Digital Access to Comparison-Based Tree Data Structures and Algorithms

Salvador Roura

Report LSI–99–26–R

Digital Access to Comparison-Based Tree Data Structures and Algorithms

Salvador Roura *

June 25, 1999

Abstract

This paper presents a simple method to build tree data structures which achieve just $\mathcal{O}(\log N)$ visited nodes and $\mathcal{O}(D)$ compared digits (bits or bytes) per search or update, where N is the number of keys and D is the length of the keys, irrespectively of the order of the updates and of the digital representation of the keys. The additional space required by the method is asymptotically dismissable compared to the space of keys and pointers, and is easily updated on line. The method applies to fixed-length base-2 keys and to variable-length string keys as well, and permits to save space for common prefixes. The same ideas can be applied to the sorting problem, achieving algorithms with the best properties of quicksort/mergesort and radixsort together.

1 Introduction

There are two general frameworks to build a tree data structure with a given set of keys. Within the first framework, we consider keys as units and perform comparisons between them as a whole. Given two keys `x` and `y`, we are not concerned about the particular value of their i-th bit; we are only interested on whether `x` is smaller, equal, or larger than `y`. A direct application of this idea produces the binary search tree (BST, for short) data structure, as well as several variants of balanced binary search trees. We call the trees of this kind *comparison-based* search trees.

In the second approach to the construction of tree data structures, we use the internal representation of the keys to guide the searches, instead of full key comparisons. Loosely speaking, at the i-th level of the tree, we either go left or right depending on whether the i-th bit is 0 or 1, respectively. Two of the most

*Departament de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya, E-08028 Barcelona, Catalonia, Spain. E-mail:roura@lsi.upc.es

important variants of trees built under this approach are probably tries [4] and patricia tries [10]. We call the trees of this kind *digit-based* search trees.

The two approaches for the construction of tree data structures differ intrinsically, and each one has its own interesting properties and potential drawbacks. Let N denote the number of keys in the tree, and let D denote the number of digits (bits, for the moment) of each key. (In order to easily compare the several kinds of trees, we do not consider variable-length keys, nor bases larger than 2, in this introductory section.) We assume the common situation where the number of keys N is much smaller than 2^D, the cardinality of the set of possible D-bit keys, but much larger than D, the length of the keys. For example, consider a set of 50000 keys with 128 bits each (we will repetitively use this example through this section). Then we certainly have $D = 128 << N = 50000 << 2^{128}$. More formally, if we think of N as a function of D, then we assume $N = o(2^D)$ and $D = o(N)$.

Let us contrast comparison-based search trees and digit-guided search trees. On the one hand, comparison-based search trees require some care to avoid the worst case $\Theta(N)$ for the average number of visited nodes per search or update (insertion or deletion). This can be achieved by explicitly balancing the tree (AVL-trees [1], red-black-trees [6], etc.) or by randomising it [9]. All these strategies guarantee that the (worst-case or expected) number of visited nodes per search or update is $\Theta(\log N)$, irrespectively of the order of the updates. For our example above, the difference is roughly between $50000/2 = 25000$ and just $\log_2 50000 \simeq 16$ visited nodes per search or update. The second potential drawback of comparison-based search trees appears with long keys whose digital representation requires many bits (or bytes). Since the comparisons are between full keys, if we have long common prefixes we may need to perform almost D bit comparisons (or, say, almost $\lceil D/8 \rceil$ 8-bit byte comparisons) per node, which amounts to $\Theta(D \log N)$ bit/byte comparisons per search or update on the average.

The principal property of digit-guided search trees is that they only require $\mathcal{O}(D)$ bit comparisons per search or update, irrespectively of the digits of the keys, of the shape of the tree, and of the location of the searched or updated key in the tree. In terms of our example above, the difference with balanced trees is roughly between $16 \cdot 128 = 2048$ and only 128 bit comparisons. Unfortunately, for every particular digit-based search tree, there are sets of keys with long common prefixes that induce an internal path length $\Theta(N \cdot D)$ —much more than the internal path length $\Theta(N \log N)$ of balanced BSTs. The simplest example of this phenomenon is a set of keys with exactly one key starting with the prefix 1, another key starting with the prefix 01, another one starting with the prefix 001, ..., and most of the keys starting with a long common prefix $0 \cdots 0$ and differing only in the last (approximately $\log_2 N$) bits (see the trie in Figure 1). That trie (and any patricia trie with the same set of keys) requires, on the average, approximately D visited nodes per successful search. So, in our example above with $D = 128$ and $N = 50000$, the difference is roughly between

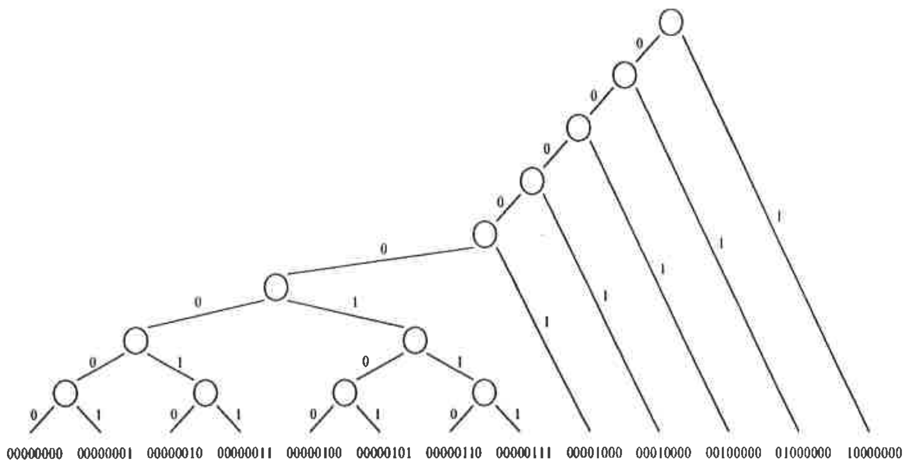


Figure 1: Trie for a bad-case set of digits for the keys.

128 and just 16 visited nodes per successful search.

This paper presents a simple strategy to achieve the best properties of both balanced search trees and digit-guided search trees together, that is,

a) $\mathcal{O}(\log N)$ visited nodes per search or update, irrespectively of the order of the insertions;

b) $\mathcal{O}(D)$ bit comparisons per search or update for every set of digits for the keys.

We call such strategy *digital access* to BSTs, where by "digital" it is meant that each bit needs to be compared just once (or that each byte needs to be compared at most a constant number of times).

This method will be explained in detail in later sections. But let us first compare it against some classic data structures. Figure 2 presents a summary of the main tree data structures, each one with its average cost per successful search or deletion, the average taken over all the keys in the tree, the cost measured as the number of visited nodes and as the number of bit comparisons. (For the sake of clarity, the discussion about unsuccessful searches and insertions is deferred to other sections, where we will see that digital access to binary search trees also competes pretty well.) The average costs labeled with the word "always" do not depend on the insertion order nor the digits of the keys. Those labeled with "insertion order" or "digits" achieve the given cost with a worst-case insertion order (say, increasing order), or with a worst-case digits for the keys (like, for instance, the ones presented in Figure 1), respectively.

Data structure	Visited nodes	Bit comparisons
BST with classic access	$\Theta(N)$ {insertion order}	$\Theta(D \cdot N)$ {digits, insertion order}
Balanced BST with classic access	$\Theta(\log N)$ {always}	$\Theta(D \log N)$ {digits}
Trie / patricia trie	$\Theta(D)$ {digits}	$\Theta(D)$ {always}
Trie / patricia trie with random keys	$\Theta(\log N)$ {with hight probability}	$\Theta(D)$ {always}
BST with digital access	$\Theta(N)$ {insertion order}	$\Theta(D)$ {always}
Balanced BST with digital access	$\Theta(\log N)$ {always}	$\Theta(D)$ {always}

Figure 2: Main binary tree data structures and their worst expected successful search cost.

The number of visited nodes for digit-guided search trees with random keys is a double average, over all the keys in the tree, and also over all the possible sets of digits for the keys. The average value $\Theta(\log N)$ could become $\Theta(D)$ in a very unlikely situation. On the other hand, the average depth of the nodes in an explicitly balanced BST is always $\Theta(\log N)$, but if we choose to balance the tree by randomising it, then that average is highly probable, but not guaranteed (in that case we have again a double average, over all the keys in the tree, as well as over all the possible random choices of the randomised insertion and deletion algorithms).

It is worth emphasising the crucial difference between the highly probable $\Theta(\log N)$ cost of randomised BSTs and that of digit-guided search trees with random keys. For the former we depend upon a (presumably sound) random-number generator, while for the later we require the keys to be truly random. In practical situations where it is not sure that the keys are random (or where it is known that they are not), there may be no reasonable guarantees that a search or update in a digit-guided search tree will not visit significantly more than $\Theta(\log N)$ nodes.

To end this introductory section, let us briefly relate the results presented in this paper with other works. The suffix array [8] is probably the first data structure that combines the best of the two worlds, comparison-based and digit-guided, to perform efficient string searches in static arrays. One major difference of our results with respect to suffix arrays is that our method deals with dynamic sets of keys. The same comparison applies to the BST version of suffix arrays, suffix binary search trees [7], which also deals with static sets of keys. It is noteworthy that our method, when applied to BSTs, is mainly a suffix binary search tree provided with a mechanism to update the tree (insertions and deletions at the leaves of the tree, and rotations). We will see that this apparently small improvement has a dramatic effect: it allows us to directly adapt most balancing strategies (those that can be set in terms of rotations, insertions and deletions at the leaves) to the digital framework. In other words, for every one of those (classic) comparison-based algorithms, we literally achieve a new digital algorithm which captures the best properties of the classic counterpart. Another interesting data structure with connections to our work is the string B-tree [3], a combination of suffix array and B-tree, the popular external memory data structure. Finally, our results have some similarities with those in [5], though both papers are independent. There, the authors provide a general framework to combine string techniques with a variety of linked data structures. By contrast, the emphasis of our paper is set on search trees and sorting (quicksort and mergesort).

