On the number of string lookups in BSTs (and related algorithms) with digital access

Leonor Frias *
Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
lfrias@lsi.upc.edu

Abstract

Binary search trees and quicksort are examples of comparison-based data structure and algorithm respectively. Comparison-based data structures and algorithms can be can be augmented so that no redundant character comparisons are made. Unnoticed, this approach also avoids looking up the string in some nodes. This paper characterizes analytically the number of string lookups in so-augmented BSTs, quicksort and quickselect. Besides, we also characterize a variant proposed in this paper to reduce further the number of string lookups.

1 Introduction

In comparison-based data structures and algorithms, comparisons are made taking the keys as a whole, whatever the key type is. This is simple, there is often no other obvious way to compare keys, and in many cases leads to a good performance. However, for string keys this approach is rough and string keys are indeed common.

String keys can be seen as a sequence of digits or characters, whose alphabet size is typically relatively small. Efficient *ad hoc* string algorithms and data structures (i.e. tries and its variants) exploit the digital structure of string keys to avoid redundant character comparisons. However, their worst-case performance is tied to the string length and in general, their performance depends greatly on the characteristics of the dataset. Also, most of them require information on the alphabet cardinality (i.e. the number of different digits). Instead, it is possible to specialize comparison-based data structures and algorithms, so that no redundant digit comparisons are made whilst keeping the rest of combinatorial properties (see [13, 11, 9]). That is, their main advantage is robustness. The application of these techniques relies on the relative order in which elements are compared. Specifically, information on the common prefixes between one element and its predecessor/successor in the access path must be kept. Besides, using this information not only redundant character comparisons are avoided but actually accessing the string data itself in some cases. This is relevant from a performance perspective, because string data is accessed through pointers to arbitrarily far locations in memory and memory hierarchies in modern computers deal efficiently only with memory accesses with high locality of reference (i.e., repetitive memory accesses to locations close in memory).

In this paper, we analyze the number of string lookups in so-enhanced binary search trees (BSTs), quicksort and quickselect. In the following, we call them respectively augmented BSTs (aBSTS), augmented quicksort (aQSort) and augmented quickselect (aQSel). In all cases, the implementation is amenable. Besides, in [5] an experimental study showed that aBSTS are also competitive in practice.

In order to characterize aBSTs, aQSort and aQSel

*Supported by Spanish project ALINEX (ref. TIN2005-05446)

from the number of string lookups perspective is key to relate them with some kind of tries. Indeed, the relationship with tries is stated in [13] to characterize them from the point of view of the digit comparisons. In Section 2, these data structures and algorithms are presented in more detail. In Section 3, the notation is introduced. Then, in Section 4, we describe precisely the relationship between aBSTs and tries and present the theoretical analysis on the number of string lookups. Following from this analysis, an extension of aBSTs is presented, which aims to avoid one specific kind of string lookups. Its analysis is on Section 5. Then, in Section 6, we analyze aQSort and a aQSel relating them to aBSTs. Finally, Section 7 closes with a summary and a discussion of further research.

2 Preliminaries

In order to search and sort strings efficiently, we can either use a trie-like data structure or algorithm, or enhance a BST-like data structure or algorithm with efficient digital access. Examples from the latter are aBSTs, aQSort and aQSel. In the following, we describe them as well as some trie-like data structures and algorithms of relevant interest in order to characterize them.

Figure 1 shows an example of an aBST, as well as the corresponding trie-based data structures.

2.1 Combining BSTs, quicksort and quickselect with digital access

Comparison-based data structures and algorithms can be augmented, so that its combinatorial properties are kept whilst enhancing efficient digital access. Thus, they are of particular interest when worst-case guarantees (such as, logarithmic cost search in the number of elements) are required. In return, linear extra space in the number of elements is needed.

aBSTs. Combining BSTs with fast digital comparisons was particularly tackled in [13, 11].

