Space-Efficient Framework for Top-k String Retrieval Problems [*]

Wing-Kai Hon
Department of Computer Science
National Tsing-Hua University, Taiwan
Email: wkhon@cs.nthu.edu.tw

Rahul Shah
Department of Computer Science
Louisiana State University, LA, USA
Email: rahul@csc.lsu.edu

Jeffrey Scott Vitter
Department of Computer Science
Texas A&M University, TX, USA
Email: jsv@tamu.edu

Abstract— Given a set $\mathcal{D} = \{d_1, d_2, ..., d_D\}$ of D strings of total length n, our task is to report the "most relevant" strings for a given query pattern P. This involves somewhat more advanced query functionality than the usual pattern matching, as some notion of "most relevant" is involved. In information retrieval literature, this task is best achieved by using inverted indexes. However, inverted indexes work only for some predefined set of patterns. In the pattern matching community, the most popular pattern-matching data structures are suffix trees and suffix arrays. However, a typical suffix tree search involves going through all the occurrences of the pattern over the entire string collection, which might be a lot more than the required relevant documents.

The first formal framework to study such kind of retrieval problems was given by Muthukrishnan [25]. He considered two metrics for relevance: frequency and proximity. He took a threshold-based approach on these metrics and gave data structures taking $O(n \log n)$ words of space. We study this problem in a slightly different framework of reporting the top k most relevant documents (in sorted order) under similar and more general relevance metrics. Our framework gives linear space data structure with optimal query times for arbitrary score functions. As a corollary, it improves the space utilization for the problems in [25] while maintaining optimal query performance. We also develop compressed variants of these data structures for several specific relevance metrics.

Keywords-document retrieval; text indexing; succinct data structures; top-k queries

1. Introduction

In text pattern matching, the most fundamental problem is to find, given a text of size n and a pattern P of length p, all the locations in the text where this pattern matches. Earlier work has focussed on developing linear-time algorithms for this problem [19]. When the text is given beforehand, and the pattern queries come online, one might want to build a data structure on the text such that pattern matching queries can be answered in $O(p + \texttt{occ})$ time, where occ denotes the number of occurrences. Suffix tree [23, 35] is the most popular data structure which achieves this goal.

Most string databases consist of a collection of multiple text documents (or strings) rather than just one single text. In this case, the most natural question is to retrieve all the documents in which the query pattern occurs. A simple suffix tree search might be inefficient in this case as the search might involve browsing through a lot more occurrences than the actual number of qualifying documents. More formally, let $\mathcal{D} = \{d_1, d_2, ..., d_D\}$ be the collection of D strings of total length n. The characters of each string in $\mathcal{D}$ are drawn from the alphabet set Σ. Given a query pattern P (with length p), let occ be the number of occurrences of this pattern over the entire collection and let ndoc be the number of documents in which the pattern occurs. Then, ndoc may be much smaller than occ. The usual suffix tree based approach would give a query time of $O(p + \texttt{occ})$, which may be inefficient. Muthukrishnan [25] gave a linear-space data structure having optimal $O(p+\texttt{ndoc})$ query time for this *document listing* problem.

Muthukrishnan [25] also initiated a more appropriate study of document retrieval with various relevance metrics. The two problems considered by [25] were K*-mine* and K*-repeats*. In the K-mine problem, the query returns all the documents which have at least K occurrences of the pattern P. This basically amounts to thresholding by frequency. In the K-repeats problem, the query returns all the documents in which there is at least one pair of occurrences of the pattern P such that these occurrences are at most distance K apart. This relates to another popular heuristic in information retrieval called proximity. He gave $O(n \log n)$-word data structures for these problems which can answer the queries in optimal $O(p + \texttt{ndoc})$ time.

Sadakane [31] showed how to solve the document listing problem using succinct data structures which take space very close to that of the compressed text. He also showed how to compute the TF-IDF scores [36] of each document with such data structures. However, one limitation of Sadakane's approach is that it needs to first retrieve all the documents where the pattern (or patterns) occurs, and then find their relevance scores. The more meaningful task from the information retrieval point of view, however, is to get only some of the highly relevant documents. In this sense, it is very costly to retrieve all the documents first. Nevertheless, Sadakane did show some very useful tools and techniques for deriving succinct data structures for these problems. Similar work was also done by Välimäki and Mäkinen [34] where they derived alternative succinct data structures for the document listing problem. In all these papers, deriving succinct data structures for the more meaningful (in the IR sense) K-mine and K-repeats problems were listed as open

* Research supported in part by Taiwan NSC Grant 96-2221-E-007-082 (W.K. Hon) and US NSF Grant CCF–0621457 (R. Shah and J.S. Vitter).

problems. Further, no succinct data structure was known till date, and no $O(n)$ words (i.e., linear space in the traditional sense) data structure was known either.

We study these problems in a slightly more natural and more general framework of retrieving top k most relevant documents (in sorted order) to the query pattern P. For a pattern P and a document d, we define a score function $\mathit{score}(P, d)$ which captures the relevance score of document d with respect to the pattern P. In our framework, the score function can be arbitrary and can capture many practical measures, such as the frequency of P in d, the distance between two closest occurrences of P in d (known as proximity), or simply the static PageRank [27] of d which is independent of pattern P. The score function may also be some combinations of these measures.

Muthukrishnan's formulation tends to capture the notions of frequency and proximity by introducing the *K-mine* and *K-repeats* problems. However, he solved these problems separately and individually. We give a uniform framework for arbitrary score functions, and in that sense problems like *K-mine* and *K-repeats* are subcases under our framework. Also, one of the major challenges with Muthukrishnan's formulation is that it relies on the user to specify the correct threshold. Moreover, it is not easy to convert these into a top-k based solution without admitting some inefficiencies. Our model is more robust in the sense that both the top-k queries and the threshold-based queries can be answered with equal ease.

The design of succinct/compressed text indexing data structures has been an emerging field of research with great practical impact. In addition to deriving a linear space data structure with optimal query times, we further show how to achieve compressed data structures. Our compressed solutions take space very near to the entropy of the text data, albeit suffering from somewhat worse query times, which answer affirmatively to an open problem listed by [34].

