Search Data Structures for Skewed Strings

Pilu Crescenzi *, Roberto Grossi **, and Giuseppe F. Italiano ***

Abstract. We consider skewed distributions of strings, in which any two such strings share a common prefix much longer than that expected in uniformly distributed (random) strings. For instance, this is the case of URL addresses, IP addresses, or XML path strings, all representing paths in some hierarchical order. As strings sharing a portion of the path have a quite long common prefix, we need to avoid the time-consuming repeated examination of these common prefixes while handling the linked data structures storing them. For this purpose, we show how to implement search data structures that can operate on strings with long prefixes in common. Despite the simplicity and the generality of the method, our experimental study shows that it is quite competitive with several optimized and tuned implementations currently available in the literature.

1 Introduction

In many applications keys are arbitrarily long, such as strings, multidimensional points, multiple-precision numbers, or multi-key data, and are modelled as k-dimensional keys for a given positive integer $k > 1$, or as variable-length keys. The latter can be virtually padded with a sufficient number of string terminators so that they can be considered having all the same length k. When dealing with skewed distributions (shortly, *skewed strings*), such as URL addresses, IP addresses, or XML path strings representing paths in some hierarchical order, we can observe that they share typically long prefixes. It is more realistic to assume that the average length of the common prefix of any two such strings is much longer than that expected in the case of uniformly distributed (random) strings.

A reasonable measure of "skewness," denoted $\mathscr{L}(S)$, for a given set S of n strings $x_1, \ldots, x_n$ over an alphabet Σ can be formalized as follows. Let m_i be the longest matched prefix of x_i against the previous strings $x_1, \ldots, x_{i-1}$. Define $m = \sum_{i=2}^{n} m_i/(n-1)$ as the average length of matched prefixes in S. Note that the definition of m does not depend on the order of the keys in S, as $(m+1)$ is equivalent to the internal path length of the $|\Sigma|$-ary trie built on S, which is independent of the order of insertions. We use the fact that $(m+1) \approx \log_{|\Sigma|} n$ when the strings in S are independent and identically distributed [31, Chap.4]. Hence, we measure the skewness (or non-randomness) of the strings in S by the ratio $\mathscr{L}(S) = (m+1)/\log_{|\Sigma|} n$. In a certain sense, a large value of m is a

* Dipartimento di Sistemi e Informatica, Università degli Studi di Firenze, Italy (piluc@dsi.unifi.it).

** Dipartimento di Informatica, Università di Pisa, Italy (grossi@di.unipi.it). Partially supported by the Italian MIUR Project ALINWEB.

*** Dipartimento di Informatica, Sistemi e Produzione, Università di Roma "Tor Vergata", Italy (italiano@disp.uniroma2.it). Partially supported by the EU Programme ALCOM-FT (n. IST-1999-14186) and by the Italian MIUR Project ALINWEB.

good indicator of the skewness of the strings, which is normalized by that same indicator for random strings. Alternatively, since $\log_{|\Sigma|} n$ is proportional to the height of a random $|\Sigma|$-ary tree with n nodes [31, Chap.4], we can see $\mathscr{L}(S)$ as the average number of characters matched per level in a random trie. The intuition is that $\mathscr{L}(S)$ is small in random strings, and the dominant cost is given by traversing the access path from the root to a node. For skewed strings, $\mathscr{L}(S)$ is large and the dominant cost is due to scanning the long prefixes of the strings.

Fast searching on a dynamic set S of skewed strings can be supported by choosing from a vast repertoire of basic data structures, such as AVL-trees [2], red-black trees [4, 14, 32], (a, b)-trees [16], weight-balanced BB[α]-trees [22], finger search trees, self-adjusting trees [30], and random search trees [27], just to name a few. Many of these data structures exhibit interesting combinatorial properties that make them attractive both from the theoretical and from the practical point of view. They are defined on an ordered set of (one-dimensional or one-character) keys, so that searching is driven by comparisons against the keys stored in their nodes: it is usually assumed that any two keys can be compared in $O(1)$ time. Their operations can be extended to long keys by performing string comparisons while routing the keys for searching and updating. This approach may work well in the case of uniformly distributed strings, since a mismatching character is found in expected constant time. However, it is not satisfactory for skewed strings, since a string comparison may take as much as $O(k)$ time per operation, thus slowing down the performance of carefully designed data structures by a multiplicative $O(k)$ factor. To improve on this simple-minded approach, we would like to avoid repeated examinations of long common prefixes while handling the linked data structures storing them efficiently.

Several approaches for solving this problem are possible, such as using tries (see e.g. [17]) or *ad hoc* data structures for k-dimensional keys, for which we have many examples [5, 6, 8, 10, 11, 13, 19, 30, 33–35]. The height of all these data structures is $O(k + \log n)$, where k is the maximum key length and n the total number of keys. A more general paradigm, described in [12], makes use of algorithmic techniques capable of augmenting many kinds of (heterogeneous) linked data structures so that they can operate on k-dimensional keys. Consequently, the richness of results on data structures originally designed for one-dimensional keys is available for k-dimensional keys in a smooth, simple and efficient way, without incurring in the previously mentioned $O(k)$-slowdown.