Each node in a BST stores a string y and pointers to the children (and possibly the parent). Let ℓ_p and ℓ_s be respectively the maximum common prefix of y with its predecessor and its successor. Each node in an aBST stores additionally ℓ, defined as the maximum of ℓ_p and ℓ_s, and a boolean b to indicate which is the maximum. Searching for a string w works as follows: while going down the tree, two lengths are kept: l_p and l_s; they are respectively the maximum common prefix of w with the current predecessor and the current successor (and thus, initially they equal to 0). The new comparison function works as follows (see Figure 2 for a possible definition). ℓ is compared against l_p or l_s (according to b). Let l be that length. If $l = \ell$, the characters of w must be compared against those of y, starting at the $\ell+1$-th position, which has the first significant character of y, until a difference is found. Besides, if at least the first couple of characters are equal (i.e. $y[\ell+1] = w[\ell+1]$), ℓ_p or ℓ_s must be updated accordingly. If $l \neq \ell$, the next node to visit is determined merely on the values of ℓ, b, and l, the string y is not looked up, and there is no change in ℓ_p or ℓ_s. In Figure 2 the comparison function is sketched. Using this method, searching for a string of length m in a BST of height h takes time $\Theta(m+h)$ in the worst-case.

```
resultComparison compare()(nodePtr t, stringType& key,
    int& l_p, int& l_s){
    if (t.b){
        if (l_p > t->l) return SMALLER;
        if (l_p < t->l or l_p < l_s) return GREATER;
        return look_string(t, key, l_p, l_s);
    }
    else{
        if (l_s > t->l) return GREATER;
        if (l_s < t->l or l_s < l_p) return SMALLER;
        return look_string(t, key, l_p, l_s);
    }
}
```

Figure 2: Compare function for aBSTs

Concerning storage needs, the overall increment in storage becomes less important as longer the string pointed by the node. Besides, looking up the fields ℓ and b in a search only increases the number of local memory accesses, which are handled efficiently by memory hierarchies.

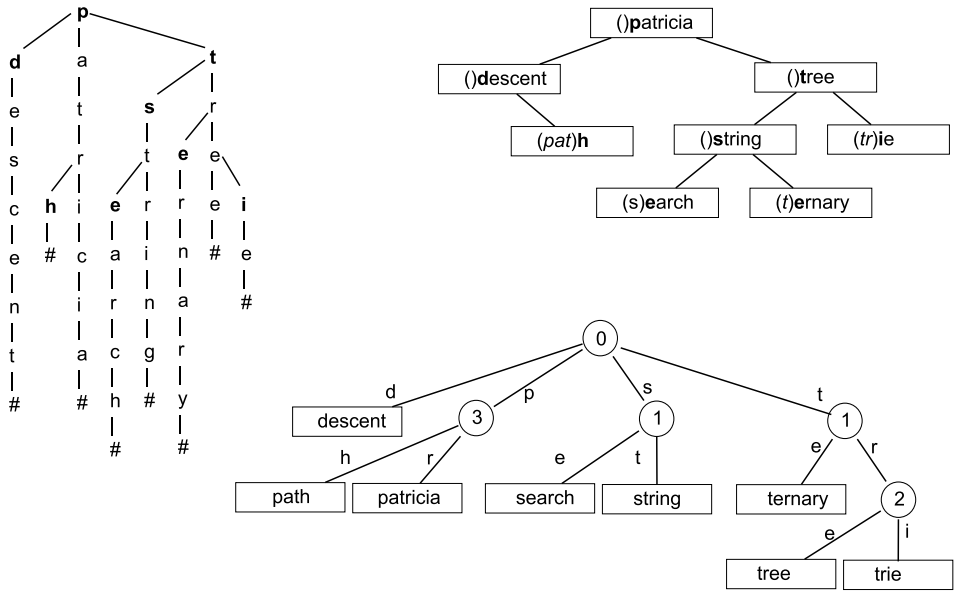

Figure 1: Example of a TST (left), an aBST (top) and a Patricia trie (bottom) storing the same set of strings. The shared prefixes are shown between parentheses.

aQSort and aQSel. Combining quicksort with fast digital comparisons was particularly tackled only in [13]. Also, recently the problem has been revisited considering their parallel versions [10].

