Data Structural Bootstrapping, Linear Path Compression, and Catenable Heap Ordered Double Ended Queues[1]

Adam L. Buchsbaum[2]
Rajamani Sundar[3]
Robert E. Tarjan[4]

Research Report CS-TR-381-92
September 1992

Abstract

A *deque with heap order* is a linear list of elements with real-valued keys which allows insertions and deletions of elements at both ends of the list. It also allows the *findmin* (equivalently *findmax*) operation, which returns the element of least (greatest) key, but it does not allow a general *deletemin* (*deletemax*) operation. Such a data structure is also called a *mindeque* (*maxdeque*). Whereas implementing mindeques in constant time per operation is a solved problem, catenating mindeques in sublogarithmic time has until now remained open.

This paper provides an efficient implementation of catenable mindeques, yielding constant amortized time per operation. The important algorithmic technique employed is an idea which is best described as *data structural bootstrapping*: We abstract mindeques so that their elements represent other mindeques, effecting catenation while preserving heap order. The efficiency of the resulting data structure depends upon the complexity of a special case of path compression which we prove requires linear time.

[1]An extended abstract of this paper will appear in the 33rd IEEE Foundations of Computer Science, 25-27 October, 1992.

[2]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.

[3]Also affiliated with DIMACS, Rutgers University, Piscataway, NJ 08855. Supported by DIMACS under NSF-STC88-09648.

[4]Also affiliated with NEC Research Institute, 4 Independence Way, Princeton, NJ 08540. Research at Princeton University partially supported by the National Science Foundation, Grant No. CCR-8920505, the Office of Naval Research, Contract No. N00014-91-J-1463, and by DIMACS under NSF-STC88-09648.

1 Introduction

A *deque with heap order* is a linear list of elements with real-valued keys which allows insertions and deletions of elements at both ends of the list. It also allows the *findmin* (equivalently *findmax*) operation, which returns the element of least (greatest) key, but it does not allow a general *deletemin* (*deletemax*) operation. Such a data structure is also called a *mindeque* (*maxdeque*). The restricted access and lack of deletemin distinguish mindeques from general heaps and allow faster operation times than do heaps. Gajewska and Tarjan [GT86] show how to implement mindeques with constant time (amortized or worst-case) per operation; they leave open the problem of how to catenate mindeques.

This paper provides an efficient implementation of catenable mindeques. The important algorithmic technique employed is an idea of Driscoll et al [DST91], which is best described as *data structural bootstrapping*: The mindeques of Gajewska and Tarjan are abstracted so that their elements represent other mindeques, effecting catenation while preserving heap order. 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 bound on *deque ordered spine-only path compression*, a case of order preserving path compression employed by our data structure.

Our result is important in the following respects. It shows how the bootstrapping technique of Driscoll et al, originally developed to create confluently persistent catenable lists, is in fact a generally 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.

Sections 2 and 3 of this paper describe our data structure for catenable mindeques, using the bootstrapping technique of Driscoll et al. Sections 4 and 5 define and analyze deque ordered spine-only path compression, proving the linearity of this special case of path compression. These latter sections are the more technically difficult of the paper. Finally, we conclude and offer some open problems in Section 6.

2 Lists, Heap Order, and Catenation

Consider the following operations, to be performed on a linear list d of elements:

$push(x, d)$	Insert x as the new first element of d; the previous ith element becomes the $(i+1)$st.
$pop(d)$	Remove and return the first element of d (or $\emptyset$ if d is empty); the previous ith element, for $i \geq 2$, becomes the $(i-1)$st.
$inject(x, d)$	Insert x as the new last element of d.
$eject(d)$	Remove and return the last element of d (or $\emptyset$ if d is empty).

