Maintaining dynamic sequences under equality–tests in polylogarithmic time*

K. Mehlhorn† R. Sundar‡ C. Uhrig†

January 16, 1996

Abstract

We present a randomized and a deterministic data structure for maintaining a dynamic family of sequences under equality–tests of pairs of sequences and creations of new sequences by joining or splitting existing sequences. Both data structures support equality–tests in $O(1)$ time. The randomized version supports new sequence creations in $O(\log^2 n)$ expected time where n is the length of the sequence created. The deterministic solution supports sequence creations in $O(\log n(\log m \log^* m + \log n))$ time for the m–th operation.

1 Introduction

We present a data structure for maintaining dynamically a family $\mathcal{F}$ of sequences over a universe U. The data structure supports the following operations on an initially empty family of sequences; let s_1, s_2 be sequences, $a_j \in U$ for $j = 1, .., n$ and i an integer.

- **Equal**(s_1, s_2)**:** Test if $s_1 = s_2$.
- **Makesequence**(s, a_1)**:** Create the sequence $s_1 = a_1$.
- **Concatenate**(s_1, s_2, s_3)**:** Create the sequence $s_3 = s_1 s_2$ without destroying the sequences s_1 and s_2.
- **Split**(s_1, s_2, s_3, i)**:** Create the two new sequences $s_2 = a_1 \dots a_i$ and $s_3 = a_{i+1} \dots a_n$ without destroying $s_1 = a_1 \dots a_n$.

*This work was supported by the ESPRIT Basic Research Actions Program, under contract No. 7141 (project ALCOM II).

†Max-Planck-Institut für Informatik, Im Stadtwald, 6600 Saarbrücken, Germany

‡Department of Computer Science and Automation, Indian Institute of Science, Bangalore 560012, India

Operation	randomized	deterministic
Equal	1	1
Makesequence	1	$\log m$
Concatenate	$O(\log^2 n)$	$O(\log n(\log m \log^* m + \log n))$
Split	$O(\log^2 n)$	$O(\log n(\log m \log^* m + \log n))$

Table 1: Time bounds

randomized	deterministic
$O(\log^2 n)$	$O(\log n \log^* m)$

Table 2: Incremental space cost

We present two solutions: one randomized and one deterministic. The deterministic solution is essentially a derandomization of the randomized solution. Table 1 lists the time bounds for the m–th operation in a sequence of operations. We use n to denote the total length of the sequences involved in the m–th operation.

The problem of maintaining dynamic sequences with equality test arises mainly in the implementation of high–level programming languages like SETL, where sets and sequences are basic data types and equality tests are permitted. The best previous deterministic solution is due to Sundar and Tarjan [ST90]. They achieve a constant time equality test and amortized update time $O(\sqrt{n \log m} + \log m)$. The amortized space required per update is $O(\sqrt{n})$. Our solution is exponentially better. Pugh and Teitelbaum [PT89] gave a randomized solution for the special case of *repetition–free* sequences. It has logarithmic expected running time per operation.

We use the standard RAM model of computation. In particular, we assume that the word size w is at least $\log \max(n, m)$, that arithmetic on words of length w takes constant time, and that a random bitstring of length w can be chosen in constant time.

We now give a brief account of our randomized solution. We compute for each sequence s a unique signature $sig(s)$ in $[0..m^3]$. This signature is used to perform equality tests. The signature of a sequence $s = a_1 a_2 \ldots a_n$ with $a_i \neq a_{i+1}$ is computed as follows (the extension to general sequences is described in section 3): Firstly, s is broken into blocks (subsequences) of length at least 2 and expected length at most $\log n$. Secondly, each block, say $b = a_i \ldots a_j$, is replaced by a single integer which is computed in a Horner–like scheme by means of a pairing function p, i.e., b is replaced by $p(a_i, p(a_{i+1}, \ldots, p(a_{j-1}, a_j)) \ldots)$. Thirdly, the same rules are applied to the shrunken sequence until the shrunken sequence has length one. The depth of nesting in this recursion is $O(\log n)$. Randomization is used to break a sequence

into blocks. Each element of the sequence chooses a random real number and blocks begin at local minima. In this way blocks (except maybe the first) have length at least 2. Also the expected length of the longest block is $O(\log n)$ (since the probability that a sequence of k random real numbers is increasing or decreasing is $2/k!$). The update algorithms need to manipulate a bounded number of blocks in each level of the recursion and hence spend time $O(\log n)$ in each level. The $O(\log^2 n)$ time bound results.

In our deterministic solution we replace the randomized strategy for breaking a sequence into blocks by a deterministic one which we exhibit in section 2. This strategy is based on the algorithm for three–coloring rooted trees (we consider a sequence to be a rooted tree) by Goldberg, Plotkin and Shannon [GPS88], which is a generalization of the so–called *deterministic coin tossing technique* of Cole and Vishkin [CV86]. It generates blocks of length at most 4 and decides for each index i whether a_i starts a new block by looking only at a_{i+l} for $0 \leq |l| = O(\log^* m^3) = O(\log^* m)$. The update algorithms have a recursion depth of $O(\log n)$. On each level they spend time $O(\log n)$ to manipulate the balanced tree imposed on the level and handle $O(\log^* m)$ blocks. For each block time $O(\log m)$ is needed.

This paper is structured as follows. In section 2 we give randomized and deterministic rules for decomposing a sequence into blocks, in section 3 we define a hierarchical representation of sequences based on the block decomposition, and in section 4 we show how to realize the various operations.

2 The Block Decomposition

2.1 The Randomized Block Decomposition

Let U be a universe and $s = a_1 \dots a_n$ with $a_i \in U$ and $a_i \neq a_{i+1}$ for $1 \leq i \leq n-1$. Each element $a \in U$ is assigned a random priority $prio(a) \in [0, 1]$. We represent these priorities with sufficiently large finite precision that guarantees that all priorities are distinct. In section 4 we show that the expected number of bits in the representation of a priority is small and that this will not affect the complexity of the operations.