The following properties on the comparisons made in quicksort are used. First, if the pivot is chosen otherwise than comparing elements, comparisons are made solely in partitioning. In one partition, each element is compared exactly once against the pivot. Besides, the implicit structure defined by the recursive calls in quicksort corresponds to a BST, where each pivot choice (and partitioning) constitutes a node and base cases in the recursion correspond to leaves. Note that, in the case of quickselect (only explicitly tackled in [10]), the structure of the recursive calls corresponds to a path, which is a particular case of BST.

Then, we proceed analogously as in the aBST. For each string x the length of the maximum common prefix of x with its predecessor and successor in the implicit BST structure is kept. Besides, the following changes must be done in partition. Let y be the string acting as pivot, and let x be an string in the array to be partitioned. Then, the comparison function of x against y is analogous to that in aBSTs: y acts as the string in the node and x acts as the string to be searched. Besides, the swap function used in partition to swap two strings x_1 and x_2 must be specialized so that, in addition, their prefix information is also swapped.

The previous assumes that the pivot is chosen without comparing elements and that it is not compared against itself. However, a good pivot choice is crucial for guaranteeing the quasilinear performance of quicksort. For instance, one widely used approach consists on using the median of 3 (or more elements). Doing so would cause making some comparisons twice and would destroy the BST properties. However, we can compare the elements as long as the prefix information is not corrupted. One possibility is providing an additional comparison function that does use prefix information but does not update it. Even better, given that we are not that interested in the exact result, we can rely merely on prefix information to

make a decision (to the detriment of the quality of the partitioning).

2.2 Tries and variants

Tries are traditional efficient string data structures. They are a digit-based data structures, that is keys are considered as a sequence of digits or characters in a fixed alphabet. Specifically, tries are trees of minimal size in which there is a node for every common prefix. Thus, the worst-case performance is tied to the string length. In many common situations the string length is relatively small (as words in a natural language) but this is not necessarily the case.

There are several representations and variations of tries. In the following we described the most relevant for the aim of this paper.

TSTs, Multikey Quicksort and Quickselect. The most typical representation of tries is using an array of characters for each node to access the children subtrees. However, this option is adequate only when the cardinality of the alphabet is small. In order to not waste so much space with unused pointers, a popular alternative are ternary search trees [1] (TSTs).

TSTs are a trie representation in which BSTs are used to guide descent. Each node in a TST contains at most two (not null) *comparison pointers* and a (not null) *descent pointer*. When searching for a string w, a comparison pointer (shown with a diagonal line in Figure 1) is followed when the output of the comparison is '<' or '>'. A descent pointer (shown with a vertical line in Figure 1) is followed when the output of the comparison is '='. In addition, the end of the strings is marked with a special end character (in Figure 1 is denoted by `#`).

There is a unique trie for a given set of strings. Instead, there might be several TSTs because for a given position and prefix, the k possible following characters can be placed in as many ways as BSTs with k nodes; binary search is used to find the following character. For instance, given the set of strings in Figure 1, there are 4 possibilities for the first position: `d`, `p`, `s` and `t` and 14 resulting BSTs depending on the choices to place them. Finally, as with BSTs, TSTs may be balanced by a variety of schemes. In [4] a detailed analysis of TSTs is given.

Finally, multikey quicksort is a divide-and-conquer sorting algorithm isomorphic to TSTs, also presented in [1]. In the case of multikey quickselect, the sequence of calls defines a path in that isomorphic TST.

Patricia tries. The classical variation of tries are *Patricia tries* [12]. A Patricia trie is a compacted trie built as a binary trie s.t. one-way branching is avoided by including in each node the number of characters to skip over before making the next test. That is, there is a node in the Patricia trie for each node that in the corresponding trie two or more pointers go out from it. Besides, as in tries, its shape is determined uniquely by the set of strings. Furthermore, they constitute a basic block of the IO-efficient string B-tree in [8] and the static cache oblivious dictionary in [3] because it is guaranteed that only one string is looked up per search.

3 Notation

In this section, we present the notation that will be used throughout the paper.

General. The following notation is used. Let a be an array of strings, $S(a)$ denotes the set of strings in a. Analogously, let u be a search tree (e.g. a TST, an aBST), let w be a string. $S(u)$ denotes the set of strings stored in u. $C(u, w)$ denotes the sequence of pairwise character comparisons made in u searching for w. Each pairwise character comparison in u is *matched* to the node that is looked up to make the comparison. $I(u, w)$ denotes the sequence of subsequences resulting from partitioning $C(u, w)$ after each inequality comparison.