We assume the existence of a *makelist* operation that returns an initially empty list. If only push and pop (or inject and eject) are allowed, d is a *stack* (formally a *list of type stack*). If only push and eject (or inject and pop) are allowed, d is a *queue*. If all the operations are allowed, d is a *double ended queue*, or *deque*. If both insertion operations but only one of the deletion operations are allowed, d is an *output restricted deque*. 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 each element in d has a real-valued key, we may also want to consider the following operation:

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

Findmin does not modify the list itself. If d is a deque, then d together with findmin is a *heap ordered deque* or *mindeque*. Analogous data structures are obtained by adding findmin to stacks, queues, and output-restricted queues. A related data structure is the *priority queue with attrition* [Sun89]. We can also consider the *findmax* operation but will restrict ourselves without loss of generality to findmin for the remainder of this paper.

Finally, if d_1 and d_2 are lists of the same type and of size (number of elements) s_1 and s_2 respectively, then we define the following operation:

catenate(d_1, d_2) Add the elements of d_2 to the back of d_1; i.e., let the (s_1+i)th element of d_1 be the ith element of d_2 for $1 \leq i \leq s_2$. The first s_1 elements of d_1 are unchanged. This operation destroys d_2.

Catenating lists in $O(1)$ time is straightforward; catenating heap ordered lists is more problematic. This paper demonstrates how to implement *catenable heap ordered deques* (or *catenable mindeques*) efficiently. In particular, our data structure performs n insertions (pushes and injects), m deletions (pops and ejects), and q catenations, all intermixed on $q+1$ catenable mindeques, in total time $O(n+m+q)$, such that findmin always takes $O(1)$ worst case time. Further, if q is fixed, each operation requires $O(1)$ worst case time.

We remark that an equivalent formulation of the problem is to have the makelist operation take an element as an argument and return a list of one element. Then, push and inject can be treated as special cases of catenation. We choose the former suite of operations in order to distinguish catenation as a special operation which complicates the implementation of mindeques.

2.1 Related Work and Applications

Queues with heap order (or *minques*) are useful in pagination [DF84, HL87, LH85, McC77] and VLSI river routing [CS84]. Booth and Westbrook [BW90] use catenable minques in the sensitivity analysis of minimum spanning trees, shortest path trees, and mininum cost network flow on planar graphs. Larmore and Hirschberg [LH85] and Cole and Siegal [CS84] independently showed how to implement minques in $O(1)$ amortized time per operation [Tar85]. Gajewska and Tarjan [GT86] modified their techniques to produce mindeques with $O(1)$ time per operation; they give both amortized and worst case solutions. Applications of mindeques include computing all pairs shortest path information [Fre91] and external farthest neighbors for simple polygons [AAA$^+$91]. Whereas a simple extension of the previous minque techniques yields efficiently catenable minques (again $O(1)$ amortized time per operation), Gajewska and Tarjan left as an open problem how to construct efficiently catenable mindeques. We address this problem.

2.2 Reviewing Heap Ordered Queues and Deques

For completeness, we begin by reviewing how to implement minques and mindeques. 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 minque 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 1a.

To pop an element from d, if the element is minimum, the new minimum element of d becomes the element's successor 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 element of d' is of lessor 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 technique of [CS84, LH85], yielding $O(1)$ amortized time bounds for all the queue operations. We note the technique also extends to implement heap ordered stacks, which will be used to create mindeques, 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.

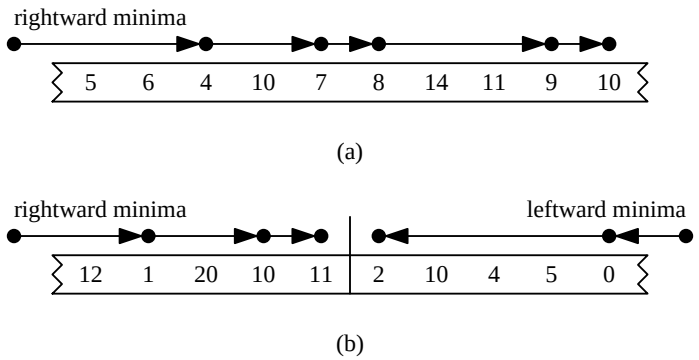

Figure 1: (a) A minque with minimum key 4. (b) A mindeque with minimum key 0; the center line is the break point between the two stacks.