An element a_i of s is a local minimum of s if it has a successor in s and its priority is a local minimum in the sequence $prio(a_1) \dots prio(a_n)$ of priorities corresponding to s.

Randomized marking rule: Every local minimum is marked.

Now a block starts at position 1 and at every marked element.

2.2 The Deterministic Block Decomposition

In this section we give a deterministic construction to divide a sequence into suitable blocks. The algorithm is essentially a sequential version of the so–called *three–coloring*

technique for rooted trees of Goldberg, Plotkin and Shannon [GPS88] (which in turn is a generalization of the *deterministic coin tossing technique* of Cole and Vishkin [CV86]) and can be considered as a constructive proof of the following lemma.

Lemma 1 For every integer N there is a function
$f : ([0..N-1] \cup \{-1\})^{\log^* N+11} \rightarrow \{0,1\}$ such that for every sequence $a_1 \dots a_n \in [0..N-1]^*$ with $a_i \neq a_{i+1}$ the sequence $d_1 \dots d_n \in \{0,1\}^*$ defined by
$d_i := f(\tilde{a}_{i-\log^* N-6}, \dots, \tilde{a}_{i+4})$ where $\tilde{a}_j = a_j$ for $1 \leq j \leq n$ and $\tilde{a}_j = -1$ otherwise, satisfies:

1) $d_i + d_{i+1} \leq 1$ for $1 \leq i < n$
2) $d_i + d_{i+1} + d_{i+2} + d_{i+3} \geq 1$ for $1 \leq i < n-3$

The sequence $d_1 \dots d_n$ is used to decompose the sequence $a_1 \dots a_n$ into blocks according to the following rule: Start a new block at index $i = 1$ and at every index i with $d_i = 1$. It is clear that no block has length exceeding four and that all but the first and last block have length at least two.

In the following subsection we will review the three–coloring technique. Afterwards we explain the decomposition rule and finally we show how one can derive lemma 1 from the coloring algorithm.

2.2.1 The Three–coloring Algorithm

Let $s = a_1 \dots a_n$ with $a_i \in [0..N-1]$ and $a_i \neq a_{i+1}$. Let's consider s as a linked list (which is a special form of rooted tree). A *coloring* of a list is an assignment $C : \{a_1, .., a_n\} \rightarrow \mathbb{N}$. A *valid* coloring is a coloring such that no two adjacent elements have the same color. In this section we review the three–coloring algorithm of Goldberg, Plotkin, and Shannon ([GPS88]). Lemmas 2 and 3 are due to them but lemma 4 of the next section is new.

Informally it is done as follows. We first compute a valid $\lceil \log N \rceil$– coloring. Afterwards we replace the elements in the list by their colors, consider the set of colors to be the new universe and iterate the coloring procedure. After $O(\log^* N)$ iteration steps we will get a valid six–coloring which we then reduce by a different procedure to a three–coloring (of course it is easy to compute a valid two–coloring for a list in time $O(n)$, but for our purpose the decisions have to be made 'locally'). The details are as follows:

Identify each a_i with its binary representation (which has $\lceil \log N \rceil$ bits). The bits are numbered from zero and the j-th bit of the representation of a color of element a_i is denoted by $C_i(j)$. The following procedure has as input the sequence $s = a_1 \dots a_n$ and computes a six–coloring for s. In each iteration every element a_i is assigned a new color by concatenating the number of the bit, where the old color differs from the old color of a_{i-1} and the value of this bit.

(1) **Procedure** Six-colors($a_1 \dots a_n$: sequence) ;

```
(2)  begin
(3)      N_C ← N ;
(4)      forall i ∈ {1,..,n} do
(5)          C_i ← a_i ;
(6)      od ;
(7)      while N_C > 6 do
(8)          j_1 ← 0 ;
(9)          b_1 ← C_1(0) ;
(10)         C_1 ← 2j_1 + b_1 ;
(11)         forall i ∈ {2,..,n} do
(12)             j_i ← min{j|C_i(j) ≠ C_{i-1}(j)} ;
(13)             b_i ← C_i(j_i) ;
(14)             C_i ← 2j_i + b_i ;
(15)         od ;
(16)         N_C ← max{C_i|i ∈ {1,..,n}} + 1 ;
(17)     od ;
(18) end ;
```

Lemma 2 The procedure Six–colors produces a valid six–coloring of a list $a_1 \ldots a_n$ where $a_i \in \{0,..,N-1\}$ for $1 \leq i \leq n$ in time $O(n \log^* N)$.

Proof. First we show that the procedure computes a valid coloring. Note that C is valid at the beginning, since $a_i \neq a_{i+1}$ for $1 \leq i \leq n-1$. Now suppose, C is valid when we enter the loop of the while–statement. Consider two adjacent elements a_i and a_{i+1} for some $i > 1$. In line (12), a_{i+1} chooses some index j_1 such that $C_{i+1}(j_1) \neq C_i(j_1)$ and a_i chooses some index j_2 such that $C_i(j_2) \neq C_{i-1}(j_2)$. The new color of a_{i+1} is $2j_1 + C_{i+1}(j_1)$ and the new color of a_i is $2j_2 + C_i(j_2)$ (note that line (14) means to concatenate the number of the least significant bit, where the old color differs from the old color of a_{i-1} and the value of this bit). If $j_1 \neq j_2$ the new colors are different and we are done. Otherwise $j_1 = j_2$ and $C_i(j_1) \neq C_{i+1}(j_1)$ by the definition of j_1 and again the new colors are different. Thus at the end of the loop the new coloring is also valid.