1.1. Our Results

The following summarizes our results:

1) We provide a general framework to design linear space data structures for top-k string retrieval problems. Our framework works with any arbitrary score function. Many popular metrics like frequency, proximity, and importance are covered by this model. We achieve query time of $O(p + k \log k)$ for retrieving the top k documents in sorted order.
2) We provide a framework for designing succinct/compressed data structures for top-k string retrieval problems. Our main idea is based on sparsification which allows us to achieve better space (at the cost of somewhat worse query times). Our framework is applicable to any score function that can be calculated in compressed space. In the specific case when we score by frequency, our data structure occupies $2|CSA| + o(n)$ bits and answers queries in $O(p + k \operatorname{polylog} n)$ time.[‖]
3) Our framework improves the following existing results in the literature:
    a) K-mine problem: Muthukrishnan gave $O(n \log n)$-word indexes which answer queries in $O(p + \mathtt{ndoc})$ time. We present optimal $O(n)$-word[†] data structures which support queries in optimal $O(p + \mathtt{ndoc})$ time. We also give a succinct data structure for these problems, taking $2|CSA| + o(n) + D\log(n/D)$ space while answering queries in $O(p + \mathtt{ndoc} \operatorname{polylog} n)$ time.
    b) K-repeats problem: Once again, we improve Muthukrishnan's $O(n \log n)$-word structure to $O(n)$-word structure.
    c) Document Listing Problem: An $O(n)$-word data structure was proposed by [25], and the space was subsequently improved through a series of papers to $|CSA| + 2n + o(n) + D\log(n/D)$ bits with query answered in $O(p + \mathtt{ndoc} \log n)$ time [13, 31, 34]. We remove the additional $2n$ bits required (with slight increase in query time) to achieve a better space bound of $|CSA| + o(n) + D\log(n/D)$ bits.

1.2. Related Work

Pattern matching is a field of research which is almost about half a century old. Some of the earliest algorithms like [19] achieved optimal linear time performance. In the data structural sense, suffix trees [23, 35] and suffix arrays [21] are the most popular linear space data structures with optimal (or near-optimal) query performance. Although thought to be linear, the practical space requirement of suffix tree turned out to be about 15–50 times that of the text and and for suffix arrays this was almost about 5–20 times the text. Due to this limitation, they compared unfavorably to inverted indexes. Recently, Grossi and Vitter [17] and Ferragina and Manzini [11] gave compressed variants of text searching data structures, based on the Burrows-Wheeler Transform [6]. These data structures not only compared well with inverted indexes in their space utilization but also provided query functionality for arbitrary patterns. Since then, designing succinct or compressed data structures for text problems has been a thriving field of research with many improvements and extensions [2, 10, 26, 29, 30]. Puglisi et al [28] indeed showed that compressed text indexes provide faster searching than inverted indexes. However, the authors also showed that if the number of occurrences

[‖]Here, $|CSA|$ denotes space (in bits) of the compressed suffix array (CSA) [17] of the given documents in $\mathcal{D}$. It is shown that the average bits per character, $|CSA|/n$, is close to the empirical entropy of the documents.

[†]Space is optimal in the non-succinct manner.

(matching locations) are too many then again inverted indexes perform better in terms of document retrieval.

The formal study of document retrieval problems is motivated by this fact that occurrences of a pattern may be too many but the number of documents carrying the pattern might be fewer. Matias et al [22] gave the first solution for the document listing problem which answers this in $O(p \log D + \texttt{ndoc})$ time. Muthukrishnan [25] improved the result to optimal $O(p + \texttt{ndoc})$. He also initiated the formal framework for capturing the notion of "relevant" documents by introducing the K-mine and K-repeats problems, and gave data structures taking $O(n \log n)$ words of space for them. His work quickly motivated a flurry of new results with some seeking to improve them and some utilizing them in specific applications [7]. Many particular algorithms in bio-informatics have been based on the frequency metric [15, 20].

One particular line of research was to obtain compressed/succinct data structures for document listing problem by solving range minimum/maximum query (RMQ) using succinct variant of cartesian tree (See [31]). Although solving RMQ is as old as Chazelle's original paper on range searching [8], many simplifications [3] and improvements have been made, culminating in Fischer et al's $2n + o(n)$ bits of space data structure [13, 14]. Even our results shall extensively use RMQ as a tool to obtain top-k in a given set of ranges.

The study of reporting top-k matching items in the given range in sorted order can be traced back to McCreight's priority search trees [24]. Bialynicka-Birula and Grossi [4] gave a general framework to add rank information to items being outputted (from any range reporting data structure) and report top-k items in sorted order. We wish to note here that, although they achieve an additive term of $O(k)$ (as against our $O(k \log k)$) for reporting k items in sorted order, their data structure necessarily takes super-linear space. The other data structures having an $O(k)$ additive term like [16, 24] do not output the top-k items in sorted order. Although these data structures do not directly address the notion of frequency or proximity (when rank score is dependent on set of items rather than a single item) they can be used as alternative tools (in place of RMQ structures) in our framework also.

Top-k query processing has been an extensive field of research in the information retrieval and database community [9, 18]. Many theoretical results have also appeared in the context of aggregating ranks from various ranked lists [1, 32].

2. Preliminaries

2.1. Generalized Suffix Tree

Given a set of D strings $\{d_1, d_2, \ldots, d_D\}$ of total length n, the *generalized suffix tree* (GST) is a compact trie storing all suffixes of all the D strings. The GST consists of n leaves, each corresponds to a distinct suffix of a particular string. Each edge in the GST is labeled by a character string; for any node v, we denote $path(v)$ to be the string formed by concatenating the edge labels from the root to v. The edge labels in GST satisfies the following: For any leaf ℓ, $path(\ell)$ is equal to the suffix corresponding to ℓ.

Given a pattern P, the locus node v of P in the GST is defined as the node, closest to the root, such that P is a prefix of $path(v)$. It is known [23] that all occurrences of p, if exist, will exactly correspond to the leaves in the subtree of the locus node.[‡] The locus of P can easily be determined in $O(p)$ time, by traversing from the root of the GST and matching characters of P along the way.

2.2. Suffix Array

Usually, the edges of a node in the GST are ordered by the lexicographical order of its label, so that the ith leaf will correspond to the ith (lexicographically) smallest suffix. We use $SA[i]$ to denote the position where this suffix starts, and the array $SA[1..n]$ is called the *suffix array* for the documents. Given a pattern P, if P appears in any of the D strings indexed by the documents, then the suffixes corresponding to all occurrences of P will be in contiguous region of SA. More precisely, there exists a range $[L, R]$ such that $SA[L..R]$ are the starting positions of all occurrences of P. We call such a range $[L, R]$ the *SA range* of P.