TSTs. The following notation on TST elements is used. A node with at least one not null comparison pointer is a *decision node*. A node that is not reached following a descent pointer is a *descent root*. A path that begins in a descent root, ends either in a decision node or a in leaf, and is made exclusively of descent pointers, is called a *descent path*. It is *proper*

if it contains at least two nodes (joint by one descent pointer). Note that intermediate nodes can also be decision nodes. Thus, there may be several descent paths beginning at the same node, but with different lengths. For instance, in Figure 1, there are three descent paths beginning from the root right child: `tr`, `tre` and `tree#`. Let x be a decision root or a leaf in a TST; $D(x)$ denotes the sequence of nodes in the descent path that ends in x.

The following notation regard to the searching path of a TST for a string w. A *decision taken node* for w is a decision node in the searching path of w such that one of its comparison pointers is followed. For instance, searching for `tree` in the TST in Figure 1, the root is a decision taken node, whilst its right child is a decision node but not a decision taken node. A *search descent path* for w is a descent path ending in a decision taken node or a leaf. Alternatively stated, it is a maximal descent path in the searching path of w. It is *proper* if it contains at least two nodes. For instance, the searching path for `tree` in the TST in Figure 1, contains only one search descent path: `tree#`. The searching path for `trie` contains two search descent paths: `tre` and `ie#`. (Note that the last character in the search descent paths are not included in the searched word; for instance, `trie` is build up concatenating `tr` and `ie`).

4 On the number of string lookups in aBSTs

In this section, we analyze the number of string accesses in aBSTs. First, we present some properties on TSTs and aBSTs (independently). Then, we relate both search trees. Finally, we relate the number of string accesses in aBSTs with known properties in TSTs.

Lemma 1. *Let t be a TST and let w be a string. Each of the subsequences in $I(t, w)$ matches to a search descent path in t for w and each search descent path in t for w is matched to a subsequence.*

Proof. $I(t, w)$ is obtained from $C(t, w)$ partitioning after each inequality comparison. In t inequality happens only if a comparison pointer is followed. Thus, the nodes matching one subsequence in $I(t, w)$ must be joint by descent pointers, the first must be a descent root and the last must be a decision taken node. This is exactly a search descent path. On the other hand, each search descent path must be matched to a subsequence in $I(t, w)$ because otherwise $I(t, w)$ would not represent the sequence $C(t, w)$. □

From the previous lemma, it follows that:

Corollary 1. *Let t be a TST. $|I(t, w)|$ corresponds with the number of search descent paths in t for searching a string w. $\sum_{w \in S(t)} |I(t, w)|$ corresponds with the cumulative number of search descent paths in t searching for every string in $S(t)$.*

Now, we analyze ABSTs.

Lemma 2. *Let b be an aBST and let w be a string. Each of the subsequences in $I(b, w)$ matches to a node in b. Moreover, if a node in the searching path of b searching for w does not match to any subsequence in $I(b, w)$, then no character is compared in the node.*

Proof. $I(b, w)$ is obtained from $C(b, w)$ partitioning after each inequality comparison. In b inequality happens only if a pointer is followed. Thus, each of the subsequences in $I(b, w)$ must be matched to exactly one node. On the other hand, if a node is not matched to any subsequence, no character is compared in the node because otherwise $I(b, w)$ would not represent the sequence $C(b, w)$. □

From the previous Lemma, it follows that:

Corollary 2. *Let b be an aBST. $|I(b, w)|$ corresponds with the number of string lookups searching for a string w. $\sum_{w \in S(b)} |I(b, w)|$ correspond with the cumulative number of string lookups in b searching for every string in $S(b)$.*

Using the previous properties, we relate TSTs and aBSTs.

Definition 1. *A TST t and an aBST b are* equivalent *iff for any string w, $I(t, w)$ equals $I(b, w)$.*

Note that, if t and b are equivalent, then $S(t)$ and $S(b)$ must be equal, so as the number of digit comparisons for a fixed w.

