \in Q_{u'} \cap Q_{u''}$ so the second term in $P(v)$ will decrease by one and therefore the value of $P(v)$ will not be changed by $\textsc{Split}(u)$.

The second case we have to consider is when the second term of $P(v)$ increases by one. This will only happen when *pass* removes v from Q_u for some node u adjacent to v. But then the next action *pass* will remove an element from Q_v and we again have that $P(v)$ will maintain unchanged — because $\{v, v\} \notin E$. If $Q_v = \emptyset$, *pass* can not remove an element from Q_v so $P(v)$ will be increased by one. This is the last time $P(v)$ is changed because the next operation will be $\textsc{Split}(v)$. We conclude that $n_1 \leq n_0 + 2(k + 1) + 1$.

When v splits into the nodes v' and v'', $|Q_{v'}|, |Q_{v''}| \leq k$ and at most k new edges will be introduced, so we get:

$$P(v') + P(v'') \leq P(v) + 2k + 2k \leq n_0 + 2(k + 1) + 1 + 2k + 2k = n_0 + 6k + 3.$$

If we can guarantee that $d(v'), d(v'') \geq 6k + 3$ after the split, we have that $P(v'), P(v'') \leq n_0$. Because $d(v') \geq \lfloor \alpha d(v) \rfloor$ we will make the constraint that $\lfloor \alpha d(v) \rfloor \geq 6k + 3$. When v is split we now have that $P(v'), P(v'') \leq n_0$. That $\lfloor \alpha d(v) \rfloor \geq 6k + 3$ is a consequence of the constraint $n_0 \geq 2(6k + 3)/\alpha + k$. Remember that we have assumed that $n_0 \geq 2\tau + k$. Let p denote $\max\{2(6k + 3)/\alpha + k, 2\tau + k\}$. The above considerations show that if $n_0 \geq p$ then $P(v') \leq n_0$ and $P(v'') \leq n_0$. Notice also that $d(v) \leq \tau$ implies that $P(v) \leq p$.

By induction on the order of the splittings we can now show that a newly created node v will satisfy $P(v) \leq p$. This gives us that $M \leq p + 2(k + 1) + 1$. $\square$

Theorem 7 shows that the assumption $s \geq 2$ is necessary for all **S** strategies that can guarantee that $M = O(1)$.

Theorem 7 *For $s = 1$ will M be unbounded.*
Proof: For all N we show that $M \geq N$. For a fixed N let 2^N be the initial number of nodes. We give a simple adversary strategy for player **I** that takes N rounds to create a node with at least N neighbours. In each round we connect the nodes that have not been split pairwise by edges. In each round at most half of the remaining nodes can be split so after N rounds at least one node is not split. More formally, we can show by induction, that after round i at least 2^{N-i} nodes are not split. We have that the degree of the node remaining after N rounds is at least N. $\square$

Chapter 5

Persistent Data Structures

In this chapter we consider the consequences of the results in Chap. 4 on persistence of data structures. At the end of the chapter we mention different data structural problems that are related to the problems involved in making data structures fully persistent.

5.1 Partial Persistence

In [13] Driscoll *et al.* presented a general *node copying* technique to make data structures of bounded degree partially persistent. This is a generalisation of the ideas of [23] where partially persistent trees are applied to solve a planar point location problem. The overhead of making data structures partially persistent in [13, 23] is worst case slowdown $O(1)$ on the query steps and amortised slowdown $O(1)$ on the update steps. The worst case update slowdown can be as bad as $\Theta(n)$ where n is the size of the data structure (for trees it is the size of the longest path in the tree).

An application of the technique of Driscoll *et al.* is shown in Fig. 5.1, where a tree is made partially persistent. The idea is that for each node in the ephemeral data structure (i.e. the data structure we want to make persistent) we have a family of nodes in the partially persistent data structure (indicated by dashed boxes). Each update is added to the last node in the family with the current version stamp. Each node in the persistent data structure contains a fixed number of fields. When a node gets full we create a new node in the family containing only the information that exists in the current version of the ephemeral data structure. When copying a node we recursively have to update the predecessors of the node in the current version of the ephemeral data structure, so that they point to the newly created node. In Fig. 5.1 we see that by updating the rightmost leaf of the tree we imply that three nodes have to be copied. Grey nodes are nodes that contain the information that exists in the current version of the ephemeral data structure, and the numbers are version numbers.

