Algorithmica (1997) 17: 183–198

Algorithmica
© 1997 Springer-Verlag New York Inc.

# Maintaining Dynamic Sequences under Equality Tests in Polylogarithmic Time[1]

K. Mehlhorn,[2] R. Sundar,[3] and C. Uhrig[2]

**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.

**Key Words.** Algorithms, Data structures, Derandomization, Randomization, Sequences.

## 1. Introduction.

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

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

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. The incremental space cost is given in Table 2. We use $n$ to denote the total length of all sequences involved in the $m$th 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. The problem of maintaining dynamic sequences with equality test arises mainly in the implementation of high-level programming languages like SETL, where sequences are supported as a primitive data type and equality tests are allowed.

The best previous deterministic solution is due to Sundar and Tarjan [ST]. They achieve constant time for the equality test and amortized time $O(\sqrt{n \log m} + \log m)$ for

[1] This work was supported by the ESPRIT Basic Research Actions Program, under Contract No. 7141 (Project ALCOM II).
[2] Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany.
[3] Department of Computer Science and Automation, Indian Institute of Science, Bangalore 560012, India.

Received November 5, 1994; revised June 15, 1995. Communicated by B. Chazelle.

**Table 1.** Time bounds.

| 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 | $O(\log^2 n)$ | $O(\log n(\log m \log^* m + \log n))$ |

an update operation. The amortized space required per update is $O(\sqrt{n})$. Our solution is exponentially better. Pugh [P] and Pugh and Teitelbaum [PT] gave a randomized solution for the special case of *repetition-free* sequences (i.e., $a_i \neq a_{i+1}$ for all $i$, $1 \leq i < n$). It has logarithmic expected running time per operation.

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 $s = a_1 a_2 \cdots a_n$ with $a_i \neq a_{i+1}$ for all $i$, $1 \leq i < n$, is computed as follows (the extension to general sequences is described in Section 3): First, $s$ is broken into blocks (subsequences) of length at least 2 and expected length at most $\log n$. Secondly, each block, say $b = a_i \cdots a_j$, 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 $p(a_i, p(a_{i+1}, \ldots, p(a_{j-1}, a_j)) \ldots)$. Afterward, the same rules are applied to the shrunken sequence until the sequence has length 1. The depth of nesting in this recursion is $O(\log n)$. Randomization is used to break a sequence into blocks. For each element of the sequence a random real number is chosen 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 only need to manipulate a constant 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. It is based on an algorithm for three-coloring rooted trees (we consider a sequence to be a rooted tree) by Goldberg *et al.* [GPS], which is a generalization of the so-called *deterministic coin-tossing technique* of Cole and Vishkin [CV]. We generate blocks of length at most 4 and decide for each index $i$ whether $a_i$ starts a new block by looking only at $O(\log^* m)$ neighbors of $a_i$. The update algorithms have a recursion depth of $O(\log n)$. On each level they have to manipulate a balanced tree of depth $O(\log n)$ spending $O(\log n)$ time. Furthermore, they have to handle $O(\log^* m)$ blocks spending $O(\log m)$ time for each.

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

**Table 2.** Incremental space cost.

| Randomized | Deterministic |
|---|---|
| $O(\log^2 n)$ | $O(\log n(\log n + \log^* m))$ |

representation of sequences based on the block decomposition, and in Section 4 we show how to realize the various operations.

**2. The Block Decomposition.** We first give the randomized block decomposition. Let $U$ be a universe and $s = a_1 \cdots a_n$ with $a_i \in U$ and $a_i \neq a_{i+1}$ for all $i$, $1 \leq i \leq n-1$. 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 $prio(a_1) \cdots prio(a_n)$ of priorities corresponding to $s$.

RANDOMIZED MARKING RULE. Every local minimum is marked.

Then at every marked position (and at position 1) a block starts. It ends just before the next marked position (the last block ends at position $n$).

We next give a deterministic construction to divide a sequence into suitable blocks. As mentioned above, the underlying algorithm is essentially a sequential version of the so-called *three-coloring technique for rooted trees* of Goldberg *et al.* [GPS] (which in turn is a generalization of the *deterministic coin-tossing technique* of Cole and Vishkin [CV]) and can be considered as a constructive proof of the following lemma.

