View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Elsevier - Publisher Connector

Theoretical Computer Science 426–427 (2012) 66–74

Contents lists available at SciVerse ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

On demand string sorting over unbounded alphabets

Carmel Kent [a], Moshe Lewenstein [b,*], Dafna Sheinwald [a]

[a] *IBM Research Lab, Haifa, Israel*

[b] *Department of Computer Science, Bar-Ilan University, Ramat Gan, Israel*

ARTICLE INFO

Article history:
Received 1 June 2011
Received in revised form 1 December 2011
Accepted 2 December 2011
Communicated by M. Crochemore

Keywords:
String matching
Data structures

ABSTRACT

On-demand string sorting is the problem of preprocessing a set of strings to allow subsequent queries for finding the k lexicographically smallest strings (and afterward the next k etc.) This on-demand variant strongly resembles the search engine queries which give you the best k-ranked pages recurringly.

We present a data structure that supports this in $O(n)$ preprocessing time, where n is the number of strings, and answer queries in $O(\log n)$ time. There is also a cost of $O(N)$ time amortized over all operations, where N is the total length of the strings.

Our data structure is a heap of strings, which supports heapify and delete-mins. As it turns out, implementing a full heap with all operations is not that simple. For the sake of completeness, we propose a heap with full operations based on balanced indexing trees that supports the heap operations in optimal times.

© 2011 Elsevier B.V. All rights reserved.

1. Introduction

Sorting strings is a fundamental algorithmic task that has been intensively researched in various flavors. The classical problem appears in textbooks [1,17], and variants of the problem have attracted much interest over the years, e.g. multikey sorting [9,22,5], parallel string sorting [12,11,14] and cache-aware string sorting [3,23]. Even integer sorting can be considered a special case of this problem.

Over the last few years, there has also been much interest in indexing structures, such as suffix trees [7,20,24,26], suffix arrays [19] and suffix trays [6]. The strong connection between suffix tree construction and suffix sorting was stressed in [7] and in the extended journal version [8]. In fact, suffix arrays are an array containing a lexicographic ordering of the suffixes. One of the exciting results of this field is a linear time algorithm to construct suffix arrays, see [15,16,18]. These results are followed along the line of the suffix tree construction of Farach [7] (note that the leaves of the suffix tree also represent a lexicographic ordering of the suffixes of the string). Nevertheless, the linear time results hold for alphabets which can be sorted in linear time. For unbounded alphabets, the time to sort the strings is still $O(n \log n)$, where n is the string length.

While all suffixes of an n length string (even over an unbounded alphabet) can be sorted in $O(n \log n)$ time, when it comes to sorting strings, one needs to take into consideration also the overall length of the strings, which we denote by N. Nevertheless, it is known that one can sort strings in time $O(n \log n + N)$, which matches the lower bound in the comparison model. To achieve this time bound, one may use a weight balanced ternary search trie [21] or adapt mergesort [13] (where the time bounds are implicit) or use balanced indexing structures [2], among others. There have also been studies of more practically efficient methods [5,4] which adapt quicksort.

The adapted mergesort technique [13] has the advantage of using very little extra memory and works well in cache. The weight balanced ternary search tries [21] and balanced indexing structures [2] have the advantage of being dynamic.

* Corresponding author. Tel.: +972 35317668.
E-mail address: moshe.lewenstein@gmail.com (M. Lewenstein).

0304-3975/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2011.12.001

In the on demand setting it is required to preprocess a collection of items for output where the user controls the number of elements it desires to see in a well-defined order. For example, a search engine query may have a long list of hits which gets ranked in some ordering of importance. However, the search engine will return a limited (digestible) number, say k, of the best hits to the user. If the user desires another k these will be fetched and returned.

In *on demand string sorting* one is given a collection of strings and desires to preprocess the strings so that subsequent queries of "return the next k smallest lexicographically strings" will execute fast, consuming time proportional to, or lightly depending on, k.

One way to solve this is to first sort the strings in one of the $O(n \log n + N)$ methods and then simply return the next k in $O(k)$ time. It is also possible to concatenate all the strings (separated by external symbols) to one large string S and then to create a suffix array for S. This will work well if the alphabet size is $O(|S|) = O(N)$ since the suffix array can then be created in $O(N)$ time and the desired ordering can be extracted from the suffix array ordering. However, if the alphabet is unbounded the suffix array construction will take $O(N \log N)$ time, which is worse than the previous solutions.

In on demand sorting of numbers a heap is a natural data structure to use. We propose to do the same for strings. We propose to use a heap where the elements of the heap are strings. However, a simple implementation will lead us to running times which can be even worse than we have just mentioned. We propose a construction of the heap in a careful manner utilizing the *longest common prefixes* (lcp's) among pairs of strings. In fact, lcp's have been used in suffix arrays, quicksort of strings [5,4], mergesort of strings [13] and in balanced indexing structures [2] (although they were not used in the ternary digital search tries [21]). However, handling the lcp's require special care with regards to heaps and we elaborate on this later.

Note that it is sufficient to support heapify and delete-mins to capture the application of the on demand string sorting, which we show how to support. Nevertheless, allowing insertions together with delete-mins does not work well with the lcp solution. To solve this we use a balanced indexing structure [2].

