Full-fledged Real-Time Indexing for Constant Size Alphabets

Gregory Kucherov[1,2] and Yakov Nekrich[3⋆]

[1] Laboratoire d'Informatique Gaspard Monge, Université Paris-Est & CNRS, Marne-la-Vallée, Paris, France, Gregory.Kucherov@univ-mlv.fr
[2] Department of Computer Science, Ben-Gurion University of the Negev, Be'er Sheva, Israel
[3] Department of Electrical Engineering & Computer Science, University of Kansas yakov.nekrich@googlemail.com

Abstract. In this paper we describe a data structure that supports pattern matching queries on a dynamically arriving text over an alphabet of constant size. Each new symbol can be prepended to T in $O(1)$ expected worst-case time. At any moment, we can report all occurrences of a pattern P in the current text in $O(|P| + k)$ time, where $|P|$ is the length of P and k is the number of occurrences. This resolves, under assumption of constant size alphabet, a long-standing open problem of existence of a real-time indexing method for string matching (see [2]).

1 Introduction

Two main versions of the string matching problem differ in which of the two components – pattern P or text T – is provided first in the input (or is considered as fixed) and can then be preprocessed before processing the other component. The framework when the text has to be preprocessed is usually called *indexing*, as it can be viewed as constructing a text index supporting matching queries.

Real-time variants of the string matching problem are about as old as the string matching itself. In the 70s, existence of real-time string matching algorithms was first studied for Turing machines. For example, it has been shown that the language $\{P\#T\}$ where P occurs in T can be recognized by a Turing machine, while the language $\{T\#P\}$ cannot [7]. In the realm of the RAM model, the real-time variant of pattern-preprocessing string matching has been extensively studied, leading to very efficient solutions (see e.g. [3] and references therein). The indexing variant, however, still has important unsolved questions.

⋆ This work was done while this author was at Laboratoire d'Informatique Gaspard Monge, Université Paris-Est & CNRS

In the real-time indexing problem, we have to maintain an indexing data structure for a text that arrives online, by spending $O(1)$ worst-case time on each new character; a string matching query must be answered in $O(|P|)$ time for a query string P. Back in the 70s, Slisenko [15] claimed a solution to the real-time indexing problem, but its complex and voluminous full description made it unacknowledged by the scientific community, and the problem remained to be considered open for many years. In 1994, Kosaraju [11] presented another solution which, however, did not support repetitive matching queries on different portions of arriving text, but assumed that the text is entirely read before the matching query is made. In 2008, Amir and Nor [2] proposed another algorithm that fixes this drawback and allows queries to be made at any moment of the text scan.

All the three existing real-time indexing solutions [15, 11, 2] support only existential queries asking whether the pattern occurs in the text, but are unable to report occurrences of the pattern. Designing a real-time text indexing algorithm that would support queries on all occurrences of a pattern is stated in [2] as the most important remaining open problem. The algorithms of [11, 2] assume a constant size alphabet and are both based on constructions of "incomplete" suffix trees which can be built real-time but can only answer existential queries. To output all occurrences of a pattern, a fully-featured suffix tree is needed, however a real-time suffix tree construction, first studied in [1], is in itself an open question. The best currently known algorithm [4] spends $O(\log \log n)$ worst-case time on each character, and a truly real-time construction seems unlikely to exist. Therefore, a suffix tree alone seems to be insufficient to solve the real-time indexing problem.

In this paper, we propose the first real-time text indexing solution that supports reporting all pattern occurrences, under the assumption of constant size alphabet. The general idea is to maintain several data structures, three in our case, each supporting queries for different pattern lengths. Our method employs the suffix tree construction technique recently proposed by Kopelowitz [9]. Similar to [9] and to previous real-time indexing solutions [11, 2], we assume that the text is read right-to-left, or otherwise the pattern needs to be reversed before executing the query. We use the word RAM computation model; the same model is also used in e.g., [4, 9].

The paper is organized as follows. In Section 2.1, we describe auxiliary data structures and Kopelowitz' technique that are essential for our algorithm. In Section 3, we describe the three data structures for different pattern lengths that constitute a basis of our solution. These data structures, however, do not provide a fully real-time algorithm. Then in Section 4, we show how to "fix" the solution of Section 3 in order to obtain a fully real-time algorithm.