LEMMA 1. *For every integer $N$ there is a function $f\colon [-1..N-1]^{\log^* N+11} \rightarrow \{0, 1\}$ such that for every sequence $a_1 \cdots a_n \in [0..N-1]^*$ with $a_i \neq a_{i+1}$ for all $i$ with $1 \leq i < n$ the sequence $d_1 \cdots d_n \in \{0, 1\}^*$ defined by $d_i := f(\tilde{a}_{i-\log^* N-6}, \ldots, \tilde{a}_{i+4})$, where $\tilde{a}_j = a_j$ for all $j$ with $1 \leq j \leq n$ 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 all* $i$, $1 \leq i < n-3$,

*i.e., among two adjacent $d_i$'s there is at most one* 1, *and among four adjacent $d_i$'s there is at least one* 1.

The sequence $d_1 \cdots d_n$ is used to decompose the sequence $a_1 \cdots a_n$ 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 4 and that all but the first and last block have length at least 2.

In the following subsection we review the three-coloring technique. Lemmas 2 and 3 are due to Goldberg *et al.* [GPS] but Lemma 4 is new. In Section 2.2 we explain the decomposition rule and show how one can derive Lemma 1 from the coloring algorithm.

2.1. *The Three-coloring Algorithm.* Let $s = a_1 \cdots a_n$ with $a_i \in [0..N-1]$ and $a_i \neq a_{i+1}$. We consider $s$ as a linked list (which is a special form of rooted tree). A *k-coloring* of a list is an assignment $C\colon \{a_1, \ldots, a_n\} \rightarrow \{0, \ldots, k-1\}$. A *valid* coloring is a coloring such that no two adjacent elements have the same color.

Informally it is done as follows. We first compute a valid $\lceil \log N \rceil$-coloring. Afterward 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 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," that means the color of an element must not depend on more than a small neighborhood of the element). The details are as follows:

Identify each $a_i$ (and its color) 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 $s = a_1 \cdots a_n$ 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 of $a_i$ differs from the old color of $a_{i-1}$ and the value of this bit. We use $C_i$ to denote the color of $a_i$ and $N_C$ denotes the number of used colors.

```
1.  Procedure Six-Colors($a_1 \cdots a_n$: sequence);
2.  begin
3.     $N_C \leftarrow N$;
4.     forall $i \in \{1, .., n\}$ do
5.        $C_i \leftarrow a_i$;
6.     od;
7.     while $N_C > 6$ do
8.        $C_1 \leftarrow C_1(0)$;
9.        forall $i \in \{2, \ldots, n\}$ do
10.          $j_i \leftarrow \min\{j | C_i(j) \neq C_{i-1}(j)\}$;
11.          $b_i \leftarrow C_i(j_i)$;
12.          $C'_i \leftarrow 2j_i + b_i$;
13.       od;
14.       $N_C \leftarrow \max\{C_i | i \in \{1, .., n\}\} + 1$;
15.    od;
16. end,
```

LEMMA 2. *The procedure Six-Colors produces a valid six-coloring of a list $a_1 \cdots a_n$ where $a_i \in \{0, .., N-1\}$ for all $i$, $1 \leq i \leq n$, 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 all $i$, $1 \leq i \leq n-1$. Now suppose $C$ is valid when we enter the while-loop (line 7). Consider two adjacent elements $a_i$ and $a_{i+1}$ for some $i$, $1 < i < n$. 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 in line 12 we 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 $j_1 = j_2$ but $C_i(j_1) \neq C_{i+1}(j_1)$ 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 give an upper bound on the number of iterations. Let $L = \lceil \log N \rceil$ and $L_k$ denote the number of bits in the longest representation of a color after $k$ iterations of the while-loop. We show that $L_k \leq \lceil \log^k L \rceil + 2$, if $\lceil \log^k L \rceil \geq 2$.