2.3. Compressed Suffix Arrays

Let $T[1..n]$ be a text over an alphabet Σ. The basic pattern matching structure we use is the compressed version of the suffix array (CSA) for the text T, which supports the following operations: (1) Report $SA[i]$, which is the starting position of the ith smallest suffix of T; (2) Report $SA^{-1}[j]$, which is the rank of the jth suffix $T[j..n]$ among all suffixes of T; (3) Given a pattern P, compute the exact range $[L, R]$ such that P is the prefix of all suffixes with starting position $SA[L], SA[L+1], ..., SA[R]$.

There are various versions of CSA in the literature which provide different performance tradeoffs. Throughout this paper, we shall assume the version by Ferragina et al [12] and assume $|\Sigma| = O(\text{polylog}\, n)$, such that $SA[i]$ and $SA^{-1}[j]$ can be reported in $O(\log^{1+\epsilon} n)$ time, while the exact range $[L, R]$ for P can be computed in $O(p + \log^{1+\epsilon} n)$ time for any $\epsilon > 0$. Note that the CSA of [12] is also a self-index in that we do not have to explicitly maintain the original text. The total space is $nH_h + o(n \log |\Sigma|)$, where H_h denotes the empirical hth-order entropy of T.[**]

[‡]This follows from the fact that P occurs at a particular location in some document d if and only if P is the prefix of a particular suffix of d.

[**]The space bound holds for all $h < \alpha \log n / \log |\Sigma|$, where α is any fixed constant with $0 < \alpha < 1$.

2.4. Top-k using RMQs

We shall use RMQ data structures extensively to report the desired documents when answering our query. The basic result is captured in the following lemma:

Lemma 1. *Let A be an array of numbers. We can preprocess A in linear time and associate A with a linear-space RMQ data structure such that given a set of t non-overlapping ranges $[L_1, R_1]$, $[L_2, R_2]$, ..., $[L_t, R_t]$, we can find:* (i) *All numbers in $A[L_1, R_1]$, $A[L_2, R_2]$, ..., $A[L_t, R_t]$ which are more than (or less than) some specified number K in $O(t + \mathsf{occ})$ time.* (ii) *Largest (or smallest) k numbers in $A[L_1, R_1] \cup A[L_2, R_2] \cup \cdots \cup A[L_t, R_t]$ in $O(t + k \log k)$ time.*

Proof: See Appendix for the proof. □

2.5. Score Functions

Given a pattern P and document d, let S denote the set of all positions in d where P matches. We study a class of score functions score(P, d) which depend on the set S. Some useful examples of the score function include: (1) $\mathit{freq}(P, d)$ which is the cardinality of S; (2) $\mathit{mindist}(P, d)$ which is the minimum distance between any two positions in S; (3) $\mathit{docrank}(P, d)$ which is simply the static "importance" value associated with document d.

The functions $\mathit{freq}(P, d)$ and $\mathit{mindist}(P, d)$ are directly associated with K-mine and K-repeats problems respectively. Importance metric captured by $\mathit{docrank}(P, d)$ can simply be considered as PageRank [27] of the document d which is static and invariant of P.

In this paper, we focus on obtaining top-k highest scoring documents given the pattern P. In the design of our succinct/compressed solutions, some of the score calculation will be done on the fly. We call a score function *succinctly calculable* if there is a compressed data structure (like CSA) on document d which can calculate $\mathit{score}(P, d)$ in $O(p + \operatorname{polylog} n)$ time. Amongst the above functions, we shall show that $\mathit{freq}(P, d)$ is succinctly calculable (and so is $\mathit{docrank}$) but we do not know if $\mathit{mindist}(P, d)$ is succinctly calculable or not.

3. Linear Space Structure

In this section, we describe our non-succinct data structure with almost optimal query times. Nevertheless, this data structure takes linear space (as opposed to $O(n \log n)$ space of the earlier known data structures for related problems). We mainly focus the discussion on our top-k retrieval result, though this approach also works for the threshold versions [25].

First, we build a generalized suffix tree (GST) of all the suffixes in all the documents. Let $\ell_1, \ell_2, \ldots, \ell_n$ be the leaves of this GST. Each leaf of this suffix tree is annotated with two values $SA[i]$ which stores the position in the (concatenated) text of this suffix and d_i which stores the document id of the document to which this suffix belongs. We shall first describe our result in terms of frequency metric and then generalize to arbitrary score function.

3.1. N-structure

At any node v of GST, we store an N-structure N_v which is an array of 5-tuples $\langle$document d, frequency c, parent pointer t, first depth δ_f, last depth $\delta_l\rangle$. First, N_v for any leaf node ℓ_i will contain exactly one entry with document d_i and frequency 1. For an internal node v, an entry for a document d occurs in N_v if and only if at least two children of v contain document d in their subtrees. In case the entry of document d is present in N_v then its corresponding value c denotes the frequency of d in the subtree of v. The parent pointer t points to the lowest ancestor of v which also has an entry of document d. In case there is no such ancestor, then the pointer t points to a dummy node which is regarded as the parent of the root of GST. For δ_f and δ_l, we shall give their definitions and describe their usage later.

Observation 1. *Let ℓ_i and ℓ_j be two leaves belonging to the same document d. If v is the lowest common ancestor $lca(\ell_i, \ell_j)$, then N_v contains an entry for document d.*

Observation 2. *If for two nodes u, w both N_u and N_w contain an entry for document d, then the node $z = lca(u, w)$ also has an entry for document d in N_z.*

Lemma 2. *For any node v and any document d which occurs at some leaf in the subtree of v, there is exactly one pointer t such that (i) t corresponds to document d, (ii) t originates at some node in the subtree of v, and (iii) t points to some node not in the subtree of v.*

Proof: It is easy to check that at least one pointer t will simultaneously satisfy the three properties. To show that t is unique, suppose on the contrary that two nodes u and w, which are in the subtree of v, both contain an entry of document d and with the corresponding parent pointers pointing to some nodes outside the subtree of v. By Observation 2, $z = lca(u, w)$ also has an entry for d. Consequently, the parent pointers at u and w must point to some nodes in the subtree of z. On the other hand, since both u and w are in subtree of v, z must be in the subtree of v. The above statements immediately imply that the parent pointers of u and w are pointing to some nodes *in the subtree of* v. Thus contradiction occurs and the lemma follows. □

Lemma 3. *The total number of internal nodes which have an entry for document d is at most $|d| - 1$, where $|d|$ denotes the number of characters in document d.*[†] □

3.2. I-structure

Based on pointer field in the N-structure, we are now ready to describe another structure I_v stored at every internal

[†]Here, we exclude the dummy node.

node v. For each pointer t in some N-structure which points to node v, I_v contains a corresponding entry which stores information about the origin of t. Specifically, the entry for t inside I_v stores a triplet $\langle$origin r, document d, frequency $c\rangle$, where r denotes the pre-order rank (in GST) of the node w from which t originates, while document id d and frequency c indicate $\langle t, d, c\rangle$ is an entry in the N-structure N_w.