The next sections are organised as follows. Section 2 covers digital searches in BSTs, both successful and unsuccessful. Section 3 deals with digital updates in BSTs: insertions, deletions and rotations. Section 4 shows how to extend the fixed-length base-2 algorithms in Sections 2 and 3 to variable-length keys with bases larger than 2. In Section 5 we will see that the same general technique can be adapted to some of the most important sorting methods, providing excellent cost bounds for the number of memory exchanges and bit/byte comparisons. Section 6 ends the paper with some final comments and conclusions.

2 Digital searches in binary search trees

In this section we will see how to perform searches in binary search trees (balanced or not) comparing each bit at most once. Thus, when we use this strategy to search in any kind of balanced BST, we achieve an optimal tree data structure regarding the number of visited nodes and bit comparisons (recall Figure 2). For plain (not balanced) BSTs, each digital search or update requires just $\mathcal{O}(D)$ bit comparisons, even under the worst-case insertion order and/or under the worst-case digits for the keys.

In order to perform digital accesses into a given binary search tree, we provide it with some additional information at every node, as follows. For any key $\mathbf{y}$ at depth d, let $[\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_D]$ be the digital representation of $\mathbf{y}$, $\mathbf{y}_1$ being the main digit (bit, for the moment), and let $\mathcal{A}(\mathbf{y}) = \{\mathbf{z}^{(0)}, \mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(d)}\}$ be the set of *ancestors* of $\mathbf{y}$, i.e., the set of keys in the path from the root of the tree to the node where $\mathbf{y}$ is located ($\mathbf{y}$ excluded). Then, in the same node with $\mathbf{y}$, we store a field with the length of the longest common prefix between $\mathbf{y}$ and the keys in $\mathcal{A}(\mathbf{y})$. More precisely, we store the smallest index i such that $[\mathbf{y}_1, \ldots, \mathbf{y}_i] \neq [\mathbf{z}^{(j)}_1, \ldots, \mathbf{z}^{(j)}_i]$ for every $0 \leq j \leq d$. We call $\mathbf{d}(\mathbf{y})$ such first position

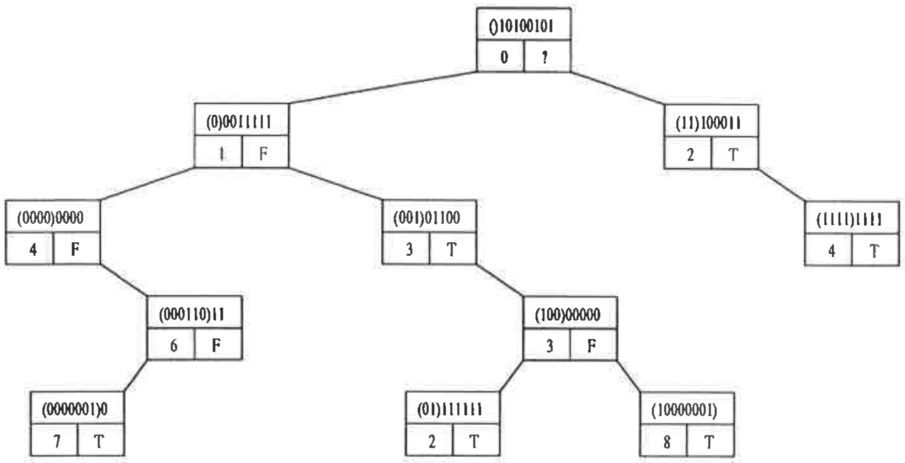


Figure 3: A binary search tree with fixed-length base-2 keys and the information to perform digital accesses.

where the prefixes of all the keys in $\mathcal{A}(\mathtt{y}) \cup \{\mathtt{y}\}$ differ.

We also store together with $\mathtt{y}$ information to allow us to discern which of the keys in $\mathcal{A}(\mathtt{y})$ shares the longest prefix with $\mathtt{y}$. We define the *predecessor* of $\mathtt{y}$ as the largest of the keys in $\mathcal{A}(\mathtt{y})$ that are smaller than $\mathtt{y}$, and the *successor* of $\mathtt{y}$ as the smallest of the keys in $\mathcal{A}(\mathtt{y})$ that are larger than $\mathtt{y}$. Note that $\mathtt{p}/\mathtt{s}$ is not well defined when $\mathtt{y}$ is smaller/larger than all the keys in $\mathcal{A}(\mathtt{y})$. Hence, we have four possible cases. First, assume that both $\mathtt{p}$ and $\mathtt{s}$ exist. If (for example) the common prefix between $\mathtt{p}$ and $\mathtt{y}$ is longer than that between $\mathtt{s}$ and $\mathtt{y}$, then no other key can share a longer prefix with $\mathtt{y}$; it is only possible that some keys smaller than $\mathtt{p}$ share with $\mathtt{y}$ exactly the same prefix, but not a longer one. Thus, it is enough to store with $\mathtt{y}$ a boolean $\mathtt{is\_p}(\mathtt{y})$, which will be true if and only if the predecessor of $\mathtt{y}$ is one of the keys with the longest common prefix with $\mathtt{y}$ (otherwise the successor of $\mathtt{y}$ will be one of such keys).

We have left some possibilities: If $\mathtt{y}$ is larger than all the keys in $\mathcal{A}(\mathtt{y})$ (this happens if $\mathtt{y}$ is located in the right spine of the tree excluded the root), then $\mathtt{s}$ is not well defined, but $\mathtt{y}$ shares the longest prefix with its predecessor $\mathtt{p}$ (and perhaps some other key smaller than $\mathtt{p}$), and hence we set $\mathtt{is\_p}(\mathtt{y})$ to true. Similarly, if $\mathtt{y}$ is smaller than all the keys in $\mathcal{A}(\mathtt{y})$, then $\mathtt{y}$ shares the longest prefix with its successor $\mathtt{s}$, and we must set $\mathtt{is\_p}(\mathtt{y})$ to false. If $\mathtt{y}$ is stored at the root of the tree, then $\mathtt{y}$ is the only key whose set of ancestors is empty, and it has no predecessor nor successor. In that case we set $\mathtt{d}(\mathtt{y})$ to 0 (as if $\mathtt{y}$ differed in a 0-th virtual bit with a non-existent key above it), and set $\mathtt{is\_p}(\mathtt{y})$ to either true or false, since it does not matter.

Figure 3 shows an example of binary search tree (in this case unbalanced) with two additional fields at each node with the information described above. The keys are shown in binary notation (8 bits). At each node, the left field stores the index d(y), and the right field stores the boolean is_p(y). The bits between parentheses do not need to be stored explicitly (we will come back to this point at the end of the section). Note that *the shape of the tree is completely independent of the set of digits for the keys*, and hence could be any other.

Now, let us describe how to digitally search for a given key x into a binary search tree that stores the information described above. The search follows exactly the same path as that of a standard search, but we perform bit comparisons instead of full key comparisons. Let c be the key stored at the root of the current subtree during the search, and let p and s be the predecessor and the successor of c, respectively (they are stored somewhere in the path up to the root, but we do not currently know their value). Then p is also the largest of the keys in $\mathcal{A}$(c) smaller than x, and s is also the smallest of the keys in $\mathcal{A}$(c) larger than x. We say that p is the current predecessor of x and s is the current successor of x. Let j be the index of the first bit where x and p differ, and let k be the index of the first bit where x and s differ. Clearly, j and k cannot be equal. To decide whether x is smaller, equal or larger than c, we use two variables xp and xs following this search invariant: At the current stage of the search, if j is larger than k, then we have $\mathtt{xp} = j$ and $\mathtt{xs} \leq k$. If j is smaller than k, then we have $\mathtt{xp} \leq j$ and $\mathtt{xs} = k$. Moreover, the first $\max\{\mathtt{xp}, \mathtt{xs}\}$ bits of x have been compared so far *just once*, the rest of bits not yet.

Figure 4 shows the five possible cases of a digital search when is_p(c) is true (when is_p(c) is false we have five symmetrical cases). Since is_p(c) is true, d(c) is the position where c and p first differ. In that figure, the position where c and s first differ is denoted by q (it is not stored anywhere, and at the current stage of the search we only know that it is smaller than d(c)). The current value of xp is denoted by r when it is not equal to d(c). The positions marked with ******, ######, and xxxxxx denote (perhaps empty) common subsequences of bits. The positions marked with ?????? are common subsequences that need to be compared to continue the search, since we do not have information about them yet. By induction hypothesis, before the comparison of x against c, the variables xp and xs fulfill the search invariant, which must also hold after the comparison. To the right of every case we see the conditions for xp and xs (denoted xp′ and xs′ after the comparison) according to the search invariant.

In the first four cases of Figure 4, the key x has a 0-bit at the q-th position, and hence $\mathtt{xs} \leq q < \mathtt{xp}$. If $\mathtt{xp} < \mathtt{d(c)}$ (Case 1), then x is larger than c, and the search must follow in the subtree to the right of the current node, with c as the new current predecessor of x. To keep the search invariant true, we need to do nothing, because xp is already the index of the first bit where x and c differ, and the current successor of x is still s and hence xs has a correct value if the invariant was true. When $\mathtt{xp} > \mathtt{d(c)}$ (Case 2), we have $\mathtt{x} < \mathtt{c}$, and c will be the

(Case 1) $q < r <$ d(c)
p = ******0######0xxxxxx0 ...
c = ******0######0xxxxxx1 ...
x = ******0######1 ...
s = ******1 ...

$$\left\{\begin{matrix} \mathtt{xp} = r \\ \mathtt{xs} \le q \end{matrix}\right\} \Longrightarrow \left\{\begin{matrix} \mathtt{xp}' = r \\ \mathtt{xs}' \le q \end{matrix}\right\}$$

(Case 2) $q <$ d(c) $< r$
p = ******0######0xxxxxx0 ...
x = ******0######0xxxxxx1 ...
c = ******0######1 ...
s = ******1 ...

$$\left\{\begin{matrix} \mathtt{xp} = r \\ \mathtt{xs} \le q \end{matrix}\right\} \Longrightarrow \left\{\begin{matrix} \mathtt{xp}' = r \\ \mathtt{xs}' \le \mathtt{d(c)} \end{matrix}\right\}$$