Throughout the paper, Σ is an alphabet of constant size σ. Since the text T is read right-to-left, it will be convenient for us to enumerate symbols of T from the end, i.e. $T = t_n \dots t_1$ and substring $t_{i+\ell}t_{i+\ell-1} \dots t_i$ will be denoted $T[i+\ell..i]$. $T[i..]$ denotes suffix $T[i..1]$. Throughout this paper, we reserve k to denote the number of objects (occurrences of a pattern, elements in a list, etc) in the query answer.

2 Preliminaries

In this Section, we describe main algorithmic tools used by our algorithms.

2.1 Range Reporting and Predecessor Queries on Colored Lists

We use data structures from [13] for searching in dynamic colored lists.

Colored Range Reporting in a List. Let elements of a dynamic linked list $\mathcal{L}$ be assigned positive integer values called *colors*. A colored range reporting query on a list $\mathcal{L}$ consists of two integers $col_1 < col_2$ and two pointers ptr_1 and ptr_2 that point to elements e_1 and e_2 of $\mathcal{L}$. An answer to a colored range reporting query consists of all elements $e \in \mathcal{L}$ occurring between e_1 and e_2 (including e_1 and e_2) such that $col_1 \leq \texttt{col}(e) \leq col_2$, where $\texttt{col}(e)$ is the color of e. The following result on colored range reporting has been proved by Mortensen [13].

Lemma 1 ([13]). *Suppose that* $\texttt{col}(e) \leq \log^f n$ *for all* $e \in \mathcal{L}$ *and some constant* $f \leq 1/4$. *We can answer color range reporting queries on* $\mathcal{L}$ *in* $O(\log\log m + k)$ *time using an* $O(m)$*-space data structure, where* m *is the number of elements in* $\mathcal{L}$. *Insertion of a new element into* $\mathcal{L}$ *is supported in* $O(\log\log m)$ *time.*

Note that the bound $f \leq 1/4$ follows from the description in [12]: the data structure in [13] uses Q-heaps [6] to answer certain queries on the set of colors in constant time.

Colored Predecessor Problem. The colored predecessor query on a list $\mathcal{L}$ consists of an element $e \in \mathcal{L}$ and a color col. The answer to a query (e, col) is the closest element $e' \in \mathcal{L}$ which precedes e such that $\texttt{col}(e) = col$. The following Lemma is also proved in [13]; we also refer to [8], where a similar problem is solved.

Lemma 2 ([13]). *Suppose that* $\texttt{col}(e) \le \log^f n$ *for all* $e \in \mathcal{L}$ *and some constant* $f \le 1/4$. *There exists an* $O(m)$ *space data structure that answers colored predecessor queries on* $\mathcal{L}$ *in* $O(\log\log m)$ *time and supports insertions in* $O(\log\log m)$ *time, where* m *is the number of elements in* $\mathcal{L}$.

2.2 On-Line Indexing for Alphabets of Small Size

Kopelowitz [9] described an online indexing method that works for an arbitrarily large alphabet A. We describe below a simplified version of his algorithm, adapted to our purposes, for the case when the alphabet size $|A| \le \log^{1/4} n$. For a current text T, Kopelowitz' algorithm maintains a list $\mathcal{S}$ of its lexicographically sorted suffixes and a suffix tree $\mathcal{T}$.[4] Besides, the following auxiliary data structures are used. For any symbol $a \in A$ that occurs in T at least once, we store a in a data structure $\mathcal{A}$. Since $\mathcal{A}$ contains at most $\log^{1/4} n$ elements, we can search in $\mathcal{A}$ in $O(1)$ time using Q-heaps [6]. For every a in $\mathcal{A}$, we store a pointer $\mathit{last}(a)$ to the last (lexicographically largest) suffix of T that starts with a. Furthermore, every suffix $T[i..]$ in $\mathcal{S}$ is colored with color t_{i+1}, i.e., the color of $T[i..]$ is the symbol that precedes the starting position of $T[i..]$ in T. We maintain the structure $\mathcal{D}$ for colored predecessor queries on $\mathcal{S}$, as in Lemma 2. For each $T[i..]$ in $\mathcal{S}$, we also store a pointer to the suffix $T[i+1..]$ in $\mathcal{S}$. Finally, we store a data structure for weighted level ancestor queries on $\mathcal{T}$: for any leaf v of $\mathcal{T}$ and for any integer q, we can find the lowest ancestor u of v such that the string depth of u is smaller than q. The data structure from [10] uses linear space, supports dynamic tree updates in expected $O(\log\log n)$ time, and answers the weighted level ancestor queries in worst-case $O(\log\log n)$ time.