A potential drawback of general techniques, such as the ones in [12], is that they may produce data structures that are difficult to use in real-life applications, which could take some advantages from their efficient implementation for skewed strings. Just to name one, Internet "interceptor" software blocks access to undesirable Internet sites, storing huge lists of URLs to be blocked, and checking by prefix searches in these URLs. Other applications may be in networking for filtering purposes and in databases for storing and searching XML records. Note that hashing is of little help in these situations as it cannot perform the search of prefixes of the keys, being scattered throughout the memory. Instead, the above data structures are more powerful than hashing, since they allow for prefix and one-dimensional range searching and for sorting keys.

In this paper, we pursue an algorithm engineering avenue to show that the techniques in [12] may be of practical value. We therefore discuss the implementing issues from the programmers' perspective. In particular, we show that the general transformation from one-character to k-character keys can be carried out by following a rather general and simple scheme, so that a given implementation of a data structure can be easily modified in order to handle k-character keys without incurring in the $O(k)$-slowdown. In principle, we can start from several basic data structures for one-character keys, and show how to make them work for k-character keys. This can be easily done as the technique itself is not invasive: if we start from a given data structure (e.g., a binary search tree, an AVL tree, an (a, b)-tree, or a skip list) or a related piece of code for one-character keys, we can produce a data structure for k-character keys which retains exactly the same topology (and the same structural properties) as the original structure.

We start out from the techniques in [12] for re-engineering the core of the search and update algorithms, and we compare the resulting codes to more optimized and tuned algorithms. In our experiments we consider the Patricia tries [17, 21], the ternary search trees [5] and a cache-aware variant of tries [1], which are known to be among the fastest algorithms for long keys. Timings were normalized with the average time of linear probing hashing [17]. Our experiments were able to locate two different thresholds on our data sets, based on the parameter $\mathscr{L}(S)$ for skewness. When the parameter is small, the adaptive tries of [1] give the fastest implementation. When it is large, our techniques deliver the fastest codes, which thus become relevant in the case of skewed strings. For intermediate values of $\mathscr{L}(S)$, the experimented algorithms have very close behavior: this is the range where the ternary search trees of [5] become very competitive. In any case, the fastest algorithms obtain a speedup factor that is bounded by 2 or 3. As a matter of fact this is not a limitation, since the data structures are highly tuned for strings and our *general* technique compares favorably in several cases. This may allow for treating 2–3 times more queries per time unit on a high-performance server, which is effective in practice. Our data structures are therefore a valid alternative to ubiquitous Patricia tries and compact tries in all their applications (indexing, sorting, compressing, etc. etc.).

2 Engineering the Algorithms for Long Keys and Strings

Our technique can be illustrated by running an example on the classical lookup procedure adopted in binary search trees. The reader can follow our example and apply the same methodology to other data structures fulfilling the requirements described in [12]. Namely, they must be linked data structures whose keys undergo a total order and whose search and update operations are driven by comparison of the keys (i.e., without hashing or bit manipulation of the keys, for which our technique does not work). With this approach, we have implemented the ANSI C code of several multi-character key data structures, starting from available code for binary search trees, treaps, AVL trees [2], (a, b)-trees [16], and skiplists [24]. In particular, for binary search trees, treaps, and AVL trees we started from the implementation described in [36], for skiplists we started from the implementation described in [24], which is available via anonymous ftp at [25], while for (a, b)-trees we started from our own implementation of [16, 20].

2.1 Preliminaries

In our description we identify the nodes with their stored keys, and assume that the keys fulfill a total order with two special keys $-\infty$ and $+\infty$ that are always the smallest and the largest ones. Given a node t, let π be the access path from the root to t. We define the *successor* π_t^+ of t along path π to be the smallest ancestor that is greater than t (i.e., we find π_t^+ by going upward in π until we cross a left link). The *predecessor* π_t^- is defined analogously: namely, π_t^- is the greatest ancestor that is smaller than t. Note that both π_t^+ and π_t^- are well defined because of the special keys. We store in each node t the maximum number of initial characters that t shares with π_t^+ and π_t^-, that is, their *longest common prefix length* or, shortly, *lcp*. As it will be clear, the *lcp* of two strings permits to compare them in $O(1)$ time by simply comparing their leftmost mismatching symbols, which occur in position *lcp* (we follow the convention of numbering string positions starting from 0). We assume that the keys are chosen from a universe `keytype`, such as a sequence of integers, characters, or reals, terminated by the special null value 0.

An example of binary search tree and its augmented version with long keys is shown in Figure 1. In the example, the first number in each node is the value of the *lcp* with its predecessor while the second number is the value of the *lcp* with its successor. For example, `seafood` has `seacoast` as predecessor and `surf` as successor: hence, the two *lcp* values are 3 and 1, respectively.