The entries in the I-structure I_v are sorted by increasing order of pre-order ranks of the origin. If there is a tie in rank of the origin, we order the entries by document id d. That is all the origin values r occur in increasing order in I_v. We store I_v by an array whose jth entry is denoted by $\langle r[j], d[j], c[j]\rangle$. Note that some entries in I_v may have the same r-values, which happens if their origin is common whose N-structure has more than one entries (corresponding to different documents) pointing to v; in this case these entries in I_v are ordered by the document ids. Also, note that in a given I_v we also have repeated values for document id (when they have different origins). Finally, we store the range-maximum query structure on the array c of frequency values.

Lemma 4. *The total number of entries $\sum_v |I_v|$ in all I-structures is $O(n)$.*

Proof: The total number of entries in I-structures is same as the total number of pointers. This in turn is the total number of entries in N-structures. By Lemma 3, the total number of such entries inside all internal nodes is at most $\sum_{i=1}^{D}(|d_i|-1) \le n$. On the other hand, the number of such entries inside all leaves is exactly n, so that the total number is at most $2n$. $\square$

3.3. Answering Queries

First, we match the pattern P in GST in $O(p)$ time and reach at its locus node v. By the property of GST, all the leaves in the subtree correspond to the occurrences of P. Now by Lemma 2, for each document that appears in the subtree of v, there will be a unique parent pointer, originated from some node in the subtree, pointing to some ancestor node of v. Thus, to answer top-k most frequent documents, it is sufficient if we can identify such pointers, and select the corresponding documents which have top-k highest frequency values.

By definition of the I-structure, we know that each of these pointers must exist in one of the entries in I-structure of one of ancestors of v. Let $rank(w)$ denote the pre-order rank of a node w in GST. For the locus v we have just reached, let $L_v = rank(v)$ and let R_v be the highest pre-order rank of any node in the subtree of v. Note that all the nodes in subtree of v have contiguous pre-order ranks. Thus, nodes in the subtree of v can be represented by the range $[L_v, R_v]$.

Now, for each ancestor u of v, the entries in the I-structure I_u is sorted according to the pre-order rank of the origins r. The contiguous range in the origin array r, with values from $[L_v, R_v]$, will correspond to parent pointers originated from the nodes in the subtree of v (that point to u). Suppose such a range can be found in the array I_u for each ancestor u. That is, we can find i_u and j_u such that $I_u[i_u]$ is the first entry such that its r value is at least L_v, and $I_u[j_u]$ is the last entry such that its r value is at most R_v. Then we can examine the frequency array $c[i_u..j_u]$ for each I_u together and apply Lemma 1(ii) to return the k documents with the highest frequency.

The range $[i_u, j_u]$ for each I_u can be found in $O(\log\log n)$ time if we have maintained a predecessor data structure over the array r. The number of ancestors of v is at most $depth(v)$. Since $depth(v) \le p$, this range translation takes at most $O(p \log\log n)$ time overall. The subsequent step by using Lemma 1(ii) then takes $O(p + k \log k)$ time. So in total, the top-k frequent documents can be returned in $O(p \log\log n + k \log k)$ time.

3.4. Improvement

Our main bottleneck is the predecessor structure for range translation which costs us $O(p \log\log n)$ time. Now, we shall see how we can convert $O(p \log\log n)$ term to $O(p)$. Notice that the $\log\log n$ factor comes from the need of translating the range $[L_v, R_v]$ in the I-structure of each of the ancestor of v. Next, we shall show how we can use the two fields δ_f and δ_l to directly map the range without having to resort to the predecessor query.

Firstly, recall that in the I_u structure of an ancestor u of v, if an entry whose origin is from the subtree of v, then its pre-order rank must be between L_v and R_v. Further, if the origin of such an entry e is the smallest, e will be the i_uth entry of I_u (i.e., the left boundary of the range $[i_u, j_u]$). Note that the origin of e must be the first node (in pre-order rank) among all nodes in the subtree of v having parent pointers to u.

The above motivates us to define the δ_f value for an entry in the N-structure N_x of any node x as follows: Let y be the ancestor of x where the parent pointer t_x points to; Let w be the node on path from x to y, closest to y, such that from the subtree of w, x is the first node (in pre-order rank) with parent pointer pointing to y. Thus, whenever the locus node for a pattern P is between w (inclusive) and x (inclusive), the left boundary of the range in I_y will be originated from x. Then δ_f is defined to be $depth(w)$, and associate this with such a parent pointer t_x. If there is no such node w, δ_f is set to be ∞. Also, if t_x is the very first pointer in I_y, then δ_f of t_x is set to $depth(y)$. Note that in the case $\delta_f \ne \infty$, let z be the origin node of the pointer just before t_x in I_y; then δ_f must be exactly one more than the depth of the lowest common ancestor (lca) of x and z. Similarly, we define δ_l with respect to the right boundary of the range in I_y.

Let us now get back to the original problem of finding left and right boundaries in I_u of each ancestor u of v. Based on the definitions of δ_f and δ_l, we have the following lemma:

Lemma 5. *Consider all the pointers originating from the subtree of v (i.e., the pointers that are in the N-structure of some descendant of v). If such a pointer satisfies $\delta_f \leq depth(v)$ (resp. $\delta_l \leq depth(v)$), then there exists an ancestor u of v such that this pointer is the first (resp. last) among all the pointers in the I-structure I_u which originate in the subtree of v.*

Conversely, for any ancestor u of v, if a pointer t is the first (resp last) pointer among all the pointers in I_u which originate in the subtree of v, then t satisfies $\delta_f \leq depth(v)$ (resp. $\delta_l \leq depth(v)$).

