DATA-STRUCTURAL BOOTSTRAPPING AND CATENABLE DEQUES

Adam Louis Buchsbaum

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE

June 1993

to

my parents

and

their parents

Abstract

The *list* is a fundamental data structure. It stores a linearly ordered collection of elements and allows access only to the front and rear elements of the list. *Catenation* can be applied to lists, unifying the rear of one list with the front of another. Absent other requirements, the basic list operations, including catenation, have straightforward implementations. If the list has certain secondary properties, however, the operations, particularly catenation, become more difficult.

Non-destructive lists, for example, support side-effect-free list operations and are fundamental in high-level programming languages such as LISP, ML, and Scheme. Actual implementations of non-destructive lists usually apply simple copying methods directly to the lists or to trees representing the lists. These copying methods have high space and time overhead, however. *Persistent data structures* allow operations on old versions, and therefore techniques for designing persistent data structures might be useful in devising non-destructive lists. Current persistence techniques, however, can be applied to only one version of a data structure at a time and therefore do not accommodate list catenation. *Heap-ordered lists*, which provide access to minimum elements, also have efficient implementations that do not allow catenation. These two data structures are related: in both cases, the addition of a secondary property complicates list catenation. In this thesis, we develop a technique called *data-structural bootstrapping* to address these problems.

Bootstrapping uses two basic ideas. First, we *abstract* a data structure by representing it in terms of its own secondary property. Inserting the abstracted data structure as an element into the same type of non-catenable data structure on a higher level effects catenation. Second, we homogeneously *decompose* a data structure into a group of smaller data structures. This produces an efficient, recursive implementation.

Data-structural bootstrapping yields efficient implementations of both non-destructive lists ($O(\log^* n)$ worst-case time and space per operation on a list of n elements) and catenable heap-ordered lists ($O(1)$ amortized time per operation). The latter have applications in pagination, network sensitivity analysis, and VLSI river routing. Catenable heap-ordered lists derive their efficiency from a special case of path compression that we prove takes only linear time.

Acknowledgments

I am indebted to my advisor, Bob Tarjan. His remarkably keen insights always seemed to lead the way out of technical cul de sacs. In addition to his technical stewardship, I also thank him for a relationship that transcends just research.

I thank my readers, Bob Sedgewick and Danny Sleator, for their invaluable comments. Bernard Chazelle and Ken Steiglitz were kind enough to sit on my defense committee.

I thank Jeff Westbrook for bringing the heap-ordered deque problem to my attention. That problem was the beginning of this work.

Rajamani Sundar was instrumental in proving the path compression result, and I spent many hours with him refining the details of that proof.

I thank Jim Driscoll and Danny Sleator for some early ideas and discussions that motivated the confluently persistent deque work.

I also benefited from technical discussions with Andrew Appel, Greg Frederickson, Milena Mihail, and Pete Winkler.

I have truly enjoyed my time at Princeton. This is due in part to the proximity to good Indian food but in even larger part to some very close friends. Karin Petersen is a true, solid friend; I know that even when geography separates us, she'll always be there. Jim Plank taught me more ways to lose at cards than can possibly be healthy to one person's mental state; he also shared a taste for the ironic and satirical. Norbert Schlenker is a kindred spirit in the devotion to poignant persiflage and bargain hunting. I shall always remember the many evenings filled with great food, cards, and Arnold films that I spent with them together with Heather Booth and Norah McCloy.

In addition to my work and my play, I will keep other fond memories of Princeton. Ken Steiglitz showed me a different way of looking at the world. Ken, David Dobkin, Bruce Maggs, Ian Munro, Jim Plank, Anne Rogers, Norbert Schlenker, and Bob Sedgewick all shared a deep appreciation of our nation's (sorry Norbert and Ian) pastime. I'll also always remember the unique implementation of that pastime by the infamous (not to mention libelous) Cache Hitters.

I cannot begin to thank my family for their support throughout my life. My parents, in particular, spared no opportunity to make my life easier, not only through the tangible, but also by setting high examples of honesty, integrity, and morality. I will always strive to keep my father's memory alive by emulating his approach to life.

Finally, I thank my wife and very best friend, Mary, for her love and caring, kindness and devotion, counsel and humor. Her companionship makes hard times easier and good times better.

This work was supported by a Fannie and John Hertz Foundation fellowship, National Science Foundation Grant No. CCR-8920505, and the Center for Discrete Mathematics and Theoretical Computer Science (DIMACS) under NSF-STC88-09648.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Algorithm design often benefits from the following approach: identify a core problem shared by several algorithms, and design one simple, efficient data structure to address it. There are numerous examples in which a general solution to a common data-structure problem leads to faster algorithms for a variety of applications. For example, Sleator and Tarjan [ST83] designed *dynamic trees* to produce fast algorithms that solve various nearest common ancestor, network flow, and constrained minimum spanning tree problems. The *Fibonacci heaps* of Fredman and Tarjan [FT87] yield fast algorithms that solve certain shortest-path, assignment, and minimum spanning tree problems. Recently, Eppstein et al. [EGIN92] developed *sparsification* as an algorithmic tool to improve the running time of many dynamic graph algorithms. In each case, the authors isolate a problem common to several different algorithms and produce one data structure or technique that addresses that problem.

Similarly, data-structure design sometimes reveals different data structures that share a common implementation problem. Rather than implementing each data structure in an ad hoc fashion, it is often better to address the common problem by applying a general data-structure design technique. We anticipate several advantages from developing such techniques. First, isolating the problems leads to better understandings of the data structures involved. Second, a few general techniques can be applied to solve many data-structure problems. Third, by concentrating on and providing solutions to common problems, the resulting data structures are less likely to have complicated, esoteric solutions and thus more likely to be useful in practice. General techniques for designing data structures, however, are rare compared to the number of data structures described in the literature. This thesis examines one data-structure design technique, namely *data-structural bootstrapping*, and applies it to implement efficiently two different yet closely related data structures.

We investigate *heap-ordered* and *persistent deques*. These are linear lists of elements that provide access only to the front and back ends of the lists. They also have some interesting secondary properties. A heap-ordered deque provides identification of the smallest element in the list; a persistent deque provides access to previous versions of the list. In each case, efficient implementations exist that provide all of the standard deque operations except *catenation*—the linking of two deques into one. We develop data-structural bootstrapping as a technique that uses these previous implementations as basic building blocks in the recursive design of catenable heap-ordered and persistent deques. We provide efficient implementations of both data structures: $O(1)$ amortized time per heap-ordered deque operation and $O(1)$ amortized time and space for each persistent deque operation except deletion, which takes $O(\log^* m)$ amortized time and space, where m is the number of deque operations performed so far. We also make the persistent deque resource bounds worst-case.

1.1 Terminology

Here we introduce some basic terminology used throughout this thesis. We define a *universe* to be a set of *items* or *elements*. Each element in the universe has a *key*; the keys are not necessarily distinct. For example, the set of natural numbers might be a universe; each number is its own key.

We refine the concept of a data structure into two variants: an *abstract data structure* and a *concrete data structure*. An *abstract data structure* is collection of operations that are allowed on a subset of a universe. For example, the *dictionary* is a common abstract data structure that allows the operations *insert*, *delete*, and *membership-query* to be performed on a subset of a universe. We say that the dictionary *permits* or *allows* the operations insert, delete, and membership-query. A *concrete data structure* is a physical implementation of some abstract data structure. For example, we might implement a dictionary as a balanced binary tree; the binary tree is a concrete data structure that implements the operations—insert, delete, and membership-query—that the dictionary permits. Sometimes we use the term *data structure* by itself; the context resolves whether we mean an abstract or concrete data structure. Distinguishing an abstract from a concrete data structure derives from the use of the term "abstraction" by Dijkstra [Dij72] and Hoare [Hoa72].

Finally, an *augmented data structure* is an abstract data structure D' that is derived from some other abstract data structure D by the addition of a new operation. For example, a *rank dictionary* is an augmented dictionary, allowing not only insert, delete, and membership-query, but also the *rank* operation (which returns the number of items in a subset with key less than that of a given element).

1.1.1 Lists and Catenation

The *list* is a fundamental abstract data structure in computer science. Lists provide sequential access to data via the following operations:

makelist(e) Create and return a new list of one element, e.[1]

[1] We could just as easily have makelist return an empty list. Creating a list of one, given element, however, simplifies the pseudocode in Chapter 3.

push(e, X) Insert e as the new first element of list X; the previous ith element becomes the $(i+1)$st. Return the new list.

pop(X) Return an ordered pair of the first element of list X and the list containing the second through the last elements of list X.

inject(e, X) Insert e as the new last element of list X, and return the new list.

eject(X) Return an ordered pair of the last element of list X and the list containing the first through the second-to-last elements of list X.

There are many different types of lists. For example, a *stack* permits only makelist, push, and pop (or makelist, inject, and eject). A *queue* permits only makelist, push, and eject (or makelist, inject, and pop). A *double-ended queue* or *deque* permits all of the above operations. If both insertion operations but only one of the deletion operations are allowed, the list is an *output-restricted deque*. Where pop (eject) is allowed, we can also consider the separate *first*(X) (*last*(X)) operation, which returns the first (last) element of list X without creating a new list. Our results are for deques and therefore encompass all of the above list types. Without loss of generality, we consider the *front* or *head* of the list (where the first element resides) to be on the left and the *back* or *tail* on the right. Knuth [Knu73a] describes these lists. Such data structures can easily be implemented by doubly-linked (in some cases singly-linked) lists yielding $O(1)$ worst-case times for each of the allowed operations [Tar83].

If X and Y are lists of the same type (i.e., queue, stack, etc.) then we define the following operation:

catenate(X, Y) Return the list containing the elements of X followed by those of Y.

Catenate can also be implemented in $O(1)$ worst-case time. For now, we assume that catenate only works on two distinct lists. Further, as described above, all of the operations are *destructive*; i.e., they destroy their arguments and allow access only to the most recent version of any list. For example, once two lists are catenated, only the resulting list remains and the two original lists are inaccessible.

It is sometimes desirable to consider certain augmented lists, i.e., lists with additional, secondary properties. This thesis investigates such lists, in particular those that are *heap-ordered* (that allow queries as to their minimum elements) and those that are *persistent* (that allow access to previous versions). We find that the addition of a secondary property can complicate list catenation while not unduly affecting the other list operations. We develop a technique called *data-structural bootstrapping* to address this problem and apply it to the construction of heap-ordered and persistent lists with catenation.

1.2 Secondary Properties

In this section, we discuss the two secondary properties *heap order* and *persistence* and explain why they are important to deques. We save discussion of how each new property actually complicates deque catenation to Chapters 2 and 3.

1.2.1 Heap Order

Knuth [Knu73b] discusses heap order in detail. If a data structure D acts on subsets of a totally ordered universe, then the data structure *D with heap order* permits all the operations that D does as well as the following additional operation:

findmin(X) Find and return an element of minimum key in X (or $\emptyset$ if X is empty).

Findmin does not modify the data structure itself, and the element returned is subject to whatever access restrictions apply to the data structure. A *heap-ordered deque* (or

	Queue	Deque	Minqueue	Mindeque
Push	X	X	X	X
Pop		X		X
Inject		X		X
Eject	X	X	X	X
Catenate	X	X	X	X
Findmin			X	X

Table 1: Operations allowed by various lists.

mindeque) thus allows queries as to its minimum element, but unless that element is the first or last of the deque, it cannot subsequently be deleted. A related data structure is the *priority queue with attrition* [Sun89].

Queues with heap order (or *minqueues*) are useful in pagination [DF84, HL87, LH85, McC77] and VLSI river routing [CS84]. Booth and Westbrook [BW90] use catenable heap-ordered queues in the sensitivity analysis of minimum spanning trees, shortest-path trees, and minimum-cost network flow on planar graphs. Larmore and Hirschberg [LH85] and Cole and Siegal [CS84] independently show how to implement heap-ordered queues in $O(1)$ amortized time per operation [Tar85]. Gajewska and Tarjan [GT86] modify their techniques to produce heap-ordered deques with $O(1)$ time per operation; they give both amortized and worst-case solutions.

Applications of heap-ordered deques include computing all-pairs shortest-path information [Fre91] and external farthest neighbors for simple polygons [AAA+91]. Whereas a simple extension of the previous heap-ordered queue techniques yields efficiently catenable heap-ordered queues (again $O(1)$ amortized time per operation), Gajewska and Tarjan leave as an open problem how to construct efficiently catenable heap-ordered deques.

Table 1 summarizes the operations allowed by queues, deques, heap-ordered queues, and heap-ordered deques. We show how to catenate heap-ordered deques efficiently in Chapter 2. As shown by Table 1, our data structure unifies all previous data structures that provide heap-ordered queue and heap-ordered deque operations. Furthermore, our data structure is extremely simple and easy to implement and therefore should be useful in practice.

1.2.2 Persistence

A data structure normally provides two types of operations to a subset of a universe: *queries*, which return information without modifying the subset, and *updates*, which may change the subset. In the terminology of Driscoll et al. [DSST89], an abstract data structure is *ephemeral* if an update destroys the version of the concrete data structure being changed; i.e., there is always one and only one valid version. As described in Section 1.1.1, destructive list operations affect ephemeral lists. A data structure that allows queries but not updates to previous versions is *partially persistent*. In this scheme, a time line of versions evolves: each new version descends from the version most recently created. Allowing previous versions to be updated yields a *fully persistent* data structure and produces a tree-like history of the data structure: each new version descends from an arbitrary older one.

There has been much previous work on making various data structures partially or fully persistent [CG93, Cha85, Col86, DM85, HM81, Mye82, Mye83, Mye84, Ove81a, RTD83, ST86]. There have also been some results providing general methods for making entire classes of data structures persistent [Die89, DR91, DSST89, Ove81b, Sar86]. In particular, Driscoll et al. [DSST89] provide a method by which many pointer-based data structures may be made partially or fully persistent with only a constant factor overhead in time (over the ephemeral version) and using only a constant amount of space per persistent update; both bounds are amortized. Their result is subject to the important restriction that any node in the ephemeral data structure may have only a fixed number of incoming and outgoing pointers. The methods of Driscoll et al. [DSST89] can make deques partially or fully persistent but cannot provide persistent deque catenation. Dietz [Die89] makes arrays fully persistent so that access and store operations on an array of size n require $O(\log\log n)$ time (amortized expected for store) and linear total space.

None of these results, however, allows updates involving more than one version; combining two versions, such as is done in deque catenation, is not possible. This type of update produces a version DAG (directed acyclic graph). A data structure with such a history is called *confluently persistent* by Driscoll, Sleator, and Tarjan [DST91].

More formally, consider every operation to have an extra vector parameter t, representing time; $t[i]$ denotes the ith component of t. E.g., pop is now specified $pop(X, t)$, and it applies to list X at time $t[1]$. The ordered pair it returns is timestamped with the current time. Without loss of generality, assume time to be ordinal by operations. If the most recently created data structure is *version* v (i.e., is timestamped v), then the next version will be $v + 1$.

Let v be the most recently created version of a data structure. In this framework, an *ephemeral data structure* allows all operations to be performed only on version v; i.e., $t[1] = v$. A *partially persistent data structure* allows queries to any version (specified in $t[1]$) but updates only to version v. A *fully persistent data structure* allows any operation to be performed on any version (again, specified in $t[1]$). A *confluently persistent data structure* allows operations to be performed on multiple versions. E.g., $catenate(X, Y, t)$ catenates version $t[1]$ of list X with version $t[2]$ of list Y; with this usage, it is likely that $Y = X$, and different lists will be distinguished by different version numbers.

Driscoll et al. [DSST89] leave open the problem of designing general techniques for building confluently persistent data structures. While we do not offer such a general solution, to date we are aware of only Driscoll, Sleator, and Tarjan [DST91] as a previous result in this direction. (They give a solution to the confluently persistent output-restricted deque problem.) In Chapter 3, we construct confluently persistent deques. Our work provides a solution to a more general problem (deques rather than output-restricted deques in a confluently persistent framework) than that of Driscoll, Sleator, and Tarjan [DST91] and also gives asymptotically improved resource bounds. We discuss the practicality of our solution in Chapter 4.

Confluently persistent deques have many uses in high-level programming languages such as LISP, ML, and Scheme [Set89], in which side-effect-free list operations are fundamental. They also can be used in the implementation of continuation passing in functional programming languages [App92]. Furthermore, some purely functional programming languages do not allow any reassignment of memory cells; such languages effectively implement all data structures, regardless of use, persistently. Therefore, any application of catenable deques becomes an application of confluently persistent deques when implemented by such languages.

1.3 Data-Structural Bootstrapping

We have described two instances of one common problem: given an abstract data structure with catenation, augmenting that data structure with a secondary property can complicate catenation. Here we give an overview of *data-structural bootstrapping*, a technique developed in this thesis to address this problem We present data-structural bootstrapping as a recursive design tool for building data structures. When applied to the problems mentioned above, bootstrapping builds catenable heap-ordered deques using only non-catenable heap-ordered deques as basic building blocks and confluently persistent deques using only fully persistent deques as basic building blocks.

We use two types of data-structural bootstrapping: *structural abstraction* and *structural decomposition*.

1.3.1 Structural Abstraction

To design confluently persistent output-restricted deques, Driscoll, Sleator, and Tarjan [DST91] obviate the general problem of catenating two versions of a fully persistent data structure by applying the following key idea: they abstract fully persistent deques by representing them by their persistent version numbers. To catenate deques, they then merely insert (push or inject) the version number of one deque into the other. This reduces catenation to simple insertion. This technique generalizes the approach of Kosaraju [Kos79], in which catenable deques are decomposed into contiguous pieces that are stored in a stack.

We extend the above technique as follows: given a data structure augmented with a secondary property, we define *structural abstraction* as the representation of the data structure by its secondary property. For example, we abstract heap-ordered deques by representing them by their minimum elements, and we abstract fully persistent deques by representing them by their persistent version numbers. In this way, we also extend the universe on which the data structure acts: items in the extended universe are instances of data structures, the keys of which are their secondary properties.

We use this technique in Chapters 2 and 3 to design catenable heap-ordered deques and confluently persistent deques. In the heap-ordered deque case, structural abstraction effects catenation; in the confluently persistent deque case, structural abstraction effects persistence.

1.3.2 Structural Decomposition

The efficiency of our catenable heap-ordered deques relies on the analysis of a related path compression problem, which we present in Chapter 2. Structural abstraction, however, suffices to build the data structure itself. Confluently persistent deques, though, derive their efficiency from the second type of bootstrapping, namely *structural decomposition*: the recursive decomposition of a data structure into smaller, homogeneous pieces. In particular, we represent a deque of n elements by a collection of deques of $O(\log n)$ elements each, and we apply this decomposition recursively.

Structural decomposition is similar to an idea of Dietz [Die82], who recursively decomposes indexed 2-3 trees so that their leaves store smaller indexed 2-3 trees, improving their performance from $O(\log n)$ to $O(\log^* n)$ amortized time. Structural decomposition differs from ideas used to solve decomposable search problems [BS80] and from other ad hoc instances of data structure decomposition (e.g., [Die91, DR91, DS87, Tsa84, Wil83, Wil84]) in that these previous results use only one level of decomposition, whereas bootstrapping is recursive.

1.4 Summary

The rest of this thesis applies data-structural bootstrapping to build efficient catenable heap-ordered deques and confluently persistent deques.

Chapter 2 discusses catenable heap-ordered deques. We use structural abstraction to represent non-catenable heap-ordered deques by their minimum elements; in this way, catenation becomes no more difficult than insertion. In order to prove that the resulting data structure achieves constant amortized time per operation, we consider *order-preserving path compression*. This is a generalization of special cases of path compression originally introduced by Hart and Sharir [HS86] and subsequently analyzed by Loebl and Nešetřil [LN88a, LN88b, LN89] and Lucas [Luc90]. We prove a linear time bound on *deque-ordered, spine-only path compression*, a case of order-preserving path compression used in our data structure. Our path compression result not only unifies the special cases of path compression considered by Loebl and Nešetřil and by Lucas but also extends their results to a more general case. Furthermore, we provide what we believe is the first practical application of this type of result, namely the application of this special case of path compression to a real data structure. Our data structure is efficient and easy to implement in practice, and it unifies previous data structures (such as heap-ordered queues and non-catenable heap-ordered deques) that are used in a wide variety of applications.