2.2 The case study of binary search trees

We begin by taking an available implementation of binary search trees, such as the one described in [36], where each node has the regular fields `key`, `left` and `right`. We may extend the functionalities to strings by using the C library function `strcmp` to perform comparisons of keys. Our first step is instead that of obtaining a preliminary version of the search procedure, called `FindA`, by augmenting each node with two fields `pred_lcp` and `succ_lcp` storing the longest common prefix of the strings pointed by the fields `key` of node t and its predecessor π_t^-, and that of node t and its successor π_t^+, respectively. We do not need to store pointers to π_t^- and π_t^+. Then, we replace `strcmp` with function `fast_scmp`, which is aware of previous comparisons of single characters. This function is at the heart of our computation as it compares efficiently a search key `x` against the key in a node `t`. Both keys have the same predecessor and successor with respect to the access path from the root to `t` (excluded), whereas their *lcp*s may be different. Function `fast_scmp` makes use of three global `static int` variables:

- `m` denotes the number of characters matched in the search key `x` so far;
- `left_best` is the *lcp* between `x` and its current predecessor;
- `right_best` is the *lcp* between `x` and its current successor.

The initial value of the variables is 0. At the generic comparison step, `m` characters have been matched, and at least one of `left_best` and `right_best` equals `m`. To compare `x` and `t`'s key, we first set a local variable `lcp` to the proper *lcp* between the key in node `t` and either its predecessor or its successor. Namely, we select one of the fields `pred_lcp` and `succ_lcp` in the node, driven by the invariant that at least one of `left_best` and `right_best` equals `m`. If `lcp` is at least `m` in value, we may have to extend the `m` matched characters by comparing one character at a time, starting from the character in position `m` of both keys. In any case, we end up storing in `lcp` the *lcp* between `x` and `t`'s key. At this point, the mismatch between the characters in position `lcp` of both keys yields the outcome of `fast_scmp`. We do not comment further the source code of function `fast_scmp` here as it is the mere implementation of the ideas in [12]:

```
int fast_scmp ( keytype x, node t ) {
    int lcp = ( left_best == m ) ? t->pred_lcp : t->succ_lcp;
    if ( m <= lcp ) {
        for ( ; (x[m] != 0) && (x[m] == t->key[m]); m++ ) ;
        lcp = m;
    }
    if ( x[lcp] <= t->key[lcp] )
        right_best = lcp;
    else
        left_best = lcp;
    return ( x[lcp] - t->key[lcp] ); }
```

As previously mentioned, once *lcp* is known, the comparison among `x` and `t->key` is trivially done in the last `if` statement. Although satisfactory from a theoretical point of view, `fast_scmp` is not always better than using `strcmp`; however, it can be reused for other kinds of data structures by introducing minimal changes in their original source codes.

2.3 Single-shot version

The resulting code of `FindA` and `fast_scmp` in Section 2.2 can be tailored to achieve greater efficiency, writing it in a single and compact form, called `FindB` and reported below. There, static variables become local variables. At each iteration in the main `while` loop, we perform the computation of `lcp`, originally in `fast_scmp`. Next, we suitably combine the comparison of the mismatching symbols at position `lcp` and the subsequent node branching to prepare for the next iteration.

```
node FindB( keytype x, node t ) {
    int m = 0,  left_best = 0,  right_best = 0, lcp = 0;
    while ( t ) {
        lcp = (left_best == m) ? t->pred_lcp : t->succ_lcp;
        if ( m <= lcp ) {
          for ( ; (x[m] != 0) && (x[m] == t->key[m]); m++) ;
          lcp = m;  }
        if ( x[lcp] == 0 ) {
            return t;
        } else if ( x[lcp] < t->key[lcp] ) {
            t = t->left; right_best = lcp;
        } else {
            t = t->right; left_best = lcp; } }
    return t; }
```

2.4 Code tuning and faster *lcp* computation

By profiling the code in `FindB`, we discovered that one potential bottleneck of our implementation was the *lcp* computation. In particular, line profiling revealed us that we had to infer the value of `lcp` computed by line

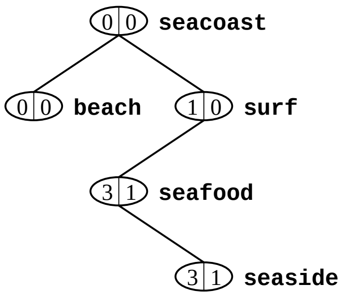
```
lcp = ( left_best == m ) ? t->pred_lcp : t->succ_lcp.
```

We remark that the purpose of the above line is to store temporarily in `lcp` the *lcp* between `t`'s key and its predecessor (resp., successor) when `x` matches the first `m` characters of that predecessor (resp., successor). We wish to avoid a direct computation. Hence, let us assume that we have that value of `lcp` as a consequence of some inductive computation, and that `lcp` is 0 initially. We unroll the `while` loop of function `FindB` and run its iterations as previously done. Then we restate each iteration according to a new scheme driven by two cases formalized in function `FindC` below, in which `x` and `t` are nonempty:

```
node FindC( keytype x, node t ) {
    int m = 0,  lcp = 0,  nextleft = 0;
    while ( 1 ) {
        if (m <= lcp) {       // CASE 1
            for ( ;  *x == t->key[m]; m++ )
                if ( !*x++ ) return t;
            if ( *x < t->key[m] ) {
                if ( !(t = t->left) ) return NULL;
                lcp = t->succ_lcp; nextleft = 0;
            } else {
                if ( !(t = t->right) ) return NULL;
                lcp = t->pred_lcp; nextleft = 1; }
        } else {                // CASE 2
            if ( nextleft ) {
                if ( !(t = t->left) ) return NULL;
                lcp = t->pred_lcp;
            } else {
                if ( !(t = t->right) ) return NULL;
                lcp = t->succ_lcp; }}}}
```