The algorithm of Kopelowitz [9] consists of three phases. During the first phase we prepend a symbol t_{n+1} to the current text $T = t_n \dots t_1$ and find the position of the new suffix $t_{n+1}T$ in the lexicographically ordered list of suffixes. To do this, we first check if t_{n+1} occurs at least once in T. We query $\mathcal{A}$ for the largest symbol $a \le t_{n+1}$; if $a \ne t_{n+1}$, then the suffix $t_{n+1}T$ should be inserted after $\mathit{last}(a)$ into $\mathcal{S}$. If $a = t_{n+1}$, at least one suffix of T starts with t_{n+1}. Using $\mathcal{D}$, we look for the predecessor of $T[n..]$ in $\mathcal{S}$ colored with t_{n+1}. Let $T[j..]$ denote such a predecessor of $T[n..]$. Then $T[j+1..]$ starts with symbol t_{n+1}, and it is easy to check that $T[j+1..]$ is the lexicographically largest suffix that precedes $t_{n+1}T$.

When we know suffixes $S' = T[i'..]$ and $S'' = T[i''..]$ of T that respectively precede and follow $t_{n+1}T$, we can find the longest common prefixes

[4] In subsequent sections we will consider $\mathcal{S}$ to be a part of $\mathcal{T}$.

$\ell' = lcp(t_{n+1}T, S')$ and $\ell'' = lcp(t_{n+1}T, S'')$ in $O(1)$ time using the data structure of [5]. If $\ell' > \ell''$, we find the leaf v of $\mathcal{T}$ that holds S' and the lowest ancestor u of v with string depth at most ℓ'; u is found using the dynamic weighted level ancestor structure from [10]. If the string depth of u equals ℓ', we create a new child of u that holds the new suffix $t_{n+1}T$. Otherwise, let w be a child of u which is an ancestor of v. We split the edge from u to w and create a new node u'. Then, we create a new child of u' that holds t_oT. The case $\ell'' \geq \ell'$ is symmetric. When the new suffix is inserted into the suffix tree, we update all the auxiliary data structures during the third phase. We refer to [9] for a detailed description of the algorithm.

The only difference between the described procedure and the original algorithm of Kopelowitz [9] is that our method assumes an alphabet of size $|A| \leq \log^{1/4} n$, which allows us to employ the colored predecessor data structure to search for the position of suffix $t_{n+1}T$ during the first phase. The algorithm in [9] works for an arbitrarily large alphabet, and therefore requires more complicated data structures.

3 Fast Off-Line Solution

In this section we describe the main part of our algorithm. The algorithm updates the index by reading the text in the right-to-left order. However, the algorithm we describe now will not be on-line, as it will have to access symbols to the left of the currently processed symbol. Another "flaw" of the algorithm is that it will support pattern matching queries only with an additional exception: we will be able to report all occurrences of a pattern except for those with start positions among a small number of most recently processed symbols of T. In the next section we will show how to fix these issues and turn our algorithm into a fully real-time indexing solution that reports all occurrences of a pattern.

The algorithm distinguishes between three types of query patterns depending on their length: *long patterns* contain at least $(\log\log n)^2$ symbols, *medium-size patterns* contain between $(\log^{(3)} n)^2$ and $(\log\log n)^2$ symbols, and *short patterns* contain less than $(\log^{(3)} n)^2$ symbols[5]. For each of the three types of patterns, the algorithm will maintain a separate data structure supporting queries in $O(|P| + k)$ time for matching patterns of the corresponding type.

[5] Henceforth, $\log^{(3)} n = \log\log\log n$.

3.1 Long Patterns

To match long patterns, we maintain a *sparse suffix tree* $\mathcal{T}_L$ storing only suffixes that start at positions $q \cdot d$ for $q \geq 1$ and $d = \log\log n/(4\log\sigma)$. Suffixes stored in $\mathcal{T}_L$ are regarded as strings over a meta-alphabet of size $\sigma^d = \log^{1/4} n$. This allows us to use the method of Section 2.2 to maintain $\mathcal{T}_L$. Recall that the method maintains a list of sorted suffixes that we denote $\mathcal{L}_L$.

Using $\mathcal{T}_L$ we can find occurrences of a pattern P that start at positions qd for $q \geq 1$, but not occurrences starting at positions $qd+\delta$ for $1 \leq \delta < d$. To be able to find all occurrences, we maintain an additional list $\mathcal{L}_E$ defined as follows.