Now we derive the number of iterations. Let L denote $\lceil \log N \rceil$ and L_k denote the number of bits in the longest representation of a color after k iterations. We will show that $L_k \leq \lceil \log^k L \rceil + 2$, if $\lceil \log^k L \rceil \geq 2$.

For $k = 1$ we have $L_1 = \lceil \log L \rceil + 1$.

Now suppose $\lceil \log^{k-1} L \rceil \geq 4$, $L_{k-1} \leq 2\lceil \log^{k-1} L \rceil + 2$ and $\lceil \log^k L \rceil \geq 2$. Then

$$\begin{aligned} L_k &= \lceil \log L_{k-1} \rceil + 1 \\ &\leq \lceil \log(2 \log^{k-1} L) \rceil + 1 \\ &\leq \lceil \log^k L \rceil + 2. \end{aligned}$$

After $\log^* N + 1$ iterations $\lceil \log^k L \rceil$ becomes one and L_k becomes three. Then there are three possible values for the index j and two possible values of the bit b_i. Therefore another iteration produces a valid six–coloring and the time bound follows. ▌**Lemma 2**

We can easily transform the six–coloring into a three–coloring, which is done by the following procedure.

```
(1)  Procedure Three–colors(a_1 ... a_n : sequence) ;
(2)  begin
(3)      Six–colors(a_1 ... a_n) ;
(4)      C_0 ← ∞ ;
(5)      C_{n+1} ← ∞ ;
(6)      for c = 3 to 5 do
(7)          forall i ∈ {1, .., n} do
(8)              if C_i = c then
(9)                  C_i ← min{{0, 1, 2} − {C_{i−1}, C_{i+1}}} ;
(10)             fi ;
(11)         od ;
(12)     od ;
(13) end ;
```

Lemma 3 The procedure Three–colors produces a valid three–coloring of a list $a_1 \dots a_n$ where $a_i \in \{0, .., N-1\}$ for $1 \leq i \leq n$ in time $O(n \log^* N)$.

Proof. In line (3) the procedure computes a valid six–coloring. Then each of the three iterations of the for–statement (line (5)) removes one color and preserves the validity of the coloring, since every list element whose color is replaced gets a new color different from the (unchanged) colors of its two neighbors. Therefore the three–coloring at the end of the third iteration is still valid. The running time of lines (4) – (12) is obviously $O(n)$ and the time bound follows. ▌**Lemma 3**

2.2.2 The Decomposition Rule

For any sequence $a_1 \dots a_n$ the sequence $d_1 \dots d_n$ is defined in the following way. We first compute a valid three–coloring by the procedure above and then set $d_i = 1$ iff the color of a_i which now is considered to be an integer in $\{0, 1, 2\}$ is a local maximum and $d_i = 0$ otherwise.

For convenience, let's define the elements a_i with $i < 1$ or $i > n$ to be *empty* elements that take no influence on the computation (in Lemma 1 these elements are written as -1).

Lemma 4 Given a sequence $a_1 \dots a_n$, the values $d_1 \dots d_n$ defined as given above have the following properties:

1) $d_i + d_{i+1} \leq 1$ for $1 \leq i < n$

2) $d_i + d_{i+1} + d_{i+2} + d_{i+3} \geq 1$ for $1 \leq i < n - 3$

3) The value of d_i only depends on the subsequence $a_{i-\log^* N-6} \dots a_{i+4}$

PROOF. Property 1 follows of the fact, that in a valid coloring any two consecutive colors (colors of consecutive elements) are different and thus there are no neighboring local maxima.

For property 2 note that any sequence of four consecutive elements either contains the color 2 which is always a local maximum or it contains the subsequence 010 where 1 is a local maximum.

Property 3 will be proven in several steps. First we prove by induction on the number of iterations of the while–statement in the procedure Six–colors that for each a_i the color of the valid six–coloring computed by the procedure only depends on the subsequence $a_{i-\log^* N-2} \dots a_i$. More precisely we argue that the color of a_i after the k-th iteration depends on the subsequence $a_{i-k} \dots a_i$. But this is easy to see. Before the first iteration the color of a_i is given by a_i directly and does not depend on another element. Now suppose that for each $1 \leq i \leq n$ the color of a_i after the $(k-1)$-th iteration depends on the subsequence $a_{i-k+1} \dots a_i$. During the next iteration each element a_i for $1 < i \leq n$ is assigned a new color by concatenating the binary string representation of the lowest index of the bit where the old color (its binary representation) differs from the old color of a_{i-1}, and the value of this bit. Therefore the new color of a_i only depends on its old color C_i and the old color C_{i-1} of element a_{i-1}. Since the C_i depended on $a_{i-k+1} \dots a_i$ and C_{i-1} depended on $a_{i-k} \dots a_{i-1}$, the new color depends on $a_{i-k} \dots a_i$ and the induction step is completed. Now note that in the proof of lemma 2 we have shown that the procedure Six–colors performs at most $\log^* N + 2$ iterations. We are done by setting $k = \log^* N + 2$.

Next we argue that for each a_i the color given by the execution of the procedure Three–colors only depends on the subsequence $a_{i-\log^* N-5} \dots a_{i+3}$. This again can be seen by induction on the number of iterations of the procedure. Before the first iteration each color C_i depends on the subsequence $a_{i-\log^* N-2} \dots a_i$ (see above). In each iteration the new color of an element depends on the old colors of its two neighbors and since there are only three iterations the color of a_i in the six–coloring depends on the six–coloring of the elements $a_{i-3} \dots a_{i+3}$ and therefore on the elements $a_{i-\log^* N-5} \dots a_{i+3}$. Finally to complete the prove for property 3 note that the value of d_i is set in dependence of the colors C_{i-1}, C_i and C_{i+1} and therefore of the subsequence $a_{i-\log^* N-6} \dots a_{i+4}$. ▌**Lemma 4**

PROOF OF LEMMA 1. Now note that by the definition of the d_i's the existence of the functions as demanded in lemma 1 is proven. ▌**Lemma 1**

Deterministic marking rule: Every position i with $d_i = 1$ is marked.