(Case 3) $q <$ d(c) $<$ i
p = ******0######0 ...
c = ******0######1??????0 ...
x = ******0######1??????1 ...
s = ******1 ...

$$\left\{\begin{matrix} \mathtt{xp} = \mathtt{d(c)} \\ \mathtt{xs} \le q \end{matrix}\right\} \Longrightarrow \left\{\begin{matrix} \mathtt{xp}' = \mathtt{i} \\ \mathtt{xs}' \le q \end{matrix}\right\}$$

(Case 4) $q <$ d(c) $<$ i
p = ******0######0 ...
x = ******0######1??????0 ...
c = ******0######1??????1 ...
s = ******1 ...

$$\left\{\begin{matrix} \mathtt{xp} = \mathtt{d(c)} \\ \mathtt{xs} \le q \end{matrix}\right\} \Longrightarrow \left\{\begin{matrix} \mathtt{xp}' \le \mathtt{d(c)} \\ \mathtt{xs}' = \mathtt{i} \end{matrix}\right\}$$

(Case 5) $q <$ d(c)
p = ******0######0 ...
c = ******0######1 ...
x = ******1xxxxxxxxxxxxx0 ...
s = ******1xxxxxxxxxxxxx1 ...
$q < u$

$$\left\{\begin{matrix} \mathtt{xp} \le q \\ \mathtt{xs} = u \end{matrix}\right\} \Longrightarrow \left\{\begin{matrix} \mathtt{xp}' \le q \\ \mathtt{xs}' = u \end{matrix}\right\}$$

Figure 4: Possible cases for a digital search when is_p(c) is true.

new current successor of x. Again, the search invariant remains true without any changes in the values of xp or xs, since the current predecessor of x is still p, and xs $\le p$ implies xs $\le$ d(c).

Let us consider now Cases 3 and 4. Here, xp = d(c), the first d(c) bits of x and c are equal, and we have not tested enough bits from x to find its first difference with c (if any). Therefore, we keep comparing bits from x and c *starting at the* (d(c) + 1)*-th position* (not from the beginning of the keys), until we reach a difference or the end of the keys. In the latter case x and c are equal, and the search for x in the tree ends successfully. Assume that, instead, we find a position i where c has a 0-bit and x has a 1-bit (Case 3). Then x is larger than c, and to keep the search invariant true we must update xp to equal i and not modify xs, because the new current predecessor of x is c and the current

```
typedef long Word;
#define D 32
typedef int Bit;
#define bit(X, I) (((X) >> (D-(I))) & 1)
typedef int Diff;
typedef int Boolean;
#define FALSE 0
#define TRUE  1
typedef struct Node *Tree;
typedef struct { Word w; Tree l, r; Diff d; Boolean is_p; } Node;
typedef enum { EMPTY, EQUAL, SMALLER, GREATER } Comp;
```

Figure 5: Type definitions in *C*.

successor of `x` is still `s`. If, on the contrary, we find a position `i` where `c` has a 1-bit and `x` has a 0-bit (Case 4), then `x` is smaller than `c`. In this case we can keep the value of `xp` because `xp` = `d(c)` implies `xp` ≤ `d(c)`, but we must set `xs` equal to `i`.

Finally, we have left considering Case 5. Here, `x` has a 1-bit at its q-th position, and so `x` is larger than `c`. The position u where `x` and `s` first differ has no relation at all with `d(c)`, and could be smaller, equal or larger than it. The current values of `xp` and `xs` fulfill the search invariant without any changes, since the current successor of `x` is still `s`, and `c`, the new current predecessor of `x`, has the same common prefix with `x` than `p`.

We have not defined yet how to start the search. Initially, no bits of `x` have been compared, so it is easy to see that `xp` = `xs` = 0 are the right initial values. We can imagine that we have compared `x` against non-existent keys above the root of the tree, and that the last compared bit of `x` has been a virtual 0-th bit. After the first comparison, if, say, the key at the root is larger than `x`, then `xs` is still zero but `xp` is updated conveniently, and we keep traversing the right spine of the tree until we find the first key larger than `x`. From that moment both `xp` and `xs` are strictly positive, since the current predecessor and successor of `x` are always well defined.

Let us use the observations above to write a *C* function to digitally search for a key into a binary search tree. Figure 5 presents most of the *C* type definitions that we will use for the rest of the paper. As it is shown, we assume that a key is a fixed-length word of 32 bits. Given a word `X` and an index `I` such that 1 ≤ `I` ≤ 32, the macro `bit(X, I)` returns the `I`-th bit of `X`. Each extracted bit is stored in a full integer for simplicity (its value will always be either 0 or 1). A tree is identified with a pointer to its root node, as usual. Each node has, apart from its key `w` and two pointers `l` and `r` to the left and right children, two

```
Comp Compare(Word x, Tree t, Diff *xp, Diff *xs)
{ Word c; Diff i; Bit bx, bc;
  if (t == NULL) return EMPTY;
  if (t->is_p) { if (*xp < t->d) return GREATER;
                 if (*xp > t->d) return SMALLER;
               }
          else { if (*xs < t->d) return SMALLER;
                 if (*xs > t->d) return GREATER;
               }
  for (c = t->w, i = t->d; ++i <= D; )
     if ((bx = bit(x, i)) != (bc = bit(c, i)))
       if (bx < bc) { *xs = i; return SMALLER; }
               else { *xp = i; return GREATER; }
  return EQUAL;
}
```

Figure 6: The function `Compare`.

additional fields storing `d(w)` and `is_p(w)`, respectively. For the sake of clarity, we avoid including other fields with the information related to the key, or for balancing the tree. Using an integer for the type `Diff` of the field `d` allows us to codify any value in the range $[0, 2^{16} - 1]$, which would suffice for most practical situations with longer keys. Bits and booleans are defined as the same basic type and always store either 0 or 1, but it is preferable to distinguish them because of their conceptual difference. The last type `Comp` is explained below.

Figure 6 shows the function `Compare`, which captures all the cases in Figure 4. Given the searched key `x`, the current subtree `t` (a pointer to its root), and pointers to the variables `xp` and `xs`, `Compare` returns the result of the comparison of `x` against the key at the root of `t` (if any), updating conveniently `xp` or `xs` when necessary. The four possible results are codified with the type `Comp`. The easiest case happens when `t` is an empty tree: the function simply returns `EMPTY`. We do not handle this case apart from non-trivial ones for aesthetic reasons (see Figure 7). Assume now that the tree is not empty and that the boolean `is_p(c)` for the current key `c` is true. Then we check if `xp` is smaller than `d(c)`. If it is, we are in Case 1 or Case 5 of Figure 4, `x` is greater than `c`, and we do not need to update `xp` nor `xs`. If, otherwise, `xp` is larger than `d(c)`, then we are in Case 2, `x` is smaller than `c`, and we do not have to update any variables, either. Finally, when `xp` equals `d(c)` we are in Case 3 or Case 4, and we search the first position `i` where `x` and `c` differ, starting at the $(\texttt{d(c)} + 1)$-th bit. If we find such difference, we update `xs` or `xp` and return `SMALLER` or `GREATER`. If not, `x` and `c` are equal, and we return so. When `is_p(c)` is false we perform similar actions.

```
Boolean R_Classic_search(Tree t, Word x)
{ if (t == NULL) return FALSE;
  if (x == t->w) return TRUE;
  if (x < t->w)  return R_Classic_search(t->l, x);
    else         return R_Classic_search(t->r, x);
}

Boolean R_Search(Tree t, Word x, Diff xp, Diff xs)
{ switch (Compare(x, t, &xp, &xs))
    { case EMPTY  : return FALSE;
      case EQUAL  : return TRUE;
      case SMALLER: return R_Search(t->l, x, xp, xs);
      case GREATER: return R_Search(t->r, x, xp, xs);
    }
}

Boolean Search(Tree t, Word x)
{ return R_Search(t, x, 0, 0);
}
```

Figure 7: Classic and digital searches in binary search trees.

Making use of the function `Compare` allows us to translate most comparison-based procedures over binary search trees into their digital counterparts. For instance, in Figure 7 we find the recursive function `R_Classic_search`, implementing the classic recursive search for a given key `x` into a given binary search tree `t`. The function only returns `TRUE` or `FALSE`, depending on whether `x` is in the tree or not. In that figure, there is also the recursive digital version `R_Search`. As it can be seen, the digital search is not significantly more complicated than the classic (comparison-based) search. In fact, the translation between one and the other is trivial, and even transparent to programmers if the function `Compare` is used. The function `Search` only performs the first call to `R_Search` with the appropriate parameters (the same as those for `R_Classic_search` plus two additional 0's).

We end the section with a few comments about the cost, in terms of time and space, of digital searches in BSTs. The number of visited nodes depends on the shape of the tree. Thus, if we assume the tree to be balanced, then the (expected) average number of visited nodes per search, either successful or unsuccessful, is $\Theta(\log N)$, and could become just $\log_2 N + o(\log N)$ depending on the balancing method. Regarding the number of compared bits, in the case of successful searches we compare each bit exactly once, which means D compared bits per search. For unsuccessful searches, we compare as many bits as necessary to distinguish the search key from the rest of keys in the tree, so in the worst case we need to compare D bits. The exact expected number of compared bits depends on the digits of the keys (or on their probability distribution), and sometimes is quite difficult to compute. Fortunately, a digital search in a binary search tree compares exactly the same bits as a search in a trie, which means that we can directly use all the previous mathematical analysis for tries. For instance, when the keys are random, it is known that the expected number of compared bits per unsuccessful search in a trie (and hence during a digital unsuccessful search in a BST) is just $\log_2 N + o(\log N)$. For other probability distributions for the digits, any cost in the range $[\log_2 N \ldots D]$ is possible.

Regarding the space required for the fields `d` and `is_p`, for the former we need $\lceil \log_2 D \rceil$ bits, while for the latter one bit suffices. Note that this is much less than the D bits of each key, though for simplicity we decided to store `d` and `is_p` in a couple of full integers.