The list $\mathcal{L}_E$ contains copies of all nodes of $\mathcal{T}_L$ as they occur during the Euler tour of $\mathcal{T}_L$. Thus, $\mathcal{L}_E$ contains one element for each leaf and two elements for each internal node of $\mathcal{T}_L$. The first copy of an internal node u precedes the copies of all nodes in the subtree of u, and the second copy of u occurs immediately after the copies of all descendants of u. To simplify the presentation, we will not distinguish between elements of $\mathcal{L}_E$ and suffix tree nodes that they represent. If a node of $\mathcal{L}_E$ is a leaf that corresponds to a suffix $T[i..]$, we mark it with the meta-symbol $\overleftarrow{T}[i,d] = t_{i+1}t_{i+2}\dots t_{i+d}$ which is interpreted as the color of the leaf for the suffix $T[i..]$. Colors are ordered by lexicographic order of underlying strings. If $S = s_1 \dots s_j$ is a string with $j < d$, then S defines an interval of colors, denoted $[minc(S), maxc(S)]$, corresponding to all strings of length d with prefix S. Recall that there are $\log^{1/4} n$ different colors. On list $\mathcal{L}_E$, we maintain the data structure of Lemma 1 for colored range reporting queries.

After reading character t_i where $i = qd$ for $q \geq 1$, we add the suffix $T[i..]$, viewed as a string over the meta-alphabet of cardinality $\log^{1/4} n$, to $\mathcal{T}_L$ according to the algorithm described in Section 2.2. In addition, we have to update the list $\mathcal{L}_E$, i.e. to insert to $\mathcal{L}_E$ the new leaf holding the suffix $T[i..]$ marked with the color $t_{i+1}t_{i+2}\dots t_{i+d}$. (Note that here the algorithms "looks ahead" for the forthcoming d letters of T.) If a new internal node has been inserted into $\mathcal{T}_L$, we also update the list $\mathcal{L}_L$ accordingly. (Details are left out and can be found e.g. in [12].)

Since the meta-alphabet size is only $\log^{1/4} n$, the navigation in $\mathcal{T}_L$ from a node to a child can be supported in $O(1)$ time. Observe that the children of any internal node $v \in \mathcal{T}_L$ are naturally ordered by the lexicographic order of edge labels. We store the children of v in a data structure $\mathcal{P}_v$ which allows us to find in time $O(1)$ the child whose edge label starts with a string (meta-symbol) $S = s_1 \dots s_d$. Moreover, we can also compute in time $O(1)$ the "smallest" and the "largest" child of v whose edge label starts with a string $S = s_1 \dots s_j$ with $j \leq d$. $\mathcal{P}_v$ will also support adding a new edge to $\mathcal{P}_v$ in $O(1)$ time. Data structure $\mathcal{P}_v$ can be implemented using e.g. atomic heaps [6]; since all elements in $\mathcal{P}_v$ are bounded by $\log^{1/4} n$, we can also implement $\mathcal{P}_v$ as described in [14].

We now consider a long query pattern $P = p_1 \dots p_m$ and show how the occurrences of P are computed. An occurrence of P is said to be a δ-occurrence if it starts in T at a position $j = qd + \delta$, for some q. For each δ, $0 \leq \delta \leq d - 1$, we find all δ-occurrences as follows. First we "spell out" $P_\delta = p_{\delta+1} \dots p_m$ in $\mathcal{T}_L$ over the meta-alphabet, i.e. we traverse $\mathcal{T}_L$ proceeding by blocks of up to d letters of Σ. If this process fails at some step, then P has no δ-occurrences. Otherwise, we spell out P_δ completely, and retrieve the closest explicit descendant node v_δ, or a range of descendant nodes $v_\delta^l, v_\delta^{l+1}, \dots, v_\delta^r$ in the case when P_δ spells to an explicit node except for a suffix of length less than d. The whole spelling step takes time $O(|P|/d + 1)$.

Now we jump to the list $\mathcal{L}_E$ and retrieve the first occurrence of v_δ (or v_δ^l) and the second occurrence of v_δ (or v_δ^r) in $\mathcal{L}_E$. A leaf u of $\mathcal{T}$ corresponds to a δ-occurrence of P if and only if u occurs in the subtree of v_δ (or the subtrees of $v_\delta^l, \dots, v_\delta^r$) and the color of u belongs to $[minc(p_\delta \dots p_1), maxc(p_\delta \dots p_1)]$. In the list $\mathcal{L}_E$, these leaves occur precisely within the interval we computed. Therefore, all δ-occurrences of P can be retrieved in time $O(\log\log n + k_\delta)$ by a colored range reporting query (Lemma 1), where k_δ is the number of δ-occurrences. Summing up over all δ, all occurrences of a long pattern P can be reported in time $O(d(|P|/d + \log\log n) + k) = O(|P| + d \log\log n + k) = O(|P| + k)$, as $d = \log\log n/(4 \log \sigma)$, $\sigma = O(1)$ and $|P| \geq (\log\log n)^2$.