The roadmap to our paper is as follows: in Section 2 we give preliminaries and definitions. In Section 3 we recall the balanced indexing data structure [2] and show what needs to be adapted for it to sort strings (instead of suffixes mentioned there). In Section 4 we introduce the heap (of strings) data structure and show how to implement heapify, delete-mins, insertions, and delete-min and insertions together. In Section 5 we show how to support a heap fully using a balanced indexing structure.

2. Definitions and preliminaries

A string S of *length* m over alphabet Σ is a sequence of m letters, $S[1], S[2], \ldots, S[m]$, each being a member of Σ. For $1 \le i \le j \le m$, we denote by $S[i, j]$ the substring $S[i], S[i+1], \ldots, S[j]$ of length $j - i + 1$. When $i > j$, substring $S[i, j]$ is defined to be the empty string of length 0. We say that $S_1 = S_2$, if $length(S_1) = length(S_2)$, and $S_1[i] = S_2[i]$ for all $1 \le i \le length(S_1)$. We say that $S_1 < S_2$, if S_1 precedes S_2 in the lexicographic order. Formally, $S_1 < S_2$, if there exists an index $1 \le j < \min\{length(S_1), length(S_2)\}$ such that $S_1[i, j] = S_2[i, j]$ and $S_1[j+1] < S_2[j+1]$, or if $length(S_1) < length(S_2)$ and $S_1 = S_2[1, length(S_1)]$.

The alphabet Σ is ordered but not bounded. Yet, we assume that any two characters of Σ can be compared in a single computational operation, consuming a constant amount of time.

For ease of description, we assume that the strings in the underlying dataset are distinct. The modifications needed to also allow equal strings are rather obvious.

2.1. Longest common prefix

Definition 1. Given strings S_1 and S_2, the largest $i \le \min\{length(S_1), length(S_2)\}$, such that $S_1[1, i] = S_2[1, i]$ is denoted by $\boldsymbol{lcp}(S_1, S_2)$. The *longest common prefix* of S_1 and S_2 is $S_1[1, \boldsymbol{lcp}(S_1, S_2)]$.

The following well-known folklore lemma has been widely used for data structures of suffixes. We prove its proof for the sake of completeness.

Lemma 1. *Given strings* $S_1 \le S_2 \le \cdots \le S_m$, *then* $\boldsymbol{lcp}(S_1, S_m) = \min_{1 \le i < m} \boldsymbol{lcp}(S_i, S_i + 1)$.

Proof. Denoting $l = \boldsymbol{lcp}(S_1, S_m)$, we have, by definition, $S_1[1, l] = S_m[1, l]$. Since $S_1 \le S_i \le S_m$ for $1 \le i \le m$, we also obtain $S_i[1, l] = S_1[1, l] = S_m[1, l]$. Hence, $\min_{1 \le i < m} \boldsymbol{lcp}(S_i, S_{i+1}) \ge l$. If $S_1 = S_m$ the lemma is trivial. Otherwise, $S_1 \ne S_m$ and then one of the following happens. Either S_1 is of length l or $S_1[l+1] < S_m[l+1]$. In the first case, where S_1 is of length l, $\boldsymbol{lcp}(S_1, S_2) = l$. In the second case, where $S_1[l+1] < S_m[l+1]$, there must be at least one $1 \le i < m$ such that $S_i[l+1] < S_{i+1}[l+1]$. Hence, $\min_{1 \le i < m} \boldsymbol{lcp}(S_i, S_{i+1}) \le l$. □

Corollary 1. *Given strings* S_1, S_2, S_3, *with* $S_1 \le S_2$ *and* $S_1 \le S_3$, *it is easy to verify that:*

[i] $S_2 \le S_3$ *implies* $\boldsymbol{lcp}(S_1, S_2) \ge \boldsymbol{lcp}(S_1, S_3)$ *and* $\boldsymbol{lcp}(S_2, S_3) \ge \boldsymbol{lcp}(S_1, S_3)$. Equivalently, $\boldsymbol{lcp}(S_1, S_2) < \boldsymbol{lcp}(S_1, S_3)$ *implies* $S_2 > S_3$.

[ii] $\boldsymbol{lcp}(S_1, S_2) > \boldsymbol{lcp}(S_1, S_3)$ *implies* $\boldsymbol{lcp}(S_2, S_3) = \boldsymbol{lcp}(S_1, S_3)$.

Proposition 1. *Given strings S_1, S_2, S_3, then:*

[i] *Identifying the smaller of S_1, S_2 and computing $\mathbf{lcp}(S_1, S_2)$ can be done in $O(\mathbf{lcp}(S_1, S_2))$ time.*

[ii] *Identifying j such that S_j is the smallest of S_1, S_2, S_3, and finding $\mathbf{lcp}(S_i, S_j)$ for $i \neq j$ can be done in $O(\max_{i \neq j} \mathbf{lcp}(S_i, S_j))$ time.*

Proof. Directly from Definition 1, by simply comparing the strings character by character (two characters for [i], and three at a time for [ii]). Note that Σ need not be bounded. □

3. Balanced indexing structures

The *Balanced Indexing Structure* (BIS, for short) [2] mentioned in the introduction is a balanced AVL search tree which is an indexing structure for an online text T. It contains one suffix (of T) in each node. The BIS allows insertion of suffixes in an online manner, spending $O(\log n)$ time inserting each new suffix while maintaining AVL balancing. The BIS supports finding all occurrences of a pattern P (in the text T that is indexed) to be found in $O(|P| + \log n + occ)$ time, where occ is the number of locations where P appears.