Chapter 3 discusses confluently persistent deques. We use structural abstraction to represent fully persistent deques by their persistent version numbers; by doing so, we make our data structure confluently persistent. Furthermore, we use structural decomposition to represent a deque of n elements by a collection of deques of $O(\log n)$ elements each; this provides efficient resource bounds. Our data structure performs pop and eject in $O(\log^* m)$ amortized time and space, where m is the number of deque operations performed so far. All of the other operations take $O(1)$ amortized time and space. We can also make all of these resource bounds worst-case, and we can implement our data structure purely functionally, with no side effects. We use m instead of n, the number of elements in the deque, since confluently persistent deques allow self-catenation (the catenation of a list with itself); therefore, it might be that $n = 2^m$. Noting that the Driscoll, Sleator, and Tarjan [DST91] solution to the output-restricted deque problem provides non-constant resource bounds ($O(\log \log m)$ amortized time and space) for catenate rather than pop and eject, we can modify our data structure to provide a solution to the output-restricted deque problem with $O(\log^{(i)} m)$ amortized time and space bounds for catenate,[2] where i is any desired constant, and $O(1)$ amortized time and space bounds for the other operations. Ours

[2] $\log^{(i)}$ is the log function iterated i times.

is the most asymptotically efficient data structure reported so far that implements confluently persistent deques; producing a practical implementation is discussed in Chapter 4.

In both cases, we use data-structural bootstrapping to obviate the problems that catenation presents to the implementation of an augmented data structure. We thus demonstrate the usefulness of data-structural bootstrapping as a tool to design high-level data structures.

Chapter 2

Catenable Heap-Ordered Deques

2.1 Introduction

This chapter provides an efficient implementation of catenable heap-ordered deques; we achieve constant amortized time per operation. We use structural abstraction to represent heap-ordered deques in terms of their minimum elements; in this way, we reduce catenation to simple insertion. In order to prove that the resulting data structure achieves constant amortized time per operation, we consider *order-preserving path compression*. This is a generalization of special cases of path compression originally introduced by Hart and Sharir [HS86] and subsequently analyzed by Loebl and Nešetřil [LN88a, LN88b, LN89] and Lucas [Luc90]. We prove a linear time bound on *deque-ordered, spine-only path compression*, a case of order-preserving path compression used in our data structure.

Our result shows how structural abstraction is a useful tool in the design of efficient data structures. Additionally, we not only unify the special cases of path compression considered by Loebl and Nešetřil and by Lucas, but we extend their results to a more general case. Furthermore, we provide what we believe is the first practical application of this type of result, namely applying this special case of path compression to a data structure that is used in practice.

Section 2.2 of this chapter describes a previous implementation of heap-ordered deques and explains why catenation cannot be applied to this data structure. Section 2.3 applies structural abstraction to the heap-ordered deques of Section 2.2, yielding catenable heap-ordered deques. Section 2.4 defines and analyzes deque-ordered, spine-only path compression, proving the linearity of this special case of path compression. We also prove a non-linear lower bound for general *order-preserving path compression.* We summarize in Section 2.5. We remark that the methods used in Chapter 3, as stated there, provide a nontrivial worst-case time bound per heap-ordered deque operation. A preliminary version of the material in this chapter appears as Buchsbaum, Sundar, and Tarjan [BST92].

2.2 Reviewing Heap-Ordered Queues and Deques

For completeness, we begin by reviewing how to implement heap-ordered queues and heap-ordered deques. As in Chapter 1, identify the head of a list (where the first element sits) as the left end of the list and the tail as the right. A *minimum element* of a list is an element of minimum key. To implement a heap-ordered queue d, maintain a secondary list d' of *rightward minima.* This list has as its first element the minimum element x of d; the second element is the minimum element of d to the right of x, and so on; see Figure 1(a).

To pop an element e from d, if e is minimum, the new minimum element of d becomes the successor of e in d' (and that element becomes the new head of d'—i.e., d' is also popped). Otherwise, no change to d' takes place. To inject an element e of key k into d, consider the rightmost element x of d'. If $key(x) > k$, eject x from d' and continue the search with the new rightmost element of d'. Once the rightmost

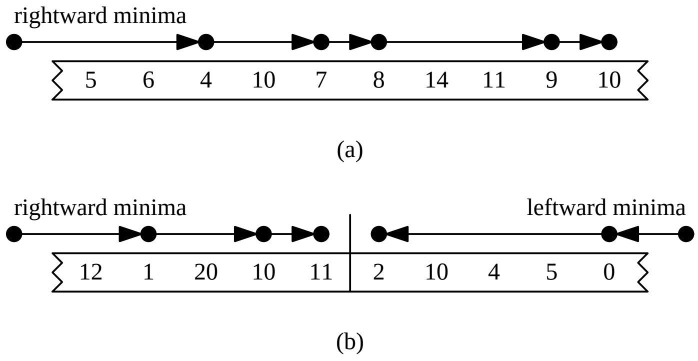


Figure 1: (a) A heap-ordered queue with minimum key 4. (b) A heap-ordered deque with minimum key 0; the center line is the break point between the two stacks.

element of d' is of lesser value than k, inject k into d'; note that k might be the only element of d' after this procedure. The findmin operation on d is now implemented by returning the head of d'. This is one possible implementation of the previous technique [CS84, LH85], yielding $O(1)$ amortized time bounds for all of the queue operations. We note that the technique also extends to implement heap-ordered stacks, which can be used to create heap-ordered deques, as well as heap-ordered output-restricted deques. Further, it is simple to add catenation to heap-ordered queues, stacks, and output-restricted deques using the same idea: the minima lists of the two lists are resolved as in the case of insertion.

Gajewska and Tarjan [GT86] implement heap-ordered deques by representing them as two heap-ordered stacks, as in Figure 1(b). When one stack is emptied, the heap-ordered deque is rebuilt so that the two stacks differ in size by no more than one. The findmin operation returns the minimum of the minimum elements in the left and right stacks. By gradually rebuilding the stacks concurrently with the deque operations, they achieve $O(1)$ worst-case time per operation. Using two stacks, however, does not allow the easy implementation of catenation that is possible with heap-ordered queues. In particular, the leftward minima list of the right stack of the left heap-ordered deque and the rightward minima list of the left stack of the right heap-ordered deque point in opposing directions. It is not clear that these minima lists can be efficiently updated to reflect the catenated heap-ordered deque.

2.3 Data-Structural Bootstrapping

We employ structural abstraction to bootstrap the heap-ordered deques and allow their efficient catenation. We wish to implement a catenable heap-ordered deque d; call the elements of d *basic elements*. We freely interchange the notion of elements and their keys (e.g., $x > y$ for two elements x and y implies that the key of x is greater than the key of y).

Our data structure D is itself a heap-ordered deque, with elements from a different universe than that of d. Each element of D consists of a *basic element* from the universe of d combined with a pointer. For an element x of D, let e be the basic element of x and p the pointer of x. If $p = \emptyset$, then e is a basic element of d; in this case we call x a *d-element* of D. Otherwise, e is the minimum basic element of a heap-ordered deque to which p points. This other heap-ordered deque has the same type of elements as does D, and the structure is recursive; the minimum element of D thus contains the minimum basic element among all those in the heap-ordered deques reachable via pointers from D. The d-elements of D occur in a natural *left-to-right* order that can be obtained by recursively visiting each element from head to tail in D, listing d-elements as they are encountered. The resulting ordered list of d-elements corresponds to the basic elements in order in the catenable heap-ordered deque d we are implementing.

In this fashion, there is a one-to-one correspondence between the elements of D and the nodes of a heap-ordered tree; see Figure 2(a)–(b). In the tree, the leaves represent d-elements of D and hence basic elements of d, and we maintain a *heap invariant* on the tree: the non-leaf (or internal) nodes contain the lowest values

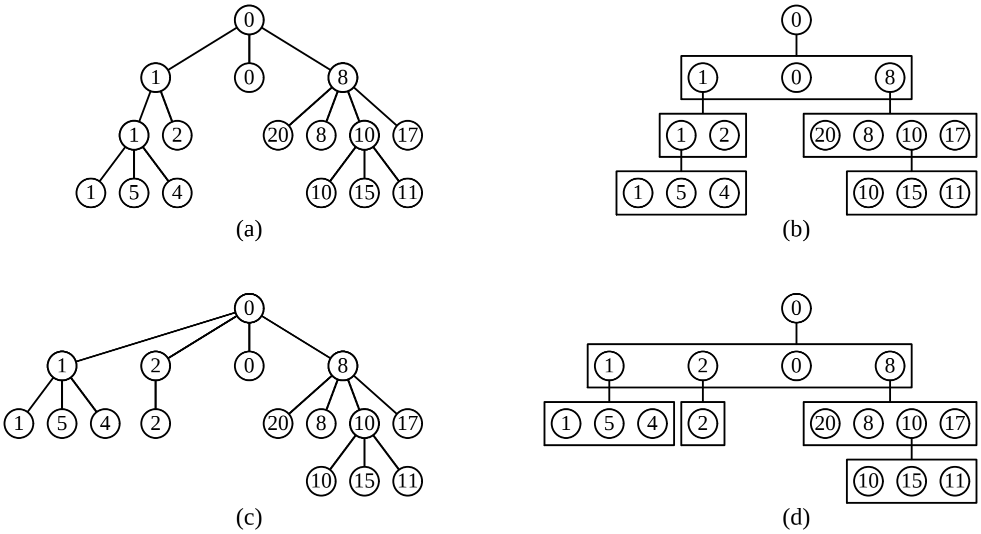


Figure 2: (a) A heap-ordered tree. (b) The corresponding catenable heap-ordered deque: boxes surround heap-ordered deques, the minimum values of which are elements in the parent heap-ordered deque. (c) The tree of (a) after a left pull. (d) The corresponding catenable heap-ordered deque.

among their children. The internal nodes are precisely the non-d-elements of D and the heap-ordered deques reachable from D. The leaves in the tree also have a natural left-to-right order, corresponding to that on the d-elements of D.

The concept of representing a heap-ordered linear list of items by exploiting the induced left-to-right order of the leaves in a normal heap-ordered tree arises in the pagodas of Françon, Viennot, and Vuillemin [FVV78]. Similar data structures are the Cartesian tree [Vui80] and the treap [AS89]. These maintain one tree under both symmetric and heap orders (on two distinct keys per node). If the symmetrically ordered key represents the position of the node in a linear list, the data structure supports heap-ordered list access operations.

2.3.1 The *Pull* Operation

We now define a *left pull* operation on the heap-ordered tree T in a way similar to the definition of the *pull* operation of Driscoll, Sleator, and Tarjan [DST91]. If the leftmost child x of the root of T is a leaf, a *left pull on* T does nothing. Otherwise, a left pull on T removes the leftmost child x' from x and makes x' a child of the root of T to the left of x; x' thus becomes the new leftmost child of the root of T, and if x is now a leaf, it is deleted. See Figure 2(c) for an example. The heap invariant of the tree is maintained by updating x, if necessary, to contain the lowest key among those of its children. We define a *right pull on* T symmetrically. Both pull operations preserve the left-to-right order of the leaves of T.

A left pull can be described in terms of the data structure D as follows.

1. If the first element x of D is a d-element (has a $\emptyset$ pointer),
2. Then do nothing.
3. Else
4. Pop D, returning x (and the rest of D).
   Denote by $d(x)$ the heap-ordered deque to which the pointer of x points.
5. Pop $d(x)$, returning x' (and the rest of $d(x)$).
6. If $d(x)$ is still non-empty, then push x back onto D.
   (The pointer of x still points to $d(x)$, but popping $d(x)$ might change the basic element of x (the minimum basic element of $d(x)$).)
7. Push x' onto D.

A right pull on D is defined symmetrically. Since we use only heap-ordered deque operations to implement pulls on D, the minimum values of all the heap-ordered deques, in particular that of D, are maintained correctly. The popping and repushing of x are executed precisely to maintain the correct minimum values; i.e., if x' is the minimum element of x, then popping x' from x and pushing it onto D necessitates changing the minima lists of D, which is effected by the popping and pushing of x before the pushing of x'. Figure 2(d) shows the result of a left pull on D. Again, the pull operations preserve the left-to-right order of the d-elements of D.

It is now straightforward to implement the catenable heap-ordered deque operations on d as follows. The makelist operation returns a heap-ordered deque D of one, given element and a $\emptyset$ pointer.

push(x, d)	Create element $\tilde{x}$ composed of basic element x and pointer $\emptyset$. Push $\tilde{x}$ onto D, returning the new D.
pop(d)	Repeatedly left pull D until its head element is a d-element. Then pop D; this returns an element $\tilde{x}$ and the rest of D. Return the basic element x of $\tilde{x}$ and the remainder of D.
inject(x, d)	Symmetric to push.
eject(d)	Symmetric to pop.
findmin(d)	Return the basic element in the minimum element of D.
catenate(d_1, d_2)	For $i \in \{1, 2\}$, let $\tilde{D}_i$ be an element containing the basic element in the minimum element of D_i and a pointer to D_i. Push $\tilde{D}_1$ onto D_2. (Alternatively, we could inject $\tilde{D}_2$ into D_1.) Return the new heap-ordered deque.

That D correctly simulates d is easily proven by induction on the number of operations.

2.3.2 Time Analysis

We now state the following.

Theorem 2.3.1 *An intermixed set of n insertions, m deletions, and q catenations on $q+1$ catenable heap-ordered deques takes total time $O(n+m+q)$; findmin always takes $O(1)$ worst-case time.*

PROOF: We first note that push (inject) on a catenable heap-ordered deque requires a constant amount of time to prepare the element (from the given basic element and a $\emptyset$ pointer) in addition to one push (inject) on a non-catenable heap-ordered deque. If we use the Gajewska-Tarjan non-catenable heap-ordered deques [GT86], each non-catenable heap-ordered deque operation takes $O(1)$ time. Either the amortized or worst-case solution they offer suffices.[1] Continuing, each catenation also entails one push or inject on a non-catenable heap-ordered deque in addition to some constant amount of preprocessing time. Each deletion entails some number of pulls, each of which requires a constant number of non-catenable heap-ordered deque operations, plus one final non-catenable heap-ordered deque pop or eject.

If we consider the corresponding heap-ordered tree, we see that the set of insertions, catenations, and deletions maps to an instance of disjoint set union [TvL84]. In particular, the insertions and catenations correspond to unions and the deletions to finds on the elements that are eventually deleted. That is, a sequence of pulls effects a path compression. Further, note that the path compressions are all *spine compressions*. This notion will be defined in detail in Section 2.4. Briefly, each compression involves a path of only leftmost children (or only rightmost children); also, each path goes from a leaf node to the root of a tree, and the compressions happen in a special order induced by access pattern of the deque. Theorem 2.4.10 will prove the linearity of such a sequence of path compressions. It is unnecessary to maintain the heap-ordered trees in a balanced fashion (e.g., doing unions by size or rank).

Findmin is performed by one non-catenable heap-ordered deque findmin, which takes $O(1)$ worst-case time. □

[1] The implementation given in Section 3.4.5 requires their worst-case solution.

Note that if $q = O(1)$—that is, if there is only a constant number of heap-ordered deques to be catenated—our data structure implements all the operations in $O(1)$ time each. Thus our structure unifies the general problem with this special case mentioned by Gajewska and Tarjan [GT86], which they solved by using the techniques of Kosaraju [Kos79].

Corollary 2.3.2 *An intermixed sequence of insertions, catenations, deletions, and findmins can be performed on a fixed number q of catenable heap-ordered deques in $O(1)$ worst-case time per operation.*

PROOF: Each pop or eject requires at most q pulls. □

2.4 Path Compression

In this section, we prove that the special case of path compression referred to in Theorem 2.3.1 takes only linear time. We begin by introducing some definitions that are necessary for our analysis.

To simplify the analysis, we assume that all path compressions affect one initial tree. We will use Definition 2.4.1 and Lemma 2.4.2 to link this framework to the situation in which path compressions affect a collection of trees (representing heap-ordered deques) that get linked together. We therefore start with a tree T that has n nodes (and $n-1$ edges). An edge (u, v) connects node u with its *parent* v in the tree; we say that $p(u) = v$ *in* T. Each node except for the root of the tree has precisely one edge joining it to its parent. The parent pointers define a *path* from x to the root of T in the natural way. A node x is a *descendant* of a node y if y lies on the path from x to the root of the tree; symmetrically, y is an *ancestor* of x. Note that x is both an ancestor and a descendant of itself. A *proper ancestor (descendant)* of x is an ancestor (descendant) y of x such that $y \neq x$.

A *path compression* $C = (x_0, \ldots, x_l)$ on a tree T takes the tree path $C = (x_0, \ldots, x_l)$ (such that $l > 0$ and $p(x_i) = x_{i+1}$ in T for $0 \leq i < l$) and updates the parent of each proper descendant of x_l on the path to be x_l, the *root of the compression*:

- $p(x_i) \leftarrow x_l,\ 0 \leq i < l.$

We say that the compression C is a *path compression from* x_0 and that it *roots at* x_l; further, x_i for $0 \leq i < l$ is a *non-root node* of the path compression. The *cost* of C is $|C| = l$.

If x_0 is a *leaf node*, i.e., a node with no children, then a *leaf deletion* $C = (x_0)$ on T removes x from T. In this case the cost of C is $|C| = 1$. A *sequence of path compressions on* $T = T_0$ is a sequence $(C_1, \ldots, C_m)$ such that C_i is a path compression or a leaf deletion on T_{i-1} and the result of C_i applied to T_{i-1} is T_i. The *cost* of a sequence of path compressions on T is $\sum_{i=1}^{m} |C_i|$.

We now define the *Rising Roots Condition* [Luc90], which links the notion of path compressions above with the well-known union and find operations used in the disjoint set union problem (see, e.g., Tarjan and van Leeuwen [TvL84]). For any node x in T, let k_x be the smallest i such that x appears as a non-root node in C_i; $k_x = \infty$ if there is no such i.

Definition 2.4.1 (Rising Roots Condition) *A sequence $(C_1, \ldots, C_m)$ of path compressions satisfies the* Rising Roots Condition *if and only if for every node x and every $i > k_x$, x appears as a non-root node in C_i if C_i is a compression from a descendant of x in T_{i-1}.*

The Rising Roots Condition tells us precisely when a sequence of path compressions on some initial tree corresponds to an intermixed sequence of unions, finds, and leaf deletions.

Lemma 2.4.2 *A sequence of path compressions (on an initial tree) satisfying the Rising Roots Condition corresponds to some sequence of intermixed union, find, and leaf deletion operations. Conversely, a sequence of intermixed union, find, and leaf deletion operations corresponds to some sequence of path compressions satisfying the Rising Roots Condition.*

PROOF: See Lucas [Luc90, Lemma 1]. □

The correspondence above is straightforward. The roots of the compressions in the path compression sequence are the roots of the finds in the union-find instance, and vice-versa. This correspondence can be used to simplify the analysis of disjoint set union instances by assuming that all the unions are done before the first find. Any result for a class of path compressions satisfying the Rising Roots Condition maps to a result for a class of disjoint set union problems.

We now introduce the notion of order to restrict the path compression sequences we shall consider. Given a tree T, embed T in the plane, yielding a left-to-right order on the children of each node. The *nearest common ancestor* of two nodes x and y ($nca(x, y)$) is the deepest node z in T such that z is an ancestor of both x and y. For x a proper descendant of z, let $c_x(z)$ be the child of z that is an ancestor of x; note that $c_x(z)$ might equal x. We define a partial order on T as follows: for any pair of nodes x and y, if $z = nca(x, y)$ and $z \notin \{x, y\}$, then $x < y$ if $c_x(z)$ is to the left of $c_y(z)$.

