A Space and Time Efficient Algorithm for Constructing Compressed Suffix Arrays

Tak-Wah Lam[1], Kunihiko Sadakane[2], Wing-Kin Sung[3]*, and Siu-Ming Yiu[1]

[1] Department of Computer Science, University of Hong Kong
Hong Kong
{twlam, smyiu}@csis.hku.hk

[2] Department of System Information Sciences
Graduate School of Information Sciences, Tohoku University
Sendai, Japan
sada@dais.is.tohoku.ac.jp

[3] Department of Computer Science, National University of Singapore
Singapore
ksung@comp.nus.edu.sg

Abstract. With the first Human DNA being decoded into a sequence of about 2.8 billion base pairs, many biological research has been centered on analyzing this sequence. Theoretically speaking, it is now feasible to accommodate an index for human DNA in main memory so that any pattern can be located efficiently. This is due to the recent breakthrough on compressed suffix arrays, which reduces the space requirement from $O(n \log n)$ bits to $O(n)$ bits. However, constructing compressed suffix arrays is still not an easy task because we still have to compute suffix arrays first and need a working memory of $O(n \log n)$ bits (i.e., more than 13 Gigabytes for human DNA). This paper initiates the study of constructing compressed suffix arrays directly from text. The main contribution is a new construction algorithm that uses only $O(n)$ bits of working memory, and more importantly, the time complexity remains the same as before, i.e., $O(n \log n)$.

1 Introduction

DNA sequences, which hold the code of life for living organisms, can be represented by strings over four characters A, C, G, and T. With the advance in bio-technology, the complete DNA sequences for a number of living organisms have been known. Even for human DNA, a draft which comprises about 2.8 billion characters, has been finished recently. This is, however, not the end, the next step is to analyse these sequences.

This paper is concerned with data structures for indexing a sequence over a fixed alphabet so that searching for an arbitrary pattern can be performed

* This research was supported in part by NUS Academic Research Grant R-252-000-119-112

efficiently. Such tools are very useful in many areas. In particular, they find applications to many biological research activities on DNA, such as gene hunting, promoter consensus identification, and motif finding. Unlike English text, DNA sequences do not have word boundaries; suffix trees [9] and suffix arrays [8] are the most appropriate solutions in the literature for indexing DNA. For a DNA sequence with n characters, building a suffix tree takes $O(n)$ time, then a pattern P can be located in $O(|P| + occ)$ time, where occ is the number of occurrences. For suffix arrays, construction and searching takes $O(n \log n)$ time and $O(|P| \log n + occ)$ time, respectively. Both data structures requires $O(n \log n)$ bits; suffix arrays is associated with a smaller constant, though. For human DNA, the best known implementation of suffix tree and suffix array requires 40 Gigabytes and 13 Gigabytes, respectively [7]. Such memory requirement far exceeds the capacity of ordinary computers. Existing approaches for indexing human DNA include (1) using supercomputers with large main memory; and (2) storing the indexing data structure in the secondary storage [1, 6]. The first approach is expensive and inflexible, while the second one is slow. As more and more DNA are decoded, it is vital that individual biologists can eventually analyze different DNA sequences efficiently with their ordinary PCs.

Recent breakthrough results in compressed suffix arrays shed light on this direction [4, 5, 10]. It is now feasible to store a compressed suffix array in the main memory, which occupies only $O(n)$ bits, yet still supporting search efficiently. Precisely, the searching time increases only by a factor of at most $\log n$. For human DNA, such a compressed suffix array occupies about 2 Gigabytes. Nowadays a PC can have up to 4 Gigabytes of main memory and can easily accommodate such a data structure.