On the other hand, if we have non-random keys with long common prefixes, then we can save space by storing any common prefix just once. The way is simple: for every key `y`, we only store the suffix $[\mathtt{y}_{\mathtt{d}(\mathtt{y})+1}, \ldots, \mathtt{y}_D]$. The rest of bits $[\mathtt{y}_1, \ldots, \mathtt{y}_{\mathtt{d}(\mathtt{y})}]$ are implicitly defined by the fields `d` and `is_p` and the keys in $\mathcal{A}(\mathtt{y})$, and can be trivially computed in the way down the tree during a search. For instance, it would be enough to store the last bit of the key `00000010` in the tree in Figure 3. Any key `y` stored at that node, with $\mathtt{d}(\mathtt{y}) = 7$ and $\mathtt{is\_p}(\mathtt{y}) = \mathtt{TRUE}$ has necessarily to start with the prefix `0000001` (the first six bits are common to `00000000`, the seventh bit is `1` because $\mathtt{is\_p}(\mathtt{y}) = \mathtt{TRUE}$). As extreme examples, the key `10100101` at the root of the tree in Figure 3 has to be stored in full, while no bits of the key `10000001` need to be stored explicitly. It is interesting to observe that the space savings of this strategy are exactly the same as those achieved in a trie when replacing full keys at leaves with the suffixes that are not computable from the trie structure.

3 Digital updates in binary search trees

There are just three fundamental operations required to change the shape of a given binary search tree: inserting a key in a new leaf of the tree, deleting a key that is stored in a leaf, and rotating a node with its parent. Moreover, most (all, to the best of the author's knowledge) balancing strategies can be written in terms of the three fundamental operations above. For instance, an insertion in a balanced BST is usually equivalent to inserting the key in a new leaf, rotating that node upwards as many times as necessary, and perhaps performing a few additional rotations. During this process, some fields with information for the balancing strategy may require to be updated, but we avoid dealing with details that depend on the chosen balancing strategy, and focus our attention on how digitally update a BST while keeping consistent the information to perform

```
Tree New_node(Word x, Diff xp, Diff xs)
{ Tree n = (Tree)malloc(sizeof(*n));
  n->w = x; n->l = n->r = NULL;
  if (xp > xs) { n->d = xp; n->is_p = TRUE; }
          else { n->d = xs; n->is_p = FALSE; }
  return n;
}

Tree R_Insert(Tree t, Word x, Boolean *nw, Diff xp, Diff xs)
{ switch (Kcompare(x, t, &xp, &xs))
    { case EMPTY  : *nw = TRUE;  return New_node(x, xp, xs);
      case EQUAL  : *nw = FALSE; return t;
      case SMALLER: t->l = R_Insert(t->l, x, nw, xp, xs); return t;
      case GREATER: t->r = R_Insert(t->r, x, nw, xp, xs); return t;
    }
}

Tree Insert(Tree t, Word x, Boolean *nw)
{ return R_Insert(t, x, nw, 0, 0);
}
```

Figure 8: Digital insertions in binary search trees.

digital accesses.

We use the following observation: for every key **y** in the tree, by definition of $\mathcal{A}$(**y**), the values of **d(y)** and **is_p(y)** do not depend at all on the information stored beneath **y**. Therefore, for every update, we only have to consider the possible effects on the fields **d** and **is_p** in the nodes directly involved and those underneath. The first immediate consequence is the digital deletion algorithm for a key stored in a leaf, too simple to include a C function for it in this paper: we first digitally search for it, afterwards we just delete the node where it is stored. No other action must be taken.

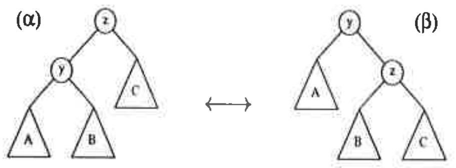
As in the case of digital searches, it is a simple matter to obtain the digital counterpart of the classic insertion algorithm. Figure 8 includes the digital version **R_Insert** of the classic insertion of a key **x** into a new leaf of a given tree **t**. The function returns the tree after the insertion, and updates a boolean ***nw** which indicates if **x** was a new key or not (if not, the tree is not modified). The variables **xp** and **xs** are first used to guide the search. If **x** was not in the tree, then we call the function **New_node**, which fills a new node with **x** as key, empty subtrees as children, and the appropriate values for the fields **d** and **is_p**, which are easily computed from the current values of **xp** and **xs**.

Let us now consider how the contents of the fields `d` and `is_p` must be updated after a rotation. Let `y` and `z` be the keys involved, with `y` < `z`, and let `A`, `B` and `C` be the subtrees below `y` and `z`, where the keys in `A` are smaller than `y`, those in `B` are larger than `y` and smaller than `z`, and those in `C` are larger than `z` (see Figure 9). Assume that we rotate the tree α to get the tree β. As already mentioned, the keys above `y` and `z` cannot be affected by the rotation. Regarding the keys in `B`, they have exactly the same ancestors in both trees, and so are not affected, either. Each key `x` in `A` gains `z` as a new ancestor, which, in principle, could imply updating `d(x)` or `is_p(x)`. However, `z` cannot be the new predecessor of `x` due to `x` < `z`, and `z` cannot be the new successor of `x` due to `x` < `y` < `z`. Hence, no changes are needed in `A`. Similarly, the keys in `C` lose `y` as ancestor, but `y` could neither be the predecessor nor the successor of any of those keys, so no changes in `C` are required. A symmetrical reasoning for the pass from the tree β to the tree α shows that we only may need to update consistently the fields `d` and `is_p` related to the keys `y` and `z`.

Figure 9 shows the five possible situations for a rotation in terms of the digits of `y` and `z`. In that figure, `p` and `s` denote, respectively, the predecessor and successor of `y` in the tree α, and the predecessor and successor of `z` in the tree β. The predecessor and successor of `z` in α are `y` and `s`. The predecessor and successor of `y` in β are `p` and `z`. The first positions where the pairs (`p`, `y`), (`y`, `z`), and (`z`, `s`) differ are denoted q, r and u, respectively. It is important to observe that only Cases 2 and 4 require updating the information for digital accesses, and that the new values for `is_p(y)`, `d(y)`, `is_p(z)`, and `d(z)` can be easily computed from their old values.

Figure 10 includes the C functions implementing the left and right rotations. For instance, when rotating from α to β, we update information if and only if `is_p(z)` is true and `d(y)` < `d(z)`. In that case, it suffices to interchange the values of `d(y)` and `d(z)` (we use a macro `swap` for it), make `is_p(z)` equal to the old value of `is_p(y)`, and set `is_p(y)` to false. When rotating from β to α we perform symmetrical actions.

We finish the section computing the cost of the several update operations. As in the case of searches, the number of visited nodes during an insertion or deletion at a leaf depends on the shape on the tree, an thus on the balancing strategy (if the tree is balanced, it is $\Theta(\log N)$). Concerning bit comparisons, deleting a key requires comparing each of its bits once to find the key, which means D bit comparisons per deletion if the key was in the tree, and at most D bit comparisons if it was not. Similarly, inserting a key requires D bit comparisons if the key was already present, and at most D bit comparisons if the key was new. Again, the previous analyses for the cost of tries tell us that a random deletion of a non-present key or a random insertion of a new key requires only $\log_2 N + o(\log N)$ bit comparisons on the average.



$$\longleftrightarrow$$

```
(Case 1)    q  <  r  <  u
p = ******0######0xxxxxx0 ...
y = ******0######0xxxxxx1 ...
z = ******0######1 ...
s = ******1 ...
```

$$\left\{\begin{array}{c}\texttt{is\_p(y)}\\ \texttt{d(y)=}\,u\\ \texttt{is\_p(z)}\\ \texttt{d(z)=}\,r\end{array}\right\} \longleftrightarrow \left\{\begin{array}{c}\texttt{is\_p(y)}\\ \texttt{d(y)=}\,u\\ \texttt{is\_p(z)}\\ \texttt{d(z)=}\,r\end{array}\right\}$$

```
(Case 2)    q  <  u  <  r
p = ******0######0 ...
y = ******0######1xxxxxx0 ...
z = ******0######1xxxxxx1 ...
s = ******1 ...
```

$$\left\{\begin{array}{c}\texttt{is\_p(y)}\\ \texttt{d(y)=}\,u\\ \texttt{is\_p(z)}\\ \texttt{d(z)=}\,r\end{array}\right\} \longleftrightarrow \left\{\begin{array}{c}\neg\texttt{is\_p(y)}\\ \texttt{d(y)=}\,r\\ \texttt{is\_p(z)}\\ \texttt{d(z)=}\,u\end{array}\right\}$$

```
(Case 3)    r  <  u
p = ******0######0 ...
y = ******0######1 ...
z = ******1xxxxxxxxxxxxx0 ...
s = ******1xxxxxxxxxxxxx1 ...
            r     <     q
```

$$\left\{\begin{array}{c}\texttt{is\_p(y)}\\ \texttt{d(y)=}\,u\\ \neg\texttt{is\_p(z)}\\ \texttt{d(z)=}\,q\end{array}\right\} \longleftrightarrow \left\{\begin{array}{c}\texttt{is\_p(y)}\\ \texttt{d(y)=}\,u\\ \neg\texttt{is\_p(z)}\\ \texttt{d(z)=}\,q\end{array}\right\}$$

```
(Case 4)    u  <  q  <  r
p = ******0 ...
y = ******1######0xxxxxx0 ...
z = ******1######0xxxxxx1 ...
s = ******1######1 ...
```

$$\left\{\begin{array}{c}\neg\texttt{is\_p(y)}\\ \texttt{d(y)=}\,q\\ \texttt{is\_p(z)}\\ \texttt{d(z)=}\,r\end{array}\right\} \longleftrightarrow \left\{\begin{array}{c}\neg\texttt{is\_p(y)}\\ \texttt{d(y)=}\,r\\ \neg\texttt{is\_p(z)}\\ \texttt{d(z)=}\,q\end{array}\right\}$$

```
(Case 5)    u  <  r  <  q
p = ******0 ...
y = ******1######0 ...
z = ******1######1xxxxxx0 ...
s = ******1######1xxxxxx1 ...
```

$$\left\{\begin{array}{c}\neg\texttt{is\_p(y)}\\ \texttt{d(y)=}\,r\\ \neg\texttt{is\_p(z)}\\ \texttt{d(z)=}\,q\end{array}\right\} \longleftrightarrow \left\{\begin{array}{c}\neg\texttt{is\_p(y)}\\ \texttt{d(y)=}\,r\\ \neg\texttt{is\_p(z)}\\ \texttt{d(z)=}\,q\end{array}\right\}$$

Figure 9: Possible cases for a rotation in a BST with the information to perform digital accesses, in terms of the digits of the involved keys.

```
#define swap(A, B) { Diff t = A; A = B; B = t; }

Tree Rotate_left(Tree t)
{ Tree r = t->r;
  if ((r->is_p) && (t->d < r->d))
    { swap(t->d, r->d); r->is_p = t->is_p; t->is_p = FALSE; }
  t->r = r->l; r->l = t;
  return r;
}

Tree Rotate_right(Tree t)
{ Tree l = t->l;
  if ((!l->is_p) && (t->d < l->d))
    { swap(t->d, l->d); l->is_p = t->is_p; t->is_p = TRUE; }
  t->l = l->r; l->r = t;
  return l;
}
```

Figure 10: The `Rotate_left` and `Rotate_right` functions.