As mentioned before, we now decompose the sequence into blocks by starting a new block at position 1 and at every marked position.

3 A Hierarchical Representation of Sequences

As mentioned in section 1 we support efficient equality tests by assigning unique signatures to sequences. In this section we explain how this is done and how sequences are represented. The maintenance of the data structure does not require that we always compute the signature for an arising sequence from scratch. A signature is a small integer. More precisely, after m operations there is no signature exceeding m^3. We need more than m signatures since signatures are also assigned to subsequences of and sequences derived from the original sequences built by the user.

Informally a signature is assigned to a sequence $s = a_1^{l_1} \dots a_n^{l_n}$ with $a_i \neq a_{i+1}$ for $1 \leq i < n$ in the following way. Each element $a_i \in U$ gets a signature. Furthermore to each power $a_i^{l_i}$ for $1 \leq i \leq n$ a signature is assigned. Then we compute a block decomposition of the sequence $sig(a_1^{l_1}) \dots sig(a_n^{l_n})$ according to the methods given in section 2. Note that all neighboring elements in this sequence are different. Then for each block a signature is computed by repeated application of a pairing function, i.e. pairs of signatures are encoded by a new signature. Afterwards the whole procedure is applied on the sequence of block encodings (instead of the original sequence) and this is repeated until the original sequence is reduced to a single integer – its signature. We now give the details.

Let S be the current set of signatures, $S = [0..max\_sig]$. Each element in S encodes either an element of U or a pair in $S \times S$ or a power in $S \times \mathbb{N}_{\geq 2}$, i.e., S is the disjoint union $S_U \cup S_P \cup S_R$ and there are injections $u : S_U \to U$, $p : S_P \to \{(a, b); a, b \in S \text{ and } a \neq b\}$ and $r : S_R \to \{(a, i); a \in S \text{ and } i \in \mathbb{N}, i \geq 2\}$. The inverses of the functions u, p, and r are maintained in dictionaries (in the randomized case based on dynamic perfect hashing and in the deterministic case based on balanced binary trees). In the randomized scheme every element $s \in S$ that encodes a power, also has a random real priority $prio(s) \in [0, 1]$ associated with it. For each such s we only store a finite approximation of $prio(s)$; the approximations are long enough to be pairwise distinct. They are chosen in a piece–meal fashion, i.e., whenever two priorities need to be compared and are found to be equal they are extended by a random word. Lemma 9 shows that only approximations of logarithmic length are needed on average.

We now define the signature $sig(s)$ of a sequence $s = a_1^{l_1} \dots a_n^{l_n}$ with $a_i \neq a_{i+1}$ for all $i, 1 \leq i \leq n$ and $n \geq 1$, and also the related functions g, $shrink$ and Sig.

The function sig is defined recursively. If $n = 1$ and $l_1 = 1$ then $sig(s) = Sig(s)$, if $n = 1$ and $l_1 > 1$ then $sig(s) = r^{-1}((a_1, l_1))$ (if $(a_1, l_1) \notin range(r)$ then increment max_sig and extend r by $(max\_sig, (a_1, l_1))$ and if $n > 1$ then $sig(s) = sig(shrink(s))$.

So assume $n > 1$. Let $g(s) = sig(a_1^{l_1}) \dots sig(a_n^{l_n})$, i.e., every power is replaced by its signature. Then $g(s) = c_1 \dots c_m$ with $c_i \neq c_{i+1}$ for all $i, 1 \leq i < m$, and some $m \geq 1$.

Let $b_1 \dots b_k$ be the block decomposition of $g(s)$. Set $shrink(s) = Sig(b_1) \dots Sig(b_k)$.

The function Sig is only defined for sequences with $l_1 = \dots = l_n = 1$ (note that's all that is needed). If $n = 1$ and $a_1 \in S$ then $Sig(s) = a_1$, if $n = 1$ and $a_1 \in U$ then $Sig(s) = u^{-1}(a_1)$ (if $a_1 \notin range(u)$ then we also increment max_sig and extend u by (max_sig, a_1)). If $n = 2$ then $Sig(s) = p^{-1}((Sig(a_1), Sig(a_2)))$ (again, extend p if necessary), and if $n > 2$, then $Sig(s) = Sig(a_1, Sig(a_2, \dots Sig(a_{n-1}, a_n) \dots))$.

Of course, in order to show the correct support of the operation Equal(s_1, s_2) we have to prove

Lemma 5 Let $s_1, s_2 \in \mathcal{F}$. Then $s_1 = s_2 \;\Leftrightarrow\; sig(s_1) = sig(s_2)$

Proof. Each $s \in S$ encodes a unique sequence in U^*. Simply run the encoding process backwards. ▌**Lemma 5**

Let's next explain how sequences are stored. As above, let $s = a_1^{l_1} \dots a_n^{l_n}$, let $g(s) = sig(a_1^{l_1}) \dots sig(a_n^{l_n})$, let $g(s) = b_1 \dots b_k$ be the sequence of blocks of $g(s)$ and finally let $shrink(s) = Sig(b_1) \dots Sig(b_k)$.

Then we represent a sequence s by $\bar{s} = (s_0 \dots s_{2t})$ where $s_0 = s$ and for all i, $1 \le i \le t$: $s_{2i-1} = g(s_{2i-2})$ and $s_{2i} = shrink(s_{2i-1})$. Note that $t = O(\log n)$ in both schemes, since blocks (except maybe the first) have length at least two in both schemes.