Definition 2.4.3 (Order Preservation) *A path compression on a tree T that yields a tree T' is* order-preserving *if $x < y$ in $T \Longrightarrow x < y$ in T'. A sequence of path compressions is* order-preserving *if each path compression in the sequence is order-preserving.*

Note that as there is in general more than one way to effect a path compression (in terms of the left-to-right order of the newly acquired children of the root of the

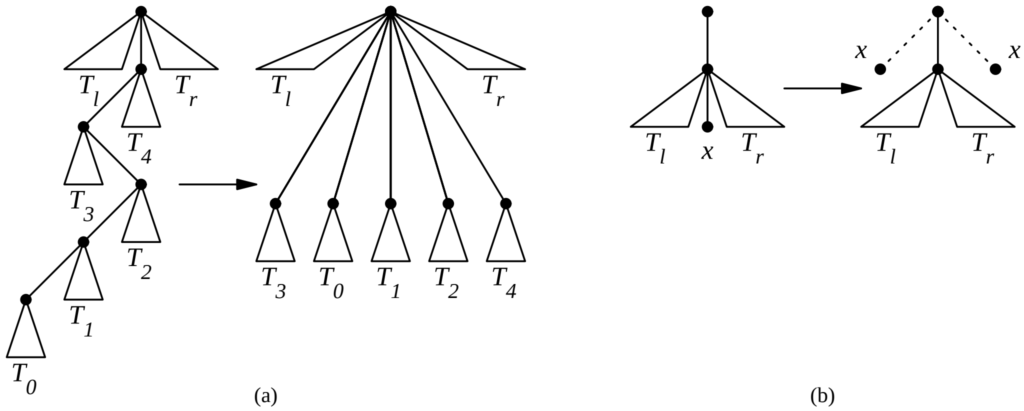


Figure 3: (a) An order-preserving path compression; triangles denote subtrees. (b) A path compression that cannot be order-preserving; x must become either left of T_l or right of T_r.

compression), Definition 2.4.3 depends upon the actual implementation of the path compressions involved. We assume for simplicity that a compression is effected in an order-preserving way if it can be. We now describe exactly when a path compression (sequence) is order-preserving.

Lemma 2.4.4 *A path compression* $(x_0, \ldots, x_l)$ *is order-preserving if and only if* x_i *is the leftmost or rightmost child of* x_{i+1} *for* $0 \leq i < l - 1$.

PROOF: First note that the last non-root node in the compression need not be an *extremal* (leftmost or rightmost) child of its parent; only the nodes whose parents change need be.

Now, assume that x_i is an extremal child of x_{i+1} for $0 \leq i < l - 1$. Then we can effect the path compression top-down in an order-preserving way as follows. If x_{l-2} is the leftmost (rightmost) child of x_{l-1}, and y is the rightmost (leftmost) sibling to the left (right) of x_{l-1}, make x_{l-2} a new child of x_l in between y and x_{l-1}; y might be nil. Continue in this fashion down to x_0; see Figure 3(a).

On the other hand, let $j < l - 1$ be such that x_j is not an extremal child of x_{j+1}. Let L be any left sibling of x_j and R be any right sibling of x_j. Before the compression we have $L < x_j < R$. If x_j becomes a left sibling of x_{j+1} during the compression, however, then after the compression we have $x_j < L$; similarly, if x_j becomes a right sibling of x_{j+1}, then after the compression we have $R < x_j$. See Figure 3(b). $\square$

Note that an only child, i.e., a node with no siblings, qualifies as both a leftmost and a rightmost child of its parent. Now consider a postorder (preorder) assigned to the nodes of a tree. Our heap-ordered deques use a restricted case of order-preserving path compression.

Definition 2.4.5 (Spine Compression) *A path compression* $(x_0, \ldots, x_l)$ *is a* left-spine compression (right-spine compression) *if* x_i *is the leftmost (rightmost) child of* x_{i+1} *for* $0 \le i < l-1$ *and the compression preserves the postorder (preorder) of the tree. A sequence of path compressions is a sequence of* spine-only *path compressions if each path compression in the sequence is a left-spine compression or a right-spine compression.*

Clearly left- and right-spine compressions are order-preserving. Lucas [Luc90] proves that a sequence of left-spine path compressions *done in postorder* and satisfying the Rising Roots Condition requires linear time; i.e., given an initial tree, each node, in postorder, is the object of one left-spine path compression and is then deleted. Loebl and Nešetřil [LN88a, LN88b, LN89] prove linearity in a more general case: a sequence of left-spine path compressions satisfying the Rising Roots Condition; they refer to this as a *local postorder*. These results derive from an open problem of Hart and Sharir [HS86]: what is the complexity of a sequence of path compressions done in postorder, without the Rising Roots Condition? Both the Lucas and the Loebl-Nešetřil problems are special cases of this problem.

We now introduce a more general order of path compressions than that of either Lucas or Loebl and Nešetřil. Let $T^{(v)}$ be the subtree rooted at v just before the first path compression that changes $p(v)$, and let $T_x^{(v)}$ be the subtree of $T^{(v)}$ rooted at x for some child x of v in $T^{(v)}$.

Definition 2.4.6 (Deque Order) *An order-preserving sequence of path compressions is a* deque-ordered sequence *if and only if for any v with two children x and y in $T^{(v)}$ such that $x < y$ the following is true: if the first path compression C' that orders v and y results in $y < v$, then all path compressions from nodes in $T_x^{(v)}$ precede C'; otherwise, if the first path compression C'' that orders v and x results in $v < x$, then all path compressions from nodes in $T_y^{(v)}$ precede C''.*

It is important to note that Definition 2.4.6 refers to nodes in a tree, e.g., $T_x^{(v)}$, with respect to $T^{(v)}$. That is, an actual compression may take place from a node that at the time of the compression is no longer a descendant of x (or v).

Intuitively, a deque-ordered sequence of path compressions with the Rising Roots Condition corresponds to an instance of disjoint set union, in which the path compressions affect leaf-to-root paths and trees can be linked between path compressions, restricted so that any path compression starts from either the leftmost or rightmost leaf, in symmetric order, of any tree. Our heap-ordered deque data structure uses this restricted type of path compression sequence.

Now we can prove our main result: a deque-ordered sequence of spine-only path compressions that satisfies the Rising Roots Condition takes at most linear time.

2.4.1 Linearity of Deque-Ordered, Spine-Only Compression

We unfortunately find it necessary to divide the following definitions into two separate cases to handle differently what happens to the left and to the right of various nodes. Let $p_i(v)$ be the parent of v in T_i. For some path compression sequence $(C_1, \ldots, C_m)$ on T_0 and a node v, we call C_i a *left compression with respect to v* if

1. C_i starts from a node x that is a descendant of v in T_0;

2. C_i is a left-spine compression.

We define a *right compression with respect to* v symmetrically by replacing (2) above with (2) C_i is a right-spine compression. Note that x might not be a descendant of v at the time the path compression occurs. Let $high_L(v)$ be the shallowest node w that is a proper ancestor of v in T_0 and that is the root of a left compression with respect to v; if there is no such path compression, then $high_L(v) = \emptyset$. Symmetrically, let $high_R(v)$ be the shallowest node w that is a proper ancestor of v in T_0 and that is the root of a right compression with respect to v; $high_R(v) = \emptyset$ if no such compression exists. Again, it is critical to realize that the compression might not be from a descendant of v at the time.

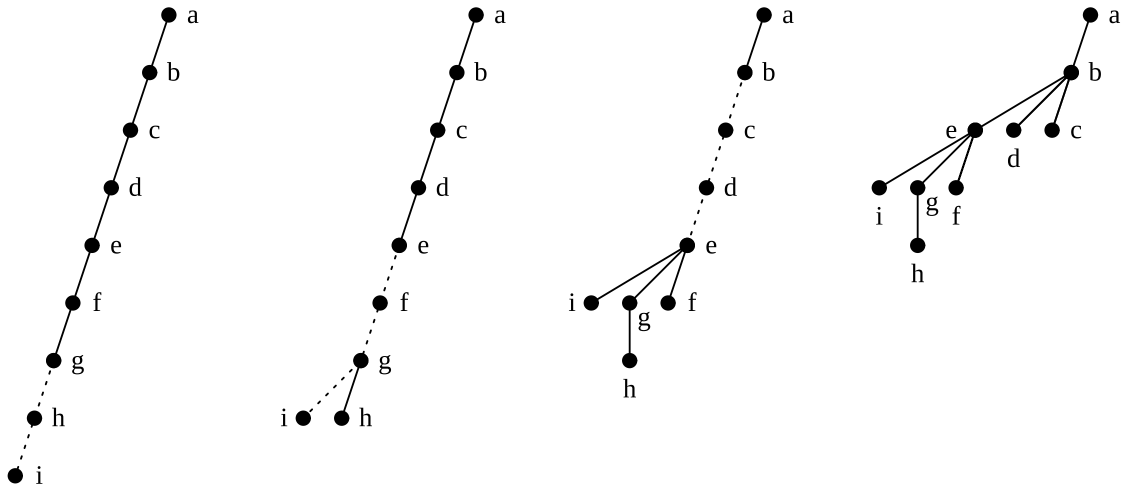
Letting $level_L(\emptyset) = level_R(\emptyset) = -1$, we now define:

$$\begin{aligned} level_L(v) &= level_L(high_L(v)) + 1 \\ last_L(v) &= \min\{i \mid v \text{ is not a proper descendant of } high_L(v) \text{ in } T_i\} \end{aligned}$$

$$\begin{aligned} level_R(v) &= level_R(high_R(v)) + 1 \\ last_R(v) &= \min\{i \mid v \text{ is not a proper descendant of } high_R(v) \text{ in } T_i\} \end{aligned}$$

See Figure 4. To make $last_L$ and $last_R$ well-defined, we say that v is never a descendant of $\emptyset$. The definitions of $level_L$ and $level_R$ are static; they are defined purely with respect to the initial tree and the sequence of path compressions. A *left-level-defining descendant* of v is a descendant (if one exists) x of v in T_0 such that a left compression with respect to v from x roots at $high_L(v)$; we call the actual compression a *left-level-defining compression* of v. Symmetrically define a *right-level-defining descendant* and *right-level-defining compression* of v.

We now prove a set of technical lemmas that localize the compressions affecting a vertex to one or two left levels and one or two right levels adjacent to those of the vertex. Again we let $p(v)$ ($p_i(v)$) be the parent of v (in T_i) (for v not the root), and $c_x(v)$ be the child of v that is an ancestor of x if x is a proper descendant of v.



(a) (b) (c) (d)

	a	b	c	d	e	f	g	h	i
$high_L$	$\emptyset$	$\emptyset$	b	b	b	e	e	e	e
$level_L$	0	0	1	1	1	2	2	2	2
$last_L$	0	0	∞	∞	∞	∞	∞	∞	∞

Figure 4: An initial tree and three successive path compressions; the compression paths are shown by dotted lines. (a) The initial tree T_0. (b) Tree T_1, after compressing path (i, h, g). (c) Tree T_2, after compressing path (i, g, f, e). (d) Tree T_3, after compressing path (e, d, c, b). Compressions 1 and 2 are left compressions with respect to each node; compression 3 is a left compression with respect to nodes a, b, c, d, and e. The table shows the $high_L$, $level_L$, and $last_L$ values for each node.

Lemma 2.4.7 *In a path compression sequence* $(C_1, \ldots, C_m)$ *on* T_0*, for any node* v*:*

1. *for any* $0 \leq i < last_L(v)$*,* $level_L(v) \geq level_L(p_i(v))$*;*

2. *for any* $0 \leq i < last_R(v)$*,* $level_R(v) \geq level_R(p_i(v))$*.*

PROOF: We prove the lemma for (1); the proof for (2) is symmetric. In T_0, consider any root-to-leaf path partitioned by nodes of $level_L$ zero. We use induction on the length of each partitioned subpath. The base case is the head r of the path; $level_L(r) = 0$. Note that $level_L(r) = 0 \Leftrightarrow high_L(r) = \emptyset \Leftrightarrow last_L(r) = 0$; the claim for the base case is thus vacuous.

For a node v such that $level_L(v) > 0$ and r is the nearest ancestor of v such that $level_L(r) = 0$, let x be a left-level-defining descendant of v. Any left compression with respect to v from x is also a left compression with respect to $p_0(v)$ from x. Thus, if $high_L(v) = r$ then $high_L(y) = r$ for every y such that y is an ancestor of v and a proper descendant of r in T_0. (No left compression with respect to r may root at a proper ancestor of r, since $level_L(r) = 0$.) By the definition of $level_L$ and the induction hypothesis, $level_L(v) \geq level_L(p_0(v))$. On the other hand, if $high_L(v) \neq r$ then $high_L(v)$ is a descendant of $high_L(p_0(v))$ in T_0. Both $high_L(v)$ and $high_L(p_0(v))$ are descendants of r in T_0. Again by the definition of $level_L$ and the induction hypothesis, $level_L(v) \geq level_L(p_0(v))$.

For a node v in T_i such that $0 < i < last_L(v)$, by definition v is a proper descendant of $high_L(v)$ in T_i. By the above proof of monotonicity, it is still the case that $level_L(v) \geq level_L(p_i(v))$.

Leaf deletions trivially preserve (1). □

Lemma 2.4.8 *In a path compression sequence* $(C_1, \ldots, C_m)$ *on* T_0*, for any node* v *not the root of* T_0*:*

1. $level_L(p(v))$ *changes at most once during compressions* C_1 *through* $C_{last_L(v)-1}$*;*

2. *$level_R(p(v))$ changes at most once during compressions C_1 through $C_{last_R(v)-1}$.*

PROOF: Again we prove the lemma for (1); the proof for (2) is symmetric. Let $level_L(v) = l$. Via the path compression sequence, v becomes a child of higher and higher nodes on the initial path from v to the root of T_0. By Lemma 2.4.7, $level_L(p(v))$ never increases during compressions C_1 through $C_{last_L(v)-1}$. By definition the highest node that becomes $p(v)$ during these compressions is of $level_L$ not less than $l-1$. Assuming $last_L(v) > 0$, Lemma 2.4.7 states that $level_L(p_0(v)) \leq level_L(v)$, and therefore $level_L(p(v))$ can change only once, from l to $l-1$, during compressions C_1 through $C_{last_L(v)-1}$. □

Note that Lemmas 2.4.7 and 2.4.8 are completely general; they apply to any sequence of path compressions. We have not yet used the Rising Roots Condition, spine-only compression, or deque order. The following lemma is critical to the counting argument used in the proof of Theorem 2.4.10, and it is here that we rely upon these conditions. We use the notation $T^{(v)}$ and $T_x^{(v)}$ as in Definition 2.4.6.

Lemma 2.4.9 *In a deque-ordered sequence $(C_1, \ldots, C_m)$ of spine-only path compressions satisfying the Rising Roots Condition, any node v can be in at most one compression of each of the following two types:*

1. (a) *The compression is a left-spine compression from a proper descendant x of v;*

   (b) *$level_L(c_x(v)) = level_L(v) = level_L(p(v))$ before the compression; and*

   (c) *$p(v)$ changes during the compression, but $level_L(p(v))$ does not.*

2. (a) *The compression is a right-spine compression from a proper descendant x of v;*

   (b) *$level_R(c_x(v)) = level_R(v) = level_R(p(v))$ before the compression; and* *(c) $p(v)$ changes during the compression, but $level_R(p(v))$ does not.*

PROOF: Again we prove the Lemma for (1); the proof for (2) is symmetric. Let $l = level_L(v)$. Consider the children of v in $T^{(v)}$; by the Rising Roots Condition v can acquire no more children. Further, each compression as specified in the lemma removes a child from v. Now, if v has more than one child of $level_L$ l just before the first such compression, consider any two such children x and y such that $x \lessdot y$. Let x' be a left-level-defining descendant of x. Note that x' is a node in $T^{(v)}$; by order preservation, $c_{x'}(v) \lessdot y$ in $T^{(v)}$.

Consider the first compression C_i that orders y and v. If C_i makes $y \lessdot v$ (i.e., it is a left-spine compression), then by Definition 2.4.6 all the compressions from nodes in $T^{(v)}_{c_{x'}(v)}$ must precede C_i. In particular, the left-level-defining compression from x', rooting at a $level_L$ $l-1$ node, must occur before C_i. By the Rising Roots Condition, C_i must therefore root at a node of $level_L$ no greater than $l-1$; thus, if $level_L(p(v)) = level_L(v) = l$ just before C_i, then $level_L(p(v))$ changes (to $l-1$) during C_i. From that point until compression $C_{last_L(v)}$, $level_L(v) \neq level_L(p(v))$, completing the proof. Note that $i < last_L(v)$. $\square$

We can now prove our main result.

Theorem 2.4.10 *Any deque-ordered sequence $(C_1, \ldots, C_m)$ of m spine-only path compressions (and intermixed leaf deletions) satisfying the Rising Roots Condition on an initial tree of n nodes incurs cost at most $3m + 4n$.*

PROOF: Consider any left-spine path compression $C = (x_0, \ldots, x_k)$. The definition of $level_L$ together with Lemma 2.4.7 shows that there are some b and l such that $level_L(x_i) = l$ for $0 \leq i \leq b$, and $level_L(x_i) = l - 1$ for $b < i \leq k$. If all the nodes on the path compression are of the same $level_L$, we say that $b = -1$. See Figure 5(a).

Whereas before the compression, $p(x_i) = x_{i+1}$ for $0 \leq i < k$, after the compression $p(x_i) = x_k$ for $0 \leq i < k$. Each node x_i for $0 \leq i < b$ has the $level_L$ of its parent

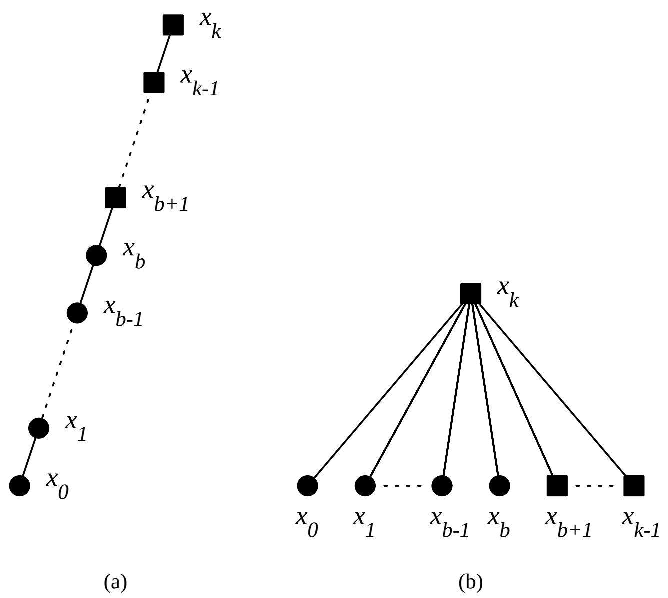


Figure 5: (a) A left-spine (incident subtrees not shown); circles are nodes of $level_L\ l$, and squares are nodes of $level_L\ l-1$. (b) After compressing the path of (a).

change, which by Lemma 2.4.8 can happen only once per node; we therefore charge each of these pointer changes to the nodes themselves. Further, for all nodes x_i for $b+1 < i < k-1$, x_i is involved in the one type-(1) compression allowed by Lemma 2.4.9. Again, the related charges are incurred against the nodes themselves. Symmetric charges can be made against the nodes for a right-spine compression. The remaining charges, those for nodes x_b (if it exists), x_{b+1}, and x_{k-1}, are incurred against the compression itself. See Figure 5(b). Since leaf deletions take unit time, the bound follows. $\square$

It is not hard to see that the data structure of Section 2.3 uses a deque-ordered sequence of spine-only path compressions satisfying the Rising Roots Condition. Thus, the proof of Theorem 2.4.10 completes the proof of Theorem 2.3.1. We again mention that the initial tree T_0 need not be balanced.