Theoretically speaking, a compressed suffix array can be constructed in $O(n \log n)$ time; however, the construction process requires much more than $O(n)$ bits of working memory. Among others, the original suffix array has to be built first, taking up at least $n \log n$ bits. In the context of human DNA, the working memory for constructing a compressed suffix array is at least 25 Gigabytes [11], far exceeding the capacity of ordinary PCs. This motivates us to investigate whether we can construct a certain kind of compressed suffix array in $O(n \log n)$ time while using $O(n)$ bits of memory. The latter requirement means construction directly from DNA sequences. This paper provides the first algorithm of such a kind, showing that the compressed suffix array proposed in [5, 10] can be built in a space and time efficient manner.

Experiments show that for human DNA, our space-efficient algorithm can run on a PC with 3 Gigabytes of memory and takes about 21 hours, i.e., about three times slower than the original algorithm implemented on a supercomputer with 64 Gigabytes of main memory [11] to accommodate the suffix array.

Technically speaking, our algorithm does not require much space other than for storing the compressed suffix array. This is based on an observation that the compressed suffix arrays of two consecutive suffixes are very similar. Thus, we can build the entire compressed suffix array directly from the DNA sequence in an incremental manner. The efficiency of our algorithm benefits from an elegant solution to an interesting data structure problem, which extends a balanced search tree to support selective increment over an arbitrary interval of up to n values in $O(\log n)$ time.

The rest of this paper is organized as follows. Section 2 gives a review of suffix arrays and compressed suffix arrays. Section 3 gives the technical background and a simple framework for computing the compressed suffix array incrementally. Sections 4 and 5 respectively detail the new data structure to support the incremental construction and the constructing algorithm.

2 Compressed suffix arrays in a nutshell

Let Σ be an alphabet, and let \$ be a special character not in Σ. We assume that \$ is lexicographically smaller than any character in Σ. Intuitively, \$ is used to mark the end of a text. Consider a length-n text T, which is represented by an array $T[1..n+1] = T[1]T[2]\dots T[n+1]$, where $T[n+1] = \$$. For $i = 1, 2, \dots, n+1$, $T_i = T[i..n+1] = T[i]T[i+1]\dots T[n+1]$ denotes a suffix of T starting from the i-th position.

i	$T[i]$	T_i
1	a	$acaaccg\$$
2	g	$caaccg\$$
3	a	$aaccg\$$
4	a	$accg\$$
5	g	$ccg\$$
6	g	$cg\$$
7	c	$g\$$
8	\$	\$

i	$SA[i]$	$T_{SA[i]}$
0	8	\$
1	3	$aaccg\$$
2	1	$acaaccg\$$
3	4	$accg\$$
4	2	$caaccg\$$
5	5	$ccg\$$
6	6	$cg\$$
7	7	$g\$$

i	$\Psi[i]$	$T[SA[i]]$
1	3	a
2	4	a
3	5	a
4	1	c
5	6	c
6	7	c
7	0	g

Fig. 1. Array, suffix array, and compressed suffix array of $acaaccg\$$

Suffix Arrays [8]: A suffix array is a sorted sequence of $n+1$ suffixes of T, denoted by $SA[0..n]$. Formally, $SA[0..n]$ is a permutation of the set of integers $\{1, 2, \dots, n+1\}$ such that, according to the lexicographic order, $T_{SA[0]} < T_{SA[1]} < \dots < T_{SA[n]}$. See Figure 1 for an example. Note that $SA[0] = n+1$. As each integer takes $\log n$ bits, the suffix array can be stored using $(n+1)\log n$ bits.[4] Given a text T together with the suffix array $SA[1..n]$, the occurrences of any pattern P in T can be found without scanning T again. Precisely, it takes $O(|P|\log n + occ)$ time, where occ is the number of occurrences.

Compressed Suffix Arrays [5, 10]: For every $i = 0, 1, 2, \dots, n+1$, define $SA^{-1}[i]$ to be the integer j such that $SA[j] = i$. The major component of a compressed suffix array for a text T is an array $\Psi[1..n]$ where $\Psi[i] = SA^{-1}[SA[i]+1]$ for $i = 1, 2, \dots, n$. See Figure 1 for an example.