In order to support the operations we store each s_j as a balanced binary tree T_j, whose symmetric traversal yields s_j. Each node v contains:

- an element a of s_j
- the size of the subtree rooted at v
- the length of the block corresponding to a in s_{j-1}
- the sum of the lengths of the blocks corresponding to the elements stored in the subtree of v
- the mark of the element a if j is odd

Each $\bar{s}$ is maintained as a linked list of the roots of the trees T_j and $\mathcal{F}$ is maintained as a linked list of the heads of these lists. In the randomized solution the dictionaries are implemented by dynamic perfect hashing (see [DKMMRT88]) and in the deterministic solution they are maintained as balanced binary trees. The operations are performed in a persistent way such that none of the sequences is destroyed (see [DSST89] for the details).

4 The Operations

The operations *Equal* and *Makesequence* are identical in both cases (randomized and deterministic).

Let s_1, s_2 be sequences. Then $Equal(s_1, s_2)$ returns *true* if $sig(s_1) = sig(s_2)$ and $false$ otherwise. This obviously needs time $O(1)$.

For $a \in U$, $Makesequence(s, a)$ creates a single node binary tree representing $s = sig(a)$. Therefore it retrieves $sig(a)$ if $a \in range(u)$; otherwise a new signature is assigned and u is extended.

This requires $O(\log m)$ time in the deterministic and $O(1)$ time in the randomized case.

The operations *Concatenate* and *Split* are more difficult to realize, but the basic idea is simple. When we concatenate s_1 and s_2 all but the $O(1)$ last blocks of s_1 and all but some few first blocks ($O(log^* m)$ for the deterministic and $O(1)$ for the randomized case) of s_2 will also be blocks of $s_1 s_2$ since the fact whether an element starts a new block is only influenced by a small neighborhood of the element (of size $O(\log^* m)$ in the deterministic and $O(1)$ in the randomized case).

4.1 The Randomized Update Operations

We first discuss the operation *Concatenate*. We are given the hierarchical representations of sequences s_1 and s_2 and need to compute the hierarchical representation of $s_3 = s_1 s_2$. The following Lemma paves the way. It states that we essentially only need to concatenate the individual elements of the hierarchical representations.

Lemma 6 Let $s_1 = a_1 \dots a_l$, $s_2 = a_{l+1} \dots a_n$ and $s_3 = s_1 s_2$ be sequences and let j be a nonnegative integer. Let $shrink^j(s_3) = c_1 \dots c_o$ and let i be such that c_i encodes the subsequence of s_3 containing a_l. Then

1. $c_1 \dots c_{i-5}$ is a prefix of $shrink^j(s_1)$ and $|shrink^j(s_1)| \leq i + 5$
2. $c_{i+4} \dots c_o$ is a suffix of $shrink^j(s_2)$ and $|shrink^j(s_2)| \leq o - i + 7$

Proof. We use induction on j.

For $j = 0$ there is nothing to prove since $shrink^0(s_i) = s_i$ for all i, $1 \leq i \leq 3$. So assume that the claim holds for some $j \geq 0$. We establish the claim for $j + 1$.

Let's denote $shrink^{j+1}(s_3)$ by $c'_1 \dots c'_{m'}$, where $c'_{i'}$ encodes the subsequence of s_3 containing a_l and $g(shrink^j(s_3))$ by $g_1 \dots g_k$, where g_z encodes the subsequence of s_3 containing a_l. By induction hypothesis we have $shrink^j(s_1) = c_1 \dots c_{i-5} d_1 \dots d_p$ and $shrink^j(s_2) = e_1 \dots e_q c_{i+4} \dots c_o$ with $p, q \leq 10$. Then the subsequence encoded by $g_1 \dots g_{z-6}$ is a proper prefix of that encoded by $c_1 \dots c_{i-5}$ and the subsequence of $g_{z+5} \dots g_k$ is a proper suffix of that encoded by $c_{i+4} \dots c_o$. Since the marks are influenced by at most one predecessor and one successor (by the definition of 'local

minimum'), the marks of the sequences $g_1 \dots g_{z-7}$ and $g_{z+5} \dots g_k$ are identical to those of the corresponding elements in $g(shrink^j(s_1))$ and $g(shrink^j(s_2))$. Since every block has size at least 2 it follows that the subsequence $c'_{i'-4} \dots c'_{i'+3}$ encodes the subsequence of $g(shrink^j(s_3))$ containing $g_{z-7} \dots g_{z+6}$. Thus $c'_1 \dots c'_{i'-5}$ exclusively depends on $c_1 \dots c_{i-5}$ and therefore is a prefix of $shrink^{j+1}(s_1)$ and $c'_{i'+4} \dots c'_{o'}$ exclusively depends on $c_{i+4} \dots c_o$ and therefore is a suffix of $shrink^{j+1}(s_2)$.

Let's denote $g(shrink^j(s_1))$ by $g_1 \dots g_{z-6} g'_1 \dots g'_y$ and $shrink^{j+1}(s_1)$ by $c'_1 \dots c'_{i'-5} d'_1 \dots d'_{p'}$. Note, that the sequence $c'_{i'-4} \dots c'_{i'}$ encodes a sequence $g_{z-x} \dots g_{z-6} \dots g_z$ and the sequence $d'_1 \dots d'_{p'}$ encodes a sequence $g_{z-x} \dots g_{z-6} g'_1 \dots g'_y$ where $y \leq p+1$. $g_{z-x} \dots g_{z-7}$ is encoded by at most 4 elements (then $c'_{i'-4} \dots c'_{i'-1} = d'_1 \dots d'_4$). $g_{z-6} g'_1 \dots g'_y$ is encoded by at most $\lceil (y+1)/2 \rceil = \lceil (p+2)/2 \rceil$ elements. Since $p \leq 10$, $p' \leq 4 + 6 = 10$. A similar argument shows that $q' \leq 10$ and we are done.