Proof: For the first part of the lemma, consider a pointer t originating in the subtree of v satisfies $\delta_f \leq depth(v)$. Suppose that t points to I-structure I_u for some ancestor u of v. Now assume to the contrary that this is not the first pointer originating in subtree of v which reaches I_u. Then, there exists another pointer s originating in the subtree of v also reaching I_u and such that pre-order rank of origin of s is less than that of t. In this case, δ_f of t must be strictly more that the depth of lca of these two originating nodes. Since, both the nodes are in subtree of v, the lca is also in the subtree of v. Thus, δ_f of t is strictly more than $depth(v)$.

For the converse, suppose t is the first pointer to reach I_u from the subtree of v, then consider a pointer s which appears just before t in I_u. The origin of s must be outside the subtree of v. Thus, δ_f of t is strictly more than the lca of the origins of s and t. Since this lca must be some proper ancestor of v, δ_f of t is at most $depth(v)$.

Similar arguments work for the case of δ_l. □

By the above lemma, if we can search all the pointers originating in the subtree of v satisfying $\delta_f \leq depth(v)$ (resp. $\delta_l \leq depth(v)$), we can find the desired left (resp. right) boundaries in I_u for each ancestor u. To facilitate the above search, we shall visit each node of the GST in pre-order, and concatenate the N-structures for all the nodes in one single array N, and construct two RMQ data structures over the δ_f entries and δ_l entries, respectively. Thus, there is a contiguous range $[start, end]$ in N corresponding to the subtree of v. Now we find all the δ_f and δ_l values in this range which are less than $depth(v)$ using Lemma 1(i), and thus obtain the desired pointers. As there are at most $2 \times depth(v)$ such pointers reported, the total time is $O(depth(v))$, which is $O(p)$.

Theorem 1. *We can design an $O(n)$ words data structure which can answer the top-k most frequent documents for a given pattern P in $O(p + k \log k)$ time.* □

3.5. Comparison with Muthukrishnan's K-mine Problem

In K-mine problem, one needs to output all the documents that have K or more occurrences of the pattern p. This can be easily done by applying Lemma 1(i) once the range $[i_u, j_u]$ in each ancestor u is ready. This gives the following result:

Theorem 2. *We can design an $O(n)$ data structure for K-mine problem which answers the queries in $O(p + \texttt{ndoc})$ time.* □

Thus our structure directly compares with Muthukrishnan's data structure, saving space by a factor of $O(\log n)$ while matching Muthukrishnan's optimal query time.

By simply replacing the frequency value by "score", we can easily generalize this structure to use arbitrary score functions (We remark that only construction algorithm may change depending on how easy it is to evaluate a given score function).

Theorem 3. *We can design an $O(n)$ words data structure such that we can answer top-k best scoring documents for a given pattern p in $O(p+k \log k)$ time. If our task is to output all the documents with more than some threshold score s, then it can be done in $O(p + \texttt{ndoc})$ time.* □

Corollary 1. *We can design $O(n)$-word data structure for K-repeats problem of [25] which can answer queries in $O(p + \texttt{ndoc})$ time.*

Again, we save the space by a factor of $O(\log n)$ over Muthukrishnan's K-repeats data structure.

3.6. Construction Algorithms

Although our data structural framework is very general for arbitrary score functions, the running time of our construction algorithm depends on how easily we can calculate the score for a given set of positions.

Firstly, we construct a generalized suffix tree in $O(n)$ time. Next, we construct the LCA data structure of [3], also in $O(n)$ time, so that the lca of any two nodes in the GST can be reported in $O(1)$ time. Then, for each document d, we traverse all the leaves corresponding to d in GST and add an entry for d in each node which is an lca of successive leaves from document d. Next, we construct a suffix tree for document d in $O(|d|)$ time, then traverse this tree in a post-order. Note that there is a one-to-one correspondence between the nodes in this tree and the lca's found in the GST. Consequently, the parent pointer values of all entries can be determined while the frequency counts can be calculated by maintaining subtree sizes along the traversal. In total, the first three tuples of all entries in all N-structures (i.e., document d, frequency c, and parent pointer t) can be initialized in $O(n)$ time.

Next, we traverse the GST in pre-order fashion, and corresponding to each parent pointer in the N-structure encountered, we add an entry to I-structure of respective node. Once, the entries in each I-structure are ready, we visit each I-structure and construct an RMQ data structure over it. This overall takes $O(n)$ time.

Now, it remains to show how to calculate the δ_f (similarly δ_l) values. For this, we traverse each of the I-structure I_w sequentially and get the list of pointing nodes (they appear in pre-order). Now, again we take successive lca's between consecutive pointing nodes. The δ_f value for a particular node v is exactly equal to 1 plus the depth of the lca of v and its previous node (in I_w), which can be computed in $O(1)$ time. After computing all the δ_f values in all entries, we traverse all the N-structures in pre-order and construct RMQ structure over δ_f values. All of this can be accomplished in $O(n)$ time.

In case of *mindist* score function, we need more time to calculate the score function (this is the only change). In this case, the scores are first calculated over the suffix tree of document d. For this, we do a recursive computation. Say at a node v, we have two children v_1 and v_2. Also assume that following is available at v_1 (and v_2): (1) $\mathit{mindist}(v_1)$[††]; (2) a list $\mathcal{L}_1$ of text positions appearing in the subtree of v_1 in sorted format (stored as a binary search tree). Then, we first merge the list $\mathcal{L}_1$ at v_1 and the list $\mathcal{L}_2$ at v_2 to obtain the list $\mathcal{L}$ at v and also during this merge operation we find out the closest pair of positions with one coming from the list at v_1 and the other from v_2. Now we compare the distance of this pair with $\mathit{mindist}(v_1)$ and $\mathit{mindist}(v_2)$ and obtain $\mathit{mindist}(v)$ for v. This merging step can be done in $O(|\mathcal{L}_1|\log(|\mathcal{L}_2|/|\mathcal{L}_1|))$ time (assuming $|\mathcal{L}_1| \le |\mathcal{L}_2|$) using Brown and Tarjan's fast merging algorithm [5]. The total time can be shown to be $O(n \log n)$ for processing the entire document (See a similar analysis in [33]). This thus gives us an $O(n \log n)$ algorithm for calculating *mindist* scores over the GST.