Lemma 3. *Let S be a set of strings. If an aBST b and a TST t are built inserting the strings in S in the same order (and apply no rotations), t and b are equivalent.*

Proof. The proof is by induction. After inserting the first string w_1, b contains only one node, which contains w; t contains a descent path of $|w|+1$ nodes, i.e. one for each character in w plus the special end character. Searching any string y in b and in t, results in $I(b,y)$ and $I(t,y)$ containing only one subsequence. Besides, the subsequence is the same because only w_1 is stored.

Before inserting the string w_{i+1} in t and in b, i strings have already been inserted in each tree, and they are equivalent. Given the equivalence, it holds that $I(t,w_{i+1})$ equals $I(b,w_{i+1})$. Let r be the leaf in t and let s be the leaf in b for searching w_{i+1}. Let x be the length of the longest common prefix of w_{i+1} with the strings already inserted. Inserting w_{i+1} in t and in b produces the following modifications. In t, a descent path of $|w_{i+1}|-x+1$ nodes is added as a child of r containing each of the characters in w starting at the $(x+1)$-th position (in sequence order) plus the special end character. In b, exactly only one node is added as a child of s containing w_{i+1}. In any case, given a searched string y, $I(t,y)$ and $I(b,y)$ change only if nodes r and s are reached respectively. In particular, $I(t,y)$ and $I(b,y)$ contain one additional subsequence. Besides, the resulting new subsequence is the same in t and in b, because the same substring is reached from r and s respectively. □

Note that an aBST has a unique equivalent TST, but a TST may have several equivalent aBSTs.

Let t be a TST and let b be an aBST which are equivalent. The following holds on the number of string lookups in aBSTs.

Lemma 4. *The number of string lookups in b searching for a string w coincides with the number of search descent paths in t for w. The number of string lookups in b searching for all the strings in b coincides with the cumulative number of search descent paths in t searching for every string in $S(t)$.*

Proof. From Corollary 1, $|I(t,w)|$ corresponds with the number of search descent paths in t for w. From Corollary 2, $|I(b,w)|$ corresponds with the number of string lookups in b searching for w. Given that t and b are equivalent, $I(t,w)$ equals $I(b,w)$. Thus, $|I(t,w)|$ equals $|I(b,w)|$ and $\sum_{w\in S(t)}|I(t,w)|$ equals $\sum_{w\in S(b)}|I(b,w)|$ and the lemma follows. □

Recall that in the case of BSTs, the number of string lookups in searching coincides with the number of key comparisons because each comparison requires looking up a string. Thus, it is greater or equal than the number of string lookups in searching in aBSTs. Note that, in a TST there are no string lookups (because each node contain one character not a string). In this case, the most relevant performance parameter is the number of visited nodes (which depends on the string length and the alphabet cardinality).

Finally, we determine precisely the number of string accesses in searching in an ABST using the following fact.

Fact 1. *In [4] the* (comparison) search cost $R(t,w)$ *of a string w in a TST t is defined as the number of comparison pointers in the searching path of w in t. This is equivalent to counting the number of search descent paths in t for w except the last one. Besides, the* (comparison) path length $L(t)$ *of a TST t is defined as the sum of the distances of all external nodes to the root of the tree, where distance is measured in the number of comparison pointers. This is equivalent to the cumulative number of search descent paths in t minus $S(t)$.*

An exact mathematical expression for $R(t,w)$ and $L(t)$ is given in [4, Theorem 2.1]. An asymptotic expression for $R(t,w)$ and $L(t)$ is given in [4, Theorem 2.2]. Then, from Lemma 4 and the previous fact, it follows that:

Corollary 3. *The number of string lookups in b searching for string w is $R(t,w)+1$. The number of string lookups in b searching for every string in $S(b)$ is $L(t)+S(t)$.*

Note that, from Lemma 3, if b is generated using a probability distribution α, α is the probability distribution for generating t.

5 On the number of string lookups in CaBSTs, an extension of aBSTs

aBSTs not only avoid redundant character comparisons but also looking up some strings. In the following, an extension of aBSTs is presented and analyzed aimed to avoid the string lookups due to binary searching. We call them CaBSTs. We relate the number of string lookups in CaBSTs with properties in TSTs and in turn with properties in Patricia tries.