▌**Lemma 6**

Lemma 6 tells that all but a small middle part of $shrink^j(s_3)$ can be copied from either $shrink^j(s_1)$ or $shrink^j(s_2)$. The proof of Lemma 6 also gives the recipe for computing the missing part from $shrink^j(s_1)$, $shrink^j(s_2)$ and $g(shrink^{j-1}(s_3))$: Again, let g_z be the element of $g(shrink^{j-1}(s_3))$ encoding the subsequence of s_3 containing a_l. Compute the marks for the elements $g_{z-6} \dots g_{z+5}$ and then compute the middle part $c_{i-4} \dots c_{i+3}$ (by retrieving or assigning signatures). To get $shrink^j(s_3)$ combine this middle part with the parts $c_1 \dots c_{i-5}$ of $shrink^j(s_1)$ and $c_{i+4} \dots c_o$ of $shrink^j(s_2)$ (which we easily get with the help of the information stored in the nodes of the trees).

Now the computation of the hierarchical representation $\bar{s_3}$ of s_3 is easy to understand. Generally all operations are performed persistently. This is essentially done by copying all nodes that are to be changed and then changing these copies. The details of this technique can be seen in [DSST89].

In the following $s_1 = a_1 \dots a_l$, $s_2 = a_{l+1} \dots a_n$, s denotes a sequence, $g(s)$ is denoted by $g_1 \dots g_m$ and T_s denotes the balanced binary tree for a sequence s.

Procedure RanConcatenate(s_1, s_2, s_3 : sequence);

1. Compute T_{s_3} by joining T_{s_1} and T_{s_2}.
2. Compute $T_{g(s_3)}$ by joining $T_{g(s_1)}$ and $T_{g(s_2)}$ (in the case that $a_l = a_{l+1}$ recompute the corresponding element of g_{s_3}).
3. Let $s = s_3$, let z be such that g_z encodes the subsequence of s_3 containing a_l and let $\bar{s_3}$ be an empty list.
4. **while** $|s| > 1$ **do** steps (5) to (8).
5. Append s and $g(s)$ at the end of the list $\bar{s_3}$.

6. Choose (or retrieve) the priorities of the subsequence $g_{z-5} \dots g_{z+4}$ of $g(s)$. Then compute the marks of $g_{z-6} \dots g_{z+4}$ in $T_{g(s)}$.
7. Let $s = shrink(s)$, where $shrink(s)$ is computed as described above.
8. Compute T_s from the corresponding trees of s_1 and s_2 and the new middle part. If $|s| > 1$ then compute $T_{g(s)}$ and update z.
9. Append s at the end of $\bar{s_3}$.

The complexity of the operation *RanConcatenate* is given by

Lemma 7 A *RanConcatenate* operation requires expected time $O(\log^2 n)$ and expected space $O(\log^2 n)$.

PROOF. Lines (1) and (2) require time $O(\log n)$. Lines (7) and (8) can also be done in $O(\log n)$ by use of the informations stored in the nodes of the trees (see section 3). Let L be the number of bits of precision needed to represent a random priority so that all of the random priorities will be distinct and let $\bar{l} = \lceil L/w \rceil$ be the maximal number of memory words needed to represent a priority. Then line (6) needs time $O(\bar{l})$. In line (7) we have to recompute the signatures of $O(1)$ blocks. Let l be the maximal length of a block in $\bar{s_3}$. Then line (7) needs time $O(l)$ to retrieve or create the signatures. Note that priorities are only assigned to those signatures being element of a sequence $g(s)$ (see line (6)). Line (8) again needs time $O(\log n)$. Thus we spend time $O(\log n + l + \bar{l})$ per level of the hierarchy. Since there are $O(\log n)$ recursion steps we need time $O(\log n(\log n + l + \bar{l}))$.

Now we want to compute the expected size of the largest block.

Lemma 8 $E[l] \leq 2 \log n + 2$

PROOF. Let l' be the length of the longest subsequence of increasing priorities in a sequence s. Since every block of s is a sequence of elements of increasing priorities followed by a sequence of elements of decreasing priorities it follows that $E[l] \leq 2E[l']$. Let us estimate $E[l']$. Suppose that $s = a_1 \dots a_k$ and let j and t be positive integers.

$$
\begin{aligned}
Pr[[prio(a_j) < prio(a_{j+1}) < \dots < prio(a_{j+t-1})] &= 1/t! \text{ and so} \\
Pr[\exists j : prio(a_j) < prio(a_{j+1}) < \dots prio(a_{j+t-1}) &\leq k/t! \\
\text{Hence } E[l'] &\leq \lceil \log k \rceil + \sum_{t=\lceil \log k \rceil + 1}^{k} k/t! \\
&\leq \lceil \log k \rceil + 1. \\
E[l'] &\leq 2\lceil \log k \rceil + 2 \leq 2\lceil \log |s| \rceil + 2
\end{aligned}
$$

▌**Lemma 8**

Note that the expected number of signatures (and therefore the incremental space cost) produced by a *Concatenate* operation is $\log^2 n$. Furthermore since n is bounded by 2^m, the expected value of max_sig is at most m^3.

Next we compute the expected number of bits for the priorities. Let m be the number of sequences in the family and $Prio$ the set of priorities. Note that on each level of *Concatenate* at most 10 priorities are chosen (line (6)). Since there are $\log n$ levels and n is bounded by 2^m there are at most $10m^2$ priorities assigned.

Lemma 9 $E[L] \leq 40\lceil \log m \rceil + 11$.