1. Case $\mathtt{m} \leq \mathtt{lcp}$.[1] We compare the characters in `x` and `t->key` as before (the inner `for` loop). After that, `m` (and `lcp`) stores their *lcp*. We branch accordingly with the second `if` statement met in the iteration. Going left, the key in the current node will become the successor of `x` matching its first `m` characters. So, `lcp` will equal `t->succ_lcp`. Analogously, going right, the current key will become the predecessor of `x` matching its first `m` characters and `lcp` will equal `t->pred_lcp`. This keeps the induction on `lcp` for the next `while` iteration.
2. Case $\mathtt{m} > \mathtt{lcp}$. We do not enter the first `if` condition as `lcp` already stores the *lcp* between `x` and `t->key`. Then, we branch with the second `if` statement as in case 1. An important difference with case 1 is that, here, we have the opposite situation to keep the induction on `lcp`. Indeed, the key in the current node is not matching the first `m` characters of `x`. Hence, going left, we surely know that the predecessor of `x` matches its first `m` characters rather than the current key (the successor). So, `lcp` will equal `t->pred_lcp`. Going right, `lcp` will equal `t->succ_lcp`.

[1] Profiling and experiments show that splitting case 1 into two further cases $<$ and $=$ does not pay off.




```c
#define goleft(t)  if ( !(t=t->left) )  return NULL
#define goright(t) if ( !(t=t->right) ) return NULL

node Find(  keytype x, node t ) {
   int m = 0;
   while ( 1 ) {
      if ( x[m] < t->key[m] ) {
           goleft(t);
           while ( t->succ_lcp < m ) goright(t);
      } else if ( x[m] > t->key[m] ) {
           goright(t);
           while ( t->pred_lcp < m ) goleft(t);
      } else {
           if ( x[m++] == '\0' ) return t;
           while ( x[m] == t->key[m] )
               if ( x[m++] == '\0' ) return t; }}}
```

Fig. 1. An augmented binary search tree for strings and its search function `Find`.

In order to better optimize the algorithm behind `FindC`, we avoid a three-way character comparison in case 2 by introducing a variable `nextleft` that stores the negation of the outcome of the comparison in the least recent execution of case 1. In other words, if we branched left the last time we executed case 1 (so, `nextleft` is 0), we have to go right in the following executions of case 2, until another execution of case 1 comes into play. We have an analogous situation for the right branch (`nextleft` is 1). Note that we do not need anymore variables `left_best` and `right_best`.

2.5 Simplification and re-engineering of the final version

Although being optimized, the code of function `FindC` looks rather cryptic. For this reason, we completely restructure it by eliminating the need for the local variables, except `m`, and by running the main `while` loop in classical three-way branching of binary search trees. Namely, we divide the top level search in the three standard cases for tree searching $[<, =, >]$, rather than cases 1–2 of Section 2.4, and design a new `while` loop. Even if variable `lcp` disappears, we refer to cases 1–2 in equivalent terms. We say that case 1 holds if the key in current node matches the first `m` characters of `x` or more; we say that case 2 holds if the key matches strictly less than `m` characters of `x`.

We now describe the new search function shown in Figure 1. Let us assume that initially case 1 holds with the `m`-th character of the key. As the new `while` loop goes on, we keep the invariant that case 1 holds at the beginning of each iteration. That is, the key `x` shares *at least* `m` initial characters with `t` itself (and no ancestor can share more). Then, we restate cases 1–2 in terms of the classical three-way branching, according to the outcome of the comparison between the characters in position `m` of `x` and `t->key`:

[<] We branch to the left child. We start an inner `while` loop that branches rightward as long as case 2 holds. That is exactly what the more complicated procedure `FindC` does. We exit from the inner `while` loop when case 1 holds again, so that we can start a new iteration of the main `while` loop.

[>] We handle this case analogously to the previous one, except that we branch to the right child and start an inner `while` loop branching leftward.

[=] We extend the match for equality as done in function `FindC`. At that point, either we find a whole match and return, or a mismatch and case 1 holds.

For example, in the tree of Figure 1, suppose that we want to search for `x` = `seashore`. We start from the root `seacoast` with `m` = 0 and compare the characters in position 0, which are equal. We therefore are in the third case, where we match `m` = 3 characters (i.e., the prefix `sea`). We then run another iteration of the main `while` loop, which leads to the second case. We apply `goright` reaching `surf` and, since its `pred_lcp` is smaller than `m`, we apply `goleft` reaching `seafood`. At this point, we compare the characters in position `m` and then apply `goright` reaching `seaside`. We run another iteration of the main `while` loop matching one further character, so that we have `m` = 4 (i.e., the prefix `seas`). Finally, we `goleft` and find a NULL pointer, completing the search with a failure. Searching for `x` = `seaside` follows the same path, except for matching all characters, thus completing the search with a success.

2.6 Properties and extensions

The code can be easily modified to handle *prefix searching*, namely, to check whether x is a prefix of one of the stored keys or to compute the longest matching prefix of x. That operation is useful, for example, in text indexing for performing full text searching and in hierarchical path strings to find a common path of two URLs. Despite its simplicity, the code is very efficient. Since the length of the access path in the tree is upper bounded by the height h of the tree in the worst case, the code reduces the cost of searching key x from $O(k \cdot h)$ to $O(k + h)$ in the worst case. To see why, we observe that each match increases the counter m and each mismatch causes traversing one or more nodes in the access path.