[4] Throughout this paper, we assume that logarithm has a base of 2.

Note that $\Psi[1..n]$ contains n integers. The trivial way to store it requires $n \log n$ bits. Nevertheless, $\Psi[1..n]$ can always be partitioned into $|\Sigma|$ strictly increasing sequences. This special property allows us to store it succinctly. This property is illustrated in the rightmost table in Figure 1 and its correctness is proved formally based on the following two lemmas.

Lemma 1. *For every $i < j$, if $T[SA[i]] = T[SA[j]]$, then $\Psi[i] < \Psi[j]$.*

Proof. Note that $i < j$ if $T_{SA[i]} < T_{SA[j]}$. If $i < j$ and $T[SA[i]] = T[SA[j]]$, then $T_{SA[i]+1} < T_{SA[j]+1}$. Let $p = \Psi[i]$ and $q = \Psi[j]$. Then $T_{SA[p]} = T_{SA[i]+1} < T_{SA[j]+1} = T_{SA[q]}$. Therefore, we have $p < q$. The lemma follows. □

Lemma 2. *$\Psi[1..n]$ can be partitioned into at most $|\Sigma|$ strictly increasing sequences S_c for all $c \in \Sigma$. For every integer $\Psi[i] \in S_c$, $T[SA[i]] = c$.*

Proof. Note that $[1..n]$ can be partitioned into at most $|\Sigma|$ intervals $[1 = i_1..j_1]$, $[i_2..j_2], \ldots, [i_{|\Sigma|}..j_{|\Sigma|} = n]$ so that for every i, j in a particular interval, $T[SA[i]] = T[SA[j]]$.

Based on Lemma 1, for every $i < j$ within a particular interval $[i_k..j_k]$, $\Psi[i] < \Psi[j]$. Thus, $\Psi[i_k..j_k]$ is a strictly increasing sequence. The lemma follows. □

For DNA sequences, Σ has only four characters. Based on Lemma 2, Ψ is partitioned into four strictly increasing sequences. To represent an increasing sequence, we can store the difference between consecutive values instead of storing the actual values. Precisely, a sequence $(v_1, \ldots, v_l)$ of increasing values can be represented as $v_1, v_2 - v_1, v_3 - v_2, \ldots, v_l - v_{l-1}$. Furthermore, each individual difference is encoded using δ-coding [3], which is for representing integers with variable length. In summary, it is shown that Ψ can be represented using $O(n)$ bits. More importantly, with Ψ and some other auxiliary data structures, the occurrences of any pattern P in the text T can be found in $O(|P| \log n)$ time. See [10] for the algorithmic details.

3 Basic Properties and Incremental Construction

The existing approach to constructing the array Ψ is as follows: First, the suffix array SA is built. Next, the values of the array $\Psi[1..n]$ are computed one by one. Finally, based on δ-coding, a succinct representation of Ψ is generated. Since suffix array requires $n \log n$ bits to store, the working space for constructing a suffix array is at least $n \log n$ bits.

To reduce the working space required, we try to avoid creating the suffix array. Our construction algorithm works directly with the DNA sequence and is based on an incremental approach. Initially, we construct the compressed suffix array for only the last character of the text. Then, we repeatedly expand the text character by character and build a bigger and bigger compressed suffix array.

Let us give the mathematical background behind such an incremental approach. Suppose we are given the compressed suffix array for the text $T[1..n+1]$ where $T[n+1] = \$$. Let $SA_T[0..n]$ be the suffix array of T. Note that $SA_T[0] = n+1$. Let $\Psi_T[1..n]$ be the compressed suffix array where $\Psi_T[i] = SA_T^{-1}[SA_T[i]+1]$ for $i = 1, 2, \ldots, n$.