By using the bucketing technique and putting small sets of polylogarithmic size into a constant number of words on the RAM model, Raman improved the update slowdown to worst case $O(1)$ [22].

In [3] we give a much simpler data structure that only needs the power of the pointer machine, and still achieves a worst case slowdown of $O(1)$ on both the query and the update steps. This is a consequence of applying Theorem 3 in Chap. 4 to the node copying technique of Driscoll *et al.* The number of fields we need in each node of the

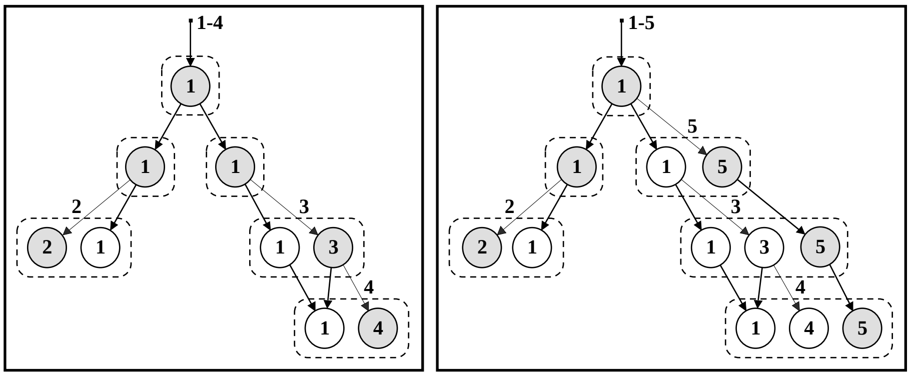

Figure 5.1: A tree made partially persistent by the technique of Driscoll *et al.* [13]. The effect of updating the rightmost leaf in the tree is shown.

persistent data structure is given by the value of M from the zeroing game. If we keep the current version of the ephemeral data structure separately, we just have to copy the node that the zeroing strategy would have zeroed, because the number of pebbles on a node corresponds to the number of fields we have used in the last node in the corresponding persistent family of nodes.

Theorem 8 (Brodal [3]) *A data structure with bounded in- and out-degree can be made partially persistent with worst case slowdown* $O(1)$ *on the pointer machine model.*

Because the data structure we present for the zeroing game is simple and easy to implement, the data structure obtained in this way would also be very efficient in practice.

5.2 Full Persistence

Two general techniques exist to make data structures fully persistent: In [8] Dietz gives a technique to make arrays (and therefore all data structures) fully persistent. The slowdown of the given technique is expected amortised $O(\log \log n)$, where n is the number of operations performed and the size of the array (the complexity is expected amortised because the data structure involves dynamic perfect hashing).

As for partial persistence [13] also contains a general technique for making bounded degree data structures fully persistent with amortised slowdown $O(1)$ per operation. The best known result on the worst case slowdown is by Dietz and Raman [10] that achieves a worst case slowdown of $O(\log \log n)$ for both query and update steps, by a simple application of the bucketing technique to the technique in [13]. Both these results only need the pointer machine model.

As mentioned in Chap. 4 the splitting game we consider is strongly related to [13]. Figure 5.2 and Fig. 5.3 show a “simple” example of a fully persistent data structure by using the technique of [13]. By $[i, j[$ we denote the valid intervals of the nodes and the edges/pointers and for simplicity we have placed values in special data nodes. Figure 5.3 shows how to modify the persistent data structure and how a node is split into two nodes. Notice that the number of new edges introduced by a split is bounded by the degree (in-degree plus out-degree) of the ephemeral data structure. For further details we refer to [13].