4. Succinct Structures

In this section, we describe succinct structure for the problem of top-k string retrieval. Our structures are based on the idea of sparsification.

4.1. Two Versions of CSA

We shall first maintain a CSA over the whole collection of concatenated text. When we concatenate all the text documents, we also need to maintain a structure which remembers the text boundaries. This is represented by bit-vector $\mathcal{B}$ over the concatenated text such that the bit is 1 if and only if it corresponds to the last character of some document in the concatenated text. We store this bit vector in compressed form and build a data structure over it supporting rank and select queries. This takes $D \log(n/D) + o(n)$ bits. Note that concatenated text is never stored explicitly. We only store CSA over it.

[††]We slightly overload the *mindist* notation where $\mathit{mindist}(v_1)$ denotes the minimum distance between the positions appearing in the subtree of v_1. If we stick to the earlier definition, this is exactly $\mathit{mindist}(\mathit{path}(v_1), d)$.

Next, for each document d, we build a separate compressed suffix array CSA_d. The sum of all these individual CSA_d's is bounded by $|CSA|$. Next we show a $o(n)$-space structure which maintains some useful top-k statistics on the top of these CSAs.

4.2. The Sparsified Top-k Structure

First, we shall show the sparsified structure for a fixed value of k. Consider a generalized suffix tree GST of the documents. We fix $g = k \log^{2+\epsilon} n$ be the group size, and traverse the leaves of GST from left to right to form groups of contiguous g leaves. Thus, the first group consists of $\ell_1, \ell_2, \ldots, \ell_g$, the next group consists of $\ell_{g+1}, \ldots, \ell_{2g}$, and so on. In total, there are n/g groups. Now for each group, we mark the lca in GST of its first and last leaf. Thus, we have marked at most n/g internal nodes. Now we do further marking. If nodes u and v are marked, then $lca(u, v)$ also gets marked. Thus, the set of marked nodes becomes closed under the lca operation, and it is easy to show that total number of marked nodes is no more than $2n/g$.

We maintain a list of size k called *F-list* corresponding to each marked node v. This F-list consists of top k most frequent (or highest scoring) documents in the subtree of v; in addition, we store the corresponding frequencies (score) along with the documents.

Next, we construct a tree τ_k consisting of only marked nodes in GST. Each node v in τ_k is originally a node in the GST; we store along with v the corresponding SA range $[L_v, R_v]$ (i.e., the range of leaves in the GST within the subtree of v) and also the corresponding F-list F_v. Now, τ_k has at most $2n/g = 2n/(k \log^{2+\epsilon} n)$ nodes, each having $O(k)$ storage due to the F-list. The total space taken by this structure is thus $O(n/\log^{2+\epsilon} n)$ words or $O(n/\log^{1+\epsilon} n)$ bits.

We shall store τ_k explicitly for the values of $k = 1, 2, 4, 8, 16, \ldots$. That is, for all values which are powers of 2. Thus, we store at most $O(\log n)$ such structures. Thus, the total space taken by these structures is $O(n/\log^{\epsilon} n) = o(n)$ bits. We remark that the original GST will not be stored.

4.3. Answering Queries

First, we round up k to its closest higher power of 2. Then, we match pattern P in CSA in $O(p + \log^{1+\epsilon} n)$ time and find out the range $[L, R]$ in the suffix array such that the pattern matches at locations $SA[L], SA[L+1], \ldots, SA[R]$. Now, we traverse τ_k based on this range $[L, R]$, starting from the root and visiting appropriate child and so on, until we reach the first node v whose SA range $[L_v, R_v]$ is contained in $[L, R]$. Note that the node v is indeed the locus of P in τ_k if we consider τ_k as a "reduced version" of the original GST.

Claim 1. *Both the quantities $L_v - L$ and $R - R_v$ are at most g.*

Proof: Suppose $L_v - L$ was more than g. Then there must have been a marked node w in GST representing the leaves $\ell_{L_v-g}, \ell_{L_v-g+1}, ..., \ell_{L_v-1}$. And hence there must have been a marked node $z = lca(v, w)$. Then, z would also satisfy the containment property and since z is an ancestor of v, this would contradict the fact that v is the first such node encountered. $\square$

Now, we check the F-list F_v of v which contains the top-k frequent documents when we restrict our attention to the positions in the L_vth to the R_vth leaves in the original GST. To get the correct top-k frequent documents, we have to take account to all positions under the subtree of v (i.e., from the Lth to the Rth leaves in the original GST). In this aspect, we now examine each leaf $\ell_L, \ell_{L+1}, ..., \ell_{L_v-1}$ and find out its corresponding document d, using $O(\log^{1+\epsilon} n)$ time per leaf (by computing the values $SA[L], SA[L+1], \ldots$). These documents may potentially be the top-k most frequent documents, so that we now calculate the frequency of each in the range $[L, R]$. This is done as follows: (1) Starting with the leaf ℓ_i, we find $SA[i]$ and thus its corresponding document d; (2) Next, we observe that for any position j in CSA_d, we can identify the position j' in CSA such that $SA_d[j]$ and $SA[j']$ are referring to the same suffix of document d.[‡‡] This is done by SA and SA^{-1} operations on the two CSAs in $O(\log^{1+\epsilon} n)$ time (and also constant number of rank/select operations on the bit-vector $\mathcal{B}$); (3) Another important observation is that the suffixes of document d have the same relative rank in SA and SA_d. Thus, we can perform binary search (using the two CSAs) to find out the range $[L_d, R_d]$ in SA_d which corresponds to those suffixes of d stored within $[L, R]$ in SA. The total time to report L_d and R_d is $O(\log^{2+\epsilon} n)$ time. (4) Finally, we return $R_d - L_d + 1$ as the frequency of document d in range $[L, R]$ in GST. We repeat the same for leaves $\ell_{R_v+1}, \ldots, \ell_R$. Totally we find the frequencies of these at most $2g$ documents (each costing us $O(\log^{2+\epsilon} n)$ time). The total time taken by this is $O(k \log^{4+\epsilon} n)$.

Once, we have this set of frequencies (the ones we calculated and ones we got from F-list of v), we report top-k amongst them using the standard linear-time selection algorithm (in $O(g + k)$ time), followed by a sorting of the k selected documents (in $O(k \log k)$ time).