Suppose a character c is added to the beginning of $T[1..n+1]$, giving us a longer text $T'[1..n+2]$, which is equal to $cT[1..n+1]$. Let $SA_{T'}[0..n+1]$ be the suffix array of T'. Let $\Psi_{T'}[1..n+1]$ be the compressed suffix array where $\Psi_{T'}[i] = SA_{T'}^{-1}[SA_{T'}[i]+1]$ for $i = 1, 2, \ldots, n+1$.

Let us have a close look of the relationship between $T[1..n+1]$ and $T'[1..n+2]$. Suppose we search T' against the suffixes of T according to the order specified by SA_T. Let $x \in \{0, \cdots, n\}$ be the integer such that $T[SA_T[x]..n+1] < T' < T[SA_T[x+1]..n+1]$. The following lemma states the relationship between $SA_T[0..n]$ and $SA_{T'}[0..n+1]$. Figure 2 shows an example to illustrate the relationship between SA_T and $SA_{T'}$.

i	$\Psi_T[i]$	$SA_T[i]$	$T_{SA_T[i]}$
0		8	\$
1	3	3	*aaccg\$*
2	4	1	*acaaccg\$*
3	5	4	*accg\$*
4	1	2	*caaccg\$*
5	6	5	*ccg\$*
6	7	6	*cg\$*
7	0	7	*g\$*

i	$\Psi_{T'}[i]$	$SA_{T'}[i]$	$T'_{SA_{T'}[i]}$
0		$8 \Rightarrow 9$	\$
1	3	$3 \Rightarrow 4$	*aaccg\$*
2	4	$1 \Rightarrow 2$	*acaaccg\$*
3	$5 \Rightarrow 6$	$4 \Rightarrow 5$	*accg\$*
4	1	$2 \Rightarrow 3$	*caaccg\$*
5	$SA_T^{-1}[1] = 2$	1	*cacaaccg\$*
5	$6 \Rightarrow 7$	$5 \Rightarrow 6$	*ccg\$*
6	$7 \Rightarrow 8$	$6 \Rightarrow 7$	*cg\$*
7	0	$7 \Rightarrow 8$	*g\$*

Fig. 2. Suffix array and compressed suffix array of $T = acaaccg\$$ and $T' = cT$. Note that $x = 4$, that is, $T_{SA_T[x]} < T' < T_{SA_T[x+1]}$.

Lemma 3.

$$SA_{T'}[i] = \begin{cases} SA_T[i]+1 & \text{if } 0 \le i \le x \\ 1 & \text{if } i = x+1 \\ SA_T[i-1]+1 & \text{if } x+2 \le i \le n+1 \end{cases}$$

The above lemma states that $SA_{T'}[x+1] = 1$. Thus, $x = SA_{T'}^{-1}[1] - 1$. The next lemma states the relationship between $\Psi_T[1..n]$ and $\Psi_{T'}[1..n+1]$. See Figure 2 for an example.

Lemma 4. *For* $i \le x$,

$$\Psi_{T'}[i] = \begin{cases} \Psi_T[i] & \text{if } \Psi_T[i] \le x \\ \Psi_T[i]+1 & \text{if } \Psi_T[i] > x \end{cases}$$

For $i = x+1$,

$$\Psi_{T'}[i] = \begin{cases} SA_T^{-1}[1] & \text{if } SA_T^{-1}[1] \le x \\ SA_T^{-1}[1]+1 & \text{if } SA_T^{-1}[1] > x \end{cases}$$

For $i \ge x+2$,

$$\Psi_{T'}[i] = \begin{cases} \Psi_T[i-1] & \text{if } \Psi_T[i-1] \le x \\ \Psi_T[i-1]+1 & \text{if } \Psi_T[i-1] > x \end{cases}$$

Lemma 4 suggests the following algorithm to compute $\Psi_{T'}$ from Ψ_T.

Incremental Construction Algorithm:

Input: $\Psi_T[1..n]$ for $T[1..n+1]$, $SA_T^{-1}[1]$, and $T' = cT$ for some $c \in \Sigma$.
Output: $\Psi_{T'}$ for T' and $SA_{T'}^{-1}[1]$.