PROOF. Let agr be a shorthand for 'Some two priorities $prio_1$ and $prio_2$, where $prio_1, prio_2 \in Prio$, agree in the first k bits'. Then

$$Pr[agr] \leq |Prio|^2/2^{k+1}.$$

$$\text{Hence } E[L] \leq 2\lceil \log |Prio| \rceil + \sum_{k=2\lceil \log |Prio| \rceil + 1}^{\infty} |Prio|^2/2^{k+1} \leq 2\lceil \log |Prio| \rceil + 1.$$

Since $|Prio| \leq 10m^2$ it follows that:

$$E[L] \leq 40 \log m + 11.$$

▌**Lemma 9**

Thus the expected number of bits needed to represent priorities is small enough to be represented in $O(1)$ words of memory ($\bar{l}$ is a constant) and the complexity of the operations is not affected by more than a constant. It follows that each recursion step takes expected time $O(\log n)$ and the Lemma is proven. ▌**Lemma 7**

Now let's turn to the split operation. Let $s_1 = a_1 \ldots a_n$, $s_2 = a_1 \ldots a_i$ and $s_3 = a_{i+1} \ldots a_n$. Lemma 6 also suggests how to compute $shrink^j(s_2)$ and $shrink^j(s_3)$ if $shrink^j(s_1)$, $g(shrink^{j-1}(s_2))$ and $g(shrink^{j-1}(s_3))$ are given: Let $shrink^j(s_1)$ be denoted by $c_1 \ldots c_k$ where c_z encodes the subsequence of s_1 containing a_l, let $g(shrink^{j-1}(s_2))$ be denoted by $g_1 \ldots g_p$ and let $g(shrink^{j-1}(s_3))$ be denoted by $h_1 \ldots h_q$. Then choose priorities for the elements $g_{p-12} \ldots g_p$ and $h_1 \ldots h_{12}$; compute the marks for $g_{p-13} \ldots g_p$ and $h_1 \ldots h_{13}$. Lemma 6 guarantees that now all information required to compute $shrink^j(s_1)$ is available. $c_1 \ldots c_{z-5}$ is a prefix of $shrink^j(s_2)$, $g_{c-4} \ldots c_k$ is a suffix of $shrink^j(s_3)$ and the missing parts can easily be computed.

In the following s and s' denote sequences, $g(s)$ is denoted by $g_1 \ldots g_p$, and $g(s')$ by $h_1 \ldots h_q$.

Procedure RanSplit(s_1, s_2, s_3: sequence; i: integer);

1. Compute $T_{s_2}, T_{s_3}, T_{g(s_2)}$ and $T_{g(s_3)}$.
2. Let $s = s_2$ and $s' = s_3$; let $\bar{s_2}$ and $\bar{s_3}$ be empty lists.

3. **while** $|s| > 1$ **do** steps (4) to (7).
4. Choose the priorities of the sequence $g_{p-12} \dots g_p$ if necessary; compute the marks of $g_{p-13} \dots g_p$ in $T_{g(s)}$ according to the randomized marking rule.
5. Append s and $g(s)$ at the end of the list $\bar{s_2}$.
6. Let $s = shrink(s)$, where $shrink(s)$ is computed as explained above.
7. Compute T_s from the corresponding tree of s_1 and the new end part; if $|s| > 1$ compute $T_{g(s)}$.
8. **while** $|s'| > 1$ **do** steps (9) to (12).
9. Choose the priorities of the sequence $h_1 \dots h_{12}$ if necessary; compute the marks of $h_1 \dots h_{13}$ in $T_{g(s')}$ according to the randomized marking rule.
10. Append s' and $g(s')$ at the end of the list $\bar{s_3}$.
11. Let $s' \leftarrow shrink(s')$, where $shrink(s')$ is computed as explained above.
12. Compute $T_{s'}$ from the corresponding tree of s_1 and the new beginning part; if $|s'| > 1$ compute $T_{g(s')}$.
13. Append s at the end of $\bar{s_2}$ and s' at the end of $\bar{s_3}$.

Lemma 10 A *RanSplit* operation requires expected time $O(\log^2 n)$ and expected space $O(\log^2 n)$.

The proof is analogous to that of Lemma 7.

4.2 The Deterministic Update Operations

The deterministic operations are essentially implemented in the same way as the randomized operations. As pointed out above, the main difference is the computation of the block decomposition. The analogous Lemma to Lemma 6 is the following:

Lemma 11 Let $s_1 = a_1 \dots a_l$, $s_2 = a_{l+1} \dots a_n$ and $s_3 = s_1 s_2$ be sequences and $j \geq 0$ an integer. Let $shrink^j(s_3) = c_1 \dots c_o$ and let i be such that c_i encodes the subsequence of s_3 containing a_l. Then

1. $c_1 \dots c_{i-8}$ is a prefix of $shrink^j(s_1)$ and $|shrink^j(s_1)| \leq i + 7$
2. $c_{i+\log^* m^3+10} \dots c_o$ is a suffix of $shrink^j(s_2)$ and $|shrink^j(s_2)| \leq o - i + \log^* m^3 + 11$

The proof is completely analogous to that of Lemma 6. The computation of $shrink(s_3)$ is done as follows: let's denote $g(shrink^{j-1}(s_3))$ by $g_1 \dots g_k$ and let g_z be the element encoding the subsequence of s_3 containing a_l. The marks of the elements $g_1 \dots g_{z-13}$ and $g_{z+2\log^* m^3+16} \dots g_k$ are identical to the corresponding marks in $g(shrink^{j-1}(s_1))$ and $g(shrink^{j-1}(s_2))$. To compute new marks for the elements $g_{z-12} \dots g_{z+2\log^* m^3+15}$ we run the algorithm $Three - colors$ on the subsequence $g_{z-\log^* m^3-18} \dots g_{z+2\log^* m^3+19}$ since all these elements take influence on the missing marks. Afterwards we can compute $shrink^j(s_3)$ by computing the middle part $c_{i-7} \dots c_{i+\log^* m^3+10}$ and copying the other parts from $shrink^j(s_1)$ and $shrink^j(s_2)$. Now it is easy to formulate the procedure $DetConcatenate$.