Gajewska and Tarjan [GT86] implement mindeques by representing them as two heap ordered stacks, as in Figure 1b. When one stack is emptied, the mindeque is rebuilt such 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 minques.

3 Data Structural Bootstrapping and Catenable Mindeques

We employ a technique of Driscoll et al [DST91] to bootstrap the mindeques and allow their efficient catenation. We wish to implement a catenable mindeque 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 mindeque. Each element of D consists of a basic element of d combined with a pointer. For an element x of D, let e be x's basic element and p x's pointer. 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 mindeque to which p points. This other mindeque 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 in all the mindeques which D subtends. The d-elements of D occur in a natural *left-to-right* order which can be enumerated 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 mindeque 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 Figures 2a and 2b. 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 among their children. The internal nodes are precisely those elements of D and its subtended mindeques which are not d-elements. 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 et al [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 node's position in a linear list, the data structure supports heap ordered list access operations. The idea of bootstrapping mindeques to implement catenable mindeques in the above recursive fashion generalizes the technique of Kosaraju [Kos79], by which he designs catenable deques (not heap ordered) by decomposing the deques into contiguous pieces and storing

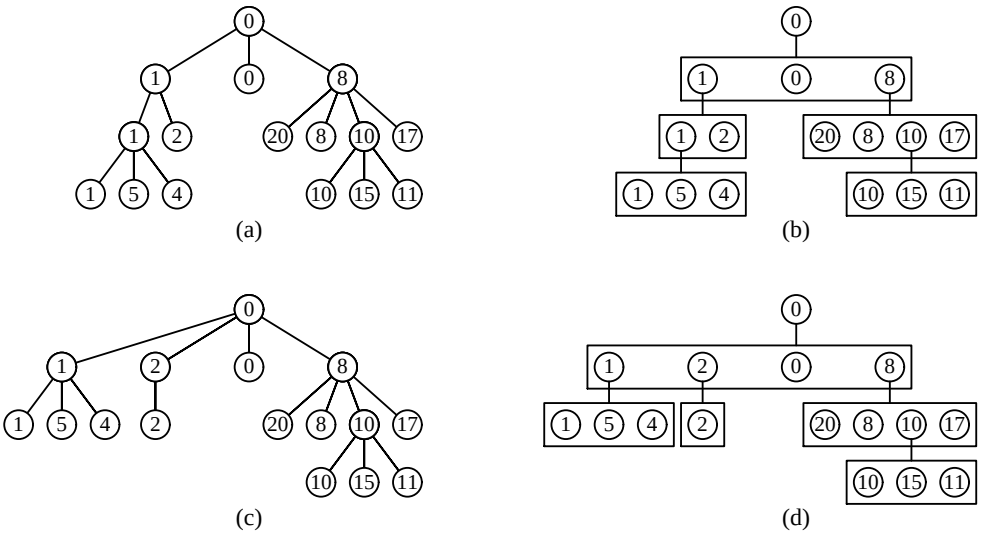

Figure 2: (a) A heap ordered tree. (b) The corresponding catenable mindeque; boxes surround mindeques, the minimum values of which are elements in the parent mindeque. (c) The tree of (a) after a left pull. (d) The corresponding catenable mindeque.

those pieces in a stack. His data structure can be extended to maintain heap order, but it only accommodates a fixed number of deques. Driscoll et al [DST91] bootstrap fully persistent lists to implement confluently persistent catenable lists.