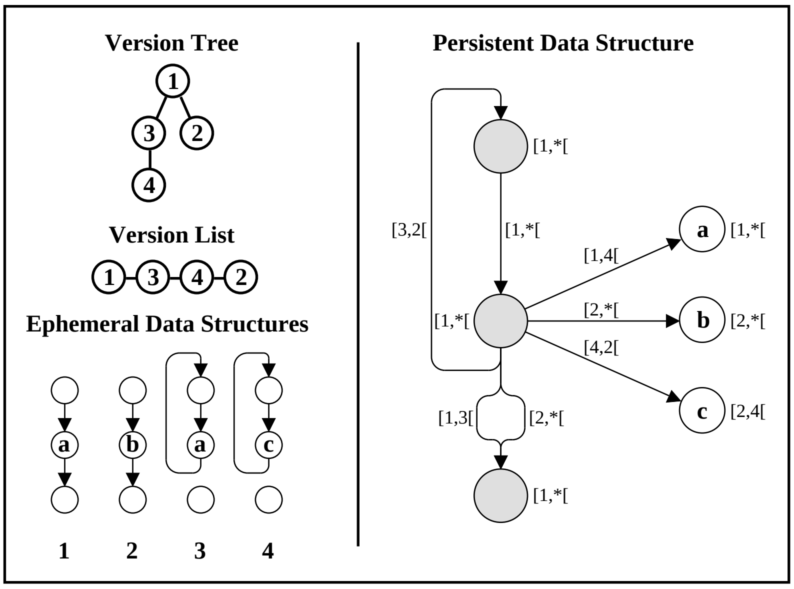

Figure 5.2: A simple example of a fully persistent data structure. With $[3,2[$ we denote the versions from 3 to 2 in the version list not containing 2, so $[3,2[= \{3,4\}$. With $*$ we denote the rightmost end of the list, so $[1,*[= \{1,2,3,4\}$.

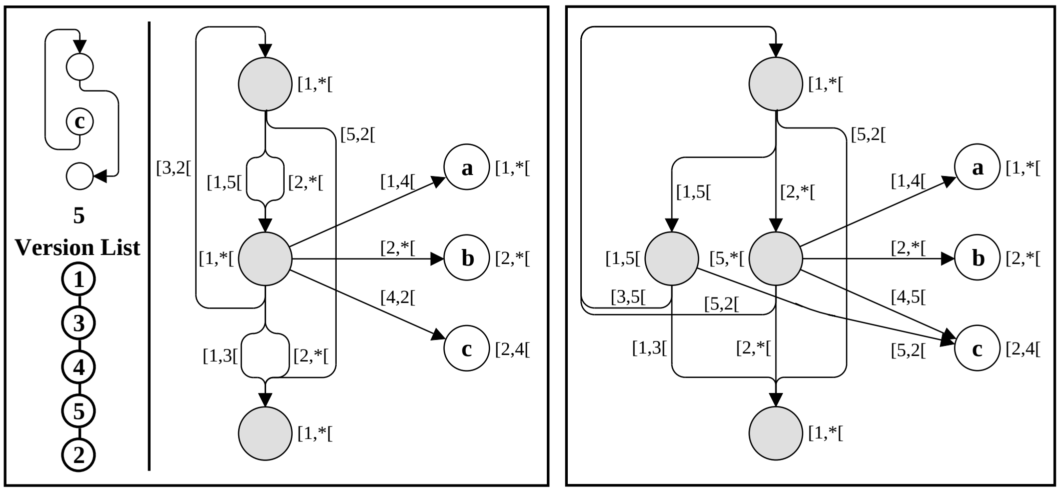

Figure 5.3: To the left is shown the effect of inserting a new pointer into version 5 of the data structure of Fig. 5.2. Version 5 is a modification of version 4. To the right is shown the effect of splitting a node.

The splitting game defined in Chap. 4 assumes that there are no multiple edges. But to apply a splitting game strategy to the full persistence technique we have to allow multiple edges. By modifying the definition of the game to allow multiple edges and by modifying the strategy such that a node can appear several times in a queue Q_v, it is possible to modify the proof of Theorem 6 and show that the degree of all nodes will still be $O(1)$. We omit the details and just state the result.

Theorem 9 *On the pointer machine model we can make a data structure of bounded degree fully persistent with worst case query slowdown $O(1)$ and with amortised update slowdown $O(1)$ and each update requires only worst case $O(1)$ structural changes.*

In Chap. 1 we described the parallel framework of Smid [25] where we have to maintain several client structures of a central structure in a network of processes. When letting the central structure be the above version of the fully persistence technique applied to a data structure and the client structures the same structure without the queues at the nodes, we immediately get the following corollary.

Corollary 1 *It is possible to maintain several copies of a fully persistent data structure of bounded degree in a network of processes with worst case client update time $O(1)$ and amortised central update time $O(1)$.*

We will mention a technical difference between our result and the result of Driscoll *et al.* [13]. Our approach guarantees that the degree of a node is always bounded by a constant. This is not true in the original technique. Temporarily while splitting nodes in an update a node can become of arbitrary degree, so the splitting of nodes should be done carefully to avoid getting amortised update time $\omega(1)$. We avoid this problem.