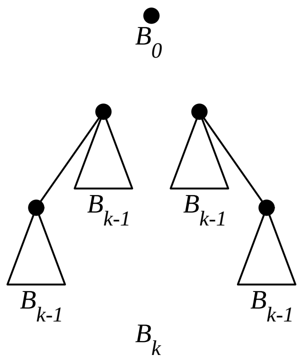


Figure 6: A B_0-tree and the two ways to make a B_k-tree.

2.4.2 A Lower Bound for Order-Preserving Path Compression

In this section, we give an example demonstrating $2n$ order-preserving path compressions and leaf deletions satisfying the Rising Roots Condition on an initial tree of size $2n$ with total cost $2n + n \log n$. Thus, at least one of the additional restrictions of spine-only path compression and deque order is crucial to the linearity of the path compression sequence used by the heap-ordered deque data structure.

To begin, we define the following trees inductively: a B_0-tree is a singleton node; a B_k-tree, $k > 0$, is formed by linking two B_{k-1}-trees as in Figure 6. These trees resemble the binomial trees of Vuillemin [Vui78]; since we allow linking in two directions, however, there are 2^k possible B_k-trees. It is easy to prove by induction that if T is a B_k-tree, then (1) $|T| = 2^k$, and (2) the depth of T is k.

Let L_k be a *collection* containing one B_i-tree for $0 \le i \le k$ laid out in the plane in some left-to-right order. We define a *deque order* on L_k inductively: L_0 is always *laid out in deque order*; L_k is *laid out in deque order* if and only if (1) the B_k-tree is the leftmost (or rightmost) tree, and (2) the rest of the trees form a collection L_{k-1} laid out in deque order to the right (or left) of the B_k-tree. See Figure 7. We use the following lemma in our lower bound construction.

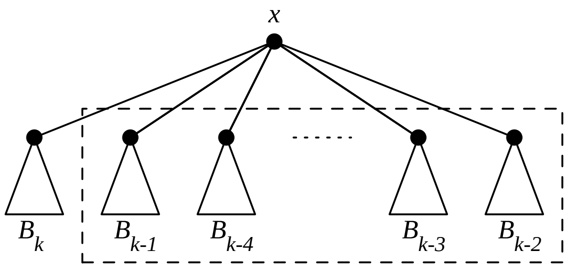


Figure 7: A collection L_k laid out in deque order. The roots of the B_i-trees are children of a common node x. The trees inside the box together with x form a B_k-tree, as described in Lemma 2.4.11.

Lemma 2.4.11 *A collection L_k laid out in deque order with the root of each B_i-tree in the collection connected to a common parent x forms a B_{k+1}-tree.*

PROOF: By induction. The base case, for $k = 0$, is trivial. For $k > 0$, consider a collection L_k laid out in deque order. Assume without loss of generality that the B_k-tree, T_l, is leftmost. By the induction hypothesis, the collection L_{k-1} laid out in deque order to the right of T_l together with the node x forms a B_k-tree T_r. The root of T_l, however, is the leftmost child of x, the root of T_r, and so by definition this structure forms a B_{k+1}-tree. See Figure 7. □

We *augment* a B_k-tree by making its root the child of a new root node. Call the augmented tree an A_k-tree; see Figure 8(a). Let r be the root of an A_k-tree. By definition, the leftmost (and rightmost) child of r is the root r_{k-1} of a B_{k-1}-tree. Either the leftmost or rightmost child of r_{k-1} is the root r_{k-2} of a B_{k-2}-tree. Continuing in this way, we derive a path $P = (s_0, r_0, r_1, \ldots, r_{k-1}, r)$ in the A_k-tree, where r_i is the root of a B_i-tree (and s_0 is the root of a B_0-tree). Note that there are two B_0-trees in the unwound construction. Further, each node in P other than r is an extremal child of its parent. See Figure 8(b). Thus, P can be compressed in an order-preserving way.

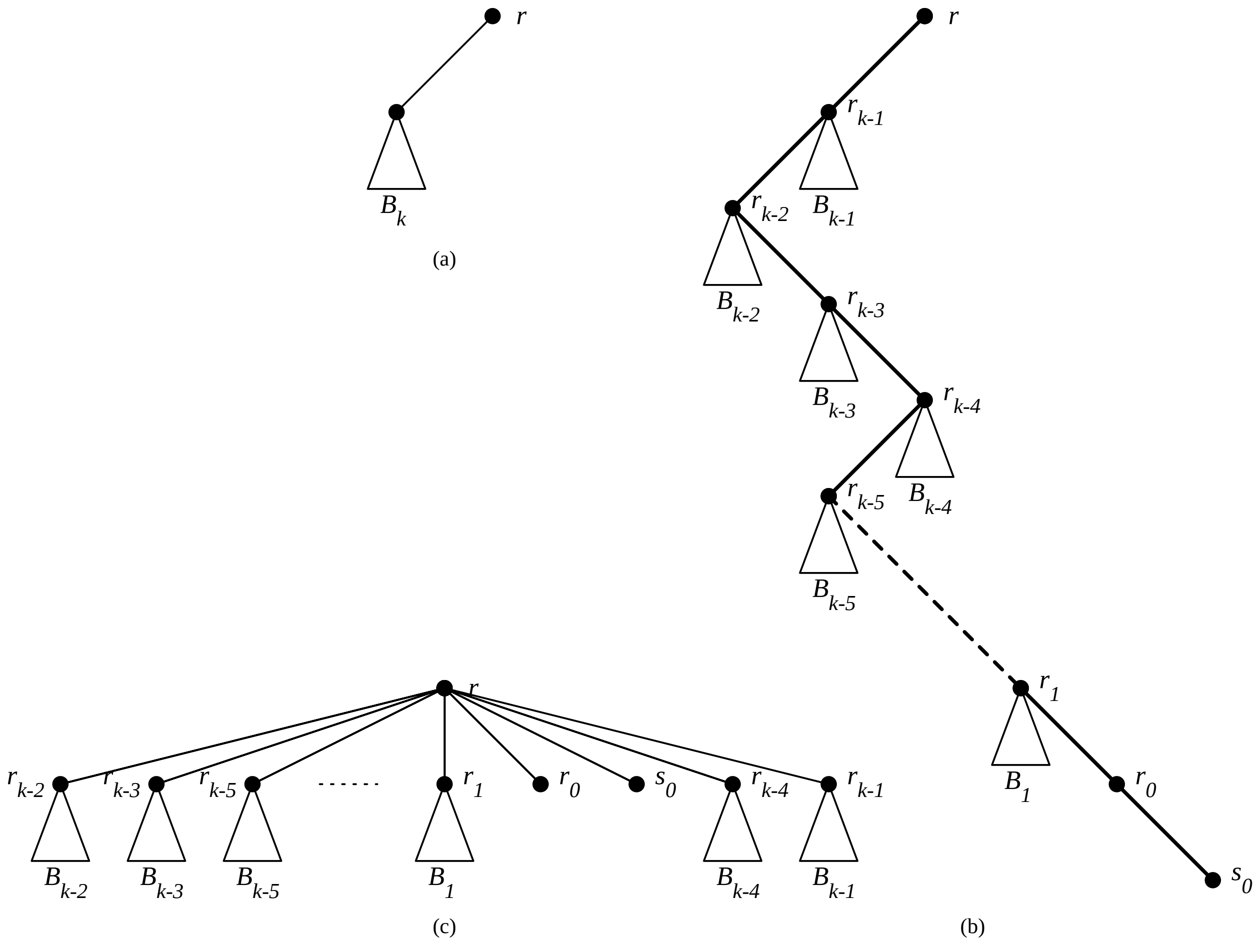


Figure 8: (a) An A_k-tree. (b) An A_k-tree unwound to show path P, in bold. (c) After compressing path P.

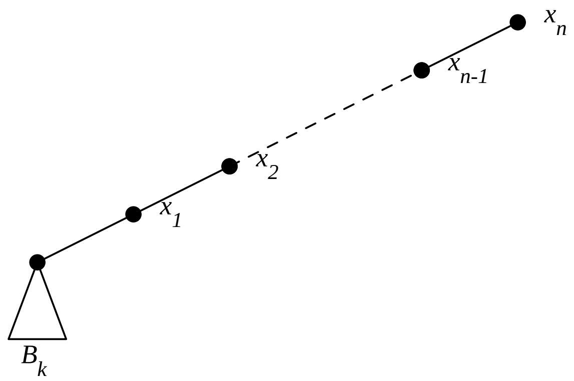


Figure 9: A B_k-tree augmented n times.

Lemma 2.4.12 *Compressing P in an order-preserving way yields a collection L_{k-1} of trees laid out in deque order and an extra B_0-tree, all connected to the common node r.*

PROOF: After the compression, the leftmost (or rightmost) child of r is r_{k-1} (descending from which is the B_{k-1}-tree). By induction, the B_i-trees descending from nodes r_i, $0 \le i < k-1$, form a collection L_{k-2} laid out in deque order to the right (or left) of the B_{k-1}-tree; their roots, along with s_0, all become children of r. See Figure 8(c). □

By compressing P, we can change the A_k-tree into a B_k-tree and an extra node. Deleting the extra node and repeating the procedure yields our result.

Theorem 2.4.13 *A sequence of n order-preserving path compressions and leaf deletions satisfying the Rising Roots Condition on an initial tree of n nodes can incur $\Theta(n \log n)$ cost.*

PROOF: First, we derive the lower bound. Form the initial tree T by augmenting a B_k-tree n times, where $n = 2^k$. This produces a B_k-tree attached to a path $(x_1, \ldots, x_n)$ of n nodes, as depicted in Figure 9. Form the path P as used in Lemma 2.4.12 from the B_k-tree and node x_1. By Lemma 2.4.12, compressing path P yields a collection L_{k-1} of trees laid out in deque order and an extra B_0-tree, the roots of all of which are children of x_1. Deleting the extra B_0-tree, which is just a singleton node, yields a B_k-tree, the root of which is a child of node x_2; Lemma 2.4.11 shows this. We can perform this procedure n times, each time performing a path compression of length $k+1$ and a leaf deletion. The path compressions are order-preserving and satisfy the Rising Roots Condition. Initially, $|T| = 2n$ and $k = \log_2 n$, thus proving the lower bound.

Since the path compressions satisfy the Rising Roots Condition, Lemma 2.4.2 shows that the path compression sequence corresponds to an instance of disjoint set union with path compression. Tarjan and van Leeuwen [TvL84] prove that such a sequence incurs $O(n \log n)$ cost. $\square$

2.5 Summary

We have described how to implement catenable heap-ordered double-ended queues in constant amortized time per operation (worst-case if the number of deques is fixed). In Chapter 3, we give an $O(\log^* n)$ worst-case time per operation solution for an arbitrary number of deques. The important contributions of our work are the use of structural abstraction in designing the data structure and the analysis of deque-ordered, spine-only path compression in proving its efficiency. Structural abstraction obviates the problem of resolving minima lists when catenating heap-ordered deques by reducing catenation to simple insertion. Our data structure is easy to implement and unifies previous data structures (such as heap-ordered queues and non-catenable heap-ordered deques) that have many applications. Our path compression result generalizes the work of Loebl and Nešetřil [LN88a, LN88b, LN89] and of Lucas [Luc90] and provides what we believe is the first application for such results.

We discuss related open problems in Chapter 4.

Chapter 3

Confluently Persistent Deques

3.1 Introduction

This chapter presents a solution to the confluently persistent deque problem. Our data structure requires $O(\log^* m)$ amortized time and space for pop and eject, where m is the number of deque operations performed so far, and $O(1)$ amortized time and space for all the other operations. We also show how to make these resource bounds worst-case; our solution can be implemented purely functionally, with no side effects. Noting that the Driscoll, Sleator, and Tarjan [DST91] solution to the output-restricted deque problem provides non-constant resource bounds ($O(\log \log m)$ amortized time and space) for catenate rather than pop and eject, we can modify our data structure to provide a solution to the output-restricted deque problem with $O(\log^{(i)} m)$ amortized time and space bounds for catenate, where i is any desired constant, and $O(1)$ amortized time and space bounds for the other operations.

Again we use data-structural bootstrapping, though in this chapter we use both types. First, we use structural abstraction to represent fully persistent lists by their persistent version numbers; this effects persistence in our data structure. Second, we use structural decomposition to implement deques of n elements by decomposing them into collections of deques of $O(\log n)$ elements and applying this decomposition recursively.

We begin in Section 3.2 by explaining why the methods of Driscoll et al. [DSST89] do not facilitate construction of confluently persistent deques. We then use data-structural bootstrapping to design confluently persistent deques, using fully persistent deques as basic building blocks. We first present a data structure in which all operations require $2^{O(\log^* m)}$ amortized time and space. While $2^{O(\log^* m)}$ is an unusual function (asymptotically greater than $\log^* m$ but less than $\log^{(i)} m$ for any i), it results from a natural recurrence: $F(m) = cF(\log m) + O(1)$. This recurrence arises from the structural decomposition of a deque of n elements into a collection of deques of $O(\log n)$ elements each. Section 3.3 describes the *deque tree*, a tree whose leaves represent a deque. After discussing a way to keep deque trees balanced, we demonstrate how to implement them using only lists and catenable deques. While Driscoll, Sleator, and Tarjan [DST91] suggest a similar decomposition of trees into paths that in turn are represented by trees, we make the observation that a decomposition of trees into paths accessible only at their ends is possible and preferable. We then apply data-structural bootstrapping to the deque trees to make them confluently persistent and efficient. We improve our data structure in Section 3.4 to reduce the resource bounds for pop and eject to $O(\log^* m)$ each and those for the other operations to $O(1)$ each; we also make these bounds worst-case. We note that our data structure suffices to implement catenable heap-ordered deques of n elements such that each heap-ordered deque deletion takes $O(\log^* n)$ worst-case time and the other heap-ordered deque operations take $O(1)$ worst-case time. Section 3.5 shows a lower bound on the methods given to balance deque trees, suggesting that simple extensions of the methods presented in this chapter might not suffice to improve the bounds further. We summarize in Section 3.6. A preliminary version of the material in this chapter appears as Buchsbaum and Tarjan [BT93].

3.2 Reviewing Persistent Deques

We begin by explaining in part how to make fully persistent deques. We also discuss why techniques for making deques fully persistent do not suffice to create efficient confluently persistent deques.

3.2.1 Naïve Methods

Two naïve methods that make deques fully persistent in fact can also make them confluently persistent. The resource bounds that result, however, are unsatisfactory.

The most naïve method is *list copying*. When performing any update on a deque, we simply make a new copy of the deque and perform the operation on the new copy. This approach, however, requires $\Theta(n)$ time and space for an operation on a deque of n elements.

A slightly more sophisticated approach is *path copying* applied to trees. We represent a deque as a balanced tree, similarly to the method used in Chapter 2. The leaves of the tree in symmetric order represent the elements of the deque in left-to-right order. Catenation links (or *melds*) two trees by joining them at appropriate nodes; deletion removes the leftmost or rightmost leaf in symmetric order from a tree. To update a tree node, we copy only the path from the root to the affected node. Any balanced tree suffices; e.g., we can use a red-black tree [GS78]. Path copying requires $\Theta(\log n)$ time and space per operation on an n-element deque.

Both of these methods suffice to implement confluently persistent deques. They consume large amounts of time and space, however.

3.2.2 Sophisticated Methods

The methods of Driscoll et al. [DSST89] create fully persistent deques that require only $O(1)$ amortized time and space per operation (except catenation). As opposed

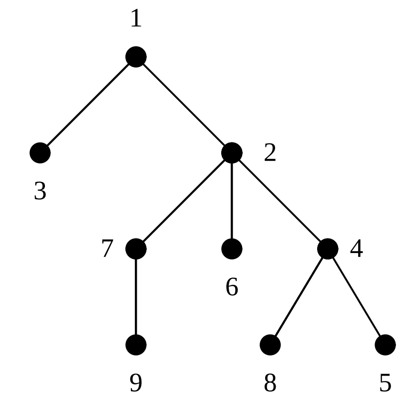


Figure 10: A version tree. The associated version list is $133'2799'7'66'488'55'4'2'1'$.

to list copying and path copying, the methods of Driscoll et al. [DSST89] do not immediately extend to allow confluently persistent deques. Here we explain why.

Key to the work of Driscoll et al. [DSST89] is the concept of linearizing a version tree. A *version tree* is a rooted tree that describes the history of the data structure. If an update to version i creates version j, then node j is a child of node i in the version tree. The persistent data structure then contains nodes that have space for extra pointers, and each pointer is tagged by the version number associated with it. When navigating in version i of the data structure, the algorithm chooses a pointer among those in the current node that are tagged as ancestors of node i in the version tree. The navigation problem thus depends on resolving ancestor relationships among pairs of nodes in the version tree.

Driscoll et al. [DSST89] observe that such ancestor queries are easy if the version tree is *linearized* into a version list as follows: traverse the version tree in symmetric order, outputting a node the first and last times it is encountered. Specifically, each node x is output once (as x) in preorder and once (as x') in postorder. In this case, x is an ancestor of y in the tree if and only if (1) x precedes y and (2) y precedes x' in the version list. Creating a new version j that is a child of version i in the version tree entails inserting jj' immediately after i in the version list. See Figure 10.

Determining ancestor queries thereby reduces to determining order in the version list. In particular, the operations performed on the version list are (1) insert an item after a given item in the list, and (2) given two items in the list, determine which precedes the other. Dietz and Sleator [DS87] show how to perform each of these operations in $O(1)$ worst-case time.

Allowing confluent persistence, however, necessitates representing the history of a data structure as a version DAG (directed acyclic graph). In particular, an operation, such as catenation, can create a new version i that descends from two (or more) older ones, j and k; in this case, there are arcs from j to i and from k to i in the version DAG. Navigating in the persistent data structure then requires determining ancestor relationships among pairs of nodes in the version DAG. How to resolve this reachability problem efficiently enough to base confluent persistence on the solution is unknown.

Chuang and Goldberg [CG93] give an alternative implementation of fully persistent deques. Their result, which is anticipated by Gajewska and Tarjan [GT86], is based upon global rebuilding, a technique that we use in Section 3.4.3. As described in Section 3.4.3, however, global rebuilding does not immediately extend to allow catenation.

The remainder of this chapter shows how we apply data-structural bootstrapping to obviate the problem of reachability in version DAGs in order to create confluently persistent deques.

3.2.3 Naïve Structural Abstraction

Why can we not just use structural abstraction to implement confluently persistent deques as we implement catenable heap-ordered deques? That is, abstract fully persistent deques, representing each by its persistent version number, and insert the version numbers into other fully persistent deques to effect catenation.

The problem is that the path compression result that makes the heap-ordered deques efficient is an amortized-time result: each heap-ordered deque operation takes $O(1)$ amortized time, because each path compression takes $O(1)$ amortized time. The amortization does not extend to the persistent setting, however. In particular, any "bad" version—one in which a long path compression occurs—can be replicated many times by a number of persistent updates. A sequence of persistent operations can repeat the same long path compression several times and therefore take more than linear time.

We can apply structural abstraction (and structural decomposition) to the confluently persistent deque problem, however, provided that we abstract an appropriate representation of deques. We begin describing this representation in the next section.