3.2 Medium-Size Patterns

Now we show how to answer matching queries for patterns P where $(\log^{(3)} n)^2 \leq |P| < (\log\log n)^2$. In a nutshell, we apply the same method as in Section 3.1 with the main difference that the sparse suffix tree will store only *truncated suffixes* of length $(\log\log n)^2$, i.e. prefixes of suffixes bounded by $(\log\log n)^2$ symbols. We store truncated suffixes starting at positions spaced by $\log^{(3)} n = \log\log\log n$ symbols. The total number of different truncated suffixes is at most $\sigma^{(\log\log n)^2}$. This small number of suffixes will allow us to search and update the data structures faster compared to Section 3.1. We now describe the details of the construction.

We store all truncated suffixes that start at positions qd', for $q \geq 1$ and $d' = \log^{(3)} n$, in a tree $\mathcal{T}_M$. $\mathcal{T}_M$ is organized in the same way as the standard suffix tree; that is, $\mathcal{T}_M$ is a compressed trie for substrings $T[qd'..qd'-(\log\log n)^2+1]$, where these substrings are regarded as strings over the meta-alphabet $\Sigma^{d'}$.[6] Observe that the same truncated suffix can occur several times. Therefore, we augment each leaf v with a list of *colors* $Col(v)$ corresponding to left contexts of the corresponding truncated suffix S. More precisely, if $S = T[qd'..qd'-(\log\log n)^2+1]$ for some $q \geq 1$, then $\overleftarrow{T}[qd', d']$ is added to $Col(v)$. Note that the number of colors is bounded by $\sigma^{\log^{(3)} n}$. Futhermore, for each color *col* in $Col(v)$, we store all positions $i = qd'$ of T such that S occurs at i and $\overleftarrow{T}[i, d'] = col$. As in Section 3.1, we store a list $\mathcal{L}_M$ that contains colored elements corresponding to the Euler tour traversal of $\mathcal{T}_M$. For each internal node, $\mathcal{L}_M$ contains two elements. For every leaf v and for each value *col* in its color list $Col(v)$, $\mathcal{L}_M$ contains a separate element colored with *col*. Observe that since the size of $\mathcal{L}_M$ is bounded by $O(\sigma^{(\log\log n)^2+\log^{(3)} n})$, updates of $\mathcal{L}_M$ can be supported in $O(\log\log(\sigma^{(\log\log n)^2})) = O(\log^{(3)} n)$ time, and colored reporting queries on $\mathcal{L}_M$ can be answered in $O(\log^{(3)} n + k)$ time (see Lemma 1).

Truncated suffixes are added to $\mathcal{T}_M$ using a method similar to that of Section 3.1. After reading a symbol $t_{qd'}$ for some $q \geq 1$, we add $S_{\text{new}} = T[qd'..qd'-(\log\log n)^2+1]$ colored with $\overleftarrow{T}[qd', d']$ to the tree $\mathcal{T}_M$. To find the place of S_{new} in the list of leaves of $\mathcal{T}_M$, here we compare truncated suffixes directly rather than using predecessor queries, as in Kopelowitz's algorithm (Section 2.2). Observe that every truncated suffix can be viewed as an integer in the range $[1..U]$ for $U = \sigma^{(\log\log n)^2}$. We store current truncated suffixes in the van Emde Boas data structure $\mathcal{V}$. Using $\mathcal{V}$, we can find the largest $S_{\text{prev}} \leq S_{\text{new}}$ and the smallest $S_{\text{next}} \geq S_{\text{new}}$ in $\mathcal{T}_M$ in $O(\log\log U) = O(\log^{(3)} n)$ time. Let $\ell' = lcp(S_{\text{prev}}, S_{\text{new}})$, $\ell'' = lcp(S_{\text{next}}, S_{\text{new}})$, and $\ell = \max(\ell', \ell'')$. Observe that *lcp* values can be computed in $O(1)$ time using standard bit operations. Once ℓ is known, we update the tree $\mathcal{T}_M$ spending $O(\log\log|\mathcal{T}_M|) = O(\log^{(3)} n)$ expected time on the weighted level ancestor query. Finally, we update $\mathcal{L}_M$: if $\mathcal{L}_M$ already contains a leaf with string value S_{new} and color $\overleftarrow{T}[qd', d']$, we add qd' to the list of its occurrences, otherwise we insert a new element into $\mathcal{L}_M$ and initialize its location list to qd'. Altogether, the addition of a new truncated suffix S_{new} requires $O(\log^{(3)} n)$ time.