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

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

$$L_k \leq \lceil \log L_{k-1} \rceil + 1 \tag{1}$$

$$\leq \lceil \log(2 \log^{k-1} L) \rceil + 1 \tag{2}$$

$$\leq \lceil \log^k L \rceil + 2. \tag{3}$$

Here (1) follows from the fact that in line 10 $j_i \leq \lceil L_{k-1} \rceil$ and (2) holds by the induction hypothesis. After $\log^* N + 1$ iterations we have $\lceil \log^k L \rceil = 1$ and hence $L_k = 3$. 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, since each iteration takes time $O(n)$, the time bound follows. □

We can easily compute a valid three-coloring by the following procedure, which replaces each color $C_i \in \{3, 4, 5\}$ of an element $a_i$ by the smallest color in $\{0, 1, 2\}$ not being assigned to one of its neighbors.

```
1.  **Procedure** *Three-Colors*($a_1 \cdots a_n$: sequence);
2.  **begin**
3.    Six-Colors($a_1 \cdots a_n$);
4.    $C_0 \leftarrow \infty$;
5.    $C_{n+1} \leftarrow \infty$;
6.    **for** $c = 3$ **to** 5 **do**
7.      **forall** $i \in \{1, \ldots, n\}$ **do**
8.        **if** $C_i = c$ **then**
9.          $C_i \leftarrow \min\{\{0, 1, 2\} - \{C_{i-1}, C_{i+1}\}\}$;
10.       **fi**;
11.     **od**;
12.   **od**;
13. **end**;
```

LEMMA 3. *The procedure Three-Colors produces a valid three-coloring of a list* $a_1 \cdots a_n$ *where* $a_i \in \{0, \ldots, N-1\}$ *for all* $i$, $1 \leq i \leq n$ *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 6) 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 7–11 is obviously $O(n)$ and the time bound follows. □

2.2. *The Decomposition Rule.* For any sequence $a_1 \cdots a_n$ we define the sequence $d_1 \cdots d_n$ in the following way. We first compute a valid three-coloring by the procedure *Three-Colors* presented above and then set $d_i = 1$ iff the color of $a_i$ (which is now considered to be an integer in $\{0, 1, 2\}$) is a local maximum in the sequence of colors and $d_i = 0$ otherwise.

For technical reasons, we define the elements $a_i$ with $i < 1$ or $i > n$ to be *empty* elements that have no influence on the computation (in Lemma 1 these elements are written as $-1$).

LEMMA 4. *Given a sequence $a_1 \cdots a_n$, the values $d_1 \cdots d_n$ defined above have the following properties*:

1. $d_i + d_{i+1} \leq 1$ *for all* $i$, $1 \leq i < n$.
2. $d_i + d_{i+1} + d_{i+2} + d_{i+3} \geq 1$ *for all* $i$, $1 \leq i < n - 3$.
3. *The value of* $d_i$ *only depends on the subsequence* $a_{i-\log^* N-6} \cdots a_{i+4}$.

PROOF. Property 1 follows from the fact that in a valid coloring any two 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.

We prove property 3 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 only depends on the subsequence $a_{i-\log^* N-2} \cdots a_i$. More precisely, we argue that the color of $a_i$ after the $k$th iteration depends on the subsequence $a_{i-k} \cdots a_i$. (Remember that $k \leq \log^* N + 2$ (see Lemma 2).) However, 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 $i$, $1 \leq i \leq n$, the color of $a_i$ after the $(k-1)$th iteration depends on the subsequence $a_{i-k+1} \cdots a_i$. During the next iteration each element $a_i$ with $1 < i \leq n$ 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 $C_i$ depended on $a_{i-k+1} \cdots a_i$ and $C_{i-1}$ on $a_{i-k} \cdots a_{i-1}$, the new color depends on $a_{i-k} \cdots a_i$, and the induction step is completed.

Next we argue that for each $a_i$ the color computed by the procedure *Three-Colors* only depends on the subsequence $a_{i-\log^* N-5} \cdots a_{i+3}$. 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 $a_{i-\log^* N-2} \cdots a_i$ (as shown above). In each iteration the new color of an element depends on the old colors of its two neighbors. Since there are only three iterations, the color of $a_i$ in the six-coloring depends on the six-coloring of the elements

$a_{i-3} \cdots a_{i+3}$ and therefore on the elements $a_{i-\log^* N-5} \cdots a_{i+3}$. To complete the proof for property 3 note that the value of $d_i$ is set in dependence on the colors $C_{i-1}$, $C_i$, and $C_{i+1}$ and therefore of the subsequence $a_{i-\log^* N-6} \cdots a_{i+4}$. □

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. □

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 the Introduction we implement efficient equality tests by assigning unique signatures to sequences. In this section we explain how this is done and how sequences are represented. A signature is a small integer. More precisely, after $m$ operations there is no signature exceeding $m^3$. Since we maintain a hierarchy of sequences, i.e., signatures are also assigned to blocks and subsequences of shrunken sequences, we need more than $m$ signatures.