3.1 The *Pull* Operation

We now define a *left pull* operation on the heap ordered tree T similar to the *pull* operation of [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' the new leftmost child of T; if x is now a leaf, it is deleted. See Figure 2c for an example. The heap invariant of the tree is maintained by making each internal node contain the lowest key 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 is now extended to the data structure D as follows: If the first element x of D is a d-element (has a $\emptyset$ pointer), a left pull on D does nothing. Otherwise, pop D, returning x. Denote by $d(x)$ the mindeque to which x's pointer points. Pop $d(x)$, returning x'. If $d(x)$ is still non-empty, push x back onto D; x's pointer still points to $d(x)$, but popping $d(x)$ may change x's basic element (the minimum basic element of $d(x)$). Then push x' onto D. A right pull on D is defined symmetrically. Since we use only mindeque operations to implement pulls on D, the minimum values of all the mindeques, in particular that of D, are maintained correctly. The popping and repushing of x is 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 2d 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 mindeque operations on d as follows. We assume that the makelist operation returns an empty mindeque D.

push(x, d)	Create element $\tilde{x}$ of basic element x and pointer $\emptyset$. Push $\tilde{x}$ onto D.
pop(d)	Repeatedly left pull D until its head element is a d-element. Then pop D; this returns an element $\tilde{x}$. Return the basic element x of $\tilde{x}$.
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; equivalently, we could inject $\tilde{D}_2$ into D_1.

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

3.2 Time Analysis

We now state:

Theorem 3.1 *An intermixed set of n insertions, m deletions, and q catenations on $q+1$ catenable mindeques takes total time $O(n+m+q)$ such that findmin always takes $O(1)$ worst case time.*

PROOF: We first note that push (inject) requires a constant amount of time to prepare the element (from the given basic element and a $\emptyset$ pointer) in addition to one "real" mindeque push (inject). If we use the Gajewska-Tarjan mindeques [GT86] (in this and what follows, either the amortized or worst case solution they offer suffices), each real mindeque operation takes $O(1)$ time. Similarly, each catenation also entails one real push or inject in addition to some constant preprocessing time. Each deletion entails some number of pulls, each of which requires a constant number of real mindeque operations, plus one final real 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 which 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 4; briefly, each compression involves a path of only leftmost children (or only rightmost children) of their parents. Theorem 5.4 will prove the linearity of these path compressions. It is unnecessary to maintain the set trees in a balanced fashion (e.g., doing unions by size or rank).

Findmin is performed by one real mindeque findmin, which takes $O(1)$ worst case time. □

Note that if $q = O(1)$, that is if there is only a constant number of mindeques 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 in [GT86], which they solved by using the techniques of Kosaraju [Kos79].

Corollary 3.2 *An intermixed sequence of insertions, catenations, deletions, and findmins can be performed on a fixed number of catenable mindeques in $O(1)$ worst case time per operation.*

PROOF: If $q = O(1)$ each sequence of pulls is of constant length. □

4 Path Compression

In this section, we introduce some definitions necessary to the analysis of our special case of path compression. Initially, we have a tree T with 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 from* x_0 on a tree T is a sequence of nodes $C = (x_0, \ldots, x_l)$ such that $l > 0$ and $p(x_i) = x_{i+1}$ in T for $0 \leq i < l$. Its effect is to update the parents of the nodes along the path, making them all point to x_l, the *root of the compression*:

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

We say that the compression C *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 $C = (x_0)$ and x_0 is a *leaf node*, i.e. a node with no children, then C is a *leaf deletion*; the effect of C is to remove 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 $\sum_{i=1}^{t} |C_i|$.

We now define the *Rising Roots Condition* [Luc90], which will link the notion of path compressions above with the well known union and find operations used in the disjoint set union problem (see, e.g., [TvL84]).

Definition 4.1 (Rising Roots Condition) *A sequence of path compressions* $(C_1, \ldots, C_m)$ *satisfies the* Rising Roots Condition *if and only if for every node* x*, if* x *appears as a non-root node in any compression* C_i*, then for every* $j > i$*,* x *appears as a non-root node in* C_j *if* C_j *is a compression from* y *and* y *is a descendant of* x *in* T_{j-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 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 [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 $x \lessdot y$ if $c_x(z)$ is to the left of $c_y(z)$.