5.3 Related Data Structural Problems

The first problem we will mention is fractional cascading. Chazelle and Guibas [7] present a data structure to handle the static problem. Mehlhorn and Näher [21] extended the data structure to handle the dynamic case. The problem is to insert *bridges* into the data structure such that gaps are of size $O(1)$. In [21] this takes amortised time $O(1)$ per update. By considering the dual graph where gaps corresponds to nodes and nodes are connected by an edge if only if the the gaps overlap we again have a splitting problem. Figure 5.4 shows an example of this relationship. So an efficiently implementable strategy for the splitting game will also have consequences for the dynamic fractional cascading data structure. If the gaps are allowed to have polylogarithmic size we can just use the bucketing technique and insert a bridge in the largest gap for each insertion we perform [9]. With some care, deletions also can be handled by the bucketing technique.

The second and most obvious problem to mention is the restructuring of an (a, b)–tree when only insertions are allowed. It is not known how to maintain the degrees in the given bounds by $O(1)$ splittings per insertion if we want to find the nodes to split in time $O(1)$. Without the time constraint we can just do lazy splitting as in the lazily recoloured red-black trees of [13].

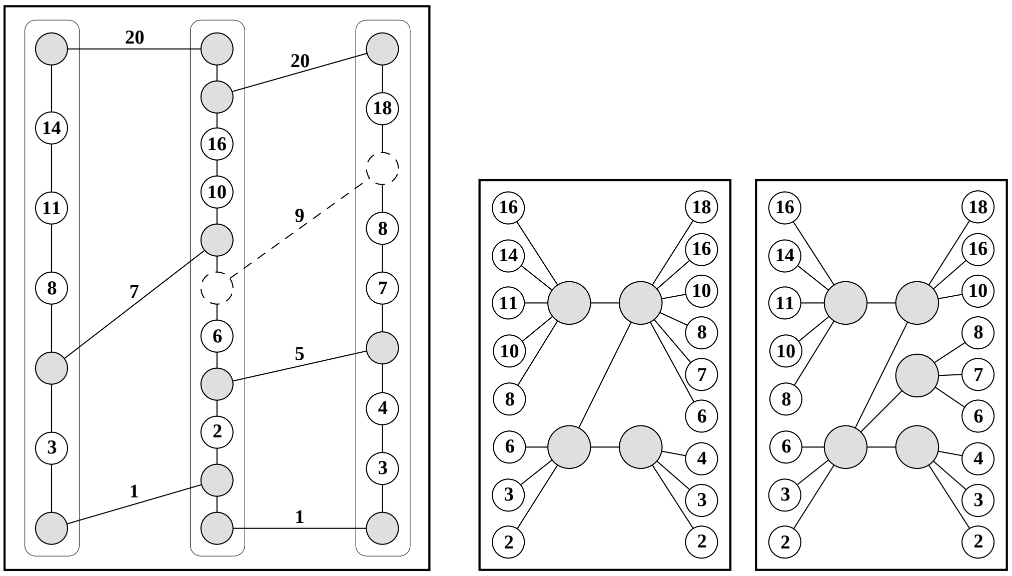

Figure 5.4: Dynamic fractional cascading. To the left is shown a dynamic fractional cascading data structure containing the three lists: $\{3, 8, 11, 14\}$, $\{2, 6, 10, 16\}$ and $\{3, 4, 7, 8, 18\}$. The two figures to the right show the corresponding graphs before and after the dashed bridge is inserted.

Chapter 6

Future Work

In this chapter we briefly summerise the open problems that we have encountered and list a few topics for further work.

- Concerning the zeroing game there are two specific problems that should be explored. The first is to show a general lower bound for the number of pebbles, that expresses the relation between b, d and M. We conjecture that M is *not* linear in d. This is because it is a consequence of Theorem 1 that if $b \ll d$ and $d = n$ and player **D** always picks the node with the most pebbles we get a bound of $M = O(b \log d) = o(d)$.

  The second topic that should be considered is to find better implementable strategies, especially whether it is possible to achieve $M = O(b + d)$. This could perhaps give the insight in how to come up with an efficiently implementable strategy for the splitting game of Chap. 4.

- For the splitting game the main research should be oriented towards finding an efficiently implementable strategy. Even a result for the very restricted case of (a, b)–trees would be interesting.