Theorem 4. *We can design a succinct data structure taking $2|CSA| + o(n) + D \log(n/D)$ space which can answer the top-k frequent query in $O(p + k \log^{4+\epsilon} n)$ time.*

4.4. Extensions

Although we described our result in terms of frequency as a scoring function, we can in fact extend it to some other scoring functions which are succinctly calculable. Unfortunately, we do not know if $mindist(P, d)$ is succinctly calculable or not. On the other hand, $docrank(P, d)$ is not only succinctly calculable, but is trivial to compute. In this case, we do not even need the individual CSA for each document. We just consider each of the $2g$ fringe leaves and get the $docrank$ of their corresponding documents (each in $O(\log^{1+\epsilon} n)$ time). After that, we combine these $2g$ documents with the k most important documents in F-list of v, then perform a linear-time selection, and then perform a sorting on the selected documents. As $g = k \log^{2+\epsilon} n$, we can get top k most important documents in $O(p + k \log^{3+\epsilon} n)$ time.

Theorem 5. *We can design a succinct data structure taking $|CSA| + o(n) + D \log(n/D)$ space which can answer the top-k most important document query in $O(p + k \log^{3+\epsilon} n)$ time.*

Although our framework is for top-k, we can easily use our data structure to answer Muthukrishnan's K-mine problem. All we do is to first search for $k = 1$, then search for $k = 2$, and then $k = 4$ and so on, until the least frequent document among the top-k documents contains less than K occurrences of P. The total number of documents reported is at most $1 + 2 + 4 + \cdots + \mathtt{ndoc} = O(\mathtt{ndoc})$. Note that we can easily combine all the τ_k's into one single tree so that searching for range of P in each of these τ_k's can be done together.

Theorem 6. *We can design a succinct data structure taking $2|CSA| + o(n) + D \log(n/D)$ space which can answer the K-mine query in $O(p + \log^2 n + \mathtt{ndoc} \log^{4+\epsilon} n)$ time.*

As a further extension of our sparsifying framework, we show that it can be applied to document listing problem also. Firstly, by using $docrank$ criteria (and setting the importance of each document to be the same), document listing can be easily done in $O(p + \mathtt{ndoc} \log^{3+\epsilon} n)$ (Theorem 5). We can further improve the query time to $O(p + \mathtt{ndoc} \log^{1+\epsilon} n)$ by combining the succinct cartesian tree technique of [31]. We sketch this result next.

We first group the leaves of GST into consecutive groups size of $\log^{\epsilon} n$ leaves each. Now we consider the array $Z[i]$ which stores the location j such that $j < i$, leaf i and leaf j belong to the same document, and j is the largest such index. To answer the document listing query, Muthukrishan [25] showed that we can repeatedly apply range minimum queries on the interval $[L, R]$ given by subtree of locus node v; then the desired documents will exactly correspond to the entries in $[L, R]$ whose Z values are less than L. To save space, we apply the following modification used by Sadakane [31]. We first create a sampled version Z' of Z. In this we only include smallest Z value in each group of size $\log^{\epsilon} n$. We make cartesian tree on Z' and find all the qualifying leaves from Z' whose Z' (and also Z) values are less than L. Now for each chosen value from Z', we check all the leaves in its group. This checking takes at most $O(\log^{1+\epsilon} n)$ in each

[‡‡]Here, SA denotes the suffix array of the concatenated text, and SA_d denotes the suffix array of the document d.

group and this cost is amortized against the chosen value in Z'.

Theorem 7. *We can design a succinct data structure taking $|CSA| + o(n) + D\log(n/D)$ space which can answer the document listing query in $O(p + \texttt{ndoc}\log^{1+\epsilon} n)$ time.*

5. Conclusions and Future Work

We have shown a robust framework for designing space-conscious data structures for a broad class of string retrieval problems. Our approach works with many natural scoring functions. Our first framework achieves optimal query performance and while taking linear space. This is an improvement over earlier data structures for some of the specific problems which took $O(n \log n)$ words of space. We have also shown a framework for deriving succinct data structures for such problems. Although its query times are not optimal in theory, this framework carries high practical appeal, in the sense that it could potentially beat inverted indexes in both space and time. The main bottleneck which comes from calculating scores of fringe documents is mainly theoretical. It leaves room for many practical tricks to optimize space and time. It will be interesting to see if practical tuning of this framework can lead to a robust string retrieval system.

References

[1] N. Ailon, M. Charikar, and A. Newman, "Aggregating Inconsistent Information: Ranking and Clustering," in *Proceedings of Symposium on Theory of Computing*, 2005, pp. 684–693.

[2] J. Barbay, M. He, J. I. Munro, and S. S. Rao, "Succinct Indexes for Strings, Binary Relations and Multi-labelled Trees," in *Proceedings of Symposium on Discrete Algorithms*, 2007, pp. 680–689.

[3] M. A. Bender and M. Farach-Colton, "The LCA Problem Revisited," in *Proceedings of Latin American Symposium on Theoretical Informatics*, 2000, pp. 88–94.

[4] I. Bialynicka-Birula and R. Grossi, "Rank-Sensitive Data Structures," in *Proceedings of International Symposium on String Processing and Information Retrieval*, 2005, pp. 79–90.

[5] M. R. Brown and R. E. Tarjan, "A Fast Merging Algorithms," *Journal of the ACM*, vol. 26, no. 2, pp. 211–226, 1979.

[6] M. Burrows and D. J. Wheeler, "A Block-sorting Lossless Data Compression Algorithm," Digital Equipment Corporation, Paolo Alto, CA, USA, Tech. Rep. 124, 1994.

[7] S. Chakrabarti, K. Puniyani, and S. Das, "Optimizing Scoring Functions and Indexes for Proximity Search in Type-Annotated Corpora," in *Proceedings of International Conference on World Wide Web*, 2006, pp. 717–726.

[8] B. Chazelle, "A Functional Approach to Data Structures and Its Use in Multidimensional Searching," *SIAM Journal on Computing*, vol. 17, no. 3, pp. 427–462, 1988.

[9] R. Fagin, A. Lotem, and M. Naor, "Optimal Aggregation Algorithms for Middleware," in *Proceedings of Symposium on Principles of Database Systems*, 2001.

[10] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan, "Structuring Labeled Trees for Optimal Succinctness, and Beyond," in *Proceedings of Symposium on Foundations of Computer Science*, 2005, pp. 184–196.