1. Locate the index x such that T' should be inserted between the suffices represented by $SA_T[x]$ and $SA_T[x+1]$.
2. Construct $\Psi_{T'}$ from Ψ_T by inserting the new suffix cT into Ψ and incremeting those values that are greater x as follows.
   - set $\Psi_{T'}[j] = \begin{cases} \Psi_T[j] & \text{if } j \le x \\ SA_T^{-1}[1] & \text{if } j = x+1 \\ \Psi_T[j-1] & \text{if } j > x+1 \end{cases}$
   - for each $\Psi_{T'}[j]$, if $\Psi_{T'}[j] > x$, increment $\Psi_{T'}[j]$ by 1.
3. Set $SA_{T'}^{-1}[1] = x+1$.

The correctness of the above algorithm follows directly from Lemma 4. By executing the incremental construction algorithm for n times, the compressed suffix array for a length-n text can be built. Observe that each incremental construction step can be implemented in $O(n)$ time. Therefore, based on the compressed suffix array data structure in Section 2, we can construct the compressed suffix array Ψ_T for $T[1..n+1]$ using $O(n^2)$ time in $O(n)$ bits working space. This time complexity seems to be tight as each step may be required to increment $\Psi_{T'}$ for $\Omega(n)$ entries. Yet we show in the following section that the time complexity can be improved to $O(n \log n)$ without increasing the space complexity.

4 New data structure for compressed suffix arrays

This section proposes a new data structure $\mathcal{D}$ for representing the compressed suffix array, which allows us to construct it in $O(n)$ bits working space while the time required remains $O(n \log n)$. The basic idea is to store the compressed suffix array in a balanced search tree like red-black tree. Furthermore, to save space, every node in the tree represents more than one value (standard trick). To support the selective increment operation efficiently as required in the previous section (see Lemma 6), each node stores relative search key instead of abslute search key. The details will be discussed in the rest of this section.

Recall from Lemma 2, $\Psi[1..n]$ can be partitioned into $|\Sigma|$ strictly increasing sequences $\{S_c | c \in \Sigma\}$. Note that $T[SA[i]] = c$ for all $\Psi[i] \in S_c$. Let $|S_c|$ be the number of integers in sequence S_c.

For a particular value $x \in S_c$, denote $rank(x)$ be its rank when the number in S_c is ordered in increasing order. The next lemma shows the relationship between the rank of S_c and the index of the Ψ function.

Lemma 5. *For $x \in S_c$, $\Psi^{-1}[x] = \sum_{d \in \Sigma, d<c} |S_d| + rank(x)$.*

Proof. Let $S' = \{y \in S_c \mid rank(y) < rank(x)\}$. We have $\Psi^{-1}[y] < \Psi^{-1}[x]$ if and only if $y \in S' \cup \bigcup_{d \in \Sigma, d<c} S_d$. Therefore, $\Psi^{-1}[x] = 1 + |S'| + |\bigcup_{d \in \Sigma, d<c} S_d|$. The lemma follows. □

Now, we describe the modified data structure $\mathcal{D}$ for the compressed suffix array. For each $c \in \Sigma$, the increasing sequence S_c is partitioned into segments of length $5 \log n$ up to $10 \log n$. Let the smallest value in each segment be the representative of the segment. To save space, for each segment, instead of storing the value of every integer, only their differences will be stored. More precisely, consider a segment with integers $v_1, \ldots, v_\ell$ where $v_1 < v_2 < \ldots < v_\ell$. v_1 is denoted as the representative of the segment and the data structure $\mathcal{D}$ stores only the differences $v_2 - v_1, v_3 - v_2, \ldots,$ and $v_\ell - v_{\ell-1}$.