3.3 Deque Trees

As in Chapter 2, we represent a deque as a tree. Consider an ordered tree [Knu73a] embedded in the plane with the root at the top and the leaves at the bottom. We can exploit the induced left-to-right order of the leaves of the tree to make the tree represent a deque. The two operations we perform on the tree are *link*, which takes two trees and makes the root of one the new leftmost or rightmost child of the root of the other (see Figure 11(a)), and *delete*, which removes either the leftmost or the rightmost leaf of the tree (see Figure 11(b)). Link corresponds to deque catenation and insertion, and delete corresponds to pop and eject. The internal nodes of the tree can be of arbitrarily high degree, but we always ensure that they are at least binary; to do this requires replacing an internal node x by its remaining child if a deletion renders x unary. See Figure 11(c). It also requires a special case of link when linking two single-node trees; in this case, a new root node is created with the two old nodes becoming its children. Since delete always preserves the left-to-right order

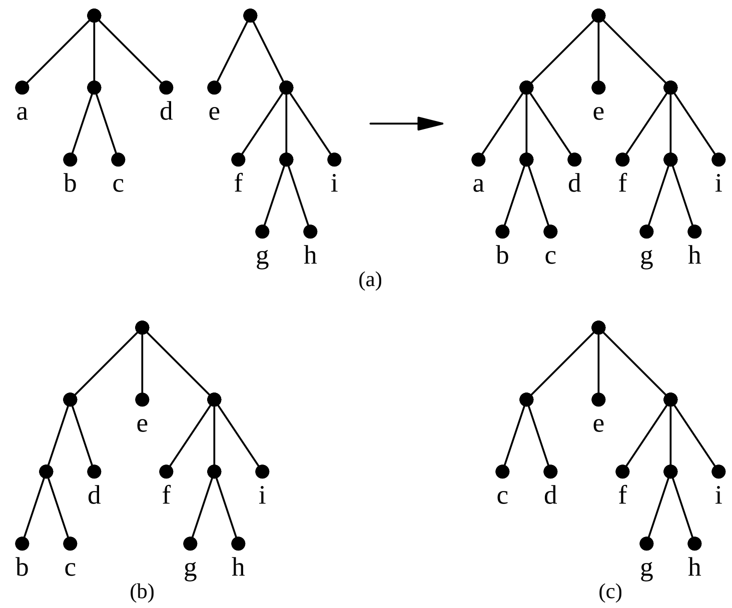


Figure 11: (a) Linking two trees. (b) After deleting the leftmost leaf of the linked tree of (a). (c) After deleting the leftmost leaf of the tree of (b).

of the remaining leaves, it is clear that such trees can be used to implement catenable deques, and we therefore call them *deque trees*.

The remainder of this section describes a method to keep deque trees "in balance." The next two sections describe a method to implement confluently persistent deque trees (and thus deques). The efficiency of the implementation derives from the deque tree balance property.

We again define a *pull* operation to apply to deque trees. This pull operation is identical to that of Driscoll, Sleator, and Tarjan [DST91] but slightly different from the left and right pulls defined in Chapter 2. Let x be the leftmost non-leaf child of the root r of a tree T, and let x' be the leftmost child of x. A *pull* on T cuts the link from x' to x and makes x' a new child of r just to the left of x. Additionally, if x is now unary, it is replaced by its remaining child. See Figure 12. We can also define this as a *left pull* and define a *right pull* symmetrically, but for our needs left pulls suffice. Note that a pull preserves the left-to-right order of the leaves of T.

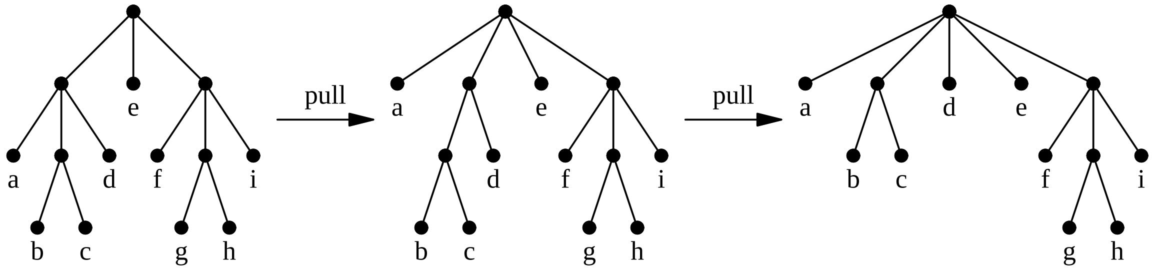


Figure 12: Two pulls applied to a tree.

We call a tree *flat* if all the children of the root are leaves or if the tree is a singleton node. A pull on a flat tree does nothing. A pull on a non-flat tree increases the degree of the root by one. In what follows, the *size* of a tree T, denoted by $|T|$, is the number of leaves in T.

We now describe how to use pulls to keep deque trees balanced, i.e., of logarithmic depth. To be precise, we define the *depth* of a deque tree to be the length of its longest root-to-leaf path. We wish to maintain the following invariant: denoting the depth of a deque tree T by d_T, we keep $|T| \geq c^{d_T}$ for some constant $c > 1$. To maintain this invariant, we augment the implementation of deque trees to perform k pulls on T every time we perform a deletion on T, for some constant k; without loss of generality, assume that the pulls precede the deletion. Also, we implement *linking by size*; i.e., when linking two trees A and B, we link the one with fewer leaves to the one with more, breaking a tie arbitrarily. This technique is also used to maintain balanced trees in the disjoint set union problem [TvL84].

Deque trees, therefore, are trees subject to the operations link, delete, and pull, as described above. Further, linking is done by size, and k pulls precede each deletion. We now prove that deque trees remain balanced.

Lemma 3.3.1 *If the number of pulls k per deletion is at least 1, then a non-root node in a deque tree T never becomes a root node unless T becomes a singleton node.*

PROOF: New root nodes are created by the makelist operation and by linking two singleton trees together. Consider a deletion on a tree T; at least one pull precedes this deletion. Before the pull, either T is flat or else the root of T is of degree at least two. In the former case, if the deletion creates a new root node of T, then T becomes a singleton node; in the latter case, the pull increases the degree of the root of T by one, to at least three, so the deletion does not change the root of T. □

Now we can prove that pulls maintain balanced deque trees.

Theorem 3.3.2 *If the number of pulls k per deletion is at least $\frac{2}{2-c}$, for some constant $c \in (1, 2)$, then $|T| \geq c^{d_T}$.*

Proof: We proceed by induction on the number of deque tree operations. A singleton tree of depth zero is created by makelist. Consider any depth-increasing link of a tree T to a tree T' yielding tree T''. We have $d_{T''} = d_T + 1 > d_{T'}$. By induction, $|T| \geq c^{d_T}$. Linking by size gives $|T'| \geq |T|$, and thus $|T''| \geq 2c^{d_T}$. As long as $c \leq 2$, the invariant holds.

Assume some deletion creates a tree X with root r such that $|X| < c^{d_X}$. We show that this produces a contradiction. Let R be the most recent tree rooted at r formed before X by a depth-increasing link. By the above analysis of linking, we know that $|R| = 2c^{d_R - 1} + x$ for some $x \geq 0$. The intermixed sequence of deletions and links that produces X from R cannot make r a non-root node; otherwise Lemma 3.3.1 shows that X is the singleton node r. Let T be the evolving tree, initially R, rooted at r as leaves are deleted from it and other trees linked to it. For $|X| < c^{d_X}$ to be true, at least $(2-c)c^{d_R - 1} + x + y$ leaves must be deleted from T, where y is the total number of leaves added to T via links. If $k \geq \frac{2}{2-c}$, then at least $2c^{d_R - 1} + \frac{2}{2-c}(x+y)$ pulls occur during these deletions. These are sufficient to flatten T before T becomes X, because (1) each pull increases the degree of the root of T by one unless T is flat, and (2) there are always at least as many pulls remaining as leaves of T. Once T is flat, the depth invariant holds until the next depth-increasing link. So no depth violation occurs. □

In particular, we can set $c = 3/2$ and $k = 4$.

Corollary 3.3.3 *If T is a deque tree, then $d_T = O(\log |T|)$.*

We prove in Section 3.5 that performing k pulls per deletion, for any constant k, can in fact result in $d_T = \Omega(\log |T|)$.

We close this section by contrasting our use of the pull operation with previous uses. Whereas Driscoll, Sleator, and Tarjan [DST91] employ the pull operation primarily to ensure that the left spine of any tree remains short (of constant length), we utilize it to keep the tree as a whole balanced in the sense that no leaf is of greater than logarithmic depth. Both results, however, use only the fact that pulls completely flatten the tree over some predictable period of time. For this reason, the full effect of the pull operation on the structure of the trees may be underutilized in these analyses. In Chapter 2, we use a pull operation to effect path compression on trees, with no local or global balance considered.

3.3.1 Decomposition into Spines

We now describe how to decompose a deque tree into a collection of *spines*. We define the spines bottom-up. A *left spine* of a tree is a maximal tree path $(x_0, \ldots, x_l)$ such that x_0 is a leaf and x_i is the leftmost child of x_{i+1} for $0 \le i < l$; we say the spine *terminates* or *ends at* x_l. A *right spine* is defined symmetrically. We show how to represent a deque tree as a set of spines so that all the deque tree operations—link, delete, and pull—require accessing the spines only at their ends. By recursively representing the spines by smaller deque trees, we obtain a bootstrapped implementation of confluently persistent deques.

Recall the deque tree operations: link, delete, and pull. We first describe how to implement them ephemerally using the spine-decomposition structure. Later we will show how to make the implementation persistent. Let $root(T)$ be the root of a deque tree T. We store the left and right spines terminating at $root(T)$ in the deques $ls(T)$ and $rs(T)$; these are called the *root spines*. For a node x with i children in T, let the children of x be $c_x^1, \ldots, c_x^i$ in left-to-right order. Let l_x^j (r_x^j) be the left (right) spine terminating at c_x^j for $1 \leq j \leq i$. We store at node x a pointer to a doubly-linked list $s\text{-}list(x)$ whose elements left-to-right are $r_x^1, l_x^2, r_x^2, \ldots, l_x^{i-1}, r_x^{i-1}, l_x^i$; i.e., we record the right spine terminating at the leftmost child, the left and right spines terminating at the middle children, and the left spine terminating at the rightmost child of x. The "missing" spines are recorded in the s-lists of some other nodes (if they are not the root spines). All the spines are stored as deques with the leaves at the front (pop) ends. The spine deques, therefore, contain as elements nodes pointing to their respective s-lists. See Figure 13. A node in a deque tree is thus stored in precisely two spines, appearing as the top node in at least one of them.

The operations on the s-lists are (1) $insert(l, x, y)$ — insert y after x in s-list l; and (2) $delete(l, x)$ — delete x from s-list l and return x. Additionally, each element x in an s-list has a $pred(x)$ and $succ(x)$ pointing to the predecessor and successor of x respectively. Special sentinels $head(l)$ and $tail(l)$ point to the first and last elements of s-list l, and the special case insert$(l, pred(head(l)), x)$ inserts a new first element x into l. The sentinels are updated by the s-list implementation machinery. Again, assume for now that these are ephemeral operations.

To make a new deque tree T with one node x, we set $ls(T)$ and $rs(T)$ to be the singleton deque (x) and $s\text{-}list(x)$ to be $\emptyset$. Now we describe how to link a deque tree A to another deque tree B, assuming at least one of them contains more than one leaf. Assume that $root(A)$ becomes the new leftmost child of $root(B)$; the other case is symmetric. First, eject $root(B)$ from $ls(B)$ and insert the new $ls(B)$ as the new

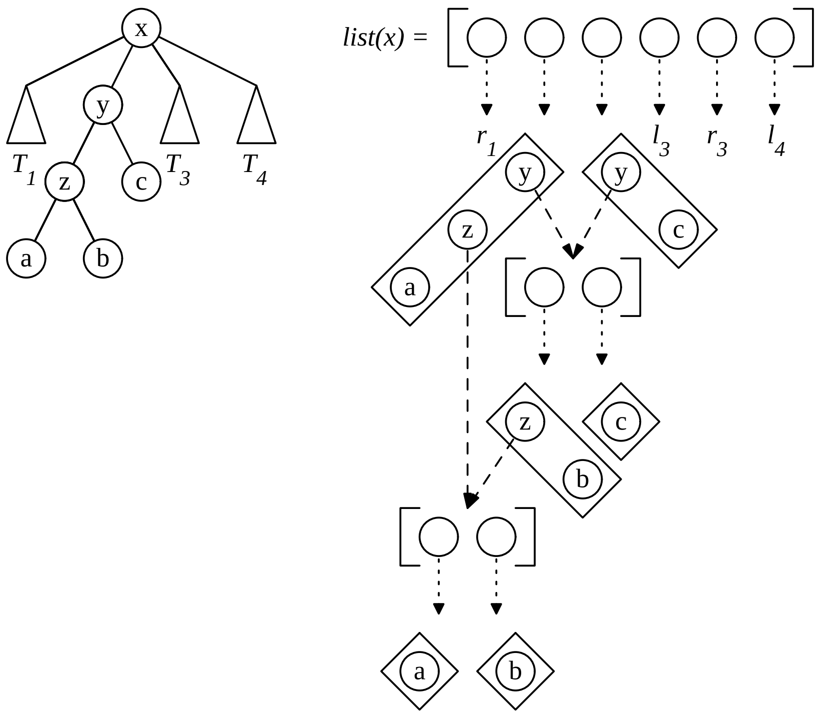


Figure 13: A subtree rooted at node x and a portion of its spine decomposition structure. Boxes surround spine deques, and s-lists are bracketed; l_i (r_i) refers to the spine deque for the left (right) spine terminating at the root of subtree T_i. Null s-lists (for leaves) are omitted.

```
(x, ls(T)) ← pop(ls(T)) // the leftmost leaf of T
(y, ls(T)) ← pop(ls(T)) // the parent of x
if (y has more than 2 children) then
    delete(s-list(y), head(s-list(y))) // discard x as the old right spine
    l ← delete(s-list(y), head(s-list(y))) // next left spine
    ls(T) ← catenate(l, catenate(makelist(y), ls(T)))
else if (y ≠ root(T)) then // must delete y and promote other child
    (g, ls(T)) ← pop(ls(T)) // the parent of y
    l ← tail(s-list(y)) // next left spine
    r ← delete(s-list(g), head(s-list(g))) // right spine terminating at y
    (y, r) ← eject(r) // remove y
    insert(s-list(g), pred(head(s-list(g))), r) // new right spine
    ls(T) ← catenate(l, catenate(makelist(g), ls(T)))
else // y = root(T) and has two children
    ls(T) ← tail(s-list(y)) // new left spine of T
    (y, rs(T)) ← eject(rs(T)) // delete y
```

Figure 14: Pseudocode for deletion of leftmost leaf of T. Double slash (//) precedes comments.

head of *s-list*(*root*(B)), as this is now the left spine terminating at the new second-leftmost child of B. Then insert $rs(A)$ as the new head of *s-list*(*root*(B)); this is now the right spine terminating at the new leftmost child of B. Finally, reset $ls(B)$ to be the catenation of $ls(A)$ and *root*(B). The special case when A and B are both singleton deque trees is handled by creating a new structure for the resulting tree of a root and the two leaves.

A deletion is no more complicated, but it involves more steps, so we merely give pseudocode for deletion. See Figure 14; recall that Figure 11 demonstrates deletion pictorially. The code in Figure 14 is for a deletion of the leftmost leaf of T; the code for deletion of the rightmost leaf is symmetric. We note that it is crucial to the functioning of the spine decomposition that the s-lists not store the first left spine or the last right spine; these are stored as parts of bigger spines in other places. This lack of duplication allows the easy implementation of the deque tree operations by simple deque and s-list operations. It is also critical that the internal nodes remain at least binary; this avoids the situation in which two spines intersect at nodes other than the top of one of them.

To implement pulls, we augment the s-lists to store as secondary information doubly-linked lists of pointers to the non-leaf children of each node. Each node also stores its degree. Using the notation above, if c_x^j is a non-leaf child, then a pointer points to l_x^j in $s\text{-}list(x)$; the exception is that if c_x^1 is a non-leaf child, then the pointer points to r_x^1. This seems unfortunate, but it turns out that we must handle the case of a pull affecting a root spine slightly differently from one affecting a middle child of $root(T)$ anyway. It is straightforward to maintain this secondary information when modifying the s-lists, and for clarity we omit this part of the implementation from the description. (In fact, we obviate this secondary information in Section 3.4.) We merely mention that maintaining the secondary information is possible since actual s-list modifications only take place at either the ends of the s-lists or near elements in the s-lists that are themselves targets of these secondary pointers. Figure 15 shows pseudocode to implement the pull operation if the first non-leaf child x of $root(T)$ is on $ls(T)$ (a similar procedure handles the case for $rs(T)$), and Figure 16 shows pseudocode for the case of x a middle child of $root(T)$. Note that the two procedures bear definite similarities. They could easily be combined into one procedure with a proper prologue and epilogue, but keeping them distinct increases their clarity. Refer back to Figure 12 for a pictorial representation of a pull.

The above explanation shows that a constant number of ephemeral deque and s-list operations suffices to implement any ephemeral deque tree operation.

```
(r, ls(T)) ← eject(ls(T)) // remove root(T) temporarily
(x, ls(T)) ← eject(ls(T)) // leftmost child of root(T)
if (degree(x) > 2) then // x remains a child of root(T)
      r_y ← delete(s-list(x), head(s-list(x))) // y is leftmost child of x
      l_z ← delete(s-list(x), head(s-list(x))) // z is next
      insert(s-list(r), pred(head(s-list(r))), catenate(l_z, makelist(x)))
            // new left spine to x
      insert(s-list(r), pred(head(s-list(r))), r_y) // right spine to y
else // delete x and promote both its children
      r_x ← delete(s-list(r), head(s-list(r))) // right spine to x
      (x, r_x) ← eject(r_x) // get rid of x
      insert(s-list(r), pred(head(s-list(r))), r_x) // now right spine to z
      insert(s-list(r), pred(head(s-list(r))), tail(s-list(x))) // left spine to z
      insert(s-list(r), pred(head(s-list(r))), head(s-list(x))) // right spine to y
ls(T) ← catenate(ls(T), makelist(r)) // restore root(T) to ls(T)
```

Figure 15: Pseudocode for pull on T when x is on $ls(T)$. Double slash (//) precedes comments.

3.3.2 Data-Structural Bootstrapping

Section 3.3.1 shows how to implement ephemeral deque trees using catenable deques and s-lists. We now describe how to implement confluently persistent deque trees using this method as a basis: we bootstrap confluently persistent deques and fully persistent s-lists.

First, note that the s-list structure is a *bounded indegree* structure, in the terminology of Driscoll et al. [DSST89]. Furthermore, each *s-list*(x) has only four *access pointers*, i.e., entry points to the s-list where access can start. These four points are the two ends and the nodes corresponding to the leftmost and rightmost non-leaf children of x. Thus, we can use the methods of Driscoll et al. [DSST89] to make s-lists fully persistent so that a non-destructive s-list operation takes $O(1)$ amortized time and space. For now, assume the existence of confluently persistent deques.

Let l_x be the left spine pointed to as the first non-leaf child of $root(T)$
$r_x \leftarrow succ(l_x)$
$p \leftarrow pred(l_x)$
$r \leftarrow root(T)$
delete($s\text{-}list(r), l_x$) // left spine to x
$(x, l_x) \leftarrow$ eject(l_x) // delete x from it
if (degree(x) > 2) then // x remains a child of $root(T)$
    $r_y \leftarrow$ delete($s\text{-}list(x), head(s\text{-}list(x))$) // y is leftmost child of x
    $l_z \leftarrow$ delete($s\text{-}list(x), head(s\text{-}list(x))$) // z is next
    insert($s\text{-}list(r), p$, catenate(l_z, makelist(x))) // new left spine to x
    insert($s\text{-}list(r), p, r_y$) // right spine to y
    insert($s\text{-}list(r), p, l_x$) // now left spine to y
else // delete x and promote both its children
    delete($s\text{-}list(r), r_x$) // right spine to x
    $(x, r_x) \leftarrow$ eject(r_x) // now remove x from it
    $r_y \leftarrow head(s\text{-}list(x))$ // y is left child of x
    $l_z \leftarrow tail(s\text{-}list(x))$ // z is right child of x
    insert($s\text{-}list(r), p, r_y$) // right spine to y
    insert($s\text{-}list(r), p, l_x$) // now left spine to y
    insert($s\text{-}list(r), r_y, r_x$) // now right spine to z
    insert($s\text{-}list(r), r_y, l_z$) // left spine to z

Figure 16: Pseudocode for pull on T when x is a middle child of $root(T)$. Double slash (//) precedes comments.