Definition 4.3 *A path compression on a tree* T *which yields a tree* T' *is* order preserving *if* $x \lessdot y$ *in* $T \Longrightarrow x \lessdot y$ *in* T'*. A leaf deletion is always taken to be* order preserving*. A sequence* $(C_1, \ldots, C_m)$ *of path compressions on* T_0 *is* order preserving *if* C_i *is order preserving for* $1 \leq i \leq m$*.*

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 compression), Definition 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 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

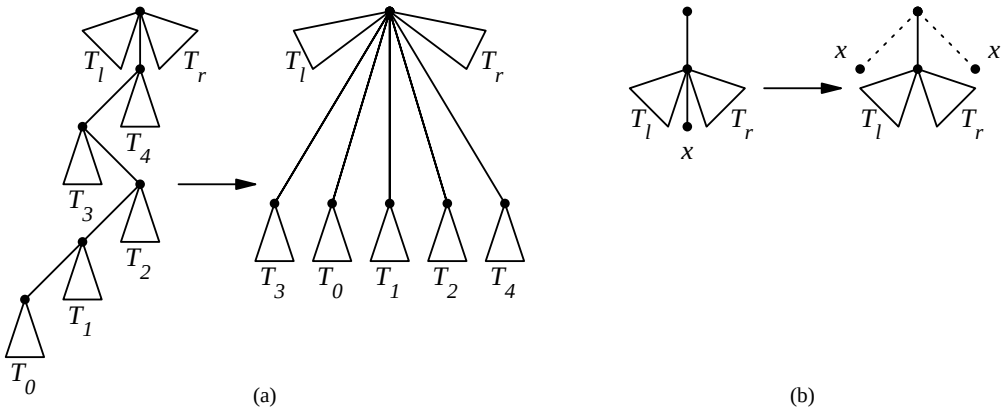

Figure 3: (a) An order preserving path compression; triangles denote subtrees. (b) A path compression which cannot be order preserving; x must become either left of T_l or right of T_r.

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 3a.

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 $L \lessdot x_j \lessdot R$. However, if x_j becomes a left sibling of x_{j+1} during the compression, then after the compression we have $x_j \lessdot L$; similarly, if x_j becomes a right sibling of x_{j+1}, then after the compression we have $R \lessdot x_j$. See Figure 3b. □

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. In this paper we consider a restricted case of order preserving path compression:

Definition 4.5 *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 $0 \leq i < l-1$ 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 initially postordered tree, each node, in postorder, is the subject of one left-spine path compression and is then deleted. Loebl and Nešetřil [LN88a, LN88b, LN89] prove the linearity of a more general problem: that of a sequence of left-spine path compressions satisfying the Rising Roots Condition; they refer to this as a *local postorder*. These works derive from an open problem of Hart and Sharir [HS86]: What is the complexity of a sequence of path compressions done in postorder? I.e., the Hart-Sharir open problem does not require 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 the 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 which 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 4.6 *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 $x \lessdot y$ the following is true: If the first path compression C' that orders v and y results in $y \lessdot v$, 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 $v \lessdot x$, then all path compressions from nodes in $T_y^{(v)}$ precede C''.*

It is important to note that Definition 4.6 refers to nodes in a tree, e.g. $T_x^{(v)}$, with respect to $T^{(v)}$. I.e., the actual compressions may take place from nodes which at that time are no longer descendants of x (or v).

In the next section we prove our main result, namely that a deque ordered sequence of spine-only path compressions that satisfies the Rising Roots Condition requires at most linear time.

5 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 which 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 may not be a descendant of v at the time the path compression occurs. Let $high_L(v)$ be the shallowest node w which is a proper ancestor of v in T_0 and which 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 which is a proper ancestor of v in T_0 and which 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 may not be from a descendant of v at the time.

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

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

To make $last_L$ and $last_R$ well-defined, we say that v is never a descendant of $\emptyset$. The definition of $level_L$ and $level_R$ is 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 which is an ancestor of x if x is a proper descendant of v.

Lemma 5.1 *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 $level_L$ 0. 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. So 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 the property. □

Lemma 5.2 *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 5.1, $level_L(p(v))$ never increases during compressions C_1 through $C_{last_L(v)-1}$. However, 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 5.1 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 5.1 and 5.2 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 5.4, and it is here that we rely upon these conditions. We use the notation $T^{(v)}$ and $T_x^{(v)}$ as in Definition 4.6.