Fact 1 *Applying procedure* `Find` *for a search key x of length k in a root-to-node access path of length h of a linked data structure requires m successful comparisons of single characters and at most h unsuccessful comparisons of single characters, where $m \leq k$ is the length of the longest matching prefix of x.*

As previously remarked, the worst situation is for skewed strings, in which the value of m can be frequently much larger than the expected value of $O(\log_{|\Sigma|} n)$, typical of independent and identically distributed strings [31, Chap.4]. Since the average height of a tree is $O(\log n)$, we expect that procedure `Find` has a cost of $O(k + \log n)$ time for n keys. What if we use the standard `strcmp` routine of the C library instead? While this may work reasonably well for random strings, the theory suggests that our techniques can be competitive for skewed strings, since `strcmp` does not exploit previously made comparisons of single characters.

We note that performing insertions is not difficult in this contest. When creating a new node s storing the search key x, we must have followed a certain root-to-node access path π. A better look at the code of `Find` reveals that we can also compute the values of `pred_lcp` and `succ_lcp` for s. We need two extra variables for this purpose, initialized to 0. In the last `else`-branch in Figure 1, after extending a match, we have that m is the length of the longest common prefix between x and the key in the current node. The character in position m can tell if we will reach either the left child or the right child of the current node in π (the next iteration of the main `while` loop). In the former case, the current node is the best candidate for being the successor of s, and so we record m to be possibly stored in field `succ_lcp` of s; in the latter case, the current node is the best candidate for being the predecessor of s, and so we record m to be possibly stored in field `pred_lcp` of s. When s is actually created at the end of π, the best candidates are employed as the predecessor and the successor of s. Having computed the values of fields `pred_lcp` and `succ_lcp`, we proceed by inserting s according to what is required by the data structure at hand (e.g., restructuring it). If needed, the *lcp* values can be suitably updated in $O(1)$ time per node as discussed in [12].

3 The Experimental Setup

In this section we describe the experiments performed. We first briefly sketch the techniques on which we experimented. Next we describe the data sets used, and finally describe the experimental results. We ran our codes on several computing platforms, but we will present only the results relative to a processor AMD Athlon (1Ghz clock and 512Mb RAM) running Linux 2.4.0, since similar results have been obtained on the other platforms.

3.1 The data structures

We developed the ANSI C code for binary search trees [17], treaps [27], AVL trees [2], (a, b)-trees [16], and skiplists [24]. In particular, for binary search trees, treaps, and AVL trees we started from the implementation described in [36], for skiplists we started from the implementation described in [24], which is available via anonymous ftp at [25], while for (a, b)-trees we started from our own implementation of [16, 20]. We used function `fast_scmp` in the same way as described in Section 2.2, leaving the tuning to a later stage. We extensively tested these five implementations on several data sets and computing platforms. We omit here the results of these preliminary tests and we only mention that the computational overhead is very small, validating the prediction of the theory as the key length k goes to infinity. At this stage, however, the resulting data structures were not always competitive with the original data structures using the standard `strcmp`, especially for small/intermediate key lengths. The purpose of these experiments was identifying the most qualified implementations to be tuned as described in Sections 2.3–2.5 and to be tested against already existing data structures. We singled out the binary search trees and the AVL trees, obtaining their tuning code, denoted `bst` and `avl`, respectively.

The string data structures on which we compared are the the adaptive tries of Acharya, Zhu, and Shen [1], the ternary search trees of Bentley and Sedgewick [5], the Patricia tries of Morrison [21], and the linear probing hashing described by Knuth [17]. To establish a connection between the measure of skewness $\mathscr{L}(S)$ given in Section 1 and the performance of the above data structures for a set S

	cat	tst	pat	bst	avl
#nodes	$O\big(n\,(m+1)\big)^*$	$O\big(n\,(m+1)\big)^*$	$O(n)$	$O(n)$	$O(n)$
height	$O(m+1)^*$	$O\big(\log n+(m+1)\big)^*$	$O(m+1)^*$	$O(\log n)^*$	$O(\log n)$

Table 1. The number of nodes and the height of the data structures considered in the experiments for a set S of n strings. The bounds are also in terms of m, the average length of matched prefixes in the formula for the skewness, $(m+1) = \mathscr{L}(S)\ \log_{|\Sigma|} n$ (see Section 1). The starred bounds hold on the average.

of n strings, Table 1 reports their number of nodes and their height in terms of m, the average length of matched prefixes in the formula for the skewness, $(m+1) = \mathscr{L}(S)\ \log_{|\Sigma|} n$. Note that $m \leq k$, the maximum string length.

The cache-aware version of tries (`cat`) uses multiple alternative data structures (partitioned arrays, B-trees and hashing arrays) for representing different trie nodes, according to their fanout and to the underlying cache. The height is $O(m+1)$ on the average and $O(k)$ in the worst case, with $O(n(m+1))$ nodes on the average and $O(nk)$ nodes in the worst case. We slightly modified their highly tuned and optimized C++ code at [1] to use our timing routines.