We use structural abstraction to represent persistent s-lists and spine deques by their persistent version numbers. In this way, we make each element in a persistent s-list point to its respective spine via a *version number* referring to the corresponding persistent deque. Similarly, each element of a persistent deque contains the version number of the persistent s-list of the corresponding node. The confluently persistent deque tree T is then fully described by two numbers: the version numbers of $ls(T)$ and $rs(T)$. We mention that the cases in the above implementation in which a node is popped off a spine only to be recatenated to the spine are present in fact to record the new version number of the node being recatenated to the spine. One implementation detail remains: updating a spine deque creates a new version of that deque, which must be stored in any s-list pointing to the spine; this creates a new version of the s-list, which must be reflected in a "higher" spine pointing to the list, and so on. This is further complicated by the fact that a node occurs in two spines, and when an s-list changes, both copies of the node that points to it must be updated.

We fix the second problem by storing the version number of any $s\text{-}list(x)$ only in the copy of node x that occurs in the middle of a spine; the copy occurring at the top of a spine is left blank. If both copies occur at the tops of spines, one is chosen arbitrarily to store the version number of $s\text{-}list(x)$. This works for the following reason. Consider how spines and s-lists are accessed: any time an $s\text{-}list(x)$ is accessed via the copy of node x occurring at the top of a spine, the spine is either a root spine or a spine actually pointed to by $s\text{-}list(r)$, where r is the root of a tree. In either case, both copies of x are readily available via a constant number of deque and s-list operations and so the version number of $s\text{-}list(x)$ may be retrieved. The important fact is that no reverse pointers are needed. These would invalidate the methods of Driscoll et al. [DSST89] in making the s-lists fully persistent, since too many access pointers would need to be accommodated. Now note that any modifications to spines and s-lists occur close to a root spine. That is, updating a spine might cause an update to an s-list, but the node pointing to this s-list will be on $ls(T)$ or $rs(T)$. Therefore, no cascading sequence of version updates occurs. As an aside, we note that Driscoll, Sleator, and Tarjan [DST91] use the pull operation to ensure that any modifications to their structure occur close to the roots of their trees, thereby keeping cascading sequences of updates short.

While it is clear that s-lists can be made fully persistent using previously known techniques [DSST89], the method described above relies upon the existence of confluently persistent deques in their own implementation. Note that to represent a confluently persistent deque of size n, however, we need only be able to implement confluently persistent deques of size $O(\log n)$; this is due to Corollary 3.3.3. This is the method of structural decomposition: a deque of n becomes a collection of deques of $O(\log n)$ elements each, and the decomposition is applied recursively.

Theorem 3.3.4 *Any operation on a confluently persistent deque of size n can be performed in $2^{O(\log^* n)}$ amortized time and space.*

Proof: We represent small confluently persistent deques (those of size less than some suitable constant, e.g., four) by doubly-linked lists. Operations on such a small deque are performed by copying the entire deque and modifying the new copy. Any time a deque increases to a size above this threshold, we implement it via the deque tree spine-decomposition structure described in Section 3.3.1.

An operation on a confluently persistent deque tree of n leaves implemented via the spine-decomposition structure requires $O(1)$ fully persistent s-list operations, each taking $O(1)$ amortized time and space [DSST89]. It also requires $O(1)$ operations on confluently persistent deques that by Corollary 3.3.3 are of size $O(\log n)$. Therefore, for some constants c_1 and c_2, the amortized time and space required by an operation on a confluently persistent deque of size n is $F(n)$, where $F(n)$ is described by the following recurrence:

$$F(n) = \begin{cases} O(1) & \text{if } n \leq c_1 \\ c_2 F(\log n) + O(1) & \text{if } n > c_1 \end{cases}$$

The solution to this recurrence is $F(n) = 2^{O(\log^* n)}$. □

We note that confluently persistent deques support "self-catenation," i.e., forming a new deque by catenating an old one to itself. Therefore, m catenations can generate a deque of size 2^m. This potential size explosion is addressed by Driscoll, Sleator, and Tarjan [DST91] with a "guessing trick" that involves guessing how many operations will be performed and keeping the lists truncated to an appropriate size, doubling or squaring the guess and rebuilding the lists each time the number of operations exceeds the previous guess. This technique is important in this previous work, since the resource bounds obtained there of $O(\log n)$ and $O(\log\log n)$ would otherwise become $O(m)$ and $O(\log m)$ respectively, which would be poor amortized results. Our method requires no such additional machinery, however, since our resource bound grows sufficiently slowly in terms of n: for $n = O(2^m)$ we have $2^{O(\log^* n)} = 2^{O(\log^* m)}$.

Finally, we conclude this section by mentioning how to improve the performance of catenate, first, and last. The latter two operations can easily be made to take $O(1)$ worst-case time by storing with each deque its actual first and last elements. This does not degrade the performance of the data structure. We improve catenate as follows. Treat the root spines ($ls(T)$ and $rs(T)$) specially by not storing the actual node $root(T)$ in them. We can now implement the standard push and inject operations by directly modifying the spine-decomposition structure of the affected deque tree using only s-list operations, rather than calling catenate. Now use push and inject in place of catenate wherever possible in the implementation. By doing all this, we remove the eject call from the link operation, and catenate becomes a non-recursive call, utilizing only a constant number of inject and s-list operations and thus taking $O(1)$ amortized time and space per call. In the next section, we further improve our data structure to reduce the cost of deletion to $O(\log^* m)$ amortized time and space. We also show how to make our resource bounds worst-case.

3.4 Improving the Data Structure

The only non-constant-cost operation in our data structure is deletion. In Section 3.3, we described how to implement a deletion on a deque of size n via a number of deletions on deques of size $O(\log n)$, yielding a $2^{O(\log^* n)}$ bound. In this section we explain how to modify our data structure so that a deletion from an n-size deque requires at most one deletion from an $O(\log n)$-size deque, thereby reducing our resource bound to $O(\log^* n)$. The bounds for the other operations are unchanged.

For clarity, we use the term *deletion* to refer to the corresponding operation on a deque of n items or a deque tree of n leaves, and we use *subdeletion* to refer to the operation on a smaller ($O(\log n)$-size) structure.

In our original implementation, one deletion can require four subdeletions to modify the spine decomposition of the deque tree plus some number of pulls (previously four), each of which potentially requires three subdeletions. In Section 3.4.1, we modify the representation of the spine decomposition to reduce to one the number of subdeletions required to update the spine decomposition of a deque tree undergoing a deletion. In order to reduce our resource bound to $O(\log^* n)$, we must then eliminate all of the subdeletions required by pulls. We do this in Sections 3.4.2 and 3.4.3 by eliminating the pulls altogether and replacing them with another mechanism to maintain balanced deque trees. Section 3.4.4 then provides the details on how to make the new implementation persistent. Section 3.4.5 applies the data structure to implement catenable heap-ordered deques.

3.4.1 Modified Representation

As previously mentioned, the present implementation can require as many as four subdeletions per deletion in order to update the spine decomposition of the deque tree:

1. to remove x, the node being deleted;

2. to remove or update y, the parent of x;

3. to update the parent of y, say g;

4. to remove the terminating node (y) of the right spine terminating at the leftmost child of g.

Our new implementation eliminates three of these subdeletions.

We keep the same spine decomposition as described in Section 3.3.1; i.e., the root spines and s-lists still record the same spines as before. We increase the degree invariant maintained at each non-root internal node so that each such node now has degree at least three instead of two; the root node still has degree at least two. Our biggest change, though, is how we store the spines. Whereas before each spine was stored as a deque, we now store the leaf node and the terminating node of each spine separately and keep only the non-leaf and non-terminating (*middle*) nodes of the spine in a deque. Section 3.4.2 makes use of these changes to provide efficient traversals of deques. Here we investigate the impact of the modifications on the algorithm for maintaining the spine decomposition. Given the record s for any spine, $\mathit{leaf}(s)$, $\mathit{deq}(s)$, and $\mathit{term}(s)$ respectively give us the leaf node, middle deque, and terminating node of s.

First, we no longer need to maintain the actual first and last elements of each deque, since we can access these elements as $\mathit{leaf}(ls(T))$ and $\mathit{leaf}(rs(T))$ each in $O(1)$ time. (T is the deque tree corresponding to the deque.) Additionally, we can now *change in place* the first and second (and last and second-to-last) elements of any deque; that is, we can modify them without performing subdeletions on the spines of T. To change the first element in place merely requires updating $\mathit{leaf}(ls(T))$. To change the second element in place, we first access $y = \mathit{leaf}(\mathit{deq}(ls(T)))$; this is the parent of $\mathit{leaf}(ls(T))$ in T. We then take from $s\text{-}\mathit{list}(y)$ the left spine l terminating at the second child of y. The second element of T is $\mathit{leaf}(l)$, and we can modify this directly. This produces a new left spine l', which we store in the proper place in $s\text{-}\mathit{list}(y)$. In a persistent setting, we now have a new version of y, which we can store in $ls(T)$ by changing $y = \mathit{first}(\mathit{deq}(ls(T)))$ in place; of course, we also have a

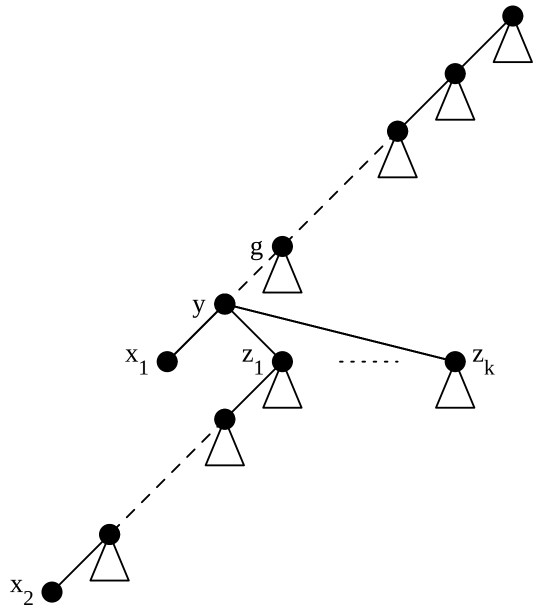


Figure 17: A deque tree T. Dashed lines are paths of nodes that comprise middle deques of spines; triangles represent other subtrees descending from the nodes. The first element of the deque is $x_1 = \mathit{leaf}(\mathit{ls}(T))$; $y = \text{first}(\mathit{deq}(\mathit{ls}(T)))$ is the parent of x_1 in T, and $g = \text{second}(\mathit{deq}(\mathit{ls}(T)))$ is the parent of y in T. Let $l = \mathit{succ}(\mathit{head}(\mathit{s\text{-}list}(y)))$; l is the left spine terminating at the second child (z_1) of y. Then $x_2 = \mathit{leaf}(l)$ is the second element of the deque. We can change x_1 and x_2 in place in T in $O(1)$ time.

new version of $\mathit{ls}(T)$. See Figure 17. Changing the last and second-to-last elements of a deque is symmetric. By the same reasoning, we can also use $\mathit{second}(T)$ and $\mathit{second\text{-}to\text{-}last}(T)$ to access the second and second-to-last elements of a deque in $O(1)$ time.

We exploit the new representation of a spine and changing in place to remove three of the four subdeletions cited above. In particular, we no longer have to pop x off the left spine of T, since x is stored apart from that spine. Similarly, any updates to y (the parent of x) and g (the parent of y) can be done *in place* on $\mathit{deq}(\mathit{ls}(T))$. If y is removed (to preserve the degree invariant), rather than pop it off $\mathit{deq}(\mathit{ls}(T))$, we merely substitute a new node for it by changing it *in place*.

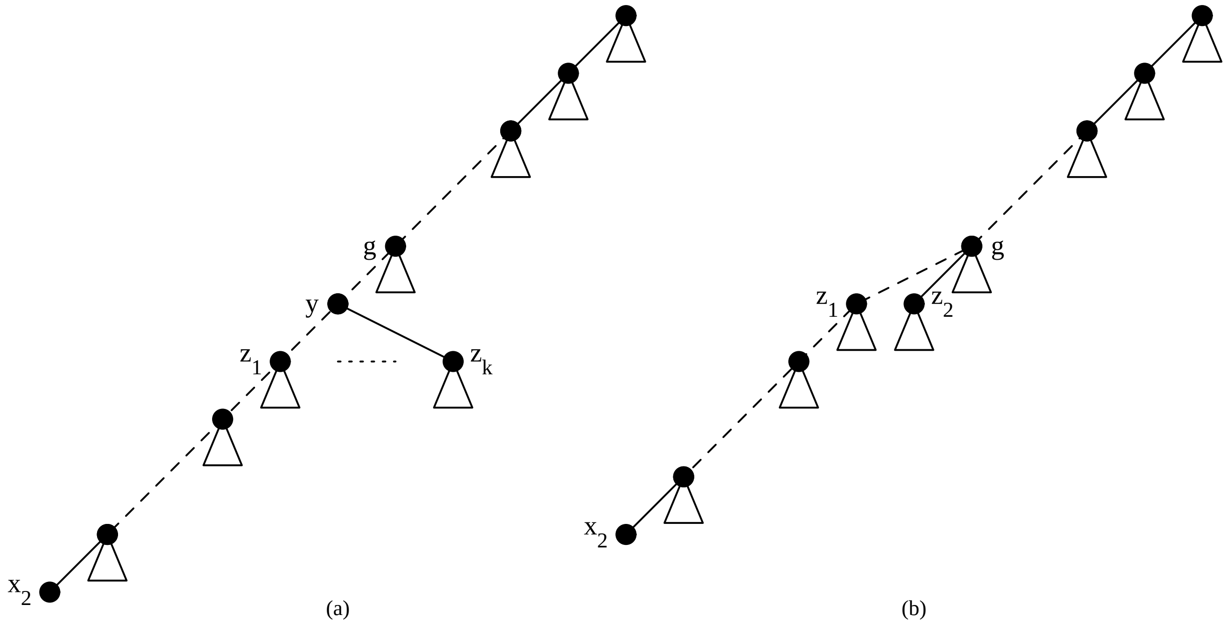


Figure 18: The deque tree of Figure 17 after deleting the leftmost node x_1 if (a) $k > 2$, and (b) $k = 2$. Again, dashed lines represent paths of nodes comprising middle deques of spines. In case (b), y is removed and its children made children of g.

The new procedure is best illustrated when we have to delete y to prevent its degree from falling too low. This now happens when the degree of y is exactly three. Let z_1 and z_2 be the middle and rightmost children, respectively, of y. Note that y is the terminating node of the right spine terminating at the leftmost child (y) of g. We eject z_2 from the middle deque of that spine, and z_2 becomes the new terminating node. We also have to change g *in place* to reflect the changes to *s-list*(g): the replacement of the right spine ending at the leftmost child of g and the addition of the left spine terminating at z_2 and the right spine terminating at z_1. As z_1 is the terminating node of the left spine ending at the second-leftmost child of y, we replace y with z_1 *in place* in $deq(ls(T))$. The new section of $ls(T)$ comes from the leaf node and middle deque of the left spine terminating at z_1, and we can catenate the appropriate segments. In fact, this case is the only time a subdeletion occurs. If y remains on the spine, then no subdeletions occur. This is depicted in Figure 18 and

Let l_x (r_x) be the left (right) spine containing the single node x.
// Note they can be constructed in $O(1)$ time.
insert($\textit{s-list}(\textit{root}(T)), \textit{pred}(\textit{head}(\textit{s-list}(\textit{root}(T)))), \textit{ls}(T)$)
insert($\textit{s-list}(\textit{root}(T)), \textit{pred}(\textit{head}(\textit{s-list}(\textit{root}(T)))), r_x$)
$\textit{ls}(T) \leftarrow l_x$

Figure 19: Pseudocode for pushing element x onto the deque described by tree T; the code for inject is symmetric. Double slash (//) precedes comments.

described via pseudocode in Figures 19–21. We also provide the pseudocode for push (and inject) and catenate. Note that the higher degree invariant slightly complicates linking deque trees: if the root of the smaller tree being linked has degree two, then we must make each of its children a child of the root of the larger tree; otherwise, we can simply link roots as before.

There are a couple of special cases that the pseudocode does not address. First, in the above case (where y is replaced by z_1), if z_1 is a leaf node then we would have to pop y from $\textit{deq}(\textit{ls}(T))$, for in this case z_1 would become $\textit{leaf}(\textit{ls}(T))$. To avoid this second subdeletion, however, we instead replace y *in place* with z_2, catenate the middle deque of the left spine terminating at z_2 to $\textit{deq}(\textit{ls}(T))$, and add z_1 as the new leftmost child $\textit{root}(T)$; this last step is just a push. If z_2 is also a leaf, then we do the pop from $\textit{deq}(\textit{ls}(T))$ (making $g = \text{first}(\textit{ls}(T))$), but no eject from the right spine terminating at y is necessary. So, again, only one subdeletion occurs; in this case, it is subdeletion (2) instead of (4). The second special case is when a spine has fewer than three elements and thus cannot support the leaf, middle deque, and terminating node representation. In this case, all the operations on the spine can be accomplished by dealing with the elements themselves.

```
if (degree(root(A)) < 3) then // must move all children to preserve
        // degree invariant
    insert(s-list(root(B)), pred(head(s-list(root(B)))), ls(B))
    insert(s-list(root(B)), pred(head(s-list(root(B)))), rs(A))
    insert(s-list(root(B)), pred(head(s-list(root(B)))), tail(s-list(root(A))))
        // it must be that degree(root(A)) = 2
    insert(s-list(root(B)), pred(head(s-list(root(B)))), head(s-list(root(A))))
    ls(T) ← ls(A)
else // can just link roots
    insert(s-list(root(B)), pred(head(s-list(root(B)))), ls(B))
    deq(rs(A)) ← inject(term(rs(A)), deq(rs(A)))
        // A becomes new leftmost child of B
    term(rs(A)) ← root(A)
    insert(s-list(root(B)), pred(head(s-list(root(B)))), rs(A))
    deq(ls(A)) ← inject(term(ls(A)), deq(ls(A)))
    term(ls(A)) ← root(A)
    ls(B) ← ls(A)
```

Figure 20: Pseudocode for linking tree A to the left of tree B. Double slash (//) precedes comments.

3.4.2 Deque Traversal

We described in Section 3.4.1 a new implementation of the spine decomposition of a deque tree. We also showed how it could be used to implement new operations on deque trees (such as changing in place). Here we introduce the notion of *traversing* a deque and show how our modified representation facilitates such an action. In Section 3.4.3 we use deque traversal to help eliminate the pulls.