The BIS is an *lcp*-based data structure. Specifically, a BIS node v is associated with a suffix $S(v)$ and maintains pointers to parent and children in the BIS, and to predecessor and successor in the lexicographic order of the suffixes, and to the smallest and largest strings in the subtree rooted at v. In addition, node v maintains *lcp_prev* with its smaller neighbor in the lexicographic order, and *lcp_extreme* which is the *lcp* of the smallest and the largest strings in the subtree rooted by v.

We show here that with a bit of adaptation this data structure will work similarly as a data structure for maintaining a collection of strings, where insert of a string S will cost $O(\log n + |S|)$, search of a string S will also cost $O(\log n + |S|)$ and deleting a string from the collection will cost $O(\log n)$.

The first difference between an indexing structure and a data structure handling a collection of strings is that there is a strong correlation between the suffixes whereas the different strings in the collection are not definitely correlated. Moreover, when seeing a string for the first time we must at least read the string (whereas, suffixes are handled one after the other, so we are really seeing only the next character). Nevertheless, we can utilize the search procedure of [2] which finds the location of a pattern P in the BIS and then reports all the occurrences of the pattern P within the text. We will be interested in Theorem 7 and Lemma 9, in the paper there which states that the algorithm finds the correct place of the pattern P in time $O(|P| + \log n)$.

Now, in our setting, when one desires to insert a string S one can utilize the procedure described there for search to find the correct location of S in time $O(\log n + |S|)$. What is still necessary is to actually insert S into the BIS. This requires insertion of a new node and rebalancing of the BIS. For brevity, we will give a condensed explanation of the method of the insertion in our setting.

The following lemma of the BIS is crucial to the method.

Lemma 2 (*Based on Lemma 8 of [2]*)**.** *For any path p leading from a node v_1 to any descendant v_2 thereof, $\mathbf{lcp}(S(v_1), S(u))$ can be computed for all nodes u on the path from v_1 to v_2 in overall $O(|p|)$ time.*

Inserting string S to a BIS rooted by string R thus starts with computing $l \leftarrow \mathbf{lcp}(S, R)$, while determining whether $S < R$, which indicates in which of R's subtree S continues. The strings in that subtree, as well as S, are either all smaller than R or all larger that R. Hence for each node v visited in the subtree, comparing l against $\mathbf{lcp}(S(v), R)$ – the *lcp* computed by Lemma 2, as S goes down the tree – suffices, in case $l \neq \mathbf{lcp}(S(v), R)$, to determine whether S continues down left or down right from v. If $l = \mathbf{lcp}(S(v), R)$ then further characters, of S and $S(v)$, are read from position $l+1$ and on until $\mathbf{lcp}(S, S(v))$ is computed while whether $S < S(v)$ is determined, which indicates in which of v's subtree S continues. The strings in that subtree, as well as S, are either all smaller than $S(v)$ or all larger. Hence, as before, with $l \leftarrow \mathbf{lcp}(S, S(v))$, a comparison of l, against $\mathbf{lcp}(S(u), S(v))$ for any node u in this subtree, can tell how S should go down from u, etc. until S is added to the BIS as a leaf. This all is done in $O(\log n + |S|)$ time, for a BIS of size n.

Removal of a string and extraction of the smallest string from a collection are not given in [2]. However, these only require the balancing procedure for the BIS, which was shown there for insertion, and can be done in $O(\log n)$ time. Hence,

Lemma 3. *An (adapted) BIS maintains a collection of strings, where insertion takes $O(\log n + |S|)$, removal of a string takes $O(\log n)$ and extracting minimum takes $O(\log n)$.*

It follows that:

Corollary 2. *A collection of strings $S_1, \ldots, S_n$, where $N = \sum_{i=1}^{n} |S_i|$, can be sorted in time $O(n \log n + N)$.*

4. Heap sorting of strings

The well-known heap data structure is a full, balanced tree maintaining the following invariant:

Heap Invariant: The value of a node is larger than the value of its parent.

Definition 2. Let C be a collection of strings. A *heap of strings* (over C) is a full, balanced binary tree where each node v is associated with a string $S(v) \in C$. The values of the nodes satisfy the heap invariant. Moreover, for each v an $lcp(v)$ field is maintained, whose value is $\mathbf{lcp}(S(v), S(parent(v)))$. (If v is the root, $lcp(v)$ can be of any value.)

One of the main observations that we will use is that our algorithms never decrease the values of the *lcp* fields (having initialized them to 0), and when nodes swap strings, they also swap *lcp* fields. In other words, each string "wanders around" the heap always accompanied by the same *lcp* field, whose value never decreases.

A naive adjustment of the integer heap sorting to string heap sorting would replace each comparison of two integers by a comparison of two strings, employing a sequence of as many character comparisons as the lengths of the compared strings, in the worst case. This, however, **multiplies** runtime by about the average string length. In the sequel, we present an effective use of the *lcp* fields in the nodes, through which we can accomplish all the needed string comparisons by only **adding** a total of $O(N)$ steps.