A query for a pattern $P = p_1 \dots p_m$, such that $(\log^{(3)} n)^2 \leq m < (\log\log n)^2$, is answered in the same way as in Section 3.1. For each $\rho =$

[6] For simplicity we assume that $\log^{(3)} n$ and $\log\log n$ are integers and $\log^{(3)} n$ divides $\log\log n$. If this is not the case, we can find d' and d that satisfy these requirements such that $\log\log n \leq d \leq 2\log\log n$ and $\log^{(3)} n \leq d' \leq 2\log^{(3)} n$.

$0, \ldots, \log^{(3)} n - 1$, we find locus nodes $v_\rho^l, \ldots, v_\rho^r$ (possibly with $v_\rho^l = v_\rho^r$) of $P_\rho = p_{\rho+1} \ldots p_m$. Then, we find all elements in $\mathcal{L}_M$ occurring between the first occurrence of v_ρ^l and the second occurrence of v_ρ^r and colored with a color *col* that belongs to $[minc(p_\rho \ldots p_1), maxc(p_\rho \ldots p_1)]$. For every such element, we traverse the associated list of occurrences: if a position i is in the list, then P occurs at position $(i+\rho)$. The total time needed to find all occurrences of a medium-size pattern P is $O(d'(|P|/d' + \log^{(3)} n) + k) = O(|P| + (\log^{(3)} n)^2 + k) = O(|P| + k)$ since $|P| \geq (\log^{(3)} n)^2$.

3.3 Short Patterns

Finally, we describe our indexing data structure for patterns P with $|P| < (\log^{(3)} n)^2$. We maintain the tree $\mathcal{T}_S$ of truncated suffixes of length $\Delta = (\log^{(3)} n)^2$ seen so far in the text. For every position i of T, $\mathcal{T}_S$ contains the substring $T[i..i - \Delta + 1]$. $\mathcal{T}_S$ is organized as a compacted trie. We support queries and updates on $\mathcal{T}_S$ using tabulation. There are $O(2^{\sigma^\Delta})$ different trees, and $O(\sigma^\Delta)$ different queries can be made on each tree. Therefore, we can afford explicitly storing all possible trees $\mathcal{T}_S$ and tabulating possible tree updates. Each internal node of a tree stores pointers to its lefmost and rightmost leaves, the leaves of a tree are organized in a list, and each leaf stores the encoding of the corresponding string Q.

The *update table* $\mathbf{T}_u$ stores, for each tree $\mathcal{T}_S$ and for any string Q, $|Q| = \Delta$, a pointer to the tree $\mathcal{T}_S'$ (possibly the same) obtained after adding Q to $\mathcal{T}_S$. Table $\mathbf{T}_u$ uses $O(2^{\sigma^\Delta} \sigma^\Delta) = o(n)$ space. The *output table* $\mathbf{T}_o$ stores, for every string Q of length Δ, the list of positions in the current text T where Q occurs. $\mathbf{T}_o$ has $\sigma^\Delta = o(n)$ entries and all lists of occurrences take $O(n)$ space altogether.

When scanning the text, we maintain the encoding of the string Q of Δ most recently read symbols of T. The encoding is updated after each symbol using bit operations. After reading a new symbol, the current tree $\mathcal{T}_S$ is updated using table $\mathbf{T}_u$ and the current position is added to the entry $\mathbf{T}_o[Q]$. Updates take $O(1)$ time.

To answer a query P, $|P| < \Delta$, we find the locus u of P in the current tree $\mathcal{T}_S$, retrieve the leftmost and rightmost leaves and traverse the leaves in the subtree of u. For each traversed leaf v_l with label Q, we report the occurrences stored in $\mathbf{T}_o[Q]$. The query takes time $O(|P| + k)$.