A few other problems that would be of immediate interest are the following.

- The idea of data structural bootstrapping [5] combined with an explicit regularity constraint [19] looks very promising. It would be interesting to try to apply these ideas to other data structuring problems where the update operations are restricted.

- A very specific problem is whether it is possible to construct mergeable priority queues where FindMin, Insert and Merge can be performed in worst case time $O(1)$ and DeleteMin in worst case time $O(\log n)$.

Acknowledgement

I want to thank the following people for encouraging discussions and for reading drafts of sections contained in this report: Thore Husfeldt, Gudmund Frandsen, Dany Breslauer and Erik M. Schmidt. Especially I want to thank Peter G. Binderup for the encouraging discussions that lead to the lower bounds for the zeroing game.

Bibliography

[1] M. Ajtai, M. Fredman, and J. Komlós. Hash functions for priority queues. *Information and Computation*, 63:217–225, 1984.

[2] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *ACTA Informatica*, 1:173–189, 1972.

[3] Gerth Stølting Brodal. Partially persistent data structures of bounded degree with constant update time. Technical Report BRICS-RS-94-35, BRICS, Department of Computer Science, University of Aarhus, 1994.

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

[5] Adam Louis Buchsbaum. *Data-Structural Bootstrapping and Catenable Deques*. PhD thesis, Princeton University, 1993. Dept. of Computer Science, Princeton U., tech report TR-423-93.

[6] Svante Carlsson 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.

[7] Bernard Chazelle and Leonidas J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1:133–162, 1986.

[8] 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.

[9] Paul F. Dietz and Rajeev Raman. A constant update time finger search tree. In *Advances in Computing and Information - ICCI '90*, volume 468 of *Lecture Notes in Computer Science*, pages 100–109. Springer Verlag, Berlin, 1990.

[10] Paul F. Dietz and Rajeev Raman. Persistence, amortization and randomization. In *Proc. 2nd ACM-SIAM SODA*, pages 78–88, 1991.

[11] 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.

[12] James R. Driscoll, Harold N. Gabow, Ruth Shrairman, and Robert E. Tarjan. Relaxed heaps: An alternative to fibonacci heaps with applications to parallel computation. *Communications of the ACM*, 31(11):1343–1354, 1988.

[13] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38:86–124, 1989.

[14] Rudolf Fleischer. A simple balanced search tree with $O(1)$ worst-case update time. Technical Report MPI-I-92-101, Max-Planck-Institut Für Informatik, 1992.

[15] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. In *Proc. 25rd Ann. Symp. on Foundations of Computer Science (FOCS)*, pages 338–346, 1984.

[16] Hania Gajewska and Robert E. Tarjan. Deques with heap order. *Information Processing Letters*, 22:197–200, 1986.

[17] Leo J. Guibas, Edward M. McCreight, Michael F. Plass, and Janet R. Roberts. A new representation for linear lists. In *Proc. 9thAnn. ACM Symp. on Theory of Computing (STOC)*, pages 49–60, 1977.

[18] Scott Huddleston and Kurt Mehlhorn. A new data structure for representing sorted lists. *ACTA Informatica*, 17:157–184, 1982.

[19] S. Rao Kosaraju. An optimal RAM implementation of catenable min double-ended queues. In *Proc. 5th ACM-SIAM SODA*, pages 195–203, 1994.

[20] Christos Levcopoulos and Mark H. Overmars. A balanced search tree with $O(1)$ worst-case update time. *ACTA Informatica*, 26:269–277, 1988.

[21] Kurt Mehlhorn and Stefan Näher. Dynamic fractional cascading. *Algorithmica*, 5:215–241, 1990.

[22] 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.

[23] Neil Sarnak and Robert Endre Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29:669–679, 1986.

[24] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary trees. In *Proc. 15th Ann. ACM Symp. on Theory of Computing (STOC)*, pages 235–245, 1983.

[25] Michiel Smid. *Dynamic Data Structures on Multiple Storage Media.* PhD thesis, University of Amsterdam, 1989.

[26] Robert Endre Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *Journal of Computer and System Sciences*, 18:110–127, 1979.

[27] Robert Endre Tarjan. Updating a balanced search tree in $O(1)$ rotations. *Information Processing Letters*, 16:253–257, 1983.

[28] Robert Endre Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6:306–318, 1985.

[29] Jean Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–315, 1978.