Each sequence $s$ can be uniquely written as $a_1^{l_1} \cdots a_n^{l_n}$ with $a_i \neq a_{i+1}$ for all $i$, $1 \leq i < n$, and all $l_i$ being positive integers, where $a_i^{l_i}$ denotes a subsequence of $l_i$ repetitions of the element $a_i$. Informally, a signature is assigned to a sequence $s = a_1^{l_1} \cdots a_n^{l_n}$ in the following way. Each element $a_i \in U$ gets a signature (this will be done by the function $\overline{sig}(s)$). In order to eliminate repetitions, to each power $a_i^{l_i}$ (for $1 \leq i \leq n$) a signature is assigned. Afterward we compute a block decomposition of the sequence $sig(a_1^{l_1}) \cdots sig(a_n^{l_n})$ according to the methods introduced 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. The resulting sequence is denoted by $shrink(s)$. Afterward the whole procedure is applied on $shrink(s)$, 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 $u$: $S_U \to U$, $p$: $S_P \to \{(a, b); a, b \in S$ and $a \neq b\}$ and $r$: $S_R \to \{(a, i); a \in S$ and $i \in \mathbb{N}, i \geq 2\}$. The inverse functions $u$, $p$, and $r$ are maintained as 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 piecemeal 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 give a constructive definition of the signature $sig(s)$ of a sequence $s$ =

$a_1^{l_1} \cdots a_n^{l_n}$ with $a_i \neq a_{i+1}$ for all $i$, $1 \leq i < n$ and $n \geq 1$. The functions *shrink*$(s)$ and $\overline{sig}(s)$ which are used in this definition are defined afterward.

The function *sig* is defined recursively. In all cases marked by $(*)$, *maxsig* is incremented and the corresponding function ($r$ in the definition of *sig*, and $u$ and $p$ in the definition of $\overline{sig}$) is extended.

$$sig(s) = \begin{cases} \overline{sig}(a_1) & \text{if } n = 1 \text{ and } l_1 = 1, \\ r^{-1}((a_1, l_1)) & \text{if } n = 1,\ l_1 > 1, \text{ and } (a_1, l_1) \in range(r), \\ maxsig + 1 & \text{if } n = 1,\ l_1 > 1, \text{ and } (a_1, l_1) \notin range(r), \quad (*) \\ sig(shrink(s)) & \text{if } n > 1. \end{cases}$$

Next we define the function *shrink*$(s)$. Let $n > 1$, then the function *elpow*$(s)$ (eliminate powers) is defined by

$$elpow(s) = sig(a_1^{l_1}) \cdots sig(a_n^{l_n}),$$

i.e., every power is replaced by its signature (which is defined above). We denote *elpow*$(s)$ by $g_1 \cdots g_n$ where $g_i = sig(a_i^{l_i})$ for all $i$, $1 \leq i \leq n$. Note that $g_i \neq g_{i+1}$ for all $i$, $1 \leq i < n$. Therefore, we can apply the block decomposition introduced in Section 2. Now let $b_1 \cdots b_k$ denote the block decomposition of *elpow*$(s)$, i.e., each $b_i$ for all $i$, $1 \leq i \leq k$, is a block. Then we define *shrink*$(s)$ by

$$shrink(s) = \overline{sig}(b_1) \cdots \overline{sig}(b_k).$$

Now note that if $\overline{sig}$ is defined for all sequences $s = a_1 \cdots a_n$ with $a_i \neq a_{i+1}$ for all $i$, $1 \leq i < n$ (i.e., $s$ contains no powers), then *sig* is completely defined.