Regarding rotations, they are usually performed before or after an insertion or deletion. The number of visited nodes is proportional to the number of rotations, and is constant (either in the worst case or on the average) for typical balancing strategies. No bits from the involved keys are compared during a rotation. When common prefixes are stored just once to save storage space, then all the cases in Figure 9 except Case 3 require moving the common subsequences of bits marked with ######, plus the final bit that distinguishes the involved keys, from the old parent to the new parent. This could have $\Theta(D)$ cost in the worst case, so for every application we should decide if the space savings compensate the time overhead for rotations. For instance, if the keys are random, then on the one hand the common prefixes are very short, and on the other hand digital insertions require just $\Theta(\log N)$ visited nodes and $\Theta(\log N)$ compared digits with balanced BSTs, which would grow to $\Theta(D)$ because rotations (or equivalents) are needed to balance the tree. In such a situation we should probably decide to store the keys in full.

4 Extensions to larger bases

In this section we consider variable-length keys in a base R (the radix) larger than 2. Without loss of generality, we think of a key as a C string, that is, an array of characters finished with `'\0'`, which is a character smaller than any

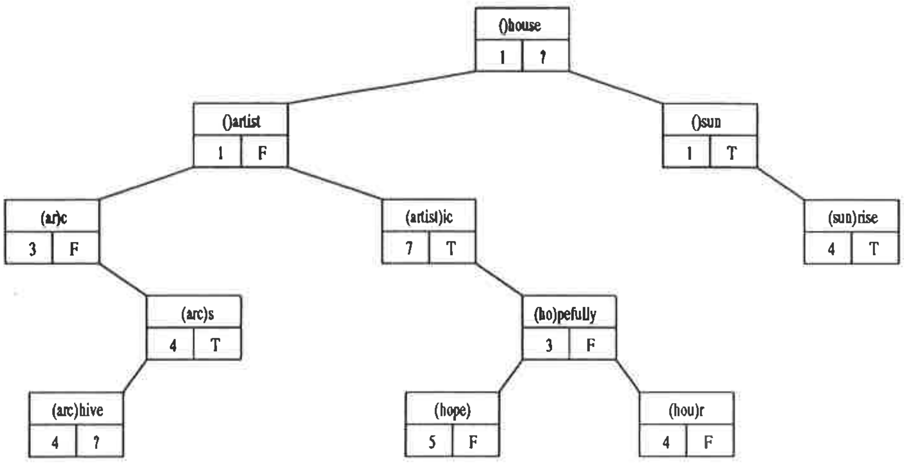


Figure 11: A binary search tree with string keys and the information to perform digital accesses.

other. If we assume that each character is stored in a 8-bit byte, then we have $R = 2^8 = 256$ possible values for the digits of the keys. Figure 11 shows a BST storing a set of English words, together with the information to perform digital accesses (again, one integer and one bit with the same meaning that in previous sections). The character `'\0'` at the end of every key is not shown explicitly.

There are several important differences with base-2 keys that are worth mentioning. When two base-2 keys $x < y$ differ at the i-th digit, we know for sure that the i-th bit of x is 0, and that the i-th bit of y is 1. By contrast, knowing that two base-R keys with $R > 2$ differ at the i-th digit does not provide enough information about the value of their i-th digits. As a consequence, for every key y in the tree we have to explicitly store at least its suffix $[y_{d(y)}, \ldots]$, since the digit $y_{d(y)}$ now is not computable from is_p(y). In Figure 11 we can see the redundant prefix of every key, which always ends at the $(y_{d(y)} - 1)$-th position. Notice that the appropriate value for the field d at the root of the tree now is not 0 but 1.

A second difference with base-2 keys is that a base-R key can differ at the same digit with its predecessor and its successor. For instance, the key `"archive"` in Figure 11 differs at the fourth digit with both `"arc"` and `"arcs"`. In such a situation we set **is_p(y)** to either true or false, since our algorithms will work properly for both values.

Finally, a digital search with base-R keys requires comparing some digits more than once. For instance, if we digitally search for the key `"home"` in the tree in Figure 11, then its first and second characters `'h'` and `'o'` are only compared at the root of the tree, while its third character `'m'` has to be compared at the root, but also at the node storing the key `"hopefully"`. No digit comparisons are needed at the nodes storing the words `"artist"`, `"artistic"`, and `"hope"`.

The search invariant for base-R keys is thus slightly different than that of base-2 keys. Let `x` be the search key, `c` the key stored at the root of the current subtree, and `p` and `s` the predecessor and the successor of `c`, respectively. Let j be the index of the first digit where `x` and `p` differ, and k the index of the first digit where `x` and `s` differ. Then the search invariant is the following: When j is larger than k we have $\mathtt{xp} = j$ and $\mathtt{xs} \leq k$, when j is smaller than k we have $\mathtt{xp} \leq j$ and $\mathtt{xs} = k$, and when j and k are equal we have $\mathtt{xp} = \mathtt{xs} = j$. At every moment only the first $\max\{\mathtt{xp}, \mathtt{xs}\}$ digits of `x` have been compared (perhaps more than once, see the end of the section), the rest of digits not yet.

When `c` differs first with its successor than with its predecessor (for the opposite situation we have symmetrical cases) the possible cases of a digital search with base-R keys resemble those in Figure 4, if we think of `0`'s and `1`'s as any pair of digits such that the first is smaller than the second. For instance, Case 1 corresponds to the situation $\mathtt{p}_q = \mathtt{c}_q = \mathtt{x}_q < \mathtt{s}_q$, $\mathtt{p}_r = \mathtt{c}_r < \mathtt{x}_r$, and $\mathtt{p}_{\mathtt{d(c)}} < \mathtt{c}_{\mathtt{d(c)}}$. The rest of cases also have their immediate equivalent in terms of base-R digits. Regarding how to perform digital accesses, Cases 1, 2 and 5 have to be handled exactly as in the binary search. Cases 3 and 4 are slightly different, since we have $\mathtt{p}_{\mathtt{d(c)}} < \mathtt{c}_{\mathtt{d(c)}}$ and $\mathtt{p}_{\mathtt{d(c)}} < \mathtt{x}_{\mathtt{d(c)}}$, but now this does not imply that $\mathtt{c}_{\mathtt{d(c)}}$ and $\mathtt{x}_{\mathtt{d(c)}}$ are equal. Therefore, we have to start comparing digits from the `d(c)`-th position, rather than from the $(\mathtt{d(c)} + 1)$-th position.

Base-R keys present two additional cases that have to be considered. The first one is similar to Case 5 but with $q = u$, which implies $\mathtt{x}_q < \mathtt{s}_q$. Here, $\mathtt{xp} = \mathtt{xs} = q$, since we have $\mathtt{p}_q = \mathtt{c}_q < \mathtt{x}_q$. In this situation we can deduce that `x` is larger than `c` without any digit comparisons. Moreover, no updates are needed for `xp` nor `xs`.

The second additional case is completely new, namely the case where `c` differs at the same digit q with `p` and `s`. If $\mathtt{xp} = \mathtt{xs} = q$, then we do not have enough information to compare `x` against `c`, so we will start comparing digits from the `d(c)`-th position. If $\mathtt{xp} > \mathtt{xs} = q$ or $\mathtt{xs} > \mathtt{xp} = q$, then we know that `x` is smaller than `c` or that `x` is greater than `c`, respectively, without comparing any digits, and without updating `xp` nor `xs`.

In Figure 12 we can see the new C definitions for keys that are strings. Now the digits of the keys are not bits but bytes (characters). Since the keys are variable-length, we assume that dynamic memory is available, and define a key to be a pointer to the first of its characters. In Figure 12 we also find the `Compare` function for strings, which has two main differences with that in Figure 6. First, the length of the keys is variable, so at the end of each iteration of the `for` we must check whether the keys have reached their end. Second, we cannot start comparing bytes at the `t->d + 1` position, as discussed previously, but rather at the `t->d` position.

```
typedef char* String;
typedef char Byte;
#define byte(X, I) (X[I-1])
typedef struct { String s; Tree l, r; Diff d; Boolean is_p; } Node;

Comp Compare(String x, Tree t, Diff *xp, Diff *xs)
{ String c; Diff i; Byte bx, bc;
  if (t == NULL) return EMPTY;
  if (t->is_p) { if (*xp < t->d) return GREATER;
                 if (*xp > t->d) return SMALLER;
               }
          else { if (*xs < t->d) return SMALLER;
                 if (*xs > t->d) return GREATER;
               }
  for (c = t->s, i = t->d; ; i++)
     { if ((bx = byte(x, i)) != (bc = byte(c, i)))
           if (bx < bc) { *xs = i; return SMALLER; }
                   else { *xp = i; return GREATER; }
         if (bx == '\0') return EQUAL;
     }
}
```

Figure 12: New C types and the **Compare** function for strings.

Regarding rotations and how to keep consistent the information to perform digital accesses with base-R keys, it is easy to see that the new cases where y or z (or both) differs at the same position with its predecessor and its successor can be handled exactly like the cases in Figure 9 (thinking of 0's and 1's as any pair of different digits). Therefore, the Rotate_left and Rotate_right functions in Figure 10 work correctly without any changes.