[11] P. Ferragina and G. Manzini, "Indexing Compressed Text," *Journal of the ACM*, vol. 52, no. 4, pp. 552–581, 2005, a preliminary version appears in FOCS'00.

[12] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro, "Compressed Representations of Sequences and Full-Text Indexes," *ACM Transactions on Algorithms*, vol. 3, no. 2, 2007.

[13] J. Fischer and V. Heun, "A New Succinct Representation of RMQ-Information and Improvements in the Enhanced Suffix Array," in *Proceedings of Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*, 2007, pp. 459–470.

[14] J. Fischer, V. Heun, and H. M. Stühler, "Practical Entropy-Bounded Schemes for $O(1)$-Range Minimum Queries," in *Proceedings of Data Compression Conference*, 2008, pp. 272–281.

[15] J. Fischer, V. Mäkinen, and N. Välimäki, "Space Efficient String Mining under Frequency Constraints," in *Proceedings of International Conference on Data Mining*, 2008, pp. 193–202.

[16] H. Gabow, J. L. Bentley, and R. E. Tarjan, "Scaling and Related Techniques for Geometry Problems," in *Proceedings of Symposium on Theory of Computing*, 1984, pp. 135–143.

[17] R. Grossi and J. S. Vitter, "Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching," *SIAM Journal on Computing*, vol. 35, no. 2, pp. 378–407, 2005, a preliminary version appears in STOC'00.

[18] I. F. Ilyas, G. Beskales, and M. A. Soliman, "A Survey of Top-K Query Processing Techniques in Relational Database Systems," *ACM Computing Surveys*, vol. 40, no. 4, 2008.

[19] D. E. Knuth, J. H. Morris, and V. B. Pratt, "Fast Pattern Matching in Strings," *SIAM Journal on Computing*, vol. 6, no. 2, pp. 323–350, 1977.

[20] V. Mäkinen, G. Navarro, J. Siren, and N. Välimäki, "Storage and Retrieval of Individual Genomes," in *Proceedings of International Conference on Computational Molecular Biology*, 2009.

[21] U. Manber and G. Myers, "Suffix Arrays: A New Method for On-Line String Searches," *SIAM Journal on Computing*, vol. 22, no. 5, pp. 935–948, 1993.

[22] Y. Matias, S. Muthukrishnan, S. C. Sahinalp, and J. Ziv, "Augmenting Suffix Trees, with Applications," in *Proceedings of European Symposium on Algorithms*, 1998, pp. 67–78.

[23] E. M. McCreight, "A Space-economical Suffix Tree Construction Algorithm," *Journal of the ACM*, vol. 23, no. 2, pp. 262–272, 1976.

[24] ——, "Priority Search Trees," *SIAM Journal on Computing*, vol. 14, no. 2, pp. 257–276, 1985.

[25] S. Muthukrishnan, "Efficient Algorithms for Document Retrieval Problems," in *Proceedings of Symposium on Discrete Algorithms*, 2002, pp. 657–666.

[26] G. Navarro and V. Mäkinen, "Compressed Full-Text Indexes," *ACM Computing Surveys*, vol. 39, no. 1, 2007.

[27] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank Citation Ranking: Bringing Order to the Web," Stanford InfoLab, Technical Report 1999-66, November 1999.

[28] S. J. Puglisi, W. F. Smyth, and A. Turpin, "Inverted Files Versus Suffix Arrays for Locating Patterns in Primary Memory," in *Proceedings of International Symposium on String Processing and Information Retrieval*, 2006, pp. 122–133.

[29] K. Sadakane, “New text indexing functionalities of the compressed suffix arrays,” *Journal of Algorithms*, vol. 48, no. 2, pp. 294–313, 2003, a preliminary version appears in ISAAC 2000.

[30] ——, “Compressed Suffix Trees with Full Functionality,” *Theory of Computing Systems*, pp. 589–607, 2007.

[31] ——, “Succinct Data Structures for Flexible Text Retrieval Systems,” *Journal of Discrete Algorithms*, vol. 5, no. 1, pp. 12–22, 2007.

[32] K. Schnaitter and N. Polyzotis, “Evaluating Rank Joins with Optimal Cost,” in *Proceedings of Symposium on Principles of Database Systems*, 2008, pp. 43–52.

[33] R. Shah and M. Farach-Colton, “Undiscretized Dynamic Programming: Faster Algorithms for Facility Location and Related Problems on Trees,” in *Proceedings of Symposium on Discrete Algorithms*, 2002, pp. 108–115.

[34] N. Välimäki and V. Mäkinen, “Space-Efficient Algorithms for Document Retrieval,” in *Proceedings of Symposium on Combinatorial Pattern Matching*, 2007, pp. 205–215.

[35] P. Weiner, “Linear Pattern Matching Algorithms,” in *Proceedings of Symposium on Switching and Automata Theory*, 1973, pp. 1–11.

[36] I. Witten, A. Moffat, and T. Bell, *Managing Gigabytes: Compressing and Indexing Documents and Images*. Los Altos, CA, USA: Morgan Kaufmann Publishers, 1999.

APPENDIX

1. *Proof of Lemma 1*

Proof: For (i), we process each range $[L_i, R_i]$ independently as follows:

(1) We issue a range maximum query and extract the maximum element m_i. (2) If $m_i < K$, we stop and process the next range. (3) Otherwise, we add m_i to our answer list, split the range $[L_i, R_i]$ into two new ranges around m_i, and recursively find the desired elements in these two ranges. The total time required is $O(t + \texttt{occ})$ for all t ranges.

For (ii), we issue range maximum queries in each of these ranges and find the the maximum value in each. Now, we take largest k of them (take all of them if $k > t$) and put them in a binary search tree. We shall use this binary search tree as a 2-sided heap which supports extract-min as well as extract-max operations. We maintain the invariant that after every iteration the number of elements in the binary search tree is at most k.

In each iteration, we do the following: (1) We first extract the maximum frequency element and add it to our answer list. (2) Then, we take the range corresponding to our extracted element and split it into two ranges (around this extracted element). (3) Next, for each of these two new ranges, we insert their respective range-maximum element in the binary search tree. (4) After that, we delete (at most two) elements with minimum values until there are at most k elements in the tree, and discard the ranges corresponding to the deleted elements from consideration.

Thus, after k iterations, we get top k largest numbers in decreasing order. The binary search tree operations take $O(\log k)$ per iteration. Hence, our total time requirement is $O(t + k \log k)$. □