$$\overline{sig}(s) = \begin{cases} a_1 & \text{if } n = 1 \text{ and } a_1 \in S, \\ u^{-1}(a_1) & \text{if } n = 1,\ a_1 \in U, \\ & \quad \text{and } a_1 \in range(u), \\ maxsig + 1 & \text{if } n = 1,\ a_1 \in U, \\ & \quad \text{and } a_1 \notin range(u), \quad (*) \\ p^{-1}((\overline{sig}(a_1), \overline{sig}(a_2))) & \text{if } n = 2 \text{ and} \\ & \quad (\overline{sig}(a_1), \overline{sig}(a_2)) \in range(p), \\ maxsig + 1 & \text{if } n = 2 \text{ and } (\overline{sig}(a_1), \overline{sig}(a_2)) \\ & \quad \notin range(p), \quad (*) \\ \overline{sig}(a_1, \overline{sig}(a_2, \ldots, \overline{sig}(a_{n-1}, a_n) \cdots)) & \text{if } n > 2. \end{cases}$$

In order to show the correctness 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. It is easy to see that each $s \in S$ encodes a unique sequence in $U^*$ by simply running the encoding process backward. □

We next explain how sequences are stored. As above, let $s = a_1^{l_1} \cdots a_n^{l_n}$, let *elpow*$(s) = sig(a_1^{l_1}) \cdots sig(a_n^{l_n})$, let $b_1 \cdots b_k$ be the sequence of blocks of *elpow*$(s)$, and finally let *shrink*$(s) = \overline{sig}(b_1) \cdots \overline{sig}(b_k)$.

Then we represent a sequence $s$ by a list of sequences $\bar{s} = (\tau_0 \cdots \tau_{2t})$ where $\tau_0 = s$ and for all $i$, $1 \leq i \leq t$, $\tau_{2i-1} = elpow(\tau_{2i-2})$ and $\tau_{2i} = shrink(\tau_{2i-2})$. Note that $t = O(\log n)$ in both schemes, since blocks (except maybe the first and the last) have length at least 2 in both schemes.

In order to support the operations we store each $\tau_j$ as a balanced binary tree $T_j$ in such a way that the symmetric traversal of $T_j$ yields $\tau_j$. Each node $v$ contains:

- An element $a$ of $\tau_j$.
- The size of the subtree rooted at $v$.
- The length of the block corresponding to $a$ in $\tau_{j-1}$.
- The sum of the lengths of the blocks corresponding to the elements stored in the subtree rooted at $v$.
- The mark of the element $a$ if $j$ is odd.

Section 4 explains how this information is used. Each $\bar{s}$ is maintained as a linked list of the roots of the trees $T_j$. $\mathcal{F}$ 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 [DKM$^+$]) and in the deterministic solution they are maintained as balanced binary trees. The operations are performed persistently such that none of the sequences is destroyed (see [DSST] 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*$(s_1, s_2)$ can be implemented by returning *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 $sig(a)$ by evaluating $u(a)$ 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 depends only on 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.* The input is the hierarchical representations of sequences $s_1$ and $s_2$ and we need to compute the hierarchical representation of $s_3 = s_1 s_2$. The following lemma paves the way. It states that if we join the suitable trees of the hierarchical representations of $s_1$ and $s_2$ to perform the concatenation, then for each tree only a small neighborhood of the concatenation position differs from the correct tree for the hierarchical representation of $s_3$ (a corresponding statement holds for the reverse operation split). Therefore, for each tree of $\bar{s}_3$ only a small middle part has to be recomputed.

LEMMA 6. *Let $s_1 = a_1 \cdots a_l$, $s_2 = a_{l+1} \cdots a_n$, and $s_3 = s_1 s_2$ be sequences and let $j \geq 0$ be an integer. Let $shrink^j(s_3) = c_1 \cdots c_r$, i.e., $c_1 \ldots c_r$ is the result of applying the shrink operation $j$ times, and let $i$ be such that $c_i$ encodes the subsequence of $s_3$ containing $a_l$. Then*:

1. *$c_1 \cdots c_{i-5}$ is a prefix of $shrink^j(s_1)$ and $|shrink^j(s_1)| \leq i + 5$.*
2. *$c_{i+4} \ldots c_r$ is a suffix of $shrink^j(s_2)$ and $|shrink^j(s_2)| \leq r - i + 7$.*

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 $j + 1$.