4 Real-Time Indexing

The indexes for long and medium-size patterns, described in Sections 3.1 and 3.2 respectively, do not provide real-time indexing solutions for several reasons. The index for long patterns, for example, requires to look ahead for the forthcoming d symbols when processing symbols t_i for $i = qd$, $q \geq 1$. Furthermore, for such i, we are unable to find occurrences of query patterns P starting at positions $t_{i-1} \dots t_{i-d+1}$ before processing t_i. A similar situation holds for medium-size patterns. Another issue is that in our previous development we assumed the length n of T to be known, whereas this may of course not be the case in the real-time setting. In this Section, we show how to fix these issues in order to turn the indexes real-time. Firstly we show how the data structures of Sections 3.1 and 3.2 can be updated in a real-time mode. Then, we describe how to search for patterns that start among most recently processed symbols. We describe our solutions to these issues for the case of long patterns, as a simple change of parameters provides a solution for medium-size patterns too. Finally, we will show how we can circumvent the fact that the length of T is not known in advance.

In the algorithm of Section 3.1, the text is partitioned into blocks of length d, and the insertion of a new suffix $T[i..]$ is triggered only when the leftmost symbol t_i of a block is reached. The insertion takes time $O(d)$ and assumes the knowledge of the forthcoming block $t_{i+d} \dots t_{i+1}$. To turn this algorithm real-time, we apply a standard deamortization technique. We distribute the cost of the insertion of suffix $T[i-d..]$ over d symbols of the block $t_{i+d} \dots t_{i+1}$. This is correct, as by the time we start reading the block $t_{i+d} \dots t_{i+1}$, we have read the block $t_i \dots t_{i-d+1}$ and therefore have all necessary information to insert suffix $T[i-d..]$. In this way, we spend $O(1)$ expected time per symbol to update all involved data structures.

Now assume we are reading a block $t_{i+d} \dots t_{i+1}$, i.e. we are processing some symbol $t_{i+\delta}$ for $1 \leq \delta < i$. At this point, we are unable to find occurrences of a query pattern P starting at $t_{i+\delta} \dots t_{i+1}$ as well as within the two previous blocks, as they have not been indexed yet. This concerns up to $(3d-1)$ most recent symbols. We then introduce a separate procedure to search for occurrences that start in $3d$ leftmost positions of the already processed text. This can be done by simply storing T in a compact form T_c where every $\log_\sigma n$ consecutive symbols are packed into one computer word[7]. Thus, T_c uses $O(|T|/\log_\sigma n)$ words of space. Using T_c, we can test whether $T[j..j-|P|+1] = P$, for any pattern P and any position j, in $O(\lceil |P|/\log_\sigma n \rceil) = o(|P|/d) + O(1)$ time. Therefore, checking $3d$ positions takes time $o(|P|) + O(d) = O(|P|)$ for a long pattern P.

We now describe how we can apply our algorithm in the case when the text length is not known beforehand. In this case, we assume $|T|$ to

[7] In fact, it would suffice to store $3d-1$ most recently read symbols in compact form.

take increasing values $n_0 < n_1 < \dots$, as long as the text T keeps growing. Here, n_0 is some appropriate initial value and $n_i = 2n_{i-1}$ for $i \geq 1$.

Suppose now that n_i is the currently assumed value of $|T|$. After we reach character $t_{n_i/2}$, during the processing of the next $n_i/2$ symbols, we keep building the index for $|T| = n_i$ and, in parallel, rebuild all the data structures under assumption that $|T| = n_{i+1} = 2n_i$. In particular, if $\log\log(2n_i) \neq \log\log n_i$, we build a new index for long patterns, and if $\log^{(3)}(2n_i) \neq \log^{(3)} n_i$, we build a new index for meduim-size and short patterns. If $\log_\sigma(2n_i) \neq \log_\sigma n_i$, we also construct a new compact representation T_c introduced earlier in this section. Altogether, we distribute the construction cost of the data structures for $T[n_i..1]$ under assumption $|T| = 2n_i$ over the processing of $t_{n_i/2+1} \dots t_{n_i}$. Since $O(n_i) = O(n_i/2)$, processing these $n_i/2$ symbols remains real-time. By the time t_{n_i} has been read, all data structures for $|T| = 2n_i$ have been built, and the algorithm proceeds with the new value $|T| = n_{i+1}$. Observe finally that the intervals $[n_i/2+1, n_i]$ are all disjoint, therefore the overhead per letter incurred by the procedure remains constant. In conclusion, the whole algorithm remains real-time. We finish with our main result.