The ternary search trees (`tst`) are a blend of tries and binary search trees. A ternary search tree has $O(n(m+1))$ nodes on the average and $O(nk)$ nodes in the worst case. Its height is $O(m+\log n)$ on the average [9] and contributes to the cost of searching, $O(k+\log n)$ time. We downloaded their C code at [5] and slightly modified it with our timing routines.

The Patricia tries (`pat`) are compacted tries storing the strings in their leaves in a total of $O(n)$ nodes, and their height is $O(k)$ in the worst case and $O(m+1)$ on the average. Searching in Patricia tries takes $O(k+\log n)$ time on the average [9, 17] by comparing just one character per node, at the price of a full comparison of the key in the node where the search ends. We employed the implementation described in [26] and removed the recursion.

Finally, we implemented linear probing hashing (`hsh`) by using the hash function in [18] as suggested in [29], with load factor 0.5 yielding $O(n)$ occupied space. Although hashing does not permit to perform prefix searching, we employed its average search time to *normalize* the timings of the other data structures, believing that it may reveal their qualities better than absolute timings.

3.2 The data sets

1. A word data set. It consists of two dictionaries `dictwords1` and `dictwords2`. The former is available at [5]: it contains 25,481 words of average length of 8.23 characters. The latter is the dictionary available on Unix machines in file `/usr/dict/words`, containing 45,407 words of average length 9.01.
2. A book data set. It consists of `mobydick` (i.e., Melville's "Moby Dick" available from [23]) with 212,891 words of average length 6.59 and of `warandpeace` (i.e., Tolstoj's "War and Peace" available from the same URL) with 565,551 words of average length 5.64.

3. A library call number data set. Used in one of the DIMACS Implementation Challenges, it is available from [28]. Each entry is a library card number (such as `WGER______2455___55___20`). From this data set, we picked `circ2.10000`, which consists of 9,983 strings of average length 22.52, and `circ2.100000`, which consists of 100,003 strings of average length 22.54. We also generated `libcall` from other files, consisting of 50,000 strings of average length 22.71.
4. A source code data set. Here, keys are the code lines. We choose `gprof`, the source code of a profiler consisting of 7,832 code lines of average length 25.87, `kernel`, the source code of a Linux Kernel consisting of 48,731 code lines of average length 29.11, and `ld`, the source code of a linker containing 19,808 lines of average length 27.77.
5. A URL data set. Each key is a uniform resource locator in the Web, where we dropped the initial `http://` prefix. Files range from 25,000 to 1,500,000 URLS. The URLs were collected in the Web domain `*.*.it` by a spider for the search engine in [3] and by the proxy server at the University of Pisa [7].

The dictionary and book data sets in points 1 and 2 tend to have the smaller values of skewness $\mathscr{L}(S)$: they consist of many words having small average lengths. On the other extreme of the spectrum we have the source code and the URL data sets in points 4–5, which are characterized by large values of $\mathscr{L}(S)$. In between, we find the library call number data sets in point 3.

3.3 The results

We ran several experiments on our data sets. One experiment was using two different data sets. The first data set (e.g., a dictionary, such as `dictcall1` or `dictcall2`) was inserted into the data structure at hand, one word at the time. Against this data set, we ran searches using all the words of the second data set (e.g., a book such as `mobydick` and `warandpeace`). The experiment thus consisted of carrying out a batch of insertions followed by a batch of lookups. We performed several experiments of this type, according to different data sets, and to whether the data sets were scrambled randomly or not. As a special case of this experiment, we also used the same data set twice: i.e., we first inserted the words of a data set (randomly scrambled or not) one at the time in the data structure at hand; next, we scrambled the same data set, and searched the data structure for each word in this random order. The goal of the data scrambling was to avoid sorted data, and to force the batches of insertions and lookups to follow completely different patterns in the data structure. We considered the searches both in the successful case (the key is stored in the trees) and in the unsuccessful case (the key is not stored). We ran each experiment several times and retained the average of their running times. Times were measured in milliseconds with the Unix command `getrusage()`.

The running times of the insert operations, the successful searches and the unsuccessful searches are reported in Tables 2–4, respectively. All these experiments yielded very similar results. Indeed for different experiments, there seemed to be very little difference in the relative behavior of our algorithms (even though, clearly, the absolute CPU times were substantially different). What we learned

$\mathscr{L}(S)$	cat	pat	tst	bst	avl
1.98	1.00	3.33	3.01	2.33	3.00
2.53	0.88	2.00	1.34	1.55	2.05
4.91	1.93	3.00	2.57	2.53	3.61
5.60	2.58	1.7	2.66	1.25	1.67
10.95	2.56	4.62	4.10	3.49	4.42
11.75	2.69	3.66	3.60	2.72	3.56
12.94	3.26	3.06	3.41	2.25	3.03
14.70	3.86	2.66	3.26	1.93	2.60
16.64	5.48	2.57	3.28	1.71	2.14
19.14	4.00	3.15	3.73	2.31	3.03
22.26	4.80	2.66	3.56	1.93	2.43
26.07	5.14	2.42	3.47	1.67	2.04

Table 2. Running times of the insert operation for the data sets S in order of skewness $\mathscr{L}(S)$. The running times are normalized with the corresponding ones of `hsh`, whose value is hence 1 in each entry (not reported).

from our experiments is that `cat` appeared to be fast on large data sets with small keys (i.e., sets S with small skewness $\mathscr{L}(S)$). On the opposite side, `bst` and `avl` produced data structures which were effective on data sets with skewed keys (i.e., with large $\mathscr{L}(S)$). In between, the performance of these data structures were not so much different, depending on the underlying architecture: however, in all our experiments `tst` started being competitive with the other two approaches exactly in this range. Note that `pat` was sometimes less competitive because of the double traversal needed by the search and update operations.