We denote $shrink^{j+1}(s_3)$ by $c'_1 \cdots c'_{r'}$, where $c'_{i'}$ encodes the subsequence of $s_3$ containing $a_l$ and $elpow(shrink^j(s_3))$ by $g_1 \cdots g_k$, where $g_z$ encodes the subsequence of $s_3$ containing $a_l$. By the induction hypothesis we have $shrink^j(s_1) = c_1 \cdots c_{i-5} e_1 \cdots e_p$ and $shrink^j(s_2) = f_1 \cdots f_q c_{i+4} \cdots c_r$ with $p, q \leq 10$. Then the subsequence encoded by $g_1 \cdots g_{z-6}$ is a proper prefix of that encoded by $c_1 \cdots c_{i-5}$ and the subsequence of $g_{z+5} \cdots g_k$ is a proper suffix of that encoded by $c_{i+4} \cdots c_r$. Since the marks are influenced by at most one predecessor and one successor (by the definition of "local minimum"), the marks of the sequences $g_1 \cdots g_{z-7}$ and $g_{z+5} \cdots g_k$ are identical to those of the corresponding elements in $elpow(shrink^j(s_1))$ and $elpow(shrink^j(s_2))$. Since every block has size at least 2 it follows that the subsequence $c'_{i'-4} \cdots c'_{i'+3}$ encodes the subsequence of $elpow(shrink^j(s_3))$ containing $g_{z-7} \cdots g_{z+6}$. Thus $c'_1 \cdots c'_{i'-5}$ exclusively depends on $c_1 \cdots c_{i-5}$ and therefore is a prefix of $shrink^{j+1}(s_1)$ and $c'_{i'+4} \cdots c'_{r'}$ exclusively depends on $c_{i+4} \cdots c_r$ and therefore is a suffix of $shrink^{j+1}(s_2)$.

Let $elpow(shrink^j(s_1))$ be denoted by $g_1 \cdots g_{z-6} g'_1 \cdots g'_y$ and $shrink^{j+1}(s_1)$ by $c'_1 \cdots c'_{i'-5} e'_1 \cdots e'_{p'}$. Note that the sequence $c'_{i'-4} \cdots c'_{i'}$ encodes a sequence $g_{z-x} \cdots g_{z-6} \cdots g_z$ and the sequence $e'_1 \cdots e'_{p'}$ encodes a sequence $g_{z-x} \cdots g_{z-6} g'_1 \cdots g'_y$ where $y \leq p + 1$. $g_{z-x} \cdots g_{z-7}$ is encoded by at most four elements (then $c'_{i'-4} \cdots c'_{i'-1} = e'_1 \cdots e'_4$). $g_{z-6} g'_1 \cdots g'_y$ 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 tells us that all but a small middle part of $shrink^j(s_3)$ can be copied from $shrink^j(s_1)$ 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 $elpow(shrink^{j-1}(s_3))$: Let $elpow(shrink^{j-1}(s_3)) = g_1 \cdots g_k$ and let $g_z$ be the element encoding the subsequence of $s_3$ containing $a_l$. The marks of the elements $g_1 \cdots g_{z-7}$ and $g_{z+6} \cdots g_k$ are identical to the corresponding marks in $elpow(shrink^{j-1}(s_1))$ and $elpow(shrink^{j-1}(s_2))$. We compute new marks for the elements $g_{z-6} \cdots g_{z+5}$. Afterward we can compute $shrink^j(s_3)$ by computing the middle part $c_{i-4} \cdots c_{i+3}$ and copying the other parts from $shrink^j(s_1)$ and $shrink^j(s_2)$. The split operations on the corresponding trees can easily be performed in $O(\log n)$ each: we know the length of those subsequences of $shrink^{j-1}(s_1)$ and $shrink^{j-1}(s_2)$, for which we want to copy the encoding subsequences of $shrink^j(s_1)$ and $shrink^j(s_2)$. Note that in every node $v$ in the trees $T_{shrink^j(s_1)}$ and $T_{shrink^j(s_2)}$ the length of the block corresponding to $v$ (resp. to its element) as well as the sum of the lengths of the blocks corresponding to the nodes in the subtree rooted at $v$ are stored. Therefore, it suffices to visit a single path to split the tree.

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 [DSST].