Let us analyse the cost, in terms of time and space, of our method applied to base-R keys with D digits (we assume fixed-length keys for clarity). As happened with fixed-length base-2 keys, the number of visited nodes depends on the balancing strategy, so we can assume it to be $\Theta(\log N)$ if desired.

The number of compared digits is not so easy to compute, since some digits may be compared more than once. However, we can provide a good upper bound based on the following fact: for each comparison between different digits we move down in the tree, while for each comparison between equal digits we move to the right in the search key. Hence, the number of digit comparisons is always bounded by above by the number of visited nodes plus the length of the prefix that has been inspected. Note that this sum is only an upper bound

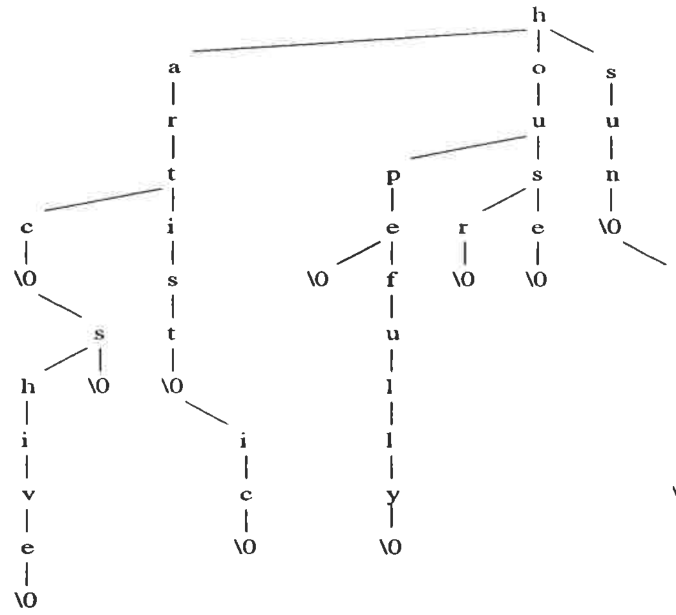


Figure 13: The ternary search tree associated to the BST in Figure 11.

because some steps down the tree may be made without comparing any digits at all (recall the example of the search for the word "home" in Figure 11). As a consequence, the number of compared digits per successful search, successful deletion, or unsuccessful insertion is between about D/R and about $D/R + \Theta(\log N)$, that is, roughly D/R for any set of digits for the keys. The number of compared digits per unsuccessful search, unsuccessful deletion, or successful insertion strongly depends on the digits of the keys. For instance, when the digits are random, it is known that the length of the expected common prefix is about $\log_R N = \ln N / \ln R$. Hence, any balancing strategy for BSTs which achieves about $\log_2 N$ visited nodes per search, together with our method, would require at most about $\log_2 N + \log_R N = (1 + 1/\log_2 R) \log_2 N$ expected digit comparisons. The actual factor could be smaller depending on the balancing strategy, but the related mathematical analysis goes far beyond the scope of this paper.

On the other hand, let us define the (unique) ternary search tree [2] associated to a given BST as the ternary search tree built by inserting the keys in the BST in any of the orders implicitly defined by the BST (say, left-preorder). (Figure 13 shows the ternary search tree —TST, for short— associated to the BST in Figure 11.) Then, it is not difficult to see that the digit comparisons of our method are exactly those that would be made in the associated TST. By contrast, every TST has in general several associated BSTs (for instance, from the TST in Figure 13 we cannot deduce if `"artist"` has been inserted before or after `"hopefully"`). Since the shape of a TST depends not only on the digits of the keys but on the insertion order of the keys as well, we could argue that the method in this paper captures the best properties of TSTs without suffering from those two potential sources of unbalancing.

Finally, the space of the fields `d` and `is_p` is $\Theta(\log D)$ per node. This is asymptotically dismissable even when we store pointers to keys instead of full keys at the nodes, because each pointer needs at least $\Theta(\log N)$ space. The space savings achieved by our method when we store each common prefix just once are exactly the same as those of any TST with the same set of keys.

5 Digital Sorting

Here, we use the ideas of previous sections to digitally sort a set of N base-R keys ($R \geq 2$) of length D always moving around $\Theta(N \log N)$ keys (or pointers to keys), and comparing at most $D \cdot N + o(D \cdot N)$ digits, which reduces to just $\Theta(N \log N)$ digit comparisons with random keys. Thus, the sorting algorithms of this section achieve together the best properties of comparison-based sort (quicksort or mergesort) and digit-guided sort (radixsort). We will see that both algorithms, quicksort and mergesort, can be adapted to the digital framework of this paper because we can view their execution as an implicit binary search tree.

Figure 14 shows the well-known quicksort algorithm. Given a couple of integers `l` and `r`, `Quicksort` sorts the (global, for the sake of clarity) array of words `K` between the `l`-th and the `r`-th positions, by first partitioning the array with respect to one of its elements, called the pivot. The partition rearranges the array into three parts, from left to right: a subarray with keys smaller or equal than the pivot, the pivot itself, and a subarray with keys greater or equal than the pivot. After that, it is enough to recursively sort both subarrays to get the whole array sorted.

In Figure 14 we also find the function `Classic_Partition`, which partitions the array `K` in the interval $[1, \ldots, r]$ with respect to `K[r]`. (If the array is not random for sure, then it would be safer to choose the pivot at random.) Since we allow for repeated keys, we use one of the easiest approaches to deal with them, namely stopping the left-to-right search not only with keys greater than the pivot, but also with keys equal to the pivot (and the symmetric strategy for the right-to-left search). The only noticeable feature in contrast to other partitioning algorithms in the literature is that `Classic_Partition` compares exactly once each element, excluded the pivot itself, against the pivot. This modification greatly simplifies the translation of classic partitioning to digital partitioning, since it guarantees that there is an implicit BST related to the

```
Word K[MAX];

#define exch_classic(A, B) { Word t = K[A]; K[A] = K[B]; K[B] = t; }

int Classic_partition(int l, int r)
{ int i = l-1, j = r;
  for (;;)
     { while (++i < j) if (K[i] >= K[r]) break;
       while (--j > i) if (K[j] <= K[r]) break;
       if (i >= j) break; else exch_classic(i, j);
     }
  exch_classic(i, r); return i;
}

void Quicksort(int l, int r)
{ int i;
  if (r <= l) return;
  i = Partition(l, r);
  Quicksort(l, i-1);
  Quicksort(i+1, r);
}
```

Figure 14: Classic quicksort for fixed-length base-2 keys.

sorting process. The macro **exch_classic** simply exchanges the values of the array K in the positions A and B. We use a macro to isolate one of the two pieces that have to be modified to get the digital version. The only other change is obviously the comparison between keys.

In Figure 15 we can see the digital counterpart (fixed length base-2 keys version) of the partitioning algorithm. The function **Quicksort** is not shown because it is exactly that in Figure 14. Apart from the array of keys **K**, we have two additional arrays **P** and **S** with the same dimensions. They store, respectively, the equivalent for every key of the variables **xp** and **xs** for the digital access to BSTs. These values (which should be initialised to 0) are used to compare the keys at each stage against the pivot, which behaves like their current parent (predecessor or successor) in the associated BST. Now, when exchanging two keys, we also have to exchange their corresponding values in P and S, as shown in the macro **exch_digital**. Like in previous sections, we could have used one boolean and one integer per key, instead of two integers, but now space considerations are less important. Since the auxiliary arrays P and S are only needed to sort the array of keys K, after sorting we can return the memory

```
Word K[MAX];
Diff P[MAX], S[MAX];

#define exch_digital(A, B) { Word t; Diff u; \
            t = K[A]; K[A] = K[B]; K[B] = t; \
            u = P[A]; P[A] = P[B]; P[B] = u; \
            u = S[A]; S[A] = S[B]; S[B] = u; }

Comp Compare(int k, int r, Boolean left)
{ Diff i; Bit bk, br;
  if (P[r] > S[r]) { if (P[k] < P[r]) return GREATER;
                     if (P[k] > P[r]) return SMALLER;
                     i = P[r];
                   }
              else { if (S[k] < S[r]) return SMALLER;
                     if (S[k] > S[r]) return GREATER;
                     i = S[r];
                   }
  while (++i <= B)
    if ((bk = bit(K[k], i)) != (br = bit(K[r], i)))
      if (bk < br) { S[k] = i; return SMALLER; }
              else { P[k] = i; return GREATER; }
  if (left) P[k] = i; else S[k] = i; return EQUAL;
}

int Digital_partition(int l, int r)
{ int i = l-1, j = r;
  for (;;)
     { while (++i < j) if (Compare(i, r, TRUE)  != SMALLER) break;
       while (--j > i) if (Compare(j, r, FALSE) != GREATER) break;
       if (i >= j) break; else exch_digital(i, j);
     }
  exch_digital(i, r); return i;
}
```

Figure 15: Digital partition for fixed-length base-2 keys.

space of P and S to the system.

The function Compare mimics that in Figure 6, with two exceptions. First, the EMPTY case makes no sense now. Second, we must inform the function if we are scanning the left or the right side of the array. Note that, since equal keys are now permitted, it is necessary to know whether a key equal to the pivot K[r] will be placed with the keys smaller than the pivot or with the keys greater than the pivot. For instance, if we are scanning the left side of the array and we find a key K[k] equal to the pivot, then that key will be placed to the right of the pivot, which will be the current predecessor of the key, and consequently P[k] is updated.

It is easy to compute the cost of digital quicksort for fixed-length base-2 keys. On the one hand, the expected number of movements of keys in the array is the same as that of classic quicksort, so it is $\Theta(N \log N)$ with random permutations of different keys, and even smaller with repeated keys. Note that this cost could be guaranteed in the worst case or with extremely high probability, by selecting the median of the array as pivot or by taking samples, respectively. On the other hand, the compared bits are exactly those in the process of inserting the keys into a trie. Hence, the number of compared bits depends on the digits of the keys, and can vary from about $N \log_2 N$ with random keys to about $N \cdot D$ in the worst case. Note that, with a bad-case set of digits for the keys, radixsort compares about $N \cdot D$ bits as well, but moves around $\Theta(N \cdot D)$ keys.