4.1. Heapify

Commonly, a full balanced binary tree of n nodes is implemented by an array T of length n. The left and right children of node $T[i]$ (if any) are $T[2i]$, and $T[2i+1]$. Given an array T of n strings, the heapify procedure changes the positions of the strings, and their associated *lcp* fields, in T, so that the resulting tree is a *heap of strings*. As with heapifying an array of integers in $O(n)$ time, our algorithm proceeds for nodes $T[n], T[n-1], \ldots, T[1]$, making a *heap of strings* from the subtree rooted at the current node $v = T[i]$, as follows:

- If v is a leaf, just initialize $lcp(v) \leftarrow 0$.
- If v has one child $u = T[2i]$ (which must be a leaf), compute $l = \boldsymbol{lcp}(S(v), S(u))$, and find the smaller of the two strings. Assign the smaller string to node v, and the larger to node u. Assign $lcp(u) \leftarrow l$, and $lcp(v) \leftarrow 0$.
- If v has two children, $u = T[2i]$ and $w = T[2i+1]$, then, by the order we process the nodes, each child now roots a *heap of strings*, and its *lcp* field is 0. We find the smallest, S, of the three strings $S(v), S(u), S(w)$, and compute the *lcp* of S with each of the other two.
  If $S = S(v)$, do not change strings in nodes, just assign the newly computed *lcp*-s: $lcp(u) \leftarrow \boldsymbol{lcp}(S, S(u))$ and $lcp(w) \leftarrow \boldsymbol{lcp}(S, S(w))$. Finally, assign $lcp(v) \leftarrow 0$.
  If $S = S(u)$ (the case $S = S(w)$ is analogous), denote the newly computed *lcp* by $l' = \boldsymbol{lcp}(S(v), S)$ and then swap $S(v)$ with $S(u)$ and assign $lcp(u) \leftarrow l'$, so that the smallest string, S, now resides in v, and the string formerly residing in v, along with its *lcp* with S, now resides in u. Then assign $lcp(w) \leftarrow \boldsymbol{lcp}(S, S(w))$ (which was already computed), and $lcp(v) \leftarrow 0$. Note that now w still roots a *heap of strings*, and in addition it satisfies the *heap of strings property*, and that u, and each of its children, now maintains an *lcp* of their string with S, which is smaller than **all** the strings in the subtree rooted at u. Invoking SiftDown(u) (Fig. 1) ensures that the subtree rooted at u, which just replaced the string of its root, is a *heap of strings*. This completes the processing of v.

Correctness of our heapify algorithm for strings follows from the known heapify for integers and Corollary 1. As for runtime, note that we process $O(n)$ nodes (including the nodes processed in the recursive calls), and that when nodes swap strings, they also swap *lcp*-s, possibly while increasing one or two. That is, each string is clearly associated with an *lcp* field, such that:

Proposition 2. [i] *for every characters comparison (of two characters, or three at a time), at least one of the strings participating in the comparison has its associated lcp field incremented.* [ii] *Once a string has its associated lcp field assigned the value of l, none of its characters in positions* $1, 2, \ldots, l$ *participates in any further character comparison.*

Corollary 3. *Heapifying into a heap of strings can be done in* $O(n+N)$ *time.*

4.2. Extracting minimal string

Having built a *heap of strings* H of size n, we now extract the smallest string therefrom, which resides at the root of H, and invoke **PumpUp**(*root*(H)) which discards one node from tree H, while maintaining that the remaining nodes form a tree, with each of them satisfying the *heap of strings property* (Fig. 2).

Correctness of **PumpUp** follows directly from Corollary 1. Observe that now H is not necessarily full and not necessarily balanced, but its height is $O(\log n)$, and hence any further extractions of minimal string does not process more than $O(\log n)$ nodes. Note that, as with our heapify procedure, each character comparison (here we only have comparison between two characters, never three) results in an increment of one *lcp* field. None of the *lcp* fields decreases from the value it had at the end of the heapify process.

In fact, if we start with heapify, and then extract smallest string by smallest (over the remaining) string, we actually sort the set of n input strings in a lexicographic order.

Corollary 4. *Sorting of n strings of total length N, over unbounded alphabet, can be implemented in* $O(n \log n + N)$ *worst case time, using* $O(n)$ *auxiliary space.*

Proof. Based on the classic heap sort: heapify followed by sequential n extractions of the data element from the root, we implement each data-element comparison by either a (constant time) *lcp* comparison, or a sequential character to character (or three characters at a time) comparisons. For the latter, we increase at least one *lcp* field by the number of steps in that sequential comparison. As the *lcp* fields never decrease, and the total of their maximal value is N, we conclude that the time complexity of the sort is $O(n \log n + N)$. □

Require: (1) All nodes in subtree rooted by v have strings larger than the string of $parent(v)$. (2) All these nodes, except, possibly, one or two children of v, satisfy the *heap of strings property*. (3) The *lcp* field of v and of each of its children reflect the *lcp* of their string with the string of $parent(v)$.
Ensure: in the subtree rooted by v, strings change nodes, and *lcp*-s are updated, such that the resulting subtree is a *heap of strings*.