In the following, $s_1 = a_1 \cdots a_l$, $s_2 = a_{l+1} \cdots a_n$, let $s$ be any sequence, $elpow(s) = g_1 \cdots g_k$, and $T_s$ is the balanced binary tree for $s$.

```
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 $T_{elpow(s_3)}$ by joining $T_{elpow(s_1)}$ and $T_{elpow(s_2)}$ (in the case that
   $a_l = a_{l+1}$ recompute the corresponding element of $elpow(s_3)$).
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
   (a) Append $s$ and $elpow(s)$ at the end of the list $\bar{s}_3$.
   (b) Choose (or retrieve) the priorities of the subsequence $g_{z-5} \cdots g_{z+4}$
       of $g(s)$ and compute the marks of $g_{z-6} \cdots g_{z+4}$ in $T_{elpow(s)}$
       accordingly.
   (c) Assign $shrink(s)$ to $s$, where $shrink(s)$ is computed as indicated
       above.
   (d) Compute $T_s$. If $|s| > 1$, then compute $T_{elpow(s)}$ and update $z$.
5. 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 4(c) and 4(d) 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 4(b) needs time $O(\bar{l})$. In line 4(c) 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 4(c) needs time $O(l)$ to retrieve or create the signatures. Note that priorities are only assigned to those signatures being elements of a sequence $g(s)$ (see line 4(b)). Line 4(d) 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 $E[l] \leq 2E[l']$.

We estimate $E[l']$. Suppose that $s = a_1 \cdots a_k$ and let $j$ and $t$ be positive integers.

$$Pr[[prio(a_j) < prio(a_{j+1}) < \cdots < prio(a_{j+t-1})] = 1/t!$$

and so

$$Pr[\exists j : prio(a_j) < prio(a_{j+1}) < \cdots < prio(a_{j+t-1})] \leq k/t!$$

Hence,

$$\begin{aligned} E[l'] &\leq \lceil \log k \rceil + \sum_{t=\lceil \log k \rceil + 1}^{k} k/t! \\ &\leq \lceil \log k \rceil + 1 \end{aligned}$$

and

$$E[l] \leq 2\lceil \log k \rceil + 2 \leq 2\lceil \log |s| \rceil + 2. \qquad \square$$

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 *maxsig* 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 let *Prio* be 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

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

and 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

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

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. $\square$

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

In the following $s$ and $s'$ denote sequences, $elpow(s) = g_1 \cdots g_p$, and $elpow(s') = h_1 \cdots h_q$.

```
Procedure RanSplit($s_1, s_2, s_3$: sequence; $i$: integer);
  1. Compute $T_{s_2}$, $T_{s_3}$, $T_{elpow(s_2)}$, and $T_{elpow(s_3)}$.
  2. Let $s = s_2$, $s' = s_3$, and let $\bar{s_2}$ and $\bar{s_3}$ be empty lists.
  3. while $|s| > 1$ do
     (a) Choose the priorities of the sequence $g_{p-12} \cdots g_p$ if necessary;
         compute the marks of $g_{p-13} \cdots g_p$ in $T_{elpow(s)}$ according to the
         randomized marking rule.
     (b) Append $s$ and $elpow(s)$ at the end of the list $\bar{s_2}$.
     (c) Let $s = shrink(s)$, where $shrink(s)$ is computed as indicated above.
     (d) Compute $T_s$; if $|s| > 1$ compute $T_{elpow(s)}$.
  4. while $|s'| > 1$ do
     (a) Choose the priorities of the sequence $h_1 \cdots h_{12}$ if necessary; com-
         pute the marks of $h_1 \cdots h_{13}$ in $T_{elpow(s')}$ according to the randomized
         marking rule.
     (b) Append $s'$ and $elpow(s')$ at the end of the list $\bar{s_3}$.
     (c) Let $s' = shrink(s')$, where $shrink(s')$ is computed as explained
         above.
     (d) Compute $T_{s'}$; if $|s'| > 1$ compute $T_{elpow(s')}$.
  5. Append $s$ at the end of $\bar{s_2}$ and $s'$ at the end of $\bar{s_3}$.
```

The complexity of the *Split* operation is given by

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

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

The proof is completely analogous to that of Lemma 6. The computation of $shrink(s_3)$ is done as follows: we denote $elpow(shrink^{j-1}(s_3))$ by $g_1 \cdots g_k$ and $g_z$ is the element encoding the subsequence of $s_3$ containing $a_l$. The marks of the elements $g_1 \cdots g_{z-13}$ and $g_{z+2\log^* m^3+17} \cdots g_k$ are identical to the corresponding marks in $elpow(shrink^{j-1}(s_1))$ and $elpow(shrink^{j-1}(s_2))$. To compute new marks for the elements $g_{z-12} \cdots g_{z+2\log^* m^3+16}$ we run the algorithm *Three-Colors* on the subsequence $g_{z-\log^* m^3-18} \cdots g_{z+2\log^* m^3+20}$ since at most these elements have influence on the missing marks. Afterward we can compute $shrink^j(s_3)$ by computing the middle part $c_{i-7} \cdots c_{i+\log^* m^3+10}$ 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 $s_1 = a_1 \cdots a_l$, $s_2 = a_{l+1} \cdots a_n$, let $s$ be any sequence, $elpow(s) = g_1 \cdots g_k$ and $T_s$ is the balanced binary tree for $s$.