An explanation of the experimental behavior of these data structures and their relation to the skewness $\mathscr{L}(S)$ can be found in Table 1, in which their number of nodes models the space occupancy and their height models the time performance. First, notice that we can classify the data structures in three classes: tries (`cat`,`tst`), compacted tries (`pat`), and lexicographic trees (`tst`,`bst`,`avl`). The members of the former two classes have depth proportional to the average length of matched prefixes, m, while the members of the latter class have a depth proportional to the logarithm of the number of strings, n, with `tst` being a hybrid member of two classes. Second, their performance is influenced by the skewness $\mathscr{L}(S)$, as larger values of $\mathscr{L}(S)$ cause larger values of m for fixed n.

With the two observations above, we now discuss the experimental behavior of data structures. When $\mathscr{L}(S)$ is small, the (compacted) tries have depth smaller than $O(\log n)$ and so their performance is superior, with the exception of `pat`, which requires a double traversal. Also the number of nodes is nearly $O(n)$. When $\mathscr{L}(S)$ is large, the lexicographic trees have depth smaller than $O(m)$, and they compare better. They always require $O(n)$ nodes, while the others do not guarantee this upper bound for skewed strings or have worse time performance. For intermediate values of $\mathscr{L}(S)$, tries and lexicographic trees are of comparable heights, and the data structures behave reasonably well. It turns out that `cat` is very fast for small skewness; `tst` is more competitive for middle skewness;

$\mathscr{L}(S)$	cat	pat	tst	bst	avl
1.98	0.30	3.33	3.04	2.33	2.67
2.53	0.55	2.11	1.27	1.47	1.94
4.91	0.96	2.85	2.44	2.62	3.14
5.60	0.92	1.33	1.93	1.00	1.40
10.95	1.15	4.16	3.63	3.19	3.58
11.75	1.21	3.29	3.18	2.52	2.87
12.94	1.49	2.62	2.91	2.05	2.43
14.70	1.77	2.11	2.61	1.66	2.11
16.64	2.10	1.55	2.34	1.33	1.55
19.14	1.65	2.46	2.92	1.92	2.36
22.26	1.77	1.89	2.64	1.53	1.97
26.07	2.18	1.70	2.55	1.33	1.74

Table 3. Running times of the successful search for the data sets S in order of of skewness $\mathscr{L}(S)$. The running times are normalized with the corresponding ones of `hsh`.

`avl` and `bst` are the choice for large skewness; `pat` is sometimes less efficient for searching and inserting; `hsh` is the fastest in all experimented cases, at the price of supporting less powerful operations. In general, `cat` takes significant more space than the other data structures when the strings are skewed.

In summary, for $m < \alpha_1 \log_{|\Sigma|} n$, for some constant α_1, we expect `cat` to be the fastest, and for $m > \alpha_2 \log_{|\Sigma|} n$, for some other constant $\alpha_2 > \alpha_1$, we expect `avl` and `bst` to take over, while `tst` are good in the middle range. These constants are highly machine-dependent, as expected. In our case, the search running times were from 1 to 4 times slower than hashing while the insert running times were slower from 1 to 5 times. The maximum speedup, computed as the maximum ratio between the running times of the slowest and the fastest algorithms for each data set (except for the first two in Table 3), is around 2 or 3. Although it may not appear as impressive as in other problems, we point out that the involved data structures are highly tuned. When employed in a high-performance server for indexing, sorting and compressing textual data, doubling the actual amount of served requests per time unit can make a difference in practice.

Some final remarks are in order. Search time in `tst` is rarely worst-case compared to that of `bst`, and `tst` has typically smaller height than that of `bst`. The extra work for keeping `avl` balanced apparently does not pay. As for the required space, `cat` and `tst` take more nodes than the others, which require to store explicitly the strings somewhere else. In some cases, `tst` may require globally less space (including that required by the strings) because it can store the strings in its structure. Being so simple, `tst` are the method of choice for tries [9] while our experiments indicate that `bst` are effective choices when compacted tries are needed, especially with skewed strings.