1: **if** v has no children **or** $lcp(v)$ is greater than each child's *lcp* **then** {$S(v)$ is a clear smallest}
2:   **return**
3: **end if**
4: **if** $lcp(v) < lcp(child1(v))$, **and:** either $child1(v)$ is the only child of v or $lcp(child2(v)) < lcp(child1(v))$ **then** {*child*1 is a clear smallest}
5:   swap strings and *lcp*-s between v and $child1(v)$
6:   **SiftDown**($child1(v)$)
7:   **return**
8: **end if**
9: **if** $lcp(v) = lcp(child1(v))$, **and:** either *child1* is the only child of v or $lcp(child2(v)) < lcp(child1(v))$ **then** {smallest is either v or *child*1}
10:   read $S(v)$ and $S(child1(v))$ from position $lcp(v) + 1$ on, character by character, until $l = \boldsymbol{lcp}(S(v), S(child1(v)))$ is computed, as the smaller of the two strings is determined.
11:   $lcp(child1(v)) \leftarrow l$
12:   **if** $S(v)$ is larger than $S(child1(v))$ **then**
13:     swap strings between v and $child1(v)$
14:     **SiftDown**($child1(v)$)
15:   **end if**
16:   **return**
17: **end if**
{v has two children; $lcp(left(v)) = lcp(v) = lcp(right(v))$}
18: read $S(v)$, $S(left(v))$, and $S(right(v))$, in parallel, from position $lcp(v) + 1$ on, until the smallest, S, of the three is determined, as the *lcp* of it with each of the other strings is computed.
19: **if** $S = S(v)$ **then**
20:   assign respective *lcp*-s to children
21: **else** {$S = S(child1(v))$}
22:   $lcp(child2(v)) \leftarrow \boldsymbol{lcp}(S, child2(v))$ {already computed}
23:   $l \leftarrow \boldsymbol{lcp}(S, S(v))$ {already computed}
24:   swap strings between v and $child1(v)$
25:   $lcp(child1(v)) \leftarrow l$
26:   **SiftDown**($child1(v)$)
27: **end if**

Fig. 1. SiftDown(v)We denote by $child1(v)$ and $child2(v)$ both $left(v)$ and $right(v)$, as applicable by the conditions on their field values.

4.3. On-demand sorting

As construction time for a *heap of strings* is $O(n)$, smaller than the $O(n \log n)$ for BIS, the *heap of strings* is better suited for cases where we will need only an (unknown) fraction of the strings from the lexicographic order.

Corollary 5. *On Demand Sorting of n strings of total length N can be done with the retrieval of the first result in $O(n + N_1)$ time, after which the retrieval of further results in $O(\log n + N_i)$ time for the i-th result, with $\sum_i N_i \leq N$.*

Proof. Using *heap of strings*, the first result can be extracted immediately after the heapify. Further results are extracted each following a **PumpUp** that rearranges the *heap of strings*, of height $O(\log n)$, following the previous extraction. Through the whole process *lcp* fields never decrease, and each character comparison incurs an *lcp* increase. This implies the time complexity. □

4.4. Find the smallest k strings

When we know in advance that we will only need the $k < n$ smallest strings from the input set of n strings of total length N, we can use a *heap of strings* of size k, and achieve the goal in $O(n \log k + N)$ time as follows. We use a heap of size k where parents hold **larger** strings than their children, and hence the largest string in the heap resides at the root node. We heapify the first k strings of the input set into such a heap, in analogy to our heapify process above. Then, for each string S of the remaining $n - k$ strings in the input, we compare S with string R at the root, while finding $\boldsymbol{lcp}(S, R)$. If S is found greater than

Require: (1) v has a null string and a non relevant *lcp* (2) All other nodes in the subtree rooted by v satisfy the *heap of strings property*. (3) The *lcp* in each of v's children reflects the *lcp* of their string with the string formerly residing in v, which is smaller than each child's string.

Ensure: (1) In the subtree rooted by v, along a path from v down to one leaf, strings and their associated *lcp*-s climb up one node, so that the leaf node becomes redundant and can be discarded from the tree. (2) All remaining node have relevant strings and *lcp*-s, and they satisfy the *heap of strings property*.

1: **if** v is a leaf **then** {discard this node from the tree, its contents were already copied to its parent}
2: discard v from the tree
3: **return**
4: **end if**
5: **if** v has only one child, *child*1, **or** $lcp(child1(v)) > lcp(child2(v))$ **then** {*child*1 is a clear smaller}
6: $S(v) \leftarrow S(child1(v))$ and $lcp(v) \leftarrow lcp(child1(v))$
7: **if** v has two children **then**
8: **PumpUp**($child1(v)$)
9: **else** {v has a single child}
10: discard $child1(v)$ from the tree, and makes its children – the children of v
11: **end if**
12: **return**
13: **end if**
{v has two children, with equal *lcp*-s: $l = lcp(left(v)) = lcp(right(v))$}
14: read $S(left(v))$ and $S(right(v))$ from position $l + 1$ on, character by character, until $l' = \textbf{lcp}(S(left(v)), S(right(v)))$ is computed, as the owner, *child*1, of the smaller of the two strings is determined.
15: $S(v) \leftarrow S(child1(v))$ and $lcp(v) \leftarrow lcp(child1(v))$
16: $lcp(child2(v)) \leftarrow l'$
17: **PumpUp**($child1(v)$)

Fig. 2. **PumpUp**(v)We denote by $child1(v)$ and $child2(v)$ both $left(v)$ and $right(v)$, as applicable by the conditions on their field values.