Theorem 1. *There exists a data structure storing a text T that can be updated in $O(1)$ worst-case expected time after prepending a new symbol to T. This data structure reports all occurrences of a pattern P in the current text T in $O(|P|+k)$ time, where k is the number of occurrences.*

5 Conclusions

In this paper we presented the first real-time indexing data structure that supports reporting all pattern occurrences in optimal time $O(|P|+k)$. As in the previous works on this topic [11, 2, 4], we assume that the input text is over an alphabet of constant size. It may be possible to extend our result to alphabets of poly-logarithmic size.

Our algorithm spends a constant *expected* worst-case time for updating the data structure when a new text symbol arrives. The expectation comes only from the updates of the weighted level ancestor structure [10], which, in turn, comes from the updates for the dynamic predecessor problem (y-fast tries). We feel that one can get rid of the expectation, however we have not found a solution to this so far.

Acknowledgements. GK has been supported by the Marie-Curie Intra-European fellowship for carrier development. We thank the anonymous reviewers for helpful comments.

References

1. A. Amir, T. Kopelowitz, M. Lewenstein, and N. Lewenstein. Towards real-time suffix tree construction. In M. Consens and G. Navarro, editors, *Proc. International Symposium on String Processing and Information Retrieval (SPIRE)*, volume 3772 of *Lecture Notes in Computer Science*, pages 67–78. Springer Berlin Heidelberg, 2005.
2. A. Amir and I. Nor. Real-time indexing over fixed finite alphabets. In *Proc. 19th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2008)*, pages 1086–1095, 2008.
3. D. Breslauer, R. Grossi, and F. Mignosi. Simple real-time constant-space string matching. In R. Giancarlo and G. Manzini, editors, *Combinatorial Pattern Matching*, volume 6661 of *Lecture Notes in Computer Science*, pages 173–183. Springer Berlin Heidelberg, 2011.
4. D. Breslauer and G. F. Italiano. Near real-time suffix tree construction via the fringe marked ancestor problem. In *Proc. 18th International Symposium on String Processing and Information Retrieval (SPIRE 2011)*, pages 156–167, 2011.
5. G. Franceschini and R. Grossi. A General Technique for Managing Strings in Comparison-Driven Data Structures. In *Proc. International Colloquium on Automata, Languages and Programming (ICALP 2004)*, Lecture Notes in Computer Science, pages 606–617. Springer Berlin / Heidelberg, 2004.
6. M. L. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comput. Syst. Sci.*, 48(3):533–551, 1994.
7. Z. Galil. String matching in real time. *J. ACM*, 28(1):134–149, 1981.
8. Y. Giyora and H. Kaplan. Optimal dynamic vertical ray shooting in rectilinear planar subdivisions. *ACM Transactions on Algorithms*, 5(3), 2009.
9. T. Kopelowitz. On-line indexing for general alphabets via predecessor queries on subsets of an ordered list. In *Proc. 53rd Annual IEEE Symposium on Foundations of Computer Science (FOCS 2012)*, pages 283–292, 2012.
10. T. Kopelowitz and M. Lewenstein. Dynamic weighted ancestors. In *Proc. 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2007)*, pages 565–574, 2007.
11. S. R. Kosaraju. Real-time pattern matching and quasi-real-time construction of suffix trees (preliminary version). In *Proc. 26th Annual ACM Symposium on Theory of Computing (STOC 1994)*, pages 310–316. ACM, 1994.
12. G. Kucherov, Y. Nekrich, and T. Starikovskaya. Cross-document pattern matching. In J. Kärkkäinen and J. Stoye, editors, *Proceedings of the 23rd Annual Symposium on Combinatorial Pattern Matching (CPM), July 3-5, 2012, Helsinki (Finland)*, volume 7354 of *Lecture Notes in Computer Science*, pages 196–207. Springer Verlag, 2012.
13. C. W. Mortensen. Fully-dynamic two dimensional orthogonal range and line segment intersection reporting in logarithmic time. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2003)*, pages 618–627, 2003.
14. G. Navarro and Y. Nekrich. Top-k document retrieval in optimal time and linear space. In *Proc. 23rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2012)*, pages 1066–1077, 2012.
15. A. Slisenko. String-matching in real time: Some properties of the data structure. In *Mathematical Foundations of Computer Science 1978, Proceedings, 7th Symposium, Zakopane, Poland, September 4-8, 1978*, volume 64 of *Lecture Notes in Computer Science*, pages 493–496. Springer, 1978.