A deque *traversal* is a sequence of steps that makes available the elements of the deque in head-to-tail or tail-to-head order. We extend this notion to deque trees, and say that a deque tree *traversal* makes available the leaves of the deque tree in left-to-right or right-to-left order. To traverse a deque implemented via spine decomposition, we need to traverse its deque tree, which involves traversing the smaller deques that

$x \leftarrow leaf(ls(T))$ // leftmost leaf of T
$y \leftarrow$ first($deq(ls(T))$) // the parent of x in T
if (degree(y) > 3) then
    delete($s\text{-}list(y), head(s\text{-}list(y))$) // right spine to x
    $l_z \leftarrow$ delete($s\text{-}list(y), head(s\text{-}list(y))$) // left spine to z
    [update y *in place* on $deq(ls(T))$]
    $deq(ls(T)) \leftarrow$ catenate($deq(l_z)$, push($term(l_z), deq(ls(T))$))
    $leaf(ls(T)) \leftarrow leaf(l_z)$
else // degree(y) $= 3$ and must promote both other children z_1 and z_2 of y
    $g \leftarrow$ second($deq(ls(T))$) // the parent of y in T
    delete($s\text{-}list(y), head(s\text{-}list(y))$) // right spine to x
    $l_{z_1} \leftarrow$ delete($s\text{-}list(y), head(s\text{-}list(y))$) // left spine to z_1
    $r_{z_1} \leftarrow$ delete($s\text{-}list(y), head(s\text{-}list(y))$) // right spine to z_1
    $l_{z_2} \leftarrow$ delete($s\text{-}list(y), head(s\text{-}list(y))$) // left spine to z_2
    $r_{z_2} \leftarrow$ delete($s\text{-}list(g), head(s\text{-}list(g))$) // right spine to y
    $(term(r_{z_2}), deq(r_{z_2})) \leftarrow$ eject($deq(r_{z_2})$) // delete y from spine,
        // now right spine to z_2
    insert($s\text{-}list(g), pred(head(s\text{-}list(g))), r_{z_2}$)
    insert($s\text{-}list(g), pred(head(s\text{-}list(g))), l_{z_2}$)
    insert($s\text{-}list(g), pred(head(s\text{-}list(g))), r_{z_1}$)
    [update g *in place* on $deq(ls(T))$]
    [change y to $term(l_{z_1}) = z_1$ *in place* on $deq(ls(T))$]
    $deq(ls(T)) \leftarrow$ catenate($deq(l_{z_1}), deq(ls(T))$)
    $leaf(ls(T)) \leftarrow leaf(l_{z_1})$

Figure 21: Pseudocode for deletion of leftmost leaf of T. Double slash (//) precedes comments.

make up its spines, and so on. Ephemeral deques allow traversal in $O(1)$ time per element. With our original data structure, it is not clear that this is possible. Our modified representation, however, allows deque traversal in $O(1)$ amortized time per element.

We traverse the deque tree in symmetric, i.e., depth-first left-to-right (or right-to-left), order. To visit the leftmost or rightmost child of a node requires performing some part of the traversal of the corresponding spine deque. Visiting a middle child of a node merely requires accessing the s-list of the node (in left-to-right or right-to-left order) to obtain and begin traversing the appropriate spine deque. Since we do not have parent pointers to facilitate the backtracking part of the traversal, we maintain a *traversal stack* that records the state of the traversal; backtracking then involves popping the top record off the stack and continuing the traversal at the node (and spine) indicated.

A deque tree with n leaves has only $c_r n$ internal nodes, for some $0 \leq c_r \leq 1$. We call c_r the *reduction factor*. Since we now store the leaves outside of the spine deques, only these internal nodes occur in smaller deques. Furthermore, since we now also store the terminating nodes of the spines outside of the spine deques, each internal node occurs in at most one smaller deque.

Let $T(n)$ be the time required to traverse a deque tree T with n leaves. Such a traversal requires one traversal of the deque of each spine of T plus a constant amount c_w (the *work constant*) of work per node in the deque tree, thus yielding

$$T(n) \leq \sum_i T(x_i) + c_w(1 + c_r)n$$

where x_i is the size of the deque containing the non-leaf/non-terminating node(s) of the ith spine of T. It is easy to show that $T(n) \leq c_t n$ if $c_t \geq \frac{c_w(1+c_r)}{1-c_r}$. We call c_t the *traversal constant*. For the analysis to be valid, we must also have $0 \leq c_r < 1$.

To finish the analysis, we must show how we obtain a suitable reduction factor. Recall that we do not store the leaves or the terminating nodes of spines of the deque tree in the spine deques but instead maintain them separately. Thus a spine consists of a terminating node, a deque, and a leaf node. Also, we have that the degree of a non-root internal node is at least c_d for some $c_d \geq 2$; in Section 3.4.1 we used $c_d = 3$. These modifications yield $c_r \leq \sum_{i=1}^{\infty} 1/c_d^i = \frac{1}{c_d - 1}$. In particular, $c_d = 3$ allows us to achieve $c_t = 3c_w$ using the above traversal strategy. Note that we can achieve $c_t = c_d c_w$. In fact, it is straightforward to show that for any traversal method involving t traversals of each smaller deque, we can achieve $c_t = c_d c_w$ by setting $c_d = t + 2$. We use linear-time deque traversal in the next section.

3.4.3 Global Rebuilding

All that remains is to eliminate the subdeletions incurred by the pulls. We accomplish that in this section by replacing the pull operations with a new mechanism to ensure that the deque trees are balanced.

The new technique is *global rebuilding*. Global rebuilding and the related idea of *partial rebuilding*, elucidated by Overmars [Ove83], are tools initially developed and used to dynamize static data structures [And89, And91, CG86, GR93]. They have also been used to turn amortized bounds into worst-case bounds [AIKO90, CG93, DS87, GT86, HM81] and to improve the space requirements of a data structure [vKO93]. The basic idea of global rebuilding is to maintain with each data structure a secondary copy of the data structure that is being gradually "rebuilt." While updates are made to the primary data structure, reducing its balance, the secondary structure is being created a few steps at a time. By the time enough updates are made to the primary structure to violate its balance condition, the secondary structure is ready; i.e., the secondary structure is then a balanced (perhaps "perfectly" so) version of the primary structure, and it then replaces the primary structure. In his monograph, Overmars [Ove83] describes how to use global rebuilding to dynamize data structures that allow insertions and deletions. His methods are not immediately extensible to allow generalized catenable data structures to be globally rebuilt. We can apply his basic idea, however, to rebuild deque trees while they are being subjected to catenations and deletions.

Globally rebuilding a deque tree entails traversing it over a sequence of operations and constructing a new, flat deque tree containing the same leaves. With each deque tree T, we associate a secondary tree $sec(T)$ and a buffer $BUF(T)$, the latter implemented as a queue. We maintain an *ongoing traversal* of T; that is we perform some number of steps of the traversal with each operation on T and continue the traversal during the next such operation. We describe in detail how to do this in Section 3.4.4.

With every operation (link or delete) on T, we perform some number (k) of *rebuilding steps*, to be described below. Also, we add the operation performed on T to the back of $BUF(T)$. That is, we inject into $BUF(T)$ a record describing the operation (e.g., pop, eject, or catenate—with the latter also storing the respective deque being catenated). With each such injection we perform another k rebuilding steps.

A *rebuilding step* is as follows. If the traversal of T is not complete, then we perform one unit of work on that traversal. Each leaf encountered is copied to $sec(T)$, and $sec(T)$ is built as a flat tree using the spine decomposition structure. If the traversal of T is complete, then we remove (pop) the front element from $BUF(T)$. If it is a pop (eject), we perform a left (right) deletion on $sec(T)$. Similarly, if it is a push (inject), we add the new element as the new leftmost (rightmost) child of the root of $sec(T)$. Otherwise, the operation is a catenate of some tree S to T: the root of S becomes the leftmost (or rightmost) child of the root of T. In this case, we begin traversing S, adding its leaves to the left (right) of the leftmost (rightmost) leaf of $sec(T)$ if S was linked to the left (right) of T. Again, during this traversal, operations on T are queued onto $BUF(T)$. It is interesting to note that any accumulated rebuilding of S (and thus $sec(S)$ and $BUF(S)$) is ignored by our algorithm; as S is traversed and its leaves added to $sec(T)$, we merely continue buffering operations on T. The ability to do this and have $sec(T)$ rebuilt in time to take over from T is what allows our data structure to undergo global rebuilding; the requirement that the rebuilding of the linked data structure somehow be saved and used is what makes global rebuilding hard to apply in general to catenable data structures.

It is straightforward to show via induction that if we finish a traversal and $BUF(T)$ is empty, then $sec(T)$ is a *flat version of* T; that is, $sec(T)$ contains the same leaves in left-to-right order as T, and each leaf in $sec(T)$ is a child of the root of $sec(T)$. Furthermore, since $sec(T)$ is always flat, we can always make the appropriate modification to $sec(T)$ in $O(1)$ time. Once $sec(T)$ is a flat version of T, we replace T by $sec(T)$ and discard the old T. We begin creating a new $sec(T)$ with the first link to the new T.

We showed in Section 3.4.2 that it takes no more than $c_t n$ time to traverse a deque tree with n leaves, for some some traversal constant c_t. We assume that it takes unit time either to buffer an operation or to pop and perform an operation from $BUF(T)$. The proof that the above scheme maintains balanced deque trees is analogous to the analysis of the effect of pulling. Depth-increasing links provide a cushion of leaves that must be deleted before depth violations can occur, and non-depth-increasing links merely delay the violations. Meanwhile, the rebuilding progresses quickly enough to finish before any depth violation occurs. Whereas pulling is a local operation, however, affecting only the area around the root of a tree, global rebuilding is, as the name implies, global. The analogue to Lemma 3.3.1 therefore requires a more complicated proof.

Lemma 3.4.1 *If the number of rebuilding steps k per operation is at least $2c_t$ (for c_t the traversal constant), then a non-root node in a deque tree T never becomes a root node unless T becomes a singleton node.*

PROOF: New root nodes are created by the makelist operation and by linking two singleton trees together. Consider when a root node r of some tree becomes a non-root node due to a link. Assume that at some point in the future r becomes a root node of some tree X. Now consider the most recent tree S formed before X by linking a tree R containing r to another tree R'. Let r' be the root of R' (and thus of S), and note that $|R'| \geq |R|$. By this definition, r' never becomes a non-root node before X is created, but it may acquire new children as a result of future links. Denote by T the evolving tree rooted at r', initially S, as leaves are deleted from it and other trees linked to it.

Let a total of x leaves be added to T via links after the formation of S; let m be the number of operations that transform T from S to X. For r to become a root node, all the leaves of R' plus all these x leaves must first be deleted from T, resulting in $k(|R'| + x + m)$ rebuilding steps being applied to T. No more than $c_t(|S| + x) + m$ rebuilding steps are needed to make $sec(T)$ a flat version of T, however. Linking by size yields $|R'| \geq |S|/2$. Thus, if $k \geq 2c_t$, there are always at least as many rebuilding steps remaining as required to produce a flat version of T, and so $sec(T)$ takes over as a flat version of T before T becomes X. If r occurs in $sec(T)$, then it does so as a leaf. For r to become the root of $sec(T)$, therefore, all the other leaves of $sec(T)$ must first be deleted, leaving the single node r as tree X. $\square$

Theorem 3.4.2 *If the number of rebuilding steps k per operation is at least $\frac{2c_t}{2-c}$, for some constant $c \in (1,2)$ and the traversal constant c_t, then $|T| \geq c^{d_T}$.*

PROOF: Apply the same argument as in the proof of Theorem 3.3.2. Using Lemma 3.4.1 and adding the new terms to account for buffered updates and the traversal constant yields the desired bounds. $\square$

In summary, we have shown how to use global rebuilding in place of pulls to keep the deque trees balanced. This eliminates all of the subdeletions associated with the pull operations. Note that therefore it is no longer necessary to maintain a secondary list of non-leaf children with each node. The operations on s-lists simply become deque operations (without catenate). Thus, a deque of n elements is decomposed into a collection of only deques, and the deques that need to be catenated are all of size logarithmic in n.

3.4.4 Persistence Details

We now consider making the improved implementation of deque trees confluently persistent. As before, we use structural abstraction to refer to the spines and s-lists by their version numbers. The spines are now persistent records, each containing the version numbers of its leaf, middle confluently persistent deque, and terminating node. Again an internal node occurs in at most two spines, but now in at most one confluently persistent deque. Each such node contains the version number of its s-list, and again we store the actual version number in the copy of the node that occurs in the middle deque (to prevent cascading update sequences). If the node occurs as the terminating node of both spines containing it, we store the version number of its s-list in the terminating node of the left spine. Again, an inspection of how we access these nodes reveals that if we ever need to access an s-list via the terminating node of a right spine, we have immediate access to the corresponding left spine as well. Thus no further deque operations are necessary. As before, all operations occur in the immediate proximity of the left spines or right spines of the trees, so cascading update sequences do not occur. In particular, the operation of changing in place can be performed as described above without causing cascading updates.

To implement global rebuilding, we store with each persistent deque tree T the version number of $sec(T)$, a pointer to the place in $sec(T)$ where the last leaf was added, the version number of the tree being traversed and copied to $sec(T)$, a persistent stack to implement the traversal of that tree, and a persistent queue to represent $BUF(T)$. Stacks and queues are bounded indegree structures, so they can be made persistent by the methods of Driscoll et al. [DSST89]; each persistent stack or queue operation thus requires $O(1)$ amortized time and space. Recall that traversing a deque tree is a recursive operation: we are also traversing the deques representing the middle nodes of spines and so on. Each such traversal requires a persistent traversal stack, and the global state of the traversal of the top-level tree is recorded in the global persistent stack (containing records "pointing to" the persistent traversal stacks via version numbers) for that tree. While a traversal step might percolate all the way through the global stack, we know that the total number of traversal steps is linear in the size of the top-level deque tree. This is the key to global rebuilding. Each time we update T, we perform the required rebuilding steps and also update the traversal stacks, $sec(T)$, etc. All of these operations are done persistently: via structural abstraction, structures are represented by their persistent version numbers. Each persistent deque tree T, therefore, has an associated persistent secondary structure $sec(T)$. When $sec(T)$ becomes a flattened version of T, we replace the appropriate version (the one just created) of T by $sec(T)$. To do this, we simply return the version numbers describing $sec(T)$: those for its spines and the new $sec(T)$, $BUF(T)$, etc. that we begin creating.

As in our original implementation, structural decomposition yields an efficient data structure.

Theorem 3.4.3 *A pop or eject on a confluently persistent deque of size n can be performed in $O(\log^* n)$ amortized time and space; catenate and makelist each require $O(1)$ amortized time and space; first and last require $O(1)$ time.*

PROOF: By the above discussion and similar arguments as used in the proof of Theorem 3.3.4. □

Corollary 3.4.4 *Pop and eject require $O(\log^* m)$ amortized time and space, where m is the total number of deque operations performed so far.*

PROOF: Since $n = O(2^m)$, $\log^* n = O(\log^* m)$. □

Chuang and Goldberg [CG93] show how to implement fully persistent deques so that each fully persistent deque operation takes $O(1)$ worst-case time; Gajewska and Tarjan [GT86], in fact, anticipate this result. Chuang and Goldberg [CG93] break such deques into two fully persistent stacks, much like Gajewska and Tarjan [GT86] break heap-ordered deques into two heap-ordered stacks, and use global rebuilding to keep the stacks non-empty. We can use these fully persistent deques in our implementation, making all of our resource bounds worst-case.

Theorem 3.4.5 *A pop or eject on a confluently persistent deque can be performed in $O(\log^* m)$ worst-case time and space, where m is the number of deque operations preceding the pop or eject; catenate and makelist each require $O(1)$ worst-case time and space; first and last require $O(1)$ worst-case time.*

PROOF: As mentioned in Section 3.4.3, our modified representation uses only persistent deques, not s-lists, to implement the spine decomposition. Therefore, we can apply the Chuang and Goldberg [CG93] implementation of fully persistent deques to our data structure. □

In fact, using the Chuang and Goldberg [CG93] fully persistent deques, our solution to the confluently persistent deque problem can be implemented purely functionally, with no side effects.

We conclude by noting that the confluently persistent output-restricted deques of Driscoll, Sleator, and Tarjan [DST91] require $O(\log \log m)$ amortized time and space for catenate and $O(1)$ amortized time and space for the other operations. We can implement output-restricted deques using a *left-spine decomposition*, i.e., storing with each node a list of left spines terminating at its non-leftmost children. No right spines are stored. With this representation, no eject operations are needed; we only pop from the spine deques (in the case above where y is removed to preserve the degree invariant and z_1 and z_2 are leaves, thereby necessitating popping y from $deq(ls(T))$). In this way, we may recurse any number of levels in our structure and then substitute the output-restricted deques of Driscoll, Sleator, and Tarjan [DST91] for the middle deques in the spines. We can thus obtain an $O(\log^{(i)} m)$ amortized time and space bound for catenate, for any desired constant i, and $O(1)$ amortized time and space bounds for the other operations. This yields strictly improved bounds for the output-restricted case in addition to our solution to the general deque problem.

3.4.5 Application to Heap-Ordered Deques

Theorems 2.3.1 and 2.4.10 prove that the heap-ordered deque data structure of Chapter 2 requires $O(1)$ amortized time per operation. The worst-case time per operation on the data structure can easily be linear, however, or at best $O(\log n)$ (for a heap-ordered deque of n elements) if we link heap-ordered deques by size.

We remark here that the spine decomposition applies also to trees representing heap-ordered deques. In our structural abstraction, however, instead of representing the tree spines and associated lists by version numbers, we represent them by their minimum elements. Thus, we can obtain the same $O(\log^* n)$ time bound per operation for catenable heap-ordered deques. Using the Chuang and Goldberg [CG93] fully persistent deques removes the amortization from our confluently persistent deque resource bounds. Similarly, by using Gajewska and Tarjan's [GT86] worst-case solution to the heap-ordered deque problem, the spine-decomposition implementation of catenable heap-ordered deques yields worst-case bounds.

3.5 A Lower Bound for Pulls

Here we show that performing k pulls per deletion, for any constant k, can yield deque trees with logarithmic-length spines. The construction extends to the global rebuilding case. Therefore, further improvements to the resource bounds will require either a deeper analysis of the structure of the lower levels of the deque trees (the smaller trees representing spine deques) or a different data structure. The following construction is due to Sleator [Sle92].

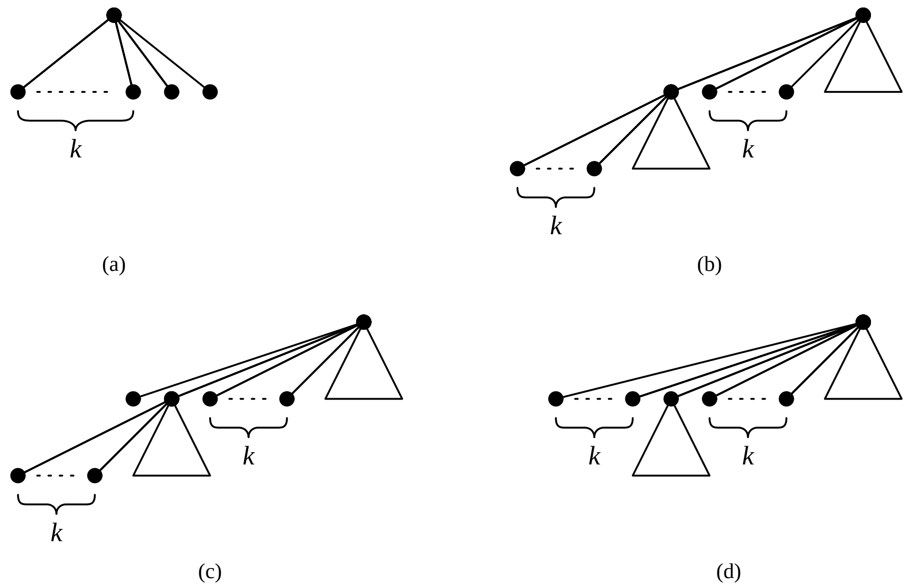


Figure 22: (a) A tree S_0. (b)–(d) Constructing a tree S_i by (b) linking two S_{i-1} trees, (c) adding a new leftmost child to the root, and (d) performing k pulls and a deletion. Triangles denote subtrees.

Theorem 3.5.1 *Performing k pulls per deletion, for $k = O(1)$, can yield an infinite set of deque trees such that any deque tree T in the set has a spine of length $\Omega(\log |T|)$.*

PROOF: Consider the tree S_0 consisting of a root node with $k+2$ children, as in Figure 22(a). We recursively construct a sequence of trees as follows: tree S_i (for $i > 0$) is formed by linking two S_{i-1} trees, adding a new leftmost child to the root of the resulting tree, and then performing k pulls and a deletion. See Figure 22(b)–(d). It is easy to prove by induction that:

1. $|S_i| = 2^i(k+2)$;
2. The length of the left spine terminating at the leftmost non-leaf child of the root of S_i is i.

$\square$

As an aside, we need two extra children of the root of S_0 so that the k pulls applied to the construction of S_1 (in Figure 22(d)) do not make the root of S_0 unary and thus remove it.