R, it is discarded. Otherwise, R is discarded, and S finds its place in the heap using procedure **SiftDown** (Fig. 1) adopted for heaps where parents hold larger strings than their children.

After this process, the heap of size k holds the smallest k strings in the collection. We can now sequentially extract the (next) largest string therefrom, retrieving the k smallest strings in decreasing lexicographic order, using **PumpUp** (Fig. 2) adopted for heaps where parents hold larger strings than their children.

As here, too, *lcp* fields only grow, we conclude that:

Corollary 6. *Finding (and sorting) the smallest k strings of a given input set of n strings of total length N, can be done in $O(n \log k + N)$ time, using $O(k)$ auxiliary space.*

4.5. Insertion of strings to a heap of strings

Insertion of strings is an operation that both heaps and search trees allow. As described in Section 3, a BIS allows the insertion of strings in $O(\log n + |S|)$. With *heap of strings*, however, build-up linearity in number of nodes does not apply for post build-up insertions. Namely, inserting an additional element to an existing heap of size n incurs the processing of $O(\log n)$ nodes, not $O(1)$. Moreover, BIS, as discussed in Section 3, allows string insertion while maintaining the tree balance and not decreasing any *lcp* fields. Insertion of data elements to a heap, however, by the known heap algorithms, causes some nodes to have their parents replace their data elements by a smaller one, without them (the children) replacing their data elements. Hence, with a *heap of strings*, by Corollary 1, the *lcp* field of such a child node, which needs to reflect the *lcp* of its string with its parent's string, might decrease.

More specifically, here is how we can insert a string, S, to a *heap of strings*, H, of size n, in $O(\log n + |S|)$ time. H remains a full balanced *heap of strings*. Only a few *lcp* fields therein might decrease. We first add an empty leaf node, denoted by *leaf* to H, such that H is still balanced, and denote by *path* the unique path $root = n_1, n_2, \ldots, n_m = leaf$ from *root* to *leaf*. We then invoke procedure **InsertString** which finds the right position for S in *path*, then pushes the suffix of *path* from that point down, making room for S, while updating *lcp* fields of nodes out of *path* whose parents now hold smaller strings than before.

InsertString($H, S, path$)

Require: (1) A *heap of strings* H of size n plus a newly added empty leaf, *leaf* (2) a path *path* from H's root to the newly added empty leaf node: $root = n_1, n_2, \ldots, n_m = leaf$ (3) a string S.

Ensure: S is inserted in the right position along *path*, and H becomes a *heap of strings* of size $n + 1$.

1: Compare S with the root string R, while computing $l = \boldsymbol{lcp}(S, R)$.
2: **if** $S > R$ **then**
3: **for** $i \leftarrow 2$ **to** $m - 1$ **do**
4: **if** $l > lcp(n_i)$ (which implies $S < S(n_i)$) **then**
5: **PushDown**$(H, S, path, i, lcp(n_i), l)$
6: **return**
7: **else if** $l = lcp(n_i)$ (the case $l < lcp(n_i)$ is not interesting since it must be that $S > S(n_i)$) **then**
8: read strings S and $S(n_i)$ from position $l + 1$ on, until $l' \leftarrow \boldsymbol{lcp}(S, S(n_i))$ is computed and determine whether $S < S(n_i)$.
9: **if** $S < S(n_i)$ **then**
10: **PushDown**$(H, S, path, i, l', l)$
11: **return**
12: **end if**
13: $l \leftarrow l'$
14: **end if**
15: **end for**
{S needs to reside in the newly added leaf}
16: $S(leaf) \leftarrow S$; $lcp(leaf) \leftarrow l$
17: **else** {$S < R$}
18: **PushDown**$(H, S, path, 1, l, 0)$
19: **end if**

Procedure **PushDown** actually modifies the heap in the same way as the classic Sift-up process of inserting a new element to an existing heap. In addition, **PushDown** also updates *lcp* fields as necessary. This is where some of these fields might decrease.

PushDown$(H, S, path, i, l', l)$

Require: (1) A *heap of strings* H of size n plus a newly added empty leaf, *leaf* (2) a string S, (3) a path *path* from H's root to the newly added leaf node: $root = n_1, n_2, \ldots, n_m = leaf$ (4) an index i to a node on *path* such that S is to be inserted between n_{i-1} and n_i in the path, (5) *lcp* value $l' = \boldsymbol{lcp}(S(n_i), S)$, (6) *lcp* value $l = \boldsymbol{lcp}(S, S(n_{i-1}))$

Ensure: Strings and *lcp*-s in nodes $n_i, n_{i+1}, \ldots, n_{m-1}$ are pushed one node down *path*, and S is placed in the evacuated node, while making H a *heap of strings* of size $n + 1$.