Lemma 5.3 *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 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., is a left-spine compression), then by Definition 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 $l - 1$, and thus $level_L(p(v))$ changes 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 $i < last_L(v)$. □

We can now prove our main result:

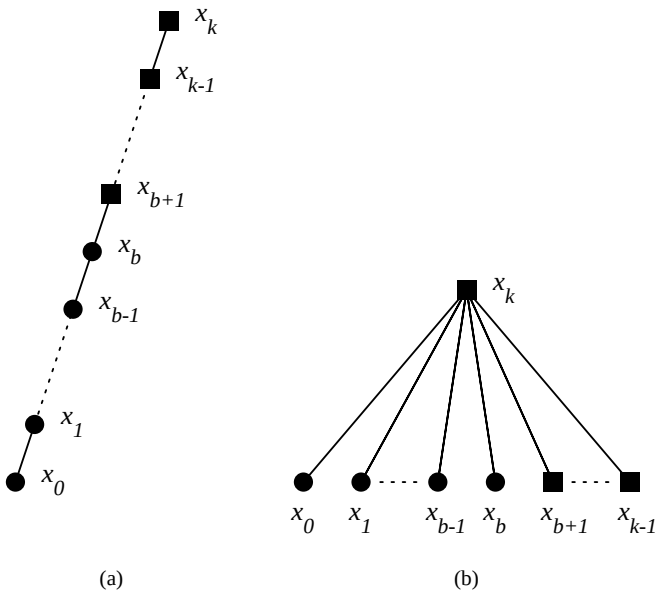

Figure 4: (a) A left-spine (incident subtrees not shown); circles are nodes of $level_L$ l, and squares are nodes of $level_L$ $l-1$. (b) After compressing the path of (a).

Theorem 5.4 *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 $C = (x_0, \ldots, x_k)$. Using the definition of $level_L$ from before, the definition together with Lemma 5.1 shows that there is 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 4a.

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 change, which by Lemma 5.2 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 5.3. 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 4b. Since leaf deletions take unit time, the bound follows. □

It is not hard to see that the data structure of Section 3 employs a deque ordered sequence of spine-only path compressions satisfying the Rising Roots Condition. Thus, Theorem 5.4 completes the proof of Theorem 3.1. We again mention that the initial tree T_0 need not be balanced.

6 Conclusion and Open Problems

We have described how to implement catenable heap ordered double ended queues in constant amortized time per operation (worst case if the number of queues is fixed). The important pieces of our work are the use of the bootstrapping technique of Driscoll et al [DST91] in designing the data structure and the analysis of deque ordered spine-only path compression in proving its efficiency. In particular, 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 leave the following open problems. First, is there an implementation of catenable mindeques that achieves constant worst case time per operation (for an arbitrary number of deques)? Also, is

there a linear time implementation of catenable mindeques that does not involve the bootstrapping technique and path compression?

Data structural bootstrapping holds promise as a general tool for designing data structures with some sort of *join* operation together with a property secondary to the original data structure. Driscoll et al [DST91] use it to effect persistence in catenable lists; we employ it to effect heap order in catenable deques. Work on formalizing this method and finding further applications seems worthwhile.

The Hart-Sharir [HS86] open problem of postorder path compression without the Rising Roots Condition remains tantalizingly open. As an intermediate step towards solving this problem, we suggest considering general order preserving deque ordered path compression (with Rising Roots). This in itself presents subtle problems not encountered when analyzing spine-only compression. Removing the Rising Roots Condition and/or the notion of some order of the path compressions seems much more difficult.

7 Acknowledgements

Jeff Westbrook initially brought the problem of catenable mindeques to our attention. Milena Mihail and Peter Winkler provided thoughtful criticism of an earlier draft of Sections 2 and 3. Greg Frederickson pointed us to [BW90].

References

[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.

[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.

[BW90] H. Booth and J. 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*.

[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.

[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.

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

[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.

[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.

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

[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.

[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.

[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.

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

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