The above construction can be extended to handle the case in which k_1 *left pulls* (like the previous pull operations) and k_2 *right pulls* (the symmetric operations on the right side) are performed per deletion, for constants k_1 and k_2. For this case, S_0 is the tree containing a root with $k_1 + k_2 + 2$ children (k_1 to the left and k_2 to the right of the arbitrary subtree in the middle). The construction also shows the same lower bound for global rebuilding: start with a root with ck children for some appropriate constant c. Since each link-insert-delete sequence doubles the size of the tree while doing only a constant amount of rebuilding, the rebuilding never finishes, and the primary tree always has a logarithmic-length spine.

3.6 Summary

We have shown how to make deques confluently persistent so that pop and eject each require $O(\log^* m)$ amortized time and space, where m is the number of deque operations performed so far, and the other deque operations each take constant amortized time and space; we also make these resource bounds worst-case. Further, our solution allows a purely functional implementation, with no side effects. We utilize two types of data-structural bootstrapping—structural abstraction and structural decomposition—to achieve our results. In particular, structural abstraction effects persistence in our spine decomposition, and structural decomposition yields an efficient data structure. Most importantly, bootstrapping obviates the problems involved in the general case of making data structures confluently persistent: we do not need to determine reachability in a version DAG.

We discuss related open problems in Chapter 4.

Chapter 4

Conclusion

4.1 Results

We have used data-structural bootstrapping to implement efficient catenable heap-ordered deques and confluently persistent deques.

Using structural abstraction, we implement catenable heap-ordered deques using only non-catenable heap-ordered deques as basic building blocks. Our data structure performs all heap-ordered deque operations in $O(1)$ amortized time (worst-case for findmin). This result derives from the linear time bound we prove on deque-ordered, spine-only path compression with the Rising Roots Condition. Not only does this path compression result unify previous work on special cases of the problem [LN88a, LN88b, LN89, Luc90], but it also provides what we believe is the first practical application of this type of result. Further, we can adapt our confluently persistent deques to implement catenable heap-ordered deques such that all operations on an n-element heap-ordered deque take $O(\log^* n)$ worst-case time, except findmin, which takes $O(1)$ worst-case time. The catenable heap-ordered deque data structure reported in Chapter 2 is easy to implement and unifies previous data structures (such as heap-ordered queues and non-catenable heap-ordered deques) that have many applications.

Using structural abstraction and structural decomposition, we implement confluently persistent deques using only fully persistent deques as basic building blocks. Our data structure performs all deque operations except deletion in $O(1)$ amortized time and space; pop and eject take $O(\log^* m)$ amortized time and space, where m is the number of deque operations performed so far; we can make these resource bounds worst-case, as well. Further, our solution allows a purely functional implementation, with no side effects. We can modify our data structure to implement confluently persistent output-restricted deques so that all operations take $O(1)$ amortized time and space except for catenate, which takes $O(\log^{(i)} m)$ amortized time and space for any desired constant i. Thus, our resource bounds for the output-restricted case strictly improve upon work of Driscoll, Sleator, and Tarjan [DST91]; we use their data structure, however, in this solution. We also give a solution to the more general problem of deques, removing the output restriction. We discuss the practicality of our data structure in Section 4.2.1.

4.2 Data-Structural Bootstrapping

We have examined *data-structural bootstrapping* as a tool to design data structures recursively. Our results provide efficient solutions to two related problems: in each case, a catenable data structure was made non-catenable by the addition of some secondary property.

We define *structural abstraction* as the abstraction of a data structure augmented with a secondary property, in the following way: we represent an instance of the data structure in terms of the secondary property; that instance, the key of which is the associated secondary property, becomes a basic element in a higher-level data structure of the same type. In each problem addressed, we use structural abstraction to obviate the problems encountered in extending methods that implement the non-catenable augmented data structure. In the catenable heap-ordered deque case, abstracting heap-ordered deques by representing them by their minimum elements avoids having to resolve minima lists that point in opposite directions. In the confluently persistent deque case, abstracting fully persistent deques by representing them by their persistent version numbers avoids having to determine reachability in version DAGs. Structural abstraction thus effects catenation in heap-ordered deques and persistence in the spine decomposition of deque trees.

We define *structural decomposition* as the recursive, homogeneous decomposition of a data structure into smaller pieces. In particular, using spine decomposition, we implement a deque of n elements as a collection of deques of $O(\log n)$ elements each. Each smaller structure undergoes the same recursive decomposition, until we obtain a data structure small enough to manipulate "explicitly" by naïve methods: in the confluently persistent deque case, we use list copying to implement small confluently persistent deques. Structural decomposition makes our implementation of confluently persistent deques efficient.

4.2.1 On General Data-Structure Design Techniques

Finally, we reconsider the anticipated benefits of using a general data-structure design technique. First, in developing data-structural bootstrapping, we isolated the problem common to the implementation of catenable heap-ordered deques and confluently persistent deques. In each case, the solution to the non-catenable data structure complicated deque catenation: heap-ordered deques use minima lists that point in opposite directions; fully persistent deques rely on determining reachability in version trees, and this reachability problem is harder to solve in version DAGs. Recognizing the reachability problem, in particular, might direct efforts towards devising general techniques that make data structures confluently persistent.

Second, by using one general technique, data-structural bootstrapping, we solved two data-structure problems. This leads directly to the third benefit: the practicality of the solutions to both problems relies on the practicality of the implementations of the simpler, non-catenable data structures. The heap-ordered deques of Gajewska and Tarjan [GT86] are easy to implement efficiently. Applying structural abstraction to these heap-ordered deques yields catenable heap-ordered deques that are also easy to implement efficiently, so we can make them useful in practice. The technicalities involved in proving the path compression result merely prove that the data structure has an efficient time bound; the data structure itself is not complicated.

The practicality of our confluently persistent deques, likewise, relies upon the efficient implementation of fully persistent deques. Producing such an implementation of fully persistent deques is thus essential to the practical viability of our confluently persistent deques. Using the methods of Driscoll et al. [DSST89] to implement fully persistent deques requires a practical implementation of the algorithm of Dietz and Sleator [DS87] that solves the list-order problem; we believe such an implementation is possible. An alternative would be to implement fully persistent deques via the method of Chuang and Goldberg [CG93] and Gajewska and Tarjan [GT86]; this also makes all of our resource bounds worst-case. Given the applications of confluently persistent deques described in Chapter 1, it would be worthwhile to implement confluently persistent deques as described in Chapter 3 based upon both of the above methods and compare the results to the list- and path-copying approaches.

4.3 Open Problems

In addition to exploring the practicality of our confluently persistent deque data structure, our work leaves several other interesting problems unresolved.

4.3.1 Catenable Heap-Ordered Deques

First, our heap-ordered deques take $O(1)$ amortized time (or $O(\log^* n)$ worst-case) time per operation. Kosaraju [Kos92] has designed a more complicated data structure that achieves constant worst-case time per operation, for an arbitrary number of heap-ordered deques. His solution uses structural abstraction; it is not immediately extensible to the confluently persistent deque problem, however, and unifying the two approaches merits more work.

The Hart-Sharir [HS86] open problem of postorder path compression without the Rising Roots Condition (see Definition 2.4.1) remains tantalizingly open. We note that our lower bound construction in Section 2.4.2 does not involve any order on the compressions themselves. Further, the construction produces an unbalanced initial tree. Thus, one can still formulate intermediate open problems incorporating different notions of order as well as the absence or presence of Rising Roots and tree balance that might be easier to solve than the Hart-Sharir problem.

4.3.2 Confluently Persistent Deques

Another open problem is implementing confluently persistent deques with better resource bounds. There is currently no reason to believe that constant time and space per operation is not possible; if it is not, however, perhaps an $O(\alpha(m))$ solution is attainable, where α is a functional inverse of the Ackermann function [Tar83]. Section 3.5 suggests that different analyses or data structures than the ones presented here are necessary for such improvements. We suggest investigating Kosaraju's [Kos92] worst-case solution to the catenable heap-ordered deque problem as a first step.

Providing general techniques for making various data structures confluently persistent remains an open problem. Of course, defining the notion of confluent persistence is the first step, as it is not well-defined for all data structures. Some linking operation is necessary for it to be applicable; it might make no sense, for instance, to talk about confluently persistent arrays. ("Linking" operations that could make arrays confluently persistent include such things as summing the respective entries of each array position and performing convolutions.) It is interesting to note that the version-tree concept of time defines full persistence for all data structures. The analogous statement concerning version DAGs and confluent persistence is not true: confluent persistence does not apply to all data structures.

As mentioned in Chapter 1, some purely functional languages effectively implement all data structures persistently, regardless of the actual application. Using techniques that make data structures persistent to facilitate the implementation of such languages is an important open area for future research.

4.3.3 Data-Structural Bootstrapping

Data-structural bootstrapping holds promise as a general tool for designing data structures. Given two data structures that allow catenation and implementations of corresponding augmented data structures without catenation, we have used data-structural bootstrapping to restore catenation to the augmented data structures. Driscoll, Sleator, and Tarjan [DST91] use structural abstraction to effect persistence in catenable lists; we employ it to effect catenation in heap-ordered deques and persistence in catenable deques. We also use structural decomposition to make our confluently persistent deques efficient. Work on formalizing these methods and finding further types and applications of bootstrapping seems worthwhile.

Bibliography

[AAA+91] P. K. Agarwal, A. Aggarwal, B. Aronov, S. R. Kosaraju, B. Schieber, and S. Suri. Computing external farthest neighbors for a simple polygon. *Discrete Applied Mathematics*, 31(2):97–111, 1991.

[AIKO90] A. Andersson, C. Icking, R. Klein, and T. Ottmann. Binary search trees of almost optimal height. *Acta Informatica*, 28(2):165–78, 1990.

[And89] A. Andersson. Improving partial rebuilding by using simple balance criteria. In *Proc. 1st Workshop on Algorithms and Data Structures*, volume 382 of *Lecture Notes in Computer Science*, pages 393–402. Springer-Verlag, 1989.

[And91] A. Andersson. Maintaining α-balanced trees by partial rebuilding. *International Journal of Computer Mathematics*, 38(1-2):37–48, 1991.

[App92] A. W. Appel. *Compiling with Continuations.* Cambridge University Press, Cambridge, 1992.

[AS89] C. R. Aragon and R. G. Seidel. Randomized search trees. In *Proc. 30th IEEE Symp. on Foundations of Computer Science*, pages 540–5, 1989.

[BS80] J. L. Bentley and J. B. Saxe. Decomposable searching problems I. Static-to-dynamic transformation. *Journal of Algorithms*, 1:301–58, 1980.

[BST92] A. L. Buchsbaum, R. Sundar, and R. E. Tarjan. Data structural bootstrapping, linear path compression, and catenable heap ordered double ended queues. In *Proc. 33rd IEEE Symp. on Foundations of Computer Science*, pages 40–9, 1992.

[BT93] A. L. Buchsbaum and R. E. Tarjan. Confluently persistent deques via data structural bootstrapping. In *Proc. 4th ACM-SIAM Symp. on Discrete Algorithms*, pages 155–64, 1993.

[BW90] H. Booth and J. R. Westbrook. Linear algorithms for analysis of minimum spanning and shortest path trees in planar graphs. Technical Report YALEU/DCS/TR-763, Yale University Dept. of Computer Science, February 1990. To appear in *Algorithmica*.

[CG86] B. Chazelle and L. J. Guibas. Fractional cascading: I. A data structure technique. *Algorithmica*, 1(2):133–62, 1986.

[CG93] T.-R. Chuang and B. Goldberg. Real-time deques, multihead Turing machines, and purely functional programming. In *Proc. 6th ACM Conf. on Functional Programming Languages and Computer Architecture*, pages 289–98, 1993.

[Cha85] B. Chazelle. How to search in history. *Information and Control*, 77(1-3):77–99, 1985.

[Col86] R. Cole. Searching and storing similar lists. *Journal of Algorithms*, 7(2):202–20, 1986.

[CS84] R. Cole and A. Siegel. River routing every which way, but loose. In *Proc. 25th IEEE Symp. on Foundations of Computer Science*, pages 65–73, 1984.

[DF84] G. Diehr and B. Faaland. Optimal pagination of B-trees with variable-length items. *Communications of the ACM*, 27(3):241–7, 1984.

[Die82] P. F. Dietz. Maintaining order in a linked list. In *Proc. 14th ACM Symp. on Theory of Computing*, pages 122–7, May 1982.

[Die89] P. F. Dietz. Fully persistent arrays (extended abstract). In *Proc. 1st Workshop on Algorithms and Data Structures*, number 382 in Lecture Notes in Computer Science, pages 67–74. Springer-Verlag, August 1989. Submitted to *J. Alg.*

[Die91] P. F. Dietz. Finding level-ancestors in dynamic trees. In *Proc. 2nd Workshop on Algorithms and Data Structures*, volume 519 of *Lecture Notes in Computer Science*, pages 32–40. Springer-Verlag, 1991.

[Dij72] E. W. Dijkstra. Notes on structured programming. In C. A. R. Hoare, editor, *Structured Programming*, pages 1–82. Academic Press, London, 1972.

[DM85] D. P. Dobkin and J. I. Munro. Efficient uses of the past. *Journal of Algorithms*, 6(4):455–65, 1985.

[DR91] P. F. Dietz and R. Raman. Persistence, amortization and randomization. In *Proc. 2nd ACM-SIAM Symp. on Discrete Algorithms*, pages 78–88, 1991.

[DS87] P. F. Dietz and D. D. Sleator. Two algorithms for maintaining order in a list. In *Proc. 19th ACM Symp. on Theory of Computing*, pages 365–72, May 1987.

[DSST89] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, February 1989.

[DST91] J. R. Driscoll, D. D. K. Sleator, and R. E. Tarjan. Fully persistent lists with catenation. In *Proc. 2nd ACM-SIAM Symp. on Discrete Algorithms*, pages 89–99, 1991. Appears as *J. ACM* **41**(5):943-59, 1994.

[EGIN92] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig. Sparsification—A technique for speeding up dynamic graph algorithms. In *Proc. 33rd IEEE Symp. on Foundations of Computer Science*, pages 60–9, 1992.

[Fre91] G. N. Frederickson. Planar graph decomposition and all pairs shortest paths. *Journal of the ACM*, 38(1):162–204, 1991.

[FT87] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[FVV78] J. Françon, G. Viennot, and J. Vuillemin. Description and analysis of an efficient priority queue representation. In *Proc. 19th IEEE Symp. on Foundations of Computer Science*, pages 1–7, 1978.

[GR93] I. Galperin and R. L. Rivest. Scapegoat trees. In *Proc. 4th ACM-SIAM Symposium on Discrete Algorithms*, pages 165–74, 1993.

[GS78] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proc. 19th IEEE Symp. on Foundations of Computer Science*, pages 8–21, 1978.

[GT86] H. Gajewska and R. E. Tarjan. Deques with heap order. *Information Processing Letters*, 22(4):197–200, April 1986.

[HL87] D. S. Hirschberg and L. L. Larmore. New applications of failure functions. *Journal of the ACM*, 34(3):616–25, 1987.

[HM81] R. Hood and R. Melville. Real-time queue operations in pure LISP. *Information Processing Letters*, 13(2):50–4, 1981.

[Hoa72] C. A. R. Hoare. Notes on data structuring. In C. A. R. Hoare, editor, *Structured Programming*, pages 83–174. Academic Press, London, 1972.

[HS86] S. Hart and M. Sharir. Nonlinearity of Davenport-Schinzel sequences and of generalized path compression schemes. *Combinatorica*, 6(2):151–77, 1986.

[Knu73a] D. E. Knuth. *The Art of Computer Programming*, volume 1: *Fundamental Algorithms*. Addison-Wesley, Reading, MA, second edition, 1973.

[Knu73b] D. E. Knuth. *The Art of Computer Programming*, volume 3: *Sorting and Searching*. Addison-Wesley, Reading, MA, 1973.

[Kos79] S. R. Kosaraju. Real-time simulation of concatenable double-ended queues by double-ended queues. In *Proc. 11th ACM Symp. on Theory of Computing*, pages 346–51, 1979.

[Kos92] S. R. Kosaraju. Personal communication. 1992.

[LH85] L. L. Larmore and D. S. Hirschberg. Efficient optimal pagination of scrolls. *Communications of the ACM*, 28(8):854–6, 1985.

[LN88a] M. Loebl and J. Nešetřil. Linearity and unprovability of set union problem strategies. In *Proc. 20th ACM Symp. on Theory of Computing*, pages 360–6, 1988.

[LN88b] M. Loebl and J. Nešetřil. Postorder hierarchy for path compressions and set union. In *Proc. 5th Int'l. Mtg. of Young Computer Scientists*, volume 381 of *Lecture Notes in Computer Science*, pages 146–51. Springer-Verlag, 1988.

[LN89] M. Loebl and J. Nešetřil. Linearity and unprovability of set union problem strategies I. Linearity of on line postorder. Technical Report 89-04, U. Chicago Dept. of Computer Science, April 1989.

[Luc90] J. M. Lucas. Postorder disjoint set union is linear. *SIAM Journal on Computing*, 19(5):868–82, October 1990.

[McC77] E. M. McCreight. Pagination of B^*-trees with variable-length records. *Communications of the ACM*, 20(9):670–4, 1977.

[Mye82] E. W. Myers. AVL dags. Technical Report 82-9, U. Arizona Dept. of Computer Science, Tucson, 1982.

[Mye83] E. W. Myers. An applicative random-access stack. *Information Processing Letters*, 17(5):241–8, 1983.

[Mye84] E. W. Myers. Efficient applicative data types. In *Proc. 11th ACM Symp. on Principles of Programming Languages*, pages 66–75, 1984.

[Ove81a] M. H. Overmars. Searching in the past, I. Technical Report RUU-CS-81-7, U. Utrecht Dept. of Computer Science, 1981.

[Ove81b] M. H. Overmars. Searching in the past, II: General transforms. Technical Report RUU-CS-81-9, U. Utrecht Dept. of Computer Science, 1981.

[Ove83] M. H. Overmars. *The Design of Dynamic Data Structures*, volume 156 of *Lecture Notes in Computer Science*. Springer-Verlag, 1983.

[RTD83] T. Reps, T. Teitelbaum, and A. Demers. Incremental context-dependent analysis for language-based editors. *ACM Transactions on Programming Languages and Systems*, 5(3):449–77, 1983.

[Sar86] N. Sarnak. *Persistent Data Structures*. PhD thesis, Department of Computer Science, New York University, 1986.

[Set89] R. Sethi. *Programming Languages Concepts and Constructs*. Addison-Wesley Series in Computer Science. Addison-Wesley, Reading, MA, 1989.

[Sle92] D. D. Sleator. Personal communication. 1992.

[ST83] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–91, 1983.

[ST86] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29(7):669–79, 1986.

[Sun89] R. Sundar. Worst-case data structures for the priority queue with attrition. *Information Processing Letters*, 31(2):69–75, 1989.

[Tar83] R. E. Tarjan. *Data Structures and Network Algorithms*. CBMS-NSF Regional Conference Series in Applied Mathematics. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[Tar85] R. E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–18, 1985.

[Tsa84] A. K. Tsakalidis. Maintaining order in a generalized linked list. *Acta Informatica*, 21(1):101–12, 1984.

[TvL84] R. E. Tarjan and J. van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM*, 31(2):245–81, 1984.

[vKO93] M. J. van Kreveld and M. H. Overmars. Union-copy structures and dynamic segment trees. *Journal of the ACM*, 30(3):635–52, 1993.

[Vui78] J. Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–15, 1978.

[Vui80] J. Vuillemin. A unifying look at data structures. *Communications of the ACM*, 23(4):229–39, 1980.

[Wil83] D. E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Information Processing Letters*, 17:81–4, August 1983.

[Wil84] D. E. Willard. New trie data structures which support very fast search operations. *Journal of Computer and System Sciences*, 28(3):379–94, 1984.