1: **for** $j = m$ down to $i + 1$ **do** {push down the end of the path}
2: $S(n_j) \leftarrow S(n_{j-1})$; $lcp(n_j) \leftarrow lcp(n_{j-1})$
3: $lcp(sibling(n_{j+1})) \leftarrow \min\{lcp(sibling(n_{j+1})), lcp(n_{j+1})\}$ {$sibling(n_{j+1})$ now has as its parent what used to be its grandparent, and hence its *lcp* might need to decrease}
4: **end for**
5: $lcp(n_{i+1}) \leftarrow l'$
6: $S(n_i)) \leftarrow S$; $lcp(n_i) \leftarrow l$
7: $lcp(sibling(n_{i+1})) \leftarrow \min\{lcp(sibling(n_{i+1})), l'\}$ { $sibling(n_{i+1})$ now has S as its parent instead of $S(n_{i+1})$}

The decrease in *lcp* values prevents from ensuring overall $O(N)$ characters comparisons for heap buildup followed by some string insertions, followed by string extractions. Number of node processings, however, remains $O(n)$ for heap buildup, followed by $O(m \log(n + m))$ for the insertion of additional m strings, followed by $O(\log(m + n))$ for each string extraction.

4.6. Inserting strings without decreasing lcp-s

One way to avoid decreasing of *lcp*-s is to insert single-childed nodes. This comes, however, at the expense of tree balancing. Before invoking InsertString, do not prepare an empty leaf node. Rather, once the right point in *path*, between n_{i-1} and n_i is found by InsertString, instead of invoking PushDown, add a new, empty node to H, and place S there, making it n_{i-1}'s child instead of n_i, and making n_i its only child. As before, number of characters processing does not exceed $|S|$. Now, however, no *lcp* field needs to decrease. Let n denote the size of the heap upon its build up, and m the number of strings thus inserted the heap post build up. It is easy to see that further extractions, using our PumpUp above, would not incur more than $O(\log n)$ node processing, since this procedure stops once a single-childed node is reached. Further insertions, however, although not exceeding $O(|S|)$ character comparisons, might incur the processing of $O(\log n + m)$ nodes, much greater than $O(\log(n + m))$ which is the number of nodes processed by BIS.

5. Heap of strings embedded with binary search trees

In this section we circumvent the non-balancing introduced in Section 4.6. Generally speaking, we embed chains of single-childed nodes on a BIS, which is *lcp* based. We start with heapify a *heap of strings* from the initial set of n strings. Then, for each string arriving post heapifying, we insert it as in Section 4.6. The path of single-childed newly born nodes, which hold strings in increasing order, is arranged on a BIS and is **not** considered part of the *heap of strings*. See Fig. 3.

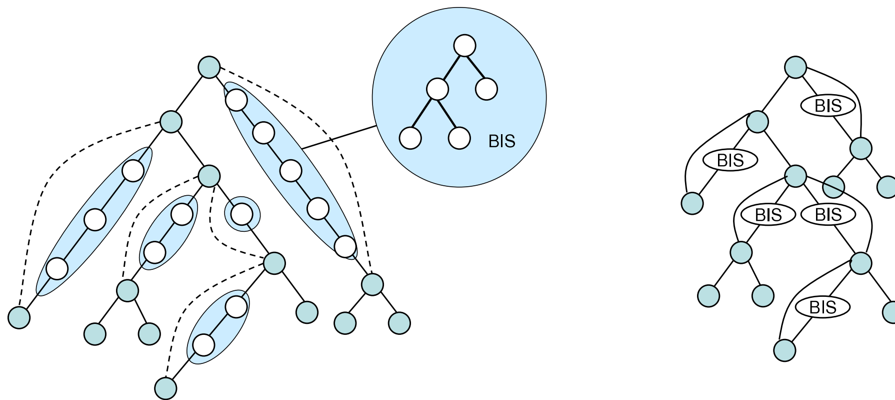

Fig. 3. *Heap of strings* embedded with *Balanced Indexing Structure*.

The BIS thus born is pointed at from both nodes of the *heap of strings*, above and below it. The smallest node in that BIS will maintain, in its *lcp_prev* field, the *lcp* of its string and the string of the *heap of strings* node above the BIS.

As in Section 4.6, before calling InsertString we do not prepare an empty leaf node. Rather, when InsertString finds the right point to insert S, between n_{i-1} and n_i of the *heap of strings*, instead of invoking PushDown, S will join the BIS that resides between these two nodes (or start a new BIS if there is not any) as described in Section 3. When we are about to insert S to the BIS "hanging" between n_{i-1} and n_i, we have already computed $l = \textbf{\textit{lcp}}(S, S(n_{i-1}))$. All the BIS strings, as well as S, are larger than $S(n_{i-1})$, and the *lcp* fields maintained in the BIS suffice to compute, in constant time, the value of $\textbf{\textit{lcp}}(R, S(n_{i-1}))$, for the BIS root R. Hence, the first step of inserting S to the BIS, namely the comparison against R, can be done without reading $S[1, l]$ again. Thus, the insertion of string S to a *heap of strings* that was heapified with n strings, incurs $O(\log n + \log m + |S|)$ time, with m denoting the size of the BIS that is to include S.

Extracting of strings from the root of the heap can continue as before, using PumpUp, which now, if visits a node v of the *heap of strings* which has a BIS leading from it to its child in the *heap of strings*, sees the smallest node of the BIS as v's single child, and pumps this child up to take the place of v. This child is removed from *BIS* and becomes a member of the *heap of strings*, and this completes the execution of PumpUp (it does not go further down). The removal of a node from BIS of size m takes $O(\log m)$ time, and our PumpUp incurs the processing of $O(\log n)$ nodes plus reading as many characters as needed, while increasing *lcp* fields.