Let us now consider the cost of digital quicksort for strings. (It is easily obtained combining the base-2 keys digital quicksort above and the string digital mergesort given below.) The number of *pointers* (not full keys) moved around is again $\Theta(N \log N)$ with random permutations of different strings. The number of times each character is compared strongly depends on the digits of the keys, and on the shape of the BST related to the quicksort execution. If that BST is well balanced (which can be achieved with high probability or for sure), then the number of digit comparisons is about $N \cdot D$ for a bad-case set of digits for N keys of length D (each digit compared once plus a few additional comparisons). The number of digit comparisons for random keys is $\Theta(N \log N)$ —see the comparison of our method against ternary search trees in the final section of this paper.

Let us now adapt mergesort to digitally sort a set of string keys. Figure 16 shows the classic version of mergesort, which sorts the array K by first sorting its left and right halves, and then merging them. This is the mission of the function Classic_merge, which scans K starting at the l-th and m-th positions, always copying the smallest of the two current keys in the auxiliary array auxK, until one of the subarrays reaches its end. From that moment, the remaining of the non-empty subarray is copied in auxK. Finally, the content of auxK is copied back to K.

As we did for quicksort, we have used global arrays for clarity, and macros to emphasise the differences with the digital version. The macro copy_inc_classic

```
String K[MAX], auxK[MAX];

#define copy_inc_classic(A, B) { auxK[A++] = K[B++]; }
#define copy_back_inc_classic(A, B) { K[A++] = auxK[B++]; }

void Classic_merge(int l, int m, int r)
{ int a = l, b = m, c = 0;
  while ((a < m) && (b < r))
    if (strcmp(K[a], K[b]) <= 0) { copy_inc_classic(c, a); }
                            else { copy_inc_classic(c, b); }
  while (a < m) copy_inc_classic(c, a);
  while (b < r) copy_inc_classic(c, b);
  a = l; c = 0;
  while (a < r) copy_back_inc_classic(a, c);
}

void Mergesort(int l, int r)
{ if (r <= l) return;
  Mergesort(l, (l+r)/2);
  Mergesort((l+r)/2+1, r);
  Merge(l, (l+r)/2+1, r+1);
}
```

Figure 16: Classic mergesort for string keys.

just copies one element from the array **K** into the auxiliary array **auxK**, and increments the indices. The macro **copy_back_inc_classic** does the opposite. The condition `strcmp(K[a], K[b]) <= 0` is conceptually equivalent to **a** $\leq$ **b**, and is evaluated comparing the digits of the keys from the beginning. However, the redundant comparisons of digits can be avoided if we imagine each of the two sorted subarrays as a sorted list, which is just a particular case of binary search tree, where the keys have been inserted in increasing order (the smallest key at the root, and so on).

In Figure 17 we find the digital version of the merging algorithm in Figure 16 (the function **Mergesort** is exactly the same). Compared to the string algorithms in Section 4 and to the digital version of quicksort, there are a couple of observations in order. First, each key has current predecessor but no current successor, which means that only the additional arrays **P** (initialised to 1) and **auxP** are needed. Second, when two keys **K[a]** and K[b] are equal, we copy **K[a]** in **auxK** (to achieve a stable version of mergesort), which means

```
String K[MAX], auxK[MAX];
Diff P[MAX], auxP[MAX];

#define copy_inc_digital(A, B) { auxK[A] = K[B]; \
                              auxP[A++] = P[B++]; }
#define copy_back_inc_digital(A, B) { K[A] = auxK[B]; \
                                   P[A++] = auxP[B++]; }

Comp Compare(int a, int b)
{ Diff i; Byte ba, bb;
  if (P[a] < P[b]) return GREATER;
  if (P[a] > P[b]) return SMALLER;
  for (i = P[b]; ; i++)
     { if ((ba = byte(K[a], i)) != (bb = byte(K[b], i)))
          if (ba < bb) { P[b] = i; return SMALLER; }
                  else { P[a] = i; return GREATER; }
       if (ba == '\0') { P[b] = i; return EQUAL; }
     }
}

void Digital_merge(int l, int m, int r)
{ int a = l, b = m, c = 0;
  while ((a < m) && (b < r))
    if (Compare(a, b) != GREATER) { copy_inc_digital(c, a); }
                             else { copy_inc_digital(c, b); }
  while (a < m) copy_inc_digital(c, a);
  while (b < r) copy_inc_digital(c, b);
  a = l; c = 0;
  while (a < r) copy_back_inc_digital(a, c);
}
```

Figure 17: Digital merge for string keys.

that K[a] always becomes the new current predecessor of K[b]. Therefore, no boolean is needed for the equality case.

Let us analyse the cost of digital mergesort for string keys. Clearly, the number of pointer copies is always about $N \log_2 N$, or about $2N \log_2 N$ if we count each movement back from the auxiliary array. Concerning the number of digit comparisons, it depends on the digits of the keys, as usual. But the worst case now can be much larger than that of quicksort, because the associated BST has the poorest possible balance, with an average depth of its keys about $N/2$. In principle, this could amount to about $N/2 + D$ digit comparisons per key (assuming keys of length D), that is, to about $N^2/2$ digit comparisons. However, we can use the following observation to get a better upper bound: every one of the D digits of each key can be compared at most R times before moving to the right in the key. Therefore, at most $N \cdot D \cdot R$ digit comparisons can be made, which is R times larger than the upper bound for quicksort (too much for large bases). Note that, anyway, this is asymptotically smaller than the worst-case number of digit comparisons of classic mergesort with strings, which is about $D \cdot N \log_2 N$. A tight upper bound for the number of compared digits with random keys is more difficult to compute with precision, but we certainly know that it is $\Theta(N \log N)$, because the common prefixes have $\Theta(\log N)$ digits on average, and each of these digits can be compared at most R times.

Finally, let us compute the cost of the digital mergesort algorithm for fixed-length base-2 keys, which is also easily obtained by combining the algorithms in this section. The number of keys moved around is again about $N \log_2 N$ or $2N \log_2 N$. Fortunately, the compared digits with $R = 2$ do not depend on the shape of the related BST, and are always those in the insertion of the keys into a trie. Hence, as it happened with quicksort, the number of compared bits varies from about $N \log_2 N$ with random keys to about $N \cdot D$ in the worst case.

6 Final Remarks and Conclusions

We have seen a simple method to build tree data structures which achieve just $\mathcal{O}(\log N)$ visited nodes and $\mathcal{O}(D)$ compared bits/bytes per search or update, irrespectively of the order of the updates and of the digital representation of the keys. In short, we have seen that the idea is to provide every key y in the tree with a boolean and an integer, which enable to know which of the keys above y shares the longest common prefix with y, and how long is that prefix. This additional information allows us to search for any given key comparing each bit once, or each byte at most a constant number of times. It is also very simple to update this information after a deletion or insertion in a leaf, or after a rotation. This implies that most balancing strategies for trees can be adapted to the digital framework presented in this paper, thus adding to their best structural (comparison-based) properties the chance to digitally access their information. Moreover, the translation is straightforward if we use some already built functions like the ones in Figures 6, 10 or 12, for instance. We have also seen that these ideas can be applied to the sorting problem, achieving algorithms with the best properties of both comparison-based sorting (quicksort or mergesort), and digit-guided sorting (radixsort). All the algorithm in this paper have been written as intuitively as possible, in order to emphasise the translation between classic and digital counterparts. Therefore, improved versions could be developed at the price of losing clarity.

To provide a broad perspective, we end the paper comparing digital access to balanced BSTs against some of the main data structures for the abstract data type "symbol table", as presented in [11]: classic access to balanced BSTs, tries, patricia tries, R-ary tries, TSTs, and hashing.

The comparison against classic access to balanced BSTs is easy: for the price of $\lceil \log_2 D \rceil + 1$ additional bits per node, we have the same cost per operation regarding the number of visited nodes, but less bit/byte comparisons in general. When the keys have long common prefixes, digital access saves much time and can also save much storage space. Moreover, since a BSTs with the information to perform digital accesses is still a BST, we have always the choice between classic search and digital search.

The comparison of digital access to balanced BSTs against tries and patricia tries is not so straightforward. First of all, the latter require access to individual bits of the keys, while the former has the flexibility to deal with accesses to bits or bytes. Moreover, tries and patricia tries do not adapt well to practical applications where the keys are strings whose characters are not used uniformly, or where the keys have an underlying format. And more important of all, with non-random keys the number of visited nodes per search in a trie or patricia trie can be much larger than $\Theta(\log N)$, and could rise up to about D and about $D/2$, respectively.

On the other hand, tries (and also TSTs, which are discussed below) allow to easily search for keys following some digital patterns, like keys differing in a certain number of digits with another key, or keys with some positions unspecified, etcetera. Although not covered by this paper, it is likely that for every pattern search algorithm for tries or TSTs there exists a (not so direct) counterpart for our method, because it also captures the digital properties of the keys (in other words, we always have an implicit trie or TST stored).

Other properties are worth being discussed. When the keys are truly random and bit access is available, tries achieve $\log_2 N + o(\log N)$ visited nodes per search more easily than the method in this paper. On the other hand, our method does not use extra nodes with null links (a trie with random keys has about $N/\ln 2 \simeq 1.44N$ nodes on the average), and uses only one kind of node, in contrast to the two kinds used normally in tries (otherwise tries waste too space). Furthermore, tries with keys like those in Figure 1 require $\Theta(D \cdot N)$ nodes to be stored. The space saved by storing every common prefix just once is exactly the same for tries and the method in this paper.