For the space required, all segments can be stored using $5n$-bits. For the representatives, we store them in a modified red-black tree R_c. For every node w in R_c, $v(w)$ is denoted as the value of the representative represent by this node. $lp(w)$ is the first parent on the left when we go up the tree R_c starting from w; if such a parent does not exist, let it be the node u such that $v(u)$ is minimized.

Every node w in the modified red-black tree R_c has the following 6 fields:

- a color bit (red or black)
- a pointer to the left subtree.
- a pointer to the right subtree.
- a pointer to the segment represented by w.
- $size(w)$, that is the total number of values represented by the subtree rooted at w.
- $d_w = v(w) - v(lp(w))$.

Although we do not store the value $v(w)$ for every node $w \in R_c$, its value can be recovered when we traverse down the tree R_c starting from the root. More precisely, when we traverse down the tree R_c, for every node w we met on the path, we can compute two information: $lp(w)$ and $v(w)$ in constant time.

Consider any node w of R_c. Suppose that for every node w' on the path from the root to w, we have already figured out the values of $lp(w')$ and $v(w')$. Let w_l and w_r be the left and right children of w, respectively. Note that the values of $lp(w_l), lp(w_r), v(w_l), v(w_r)$ can be computed as follows: When we go down to the node w_l, $lp(w_l) = lp(w)$ and $v(w_l) = d_{w_l} + v(lp(w_l))$. When we go down to the node w_r, $lp(w_r) = w$ and $v(w_r) = d_{w_r} + v(lp(w_r))$.

Observe that R_c looks very similar to a red-black tree. Thus, it has all the advantages of balanced binary search tree. In addition, it has the following property.

Lemma 6. *For any value x, let X be $\{w \in R_c \mid v(w) > x\}$. We can increase the value $v(w)$ for all $w \in X$ by some constant k using $O(\log n)$ time.*

Proof. Note that the absolute value of $v(w)$ is not stored for every node w in R_c. Instead, we store $d_w = v(w) - v(lp(w))$. This means that if the value of $lp(w)$ is increased by k, the value of w increases by k automatically.

Consider any node $w \in X$. There are two cases.

- Case 1. $v(lp(w)) > x$: In this case, since $lp(w) \in X$ and its value will be increased by k, the value of w will increase by k automatically.
- Case 2. $v(lp(w)) \leq x$: In this case, the value of $lp(w)$ will not be changed. The algorithm should increase d_w by k.

Let Y be $\{w \in X \mid v(lp(w)) \leq x\}$. Based on the above case analysis, we know that once d_w is increased for every $w \in Y$, the values of w for all $w \in X$ will increase by k automatically.

Let u be the node in R_c whose value is just larger than x. It can easily check that Y is in fact equal to $\{w \mid w$ is on the path of R_c from the root to u and $v(w) > x\}$. Since any path in R_c is of length $O(\log n)$, we can increase d_w by k for all $w \in Y$ in $O(\log n)$ time. □

For space complexity, note that the color (red or black) can be stored in 1 bit, the pointers to the left subtree or the right subtree can be stored in $\log(\frac{n}{5\log n})$-bits. Thus, each node can be stored using $5\log(\frac{n}{5\log n}+1)$-bits, which is smaller than $5\log n$. Since there are at most $\frac{n}{5\log n}$ representatives in all S_c for $c \in \Sigma$, the total space required by all modified red-black trees R_c is n.

In summary, the total space required by $\mathcal{D}$ to store the segments and the red-black trees is $5n + n = 6n$.

Lemma 7. *The total space required by the data structure $\mathcal{D}$ is $6n$ bits.*

5 An efficient incremental construction

This section shows that, if the compressed suffix array is represented by data structure $\mathcal{D}$, the time required by the incremental construction algorithm (described in Section 3) is $O(\log n)$. Recall that the incremental construction step contains three sub-steps. This section discusses them one by one.

Step 1: This step tries to locate the index x such that T' should be inserted between the suffices represented by $SA_T[x]$ and $SA_T[x+1]$. Based on data structure $\mathcal{D}$, this step can be completed as follows.