Note that a string that arrives post heapifying may spend "some time" in a node of a BIS, but once removed from there and joined the *heap of strings*, it will never go into any BIS again. Only newly arriving strings may go into a BIS. We conclude that:

Theorem 1. *It is possible to construct a heap of n strings in $O(n)$ time and support insertion and extraction of the minimal string in $O(\log(n) + \log(m))$ time, with an additional $O(N)$ amortized time over the whole sequence of operations, where m is the number of strings inserted post heapifying and were not extracted yet.*

For further reading

[10,25].

Acknowledgement

This work was partially supported by the Israel Science Foundation, grant no. 1484/04.

References

[1] A. Aho, J. Hopcroft, J. Ullman, The Design and Analysis of Computer Algorithms, Addison Wesley, Reading, MA, 1974.
[2] A. Amir, T. Kopelowitz, M. Lewenstein, N. Lewenstein, Towards real-time suffix tree construction, in: Proc. of Symp. on String Processing and Information Retrieval, SPIRE, 2005, pp. 67–78.
[3] L. Arge, P. Ferragina, R. Grossi, J.S. Vitter, On sorting strings in external memory, in: Symposium of Theory of Computing, STOC, 1997, pp. 540–548.
[4] J.-L. Baer, Y.-B. Lin, Improving quicksort performance with a codeword data structure, IEEE Transactions on Software Engineering 15 (1989) 622–631.
[5] J.L. Bentley, R. Sedgewick, Fast algorithms for sorting and searching strings, in: Proc. of Symposium on Discrete Algorithms, SODA, 1997, pp. 360–369.
[6] R. Cole, T. Kopelowitz, M. Lewenstein, Suffix trays and suffix trists: structures for faster text indexing, in: Proc. of International Colloquium on Automata, Programming and Languages, ICALP, 2006, pp. 358–369.
[7] M. Farach, Optimal suffix tree construction with large alphabets, in: Proc. 38th IEEE Symposium on Foundations of Computer Science, 1997, pp. 137–143.
[8] M. Farach-Colton, P. Ferragina, S. Muthukrishnan, On the sorting-complexity of suffix tree construction, Journal of the ACM 47 (6) (2000) 987–1011.
[9] G.H. Gonnet, R. Baeza-Yates, Handbook of Algorithms and Data Structures, Addison-Wesley, 1991.
[10] R. Grossi, G.F. Italiano, Efficient techniques for maintaining multidimensional keys in linked data structures, in: Proc. 26th Intl. Col. on Automata, Languages and Programming, ICALP, in: LNCS, vol. 1644, 1999, pp. 372–381.

[11] T. Hagerup, Optimal parallel string algorithms: sorting, merging and computing the minimum, in: Proc. of Symposium on Theory of Computing (STOC), 1994, pp. 382–391.
[12] T. Hagerup, O. Petersson, Merging and sorting strings in parallel, in: Mathematical Foundations of Computer Science, MFCS, 1992, pp. 298–306.
[13] B.R. Iyer, Hardware assisted sorting in IBM's DB2 DBMS, in: International Conference on Management of Data, COMAD 2005b, Hyderabad, India, December 20–22, 2005.
[14] J.F. JaJa, K.W. Ryu, U. Vishkin, Sorting strings and constructing difital search tries in parallel, Theoretical Computer Science 154 (2) (1996) 225–245.
[15] Juha Kärkkäinen, Peter Sanders, Simple linear work suffix array construction, in: Proc. 30th International Colloquium on Automata, Languages and Programming, ICALP 03, in: LNCS, vol. 2719, 2003, pp. 943–955.
[16] D.K. Kim, J.S. Sim, H. Park, K. Park, Linear-time construction of suffix arrays, in: Proc. of 14th Symposium on Combinatorial Pattern Matching, in: LNCS, vol. 2676, 2003, pp. 186–199.
[17] D. Knuth, The Art of Computer Programming, in: Sorting and Searching, vol. 3, Addison-Wesley, 1973.
[18] P. Ko, S. Aluru, Space efficient linear time construction of suffix arrays, in: Proc. of 14th Symposium on Combinatorial Pattern Matching, in: LNCS, vol. 2676, 2003, pp. 200–210.
[19] U. Manber, E.W. Myers, Suffix arrays: a new method for on-line string searches, SIAM Journal on Computing 22 (5) (1993) 935–948.
[20] E.M. McCreight, A space-economical suffix tree construction algorithm, Journal of the ACM 23 (1976) 262–272.
[21] K. Mehlhorn, Dynamic binary search, SIAM Journal on Computing 8 (2) (1979) 175–198.
[22] J.I. Munro, V. Raman, Sorting multisets and vectors inplace, in: Proc. of Workshop on Algorithms and Data Structures, WADS, 1991, pp. 473-479.
[23] R. Sinha, J. Zobel, D. Ring, Cache-efficient string sorting using copying, Journal of Experimental Algorithmics 11 (2006) 1084–6654.
[24] E. Ukkonen, On-line construction of suffix trees, Algorithmica 14 (1995) 249–260.
[25] Jeffrey Scott Vitter, External memory algorithms, in: Handbook of massive data sets, Kluwer Academic Publishers, Norwell, MA, USA, 2002, pp. 359–416.
[26] P. Weiner, Linear pattern matching algorithm, in: Proc. 14th IEEE Symposium on Switching and Automata Theory, 1973, pp. 1–11.