Patricia tries also achieve $\log_2 N + o(\log N)$ visited nodes per search under random keys, provided that bit access is available. Search misses in a patricia trie are faster when the keys in the tree have long common prefixes and we search for a missing key that can be identified as such inspecting a few bits at the end of the key. Compared to patricia tries, our method requires just one additional boolean per node (recall that patricia tries also store an integer, which codifies the next position in the key to be checked). It is arguable whether patricia tries are simpler (for instance, inserting or removing a key into or from a patricia trie requires two searches). In contrast to our method, patricia tries cannot save space from common prefixes. Finally, a successful search in a patricia trie requires comparing the significant bits in the longest common prefix of the key with the rest of keys, plus a final full key comparison (D bits). Usually, this is just slightly more than the D bits of our method, but could become $1.5D+o(D)$ on the average for bizarre keys like those in Figure 1. It is important to observe that, if the keys are random and accessing bits is significantly slower than accessing bytes (say, eight bit comparisons are much slower than one 8-bit byte comparison), then patricia tries are faster than our method with $R = 2$, because the final full key comparison of patricia tries can be performed as $\lceil D/8 \rceil$ byte comparisons, while our method inspects bits one by one. However, in such a situation we would choose to digitally access the tree not comparing bits but bytes. As discussed in previous sections, the number of byte comparisons per successful search would be bounded between about $D/8$ and about $D/8 + \Theta(\log N)$, that is, would always be $D/8 + o(D)$, but for our method this cost holds irrespectively of the distribution of digits for the keys.

The prime virtue of TSTs is their simplicity. However, our method outperforms TSTs in several aspects. To begin, TSTs cannot be used with $R = 2$. Moreover, TSTs need three links per digit in the worst case, while our method requires always two links *per key*. TSTs probably need only $N + o(N)$ nodes with random keys if each suffix is stored in just one node, avoiding one-way branching at the end of the keys, but this implies using two kinds of nodes. The number of visited nodes (or digit comparisons) in a successful search in a TST is always between D and $\Theta(D \cdot R)$, and is likely be, on the average, slightly higher than D for random keys. But note that with non-random keys could be much larger. Search misses in a TST are likely to require more digit comparisons in some situations. For instance, the expected common prefix with random keys has about $\log_R N = \ln N / \ln R$ digits, and at each of these digits a TST behaves more or less like a random BST over R keys. Altogether, this could amount to about $2 \ln R \cdot \ln N / \ln R = 2 \ln N = (2 \ln 2) \log_2 N$ digit comparisons per unsuccessful random search. By contrast, recall that our method applied to any balancing strategy wich achieves average depth $\log_2 N$ requires *at most* $(1 + 1/\log_2 R) \log_2 N$ digit comparisons per unsuccessful random search. Note that $1 + 1/\log_2 R$ is smaller than $2 \ln 2$ when $R > 2^{1/(2 \ln 2 - 1)} \simeq 6$. Anyway, for any TST there is at least one associated BST which implicitly stores the same TST, so the digit comparisons can always be exactly the same for both data structures. The space savings from common prefixes are identical for TSTs and our method, if we choose to do so.

Regarding R-ary tries, they achieve about $\log_R N$ visited nodes per unsuccessful search when the keys are random, which for large R is much smaller than the asymptotic upper bounds $\log_2 N$ and $(1 + 1/\log_2 R)\log_2 N$ for the number of visited nodes and digit comparisons of our method. On the other hand, R-ary tries suffer from a severe waste of space: $R/\ln R \cdot N$ links under random keys (which could become about $R \cdot D/2 \cdot N$ in the worst case), in front of $2N$ links, irrespectively of the digits of the keys, of the method in this paper. Moreover, our method uses only one type of node, in contrast to the two types normally used to save space with R-ary tries.

Finally, hashing has "constant" search and update time, which in fact is at least proportional to D (the hashing function must be computed from all the digits). Recall that the method presented in this paper has cost at most $\Theta(D)$, which sometimes is just $\Theta(\log N)$. (To be completely rigorous, that cost is at least $\Theta((\log N)^2)$, since we traverse $\Theta(\log N)$ pointers and each one has length at least $\Theta(\log N)$.) The hashing function may be difficult to build for formatted keys. BSTs are dynamic, while some variants of hashing require estimations of the number of keys to be stored. Explicitly balanced trees have worst-case guarantees, in contrast to hashing. Last but not least, in a BST the keys are implicitly sorted, which also implies that rank operations are straightforward.

Acknowledgements

I thank Conrado Martínez for his useful suggestions, and Roberto Grossi and Giuseppe Italiano for their references to previous works.

References

[1] G.M. Adel'son-Vel'skii and E. M. Landis. An algorithm for the organization of information. *Dokladi Akademia Nauk SSSR*, 146(2):263–266, 1962. English translation in Soviet Math. Doklay 3, 1259-1263, 1962.

[2] J. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *Proc. of the 8th ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 360–369, 1997.

[3] P. Ferragina and R. Grossi. The string B-tree: A new data structure for string search in external memory and its applications. *Journal of the ACM*, to appear. Preliminary version in *Proc. 27th ACM Symp. on Theory of Comp.*, 693–702, 1995.

[4] E. Fredkin. Trie memory. *Communications of the ACM*, 3:490–500, 1960.

[5] R. Grossi and G. Italiano. Efficient techniques for maintaining multidimensional keys in linked data structures. In *Proc. of the 26th International Colloquium (ICALP-99)*, LNCS. Springer, 1999. To appear.

[6] L.J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proc. of the 19th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 8–21, October 1978.

[7] R. Irving. Suffix binary search trees. Technical report, Department of Computer Science, University of Glasgow, April 1997.

[8] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22:935–948, Oct 1993.

[9] C. Martínez and S. Roura. Randomized binary search trees. *Journal of the ACM*, 45(2):288–323, March 1998.

[10] D. R. Morrison. Patricia – practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15:514–534, 1968.

[11] R. Sedgewick. *Algorithms in C.* Addison-Wesley, 3rd edition, 1998.

Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya

Research Reports – 1999

LSI–99–1–R "The Width-size Method for General Resolution is Optimal", Maria Luisa Bonet and Nicola Galesi.

LSI–99–2–R "Geometry Simplification", Carlos Andujar.

LSI–99–3–R "The Discretized Polyhedra Simplification (DPS): a Framework for Polyhedra Simplification Based on Decomposition Schemes", Carlos Andujar, Dolors Ayala and Pere Brunet.

LSI–99–4–R "On-Line Sampling Methods for Discovering Association Rules", Carlos Domingo, Ricard Gavaldà, and Osamu Watanabe.

LSI–99–5–R "Experiments on Applying Relaxation Labeling to Map Multilingual Hierarchies", Jordi Daudé, Lluís Padró and German Rigau.

LSI–99–6–R "Estructuras Geometricas Jerarquicas para la Modelizacion de Escenas 3D", P. Brunet, L. Chiarabini, G. A. Patow, F. J. Santisteve, E. Sttafetti, J. Surinyac and A. Vilanova.

LSI–99–7–R "Proposals on Mapping Multilingual Hierarchies", J. Daudé, L. Padró and G. Rigau.

LSI–99–8–R "Computing the Medial Axis Transform of Polygonal Domains by Tracing Paths", R. Joan Arinyo, L. Pŕez and J. Vilaplana.

LSI–99–9–R "ELX: Entorn Latex pel dibuix de xarxes de demostració", Josep M. Merenciano.

LSI–99–10–R "Convergence theorems for some layout measures on random lattice and random geometric graphs", Josep Diaz, Mathew D. Penrose, Jordi Petit and Maria Serna.

LSI–99–11–R "Linear Orderings of Random Geometric Graphs (Extended Abstract)", Josep Diaz, Mathew D. Penrose, Jordi Petit and Maria Serna.

LSI–99–12–R "The Proper Interval Colered Graph problem for caterpillar trees", Carme Alvarez and Maria Serna.

LSI–99–13–R "A Modular Approach to Software Process Modelling and Enaction", Xavier Franch and Josep M. Ribó.

LSI–99–14–R "Improving Mergesort for Linked Lists", Salvador Roura.

LSI–99–15–R "Axiomatic frameworks for developing BSP-style programs", A. Stewart, M. Clint and J. Gabarró.

LSI-99-16-R "Algorithms to Mesh 2D CSG Polygonals Domains from Previously Meshed CSG Primitives", R. Joan-Arinyo and M. Sole.

LSI-99-17-R "Fringe analysis of synchronized parallel insertion algorithms on 2-3 trees", R. Baeza-Yates, J. Gabarró, X.Messeguer and M.S. Busquier.

LSI-99-18-R "Beginning to Programming pilot course, IniPro, using Java.", X. Franch, J. Gabarró, A. Gómez, A. Vázquez and J. Vázquez.

LSI-99-19-R "Robust Geometric Computation (RGC), State of the Art.", F. J. Santisteve.

LSI-99-20-R "A Soft Computing Techniques Study in Wastewater Treatment Plants", Ll. Belanche , J.J. Valdés, J. Comas , I.R. Roda and M. Poch.

LSI-99-21-R "CORBA: A middleware for an heterogeneous cooperative system", Alberto Abelló Gamazo.

LSI-99-22-R "The Visibility Octree. A Data Structure for 3D Navigation", Carlos Saona-Vzquez, Isabel Navazo and Pere Brunet.

LSI-99-23-R "The Constructive Method for Query Containment Checking (extended version)", Carles Farré, Ernest Teniente, Toni Urpí.

LSI-99-24-R "Redundancy and Subsumption in High-Level Replacement Systems", H.-J Kreowski and Gabriel Valiente.

LSI-99-25-R *"Grammatica: An Implementation of Algebraic Graph Transformation on Mathematica", Gabriel Valiente.*

LSI-99-26-R "Digital Access to Comparison-Based Tree Data Structures and Algorithms", Salvador Roura.

LSI-99-27-R "Addressing Efficiency Issues During the Process of Integrity Maintenance (Extended Version)", Enric Mayol and Ernest Teniente.

LSI-99-28-R "Maximal Strategies for Paramodulation with Non-Monotonic Orderings", Miquel Bofill and Guillem Godoy.

LSI-99-29-R "Improving questionnaire screening of sleep apnea cases using fuzzy knowledge representation and aggregation techniques", David Nettleton and Lourdes Hernández.

LSI-99-30-R "Desarrollo de una aplicación CAE para el diseño de circuitos de ventilación", D. Ayala, N. Pla, A. Soto, M. Vigo and S. Vila.

LSI-99-31-R "Integration, modeling and visualization of multimodal data of the human brain", Maria Ferré Bergadà and Dani Tost Pardell.