We define CaBSTs as follows.

Definition 2. *A* CaBST *cb is an extension of an aBST b in which a character field is added to each node. This additional field stores the first significant character of the string y stored in the node. We say that cb* corresponds *to b.*

Comparisons are modified accordingly to take this character value into account. The sequence of character comparisons, though, remains the same, and so, they can be related to TSTs analogously as aBSTs are related to them.

Definition 3. *Given a string w, a character stored in a node r in a CaBST is* useful *in searching w, iff it avoids from looking up the string in r.*

Consider a node in an aBST whose string should be looked up for searching w. Intuitively, as larger the diversity of strings sharing the current prefix, greater the probability that the character in the corresponding node in a CaBST is useful.

In the following, the benefits from CaBSTs searching for a string w are described precisely taking into account the relationship between aBSTs and TSTs. Let t be a TST and let b be an aBST which are equivalent; let cb be a CaBST corresponding to b.

Definition 4. *Let t be a TST and let b be an (C)aBST that are equivalent (thus $I(t, w) = I(b, w)$); let r be a node in b and let s be a node in t. r and s are* search related *if the same subsequence in $I(t, w)$ is matched in t and in b searching for w.*

Lemma 5. *Let r be a node in cb search related to a node s in t for searching w, a character stored in r is useful in searching w iff $D(s)$ is not a proper search descent path.*

Proof. A character stored in r is useful for searching w, iff it avoids from looking up the string in r. That is, if exactly one pairwise character comparison is made in r. Given that s in t is search related to r, only one character is compared in $D(s)$. This implies that $D(s)$ is not a proper search descent path.

The opposite implication is proved as follows. If $D(s)$ is a not a proper search descent path, only one character is compared in $D(s)$. Given that s is search related to r, also only one character must be compared in r. Besides, according to CaBSTs definition, the character stored in r is the first significant character, and thus, it is compared against the current character of w. Given that only one character must be compared in r, the character in r is useful. □

That is, Lemma 5 states that CaBSTs only need one string lookup to determine the next value for a character position. Specifically, the looked up string is the one with the coinciding character value.

Lemma 6. *The number of proper search descent paths in the searching path of t for w coincides with the number of strings looked up in cb searching for w. The cumulative number of search descent paths in t searching for every string in $S(t)$ coincides with the total number of strings looked up in cb searching for every string in $S(b)$.*

Proof. From Lemma 4, the number of nodes in b whose string is looked up searching for w coincides with the number of search descent paths in t. In cb, from them, as many string lookups as useful nodes are avoided. According to Lemma 5, a node in cb is useful if it is search related to a node s in t, s.t. $D(s)$ is not a proper search descent path. Thus, a string in a node in cb is looked up in cb searching for w only if it is search related to a node s in t, s.t. $D(s)$ is a proper search descent path. Then, the lemma follows. □

However, the number of proper search descent paths in a TST is an unknown result as far as we are concerned. However, using the following fact, that

follows from the definition of a proper descent path, we can give an upper bound.

Fact 2. *The number of proper descent paths completely included in the searching path of w is greater than or equal to the number of proper search descent paths for w.*

Lemma 7. *The number of proper descent paths in t completely included in the searching path of t for w corresponds with the search cost in a Patricia trie storing $S(t)$. The cumulative number of descent paths in t for searching every string in $S(t)$ corresponds with the external path length in a Patricia trie storing $S(t)$.*

Proof. Let t be a TST, let p be a Patricia trie and let q be a trie such that $S(t) = S(p) = S(q)$. Recall that a TST is a trie representation. In particular, given the aforementioned equality between the sets of strings, the nodes in t where a proper descent path ends are related to the nodes in q in which two or more pointers go out from them and in turn, these are exactly the nodes in p (p has no other nodes).