1. We traverse the modified red-black tree R_c for S_c, return (a) a node p with value just smaller than $SA_T^{-1}[1]$; (b) $rank(p)$. (Note the special case that $SA_T^{-1}[1]$ is smaller than the smallest value in red-black tree).
2. Sequential search the segment s corresponding to p and locate the value just greater than $SA_T^{-1}[1]$. Let q and r be the location and the rank of this value in the segment.
3. $x = \sum_{b<c, b\in\Sigma} |S_b| + rank(p) + r - 1$. (Note: $SA_{T'}^{-1}[1] = x + 1$).

Note that step 1.1 just performs a search in the modified red-black tree R_c. Since R_c is a balanced tree, it requires $O(\log n)$ time. Note that during the traverse, $rank(p)$ can be computed by means of $size()$. For step 1.2, since a segment contains at most $10 \log n$ integers, a sequential search in this segment requires $O(\log n)$ time. Step 1.3 can be computed in $O(1)$ time. In total, step 1 requires $O(\log n)$ time.

Step 2: This step can be completed as follows.

1. Insert value $SA_T^{-1}[1]$ at location q of segment s.
2. Increment the total number of entries represented by the subtree at node p.
3. If the segment contains $\geq 10 \log n$ values, split the segment into two segments.
4. For every S_b where $b \in \Sigma$,
   (a) Increase all representatives in red-black tree whose values are greater than x by one. Finally, return the node p which is just smaller than or equal to x.
   (b) Sequential search the segment corresponding to p to find the value just greater than x. Increment this value (if exists) by one.

For Step 2.1, we need to insert the value $SA_T^{-1}[1]$ at location q of segment s. Assume that the segment s contains ℓ values $v_1, \ldots, v_\ell$. Also, recall that we only store the differences, that is, we store $v_2 - v_1, v_3 - v_2, \ldots, v_\ell - v_{\ell-1}$. To insert $SA_T^{-1}[1]$ into location q, Step 2.2 can be computed in $O(1)$ time. In Step 2.3, we need to split the segment, the algorithm is required to allocate a new segment and move $5 \log n$ values to the new segment. Such operation can be done in $O(\log n)$ as we need to move $5 \log n$ entries. Step 2.4(a) can be solved in $O(\log n)$ time based on Lemma 6 and the modified red-black tree data structure. Step 2.4(b) again can be solved in $O(\log n)$ time since a segment has less than $10 \log n$ values. In total, Step 2 requires $O(\log n)$ time.

Step 3: This step is simple. It takes $O(1)$ time.

In conclusion, the time required for each incremental construction is $O(\log n)$.

References

1. D. R. Clark and J. I. Munro. Efficient suffix trees on secondary storage. In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 383–391. 1996.

2. Altschul S. F., Gish W., Miller W., Myers E. W., and Lipman D. J. Basic locol alignment search tool. *Journal of Molecular Biology*, pages 403–410, 1990.
3. P. Elias. Universal codeword sets and representation of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975.
4. P. Ferragine and G. Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 390–398. 2000.
5. R. Grossi and J.S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proceedings of the 32nd ACM Symposium on Theory of Computing*, pages 397-406, 2000.
6. E. Hunt, M. P. Atkinson, and R. W. Irving. A database index to large biological sequences. In *Proceedings of the 27th VLDB Conference*, pages 410–421. 2000.
7. S. Kurtz. Reducing the space requirement of suffix trees. *Software Practice and Experiences*, 29:1149–1171, 1999.
8. U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.
9. E. M. MCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.
10. K. Sadakane. Compressed text databases with efficient query algorithms based on compressed suffix array. In *Proceedings of the 11th International Conference on Algorithms and Computation (ISAAC)*, pages 410–421. 2000.
11. K. Sadakane and T. Shibyya. Indexing huge genome sequences for solving various porblems. In *Genome Informatics*, pages 175–183. 2001.