```
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 $T_{elpow(s_3)}$ by joining $T_{elpow(s_1)}$ and $T_{elpow(s_2)}$ (in the case that
     $a_l = a_{l+1}$ recompute the corresponding element of $elpow(s_3)$).
  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
     (a) Append $s$ and $elpow(s)$ at the end of the list $\bar{s_3}$.
     (b) Run Three-Colors($g_{z-\log^* m^3-18} \dots g_{z+2\log^* m^3+20}$) and change the
         marks of $g_{z-12} \dots g_{z+2\log^* m^3+16}$ accordingly.
     (c) Assign $shrink(s)$ to $s$, where $shrink(s)$ is computed as indicated
         above.
     (d) Compute $T_s$. If $|s| > 1$, then compute $T_{elpow(s)}$ and update $z$.
  5. 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* $O(\log n(\log n + \log^* m))$.

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

Furthermore, lines 1 and 2 require time $O(\log n)$. Computing the new marks (line 4(b)) 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 4(b). 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 computing *shrink*$(s)$ (line 4(c)) 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 line 4(d) 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 the lemma follows. □

In the following $s$ and $s'$ denote sequences, $elpow(s) = g_1 \cdots g_p$, and $elpow(s') = h_1 \ldots h_q$.

```
Procedure DetSplit($s_1, s_2, s_3$: sequence; $i$: integer);
  1. Compute $T_{s_2}$, $T_{s_3}$, $T_{elpow(s_2)}$ and $T_{elpow(s_3)}$.
  2. Let $s = s_2$, $s' = s_3$, and let $\bar{s_2}$, $\bar{s_3}$ be empty lists.
  3. while $|s| > 1$ do
     (a) Run Three-Colors($g_{p-\log^* m^3-26} \cdots g_p$) and change the marks of
         $g_{p-20} \cdots g_p$ in $T_{elpow(s)}$ accordingly.
     (b) Append $s$ and $elpow(s)$ at the end of the list $\bar{s_2}$.
     (c) Let $s \leftarrow shrink(s)$, where $shrink(s)$ is computed as indicated
         above.
     (d) Compute $T_s$; if $|s| > 1$ compute $T_{elpow(s)}$.
  4. while $|s'| > 1$ do
     (a) Run Three-Colors($h_1 \cdots h_{3\log^* m^3+30}$) and change the marks of
         $h_1 \cdots h_{3\log^* m^3+26}$ in $T_{elpow(s')}$ accordingly.
     (b) Append $s'$ and $elpow(s')$ at the end of the list $\bar{s_3}$.
     (c) Let $s' \leftarrow shrink(s')$, where $shrink(s')$ is computed as indicated
         above.
     (d) Compute $T_{s'}$; if $|s'| > 1$ compute $T_{elpow(s')}$.
  5. 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* $O(\log n(\log n + \log^* m))$

The proof is analogous to that of Lemma 12.

## References

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

[DKM$^+$] 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*, pp. 524–531, 1988.

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

[GPS] A. V. Goldberg, S. A. Plotkin, and G. E. Shannon Parallel symmetry-breaking in sparse graphs. *SIAM J. Discrete Math.*, 1(4):434–446, 1988.

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

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

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