In the following let $s_1 = a_1 \dots a_l$ and $s_2 = a_{l+1} \dots a_n$. $g(s')$ is denoted by $g_1 \dots g_m$ and T_s denotes the balanced binary tree for a sequence s.

Procedure DetConcatenate(s_1, s_2, s_3 : sequence);

1. Compute T_{s_3} by joining T_{s_1} and T_{s_2}.
2. Compute $T_{g(s_3)}$ by joining $T_{g(s_1)}$ and $T_{g(s_2)}$ (in the case that $a_l = a_{l+1}$ recompute the corresponding element of g_{s_3}).
3. Let $s = s_3$, let z be such that g_z encodes the subsequence containing a_l and let $\bar{s_3}$ be an empty list.
4. **while** $|s| > 1$ **do** steps (5) to (8).
5. Append s and $g(s)$ at the end of the list $\bar{s_3}$.
6. Run Three–colors($g_{z-\log^* m^3-18} \dots g_{z+2\log^* m^3+19}$); then change the marks of $g_{z-12} \dots g_{z+2\log^* m^3+15}$ according to the output of Three–colors and the deterministic marking rule.
7. Assign s to be $shrink(s)$, where $shrink(s)$ is computed as indicated above.
8. Compute T_s. If $|s| > 1$ then compute $T_{g(s)}$ and update z.
9. Append s at the end of $\bar{s_3}$.

The complexity of the operation $DetConcatenate$ is given by

Lemma 12 A $DetConcatenate$ operation requires time $O(\log n(\log m \log^* m + \log n))$ and space $O(\log n \log^* m)$.

PROOF. First note that on every level of the hierarchical representation we create at most $O(\log^* m)$ new signatures. Thereby the space bound follows as well as the fact $max\_sig \leq m^3$, since $\log n$ is bounded by m.

Furthermore, lines (1) and (2) require time $O(\log n)$. Computing the new marks (line (6)) needs time $O((\log^* m)^2)$ (we perform $\log^* m^3$ iterations on a sequence of length about $2\log^* m^3$; see Lemma 3). Note that we only have to redecompose a subsequence of length $O(\log^* m)$ in line (6). For the remaining parts of the sequence we use the information (and the subtrees) of the hierarchical representations of s_1 and s_2. Thus when we compute $shrink(s)$ (line (7)) we need time $O(\log m \log^* m)$ to retrieve or create the signatures (time $O(\log m)$ per dictionary lookup). The building of the trees in lines (8) is done by split and join operations and needs time $O(\log n)$. Thus we spend time $O(\log m \log^* m + \log n)$ per level of the hierarchy. Since there are $O(\log n)$ recursion steps we need time $O(\log n(\log m \log^* m + \log n))$ and the Lemma follows. ▌**Lemma 12**

Now let $s_1 = a_1 \dots a_n$; $g(s)$ is denoted by $g_1 \dots g_p$, and $g(s')$ by $h_1 \dots h_q$.

Procedure DetSplit(s_1, s_2, s_3: sequence; i: integer);

1. Compute $T_{s_2}, T_{s_3}, T_{g(s_2)}$ and $T_{g(s_3)}$.
2. Let $s = s_2$ and $s' = s_3$; let $\bar{s_2}$ and $\bar{s_3}$ be empty lists.
3. **while** $|s| > 1$ **do** steps (4) to (7).
4. Run Three–colors($g_{p-\log^* m^3-26} \dots g_p$). Then change the marks of $g_{p-20} \dots g_p$ in $T_{g(s)}$ according to the output of Three–colors and the deterministic marking rule.
5. Append s and $g(s)$ at the end of the list $\bar{s_2}$.
6. Let $s \leftarrow shrink(s)$, where $shrink(s)$ is computed as indicated above.
7. Compute T_s; if $|s| > 1$ compute $T_{g(s)}$.
8. **while** $|s'| > 1$ **do** steps (9) through (12).
9. Run Three–colors($h_1 \dots h_{3\log^* m^3+30}$). Then change the marks of $h_1 \dots h_{3\log^* m^3+26}$ in $T_{g(s')}$ according to the output of Three–colors and the deterministic marking rule.
10. Append s' and $g(s')$ at the end of the list $\bar{s_3}$.
11. Let $s' \leftarrow shrink(s')$, where $shrink(s')$ is computed as indicated above.
12. Compute $T_{s'}$; if $|s'| > 1$ compute $T_{g(s')}$.

13. Append s at the end of $\bar{s_2}$ and s' at the end of $\bar{s_3}$.

The complexity of the *DetSplit* operation is given by

Lemma 13 A *DetSplit* operation requires time $O(\log n(\log m \log^* m + \log n))$ and space $O(\log n \log^* m)$.

The proof is analogous to that of Lemma 12. ▌

References

[CV86] R. Cole, and U. Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information & Control*, 70: 32–53, 1986.

[DKMMRT88] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heyde, H. Rohnert, and R.E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *Proc. 29th IEEE FOCS*, 524–531, 1988.

[DSST89] J.R. Driscoll, N. Sarnak, D.D. Sleator, and R.E. Tarjan. Making data structures persistent. *J. Comp. Sys. Sci.*, 38: 86–124, 1989.

[GPS88] A.V. Goldberg, S.A. Plotkin, and G.E. Shannon Parallel symmetry–breaking in sparse graphs. *SIAM J. Disc. Math.*, Vol. 1, No. 4: 434–446, 1988.

[P88] W. Pugh. Incremental computation and the incremental evaluation of functional programming. *Ph.D. Thesis, Cornell University*, 1988.

[PT89] W. Pugh, and T. Teitelbaum. Incremental computation via function caching. *Proc. 16th ACM POPL*, 315–328, 1989.

[ST90] R. Sundar, and R.E. Tarjan. Unique binary search tree representation and equality–testing of sets and sequences. *Proc. 22nd ACM STOC*, 18–25, 1990.