4 Conclusions

We described a general data structuring technique that, given a linked data structure for storing atomic keys undergoing comparisons, transforms it into a

$\mathscr{L}(S)$	pat	tst	bst	avl
1.98	3.00	3.02	3.00	4.02
2.53	3.02	2.50	3.01	3.01
4.91	2.25	1.75	2.50	3.00
5.60	1.33	1.66	1.00	2.01
10.95	3.88	3.17	3.30	3.80
11.75	2.79	2.45	2.39	2.87
12.94	2.43	2.43	2.12	2.68
14.70	1.77	1.88	1.33	1.88
16.64	2.00	2.33	2.00	2.66
19.14	2.43	2.43	2.10	2.66
22.26	2.06	2.25	1.62	2.18
26.07	1.63	1.91	1.36	1.90

Table 4. Running times of the unsuccessful search (not reported for `cat`) for the data sets S in order of of skewness $\mathscr{L}(S)$. The running times are normalized with the corresponding ones of `hsh`.

data structure for strings. Despite the simplicity and generality of our method, our experimental study shows that it is quite competitive with some implementations currently available in the literature. We can extend our technique to obtain text indexing data structures that are also able to perform suffix sorting. Hence, our augmented data structures provide a good alternative to popular text indexes such as suffix arrays and suffix trees [15]. They can also perform the suffix sorting of a text, which is the basis of text indexing and data compression.

Acknowledgements

We thank Salvatore Ruggieri for providing some of the data sets in [7] and Craig Silverstein for sending us a copy of [29].

References

1. A. Acharya, H. Zhu, K. Shen. Adaptive algorithms for cache-efficient trie search. *Proc. ALENEX 99*. Source code in `http://www.cs.rochester.edu/~kshen`, 1999.
2. G. M. Adel'son-Vel'skii and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3 (1962), 1259–1263.
3. Arianna Search Engine, `http://arianna.libero.it`.
4. R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica* 1 (1972), 290–306.
5. J.L. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. *SODA 1997*, 360–369 (`http://www.cs.princeton.edu/~rs/strings`).
6. S.W. Bent, D.D. Sleator and R.E. Tarjan. Biased search trees. *SIAM Journal on Computing* 14 (1985), 545–568.
7. F. Bonchi, F. Giannotti, C. Gozzi, G. Manco, M. Nanni, D. Pedreschi, C. Renso and S. Ruggieri, Web log data warehousing and mining for intelligent web caching. *Data and Knowledge Engineering*, to appear.
8. H.A. Clampett. Randomized binary searching with the tree structures. *Communications of the ACM* 7 (1964), 163–165.

9. J. Clément, Ph. Flajolet and B. Vallée. The analysis of hybrid trie structures. *SODA 1999*.
10. J. Feigenbaum and R.E. Tarjan. Two new kinds of biased search trees. *Bell Systems Technical Journal* 62 (1983), 3139–3158.
11. T.F.Gonzalez. The on-line d-dimensional dictionary problem. *SODA1992*, 376–385.
12. R. Grossi, G. F. Italiano. Efficient techniques for maintaining multidimensional keys in linked data structures. *ICALP 1999*, 372–381.
13. R.H. Gueting and H.-P. Kriegel. Multidimensional B-tree: An efficient dynamic file structure for exact match queries. *10th GI Annual Conference*, 375–388.
14. L.J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. *FOCS 1978*, 8–21.
15. D. Gusfield, *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*, Cambridge University Press, 1997.
16. S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica* 17 (1982), 157–184.
17. D.E. Knuth. *The Art of computer programming, Vol. 3 : Sorting and Searching*. Second edition, Addison-Wesley, 1973, 1998.
18. R. Jenkins, `http://burtleburtle.net/bob/hash/`.
19. K. Mehlhorn. Dynamic binary search. *SIAM J. on Computing* 8 (1979), 175–198.
20. K. Mehlhorn. *Data structures and algorithms:Searching and sorting*, Springer 1984.
21. D.R. Morrison. PATRICIA — Practical Algorithm To Retrieve Information Coded In Alphanumeric. *J. ACM*, 15, 514–534,1968.
22. J. Nievergelt and E.M. Reingold. Binary search trees of bounded balance. *SIAM Journal on Computing* 2 (1973), 33–43.
23. J.M. Ockerbloom. The on-line books page. `http://digital.library.upenn.edu/books/`
24. W. Pugh. Skip Lists: A probabilistic alternative to balanced trees. *Communications of the ACM* 33 (1990), 668–676.
25. W. Pugh. Skip Lists ftp directory. `ftp://ftp.cs.umd.edu/pub/skipLists/`.
26. R. Sedgewick. *Algorithms in C*, Addison-Wesley, 1998.
27. R. Seidel and C.R. Aragon. Randomized search trees. *Algorithmica* 16 (1996), 464–497.
28. C. Silverstein. Library call data set. `http://theory.stanford.edu/∼csilvers/libdata`.
29. C. Silverstein. A practical perfect hash algorithm. Manuscript, 1998.
30. D.D. Sleator and R.E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM* 32 (1985), 652–686.
31. W. Szpankowski, *Average Case Analysis of Algorithms on Sequences*, Wiley, 2001.
32. R.E. Tarjan. *Data structures and network algorithms*, SIAM (1983).
33. V.K. Vaishnavi. Multidimensional height-balanced trees. *IEEE Transactions on Computers* C-33 (1984), 334–343.
34. V.K. Vaishnavi. Multidimensional balanced binary trees. *IEEE Transactions on Computers* C-38 (1989), 968–985.
35. V.K. Vaishnavi. On k-dimensional balanced binary trees. *Journal of Computer and System Sciences* 52 (1996), 328–348.
36. Mark A. Weiss. *Data structures and algorithm analysis in C*, Addison Wesley, (1997). Source code in `http://www.cs.fiu.edu/~weiss/dsaa_c2e/files.html`.