Therefore, the number of descent paths followed in t searching for w corresponds with the search cost in p and similarly, the cumulative number of descent paths corresponds with the external path length. □

Useful references on the analysis of Patricia tries are [6, 2, 7]. Finally, from Lemma 6, Fact 2 and Lemma 7 follows that:

Corollary 4. *The number of strings looked up in cb searching for w is upper bounded by the search cost in a Patricia trie storing the same set of strings. The total number of strings looked up in cb searching for every string in $S(cb)$ is upper bounded by the external path length in a Patricia trie storing the same set of strings.*

Indeed, for many common datasets, strings lookups in CaBSTs can be almost eliminated by not only storing the first but the k first relevant digits, where k is a small constant (thus, memory accesses are only due to accessing to the tree nodes).

6 On the number of string lookups in (C)aQSort and (C)aQSel

In this Section, we analyze the number of string lookups in aQSort and aQSel, relating them to aBSTs.

In the following, when referring to comparisons (and related string lookups) in quicksort or quickselect, we consider only the ones due to partitioning, i.e. we do not consider the ones due to selecting the pivot (recall that special care must be taken in doing comparisons to select the pivot, see Section 2.1).

First, we present their extended versions adding one character field, namely CaQSort and CaQSel. Let a be an array of strings.

Definition 5. CaQSort *and* CaQSel *are an extension of respectively aQSort and aQSel, in which a character field is added per each element in a. This additional field stores the first significant character of the string.*

Note that from the definition of the comparison function in Section 2.1, it suffices to store the character value of the largest of the two common prefixes (namely, the one with the predecessor and the one with the successor in the implicit tree).

Now, we characterize the number of string lookups thanks to the relationship of quicksort and quickselect with BSTs. First, we consider (C)aQSort.

Lemma 8. *Let b be an aBST corresponding to the execution e of aQSort on a, let cb be a CaBST corresponding to the execution f of CaQSort on a. The number of string lookups and digit comparisons in e and f correspond respectively with the number of string lookups and digit comparisons in b and cb, searching for every string in $S(a)$.*

Proof. There is an isomorphism between a quicksort execution and a BST built in the same order as pivots are selected. So happens with the augmented versions (the fact of augmenting the BST and quicksort does not change the order in which comparisons are made), and thus, the lemma follows. □

Recall that in the case of quicksort, the number of string lookups coincides with the number of key comparisons because each comparison requires looking up a string. Thus, it is greater or equal than the number of string lookups in (C)aQSort. In the case of multikey quicksort, the number of string lookups coincides with the number of digit comparisons because each digit comparison requires looking up a string. Thus, it is also greater or equal than the number of string lookups in aQSort (but in return, multikey quicksort uses no additional space).

Besides, let t be an TST equivalent to the aBST corresponding to an execution e of aQSort. From Corollary 3 and Lemma 8 follows that:

Corollary 5. *The number of string lookups in e is $L(t) + S(t)$.*

Now, we consider (C)aQSel.

Lemma 9. *Let p be an aBST corresponding to the execution e of aQSel on a, let cp be a CaBST corresponding to the execution f of CaQSel on a. The number of string lookups and digit comparisons in e and f correspond respectively with the number of string lookups and digit comparisons in p and cp, searching for every string in $S(a)$.*

Proof. There is an isomorphism between a quickselect execution and a BST built in the same order as pivots are selected (note that the BST is actually a path). So happens with the augmented versions (the fact of augmenting the BST and quickselect does not change the order in which comparisons are made), and thus, the lemma follows. □

Recall that in the case of quickselect, the number of string lookups coincides with the number of key comparisons because each comparison requires looking up a string. Thus, it is greater or equal than the number of string lookups in (C)aQsel. In the case of multikey quickselect, the number of string lookups coincides with the number of digit comparisons because each comparison requires looking up a string. However, it cannot be directly related to the number of string lookups in aQSel. That is, the sequence of pivot selections in multikey quickselect form a path in the TST that would be isomorphic to sorting the array, whilst in the case of aQSel, the concatenation of pivots is never a path (because at least the special end character follows after the root node).

Besides, from Corollary 3 and Lemma 9, it follows that:

Corollary 6. *The number of string lookups in e is $\sum_{w \in S(a)} (R(t, w) + 1) = n + \sum_{w \in S(a)} R(t, w)$.*

It does not seem straightforward to obtain a close formula for the number of string accesses in (C)aQSel.

7 Conclusions and further work

Comparison-based data structures and algorithms can be augmented so that no redundant character comparisons are made while keeping the rest of combinatorial properties. We have noted that actually, some of the comparisons can be made without looking up any string. This is relevant from a performance perspective because saving string lookups saves random memory accesses and they are not handled efficiently by modern memory hierarchies.

In this paper, we have characterized so-augmented BSTs from the number of string lookups perspective (which we have called aBSTs). Besides, we have also proposed and characterized a variant (which we have called CaBSTs) that stores for every string the value of the first relevant character. In this way, string lookups due to binary searching when determining the next value for a character position, are avoided. Both characterizations are based on their relationship with TSTs, which are a kind of trie. Specifically, we can obtain the precise number of string lookups in searching in an aBST using known results for TSTs. In the case of CaBSTs, the exact values are unknown, but we have provided an upper bound, relating TSTs in turn with Patricia tries.

Finally, we have characterized so-augmented comparison-based algorithms, namely quicksort and quickselect (which we have called respectively (C)aQSort and (C)aQSel) relating them to (C)aBSTs (which are in turn related to TSTs). However, the behavior of (C)aQsel, not only on the number of string

lookups but on the number of digit comparisons, cannot be transposed to any known property on TSTs. That is, it remains as an open problem.

Also interesting would be to put this knowledge into practice. First, a thorough experimental analysis should compare the performance of the variants proposed in this paper against existing ones. In particular, the parallel setting should be taken into account, and so, continuing the work in [10]. Additionally, data structures libraries could be enhanced using this approach. For instance, the dictionary classes in the standard library of the C++ programming language are typically implemented with balanced BSTs. These could be augmented so that combinatorial properties are kept and fast string comparisons are offered. The latter is work in progress.

Acknowledgments

I would like to thank Jordi Petit, Salvador Roura, Amalia Duch and Conrado Martínez for their useful comments.

References

[1] J. L. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *SODA '97: Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, pages 360–369, Philadelphia, PA, USA, 1997. Society for Industrial and Applied Mathematics.

[2] Jérémie Bourdon. Size and path length of patricia tries: dynamical sources context. *Random Struct. Algorithms*, 19(3-4):289–315, 2001.

[3] G. S. Brodal and R. Fagerberg. Cache-oblivious string dictionaries. In *SODA'06: Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms*, 2006.

[4] Julien Clément, Philippe Flajolet, and Brigitte Vallée. Dynamical sources in information theory: A general analysis of trie structures. *Algorithmica*, 29(1):307–369, 2001.

[5] P. Crescenzi, R. Grossi, and G. F. Italiano. Search data structures for skewed strings. In *Experimental and Efficient Algorithms, Second International Workshop, WEA 2003, Ascona, Switzerland, May 26-28, 2003, Proceedings*, volume 2647 of *Lecture Notes in Computer Science*, pages 81–96. Springer, 2003.

[6] Luc Devroye. A study of trie-like structures under the density model. *The Annals of Applied Probability*, 2(2):402–434, 1992.

[7] Luc Devroye. Universal asymptotics for random tries and PATRICIA trees. *Algorithmica*, 42, 2005.

[8] P. Ferragina and R. Grossi. The string B-tree: a new data structure for string search in external memory and its applications. *J. ACM*, 46(2):236–280, 1999.

[9] G. Franceschini and R. Grossi. A general technique for managing strings in comparison-driven data structures. In *Proceedings of the 31st International Colloquium on Automata, Languages and Programming (ICALP)*, 2004.

[10] L. Frias and J. Petit. Combining digital access and parallel partition for quicksort and quickselect. In *IWMSE '09: Proceedings of the 2nd international workshop on Multicore software engineering*, New York, NY, USA, 2009. ACM. To appear.

[11] R. Grossi and G. F. Italiano. Efficient techniques for maintaining multidimensional keys in linked data structures. In *ICALP '99: Proceedings of the 26th International Colloquium on Automata, Languages and Programming*, pages 372–381, London, UK, 1999. Springer-Verlag.

[12] D. R. Morrison. PATRICIA: A practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, 1968.

[13] S. Roura. Digital access to comparison-based tree data structures and algorithms. *J. Algorithms*, 40(1):1–23, 2001.