Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT

Journal of Discrete Algorithms 1 (2003) 387–408

JOURNAL OF DISCRETE ALGORITHMS

www.elsevier.com/locate/jda

# The suffix binary search tree and suffix AVL tree

Robert W. Irving [*], Lorna Love

*Department of Computing Science, University of Glasgow, Glasgow G12 8RZ, Scotland, UK*

**Abstract**

Suffix trees and suffix arrays are classical data structures that are used to represent the set of suffixes of a given string, and thereby facilitate the efficient solution of various string processing problems—in particular on-line string searching. Here we investigate the potential of suitably adapted binary search trees as competitors in this context. The *suffix binary search tree* (SBST) and its balanced counterpart, the *suffix AVL-tree*, are conceptually simple, relatively easy to implement, and offer time and space efficiency to rival suffix trees and suffix arrays, with distinct advantages in some circumstances—for instance in cases where only a subset of the suffixes need be represented.

Construction of a suffix BST for an $n$-long string can be achieved in $\mathrm{O}(nh)$ time, where $h$ is the height of the tree. In the case of a suffix AVL-tree this will be $\mathrm{O}(n \log n)$ in the worst case. Searching for an $m$-long substring requires $\mathrm{O}(m + l)$ time, where $l$ is the length of the search path. In the suffix AVL-tree this is $\mathrm{O}(m + \log n)$ in the worst case. The space requirements are linear in $n$, generally intermediate between those for a suffix tree and a suffix array.

Empirical evidence, illustrating the competitiveness of suffix BSTs, is presented.
© 2003 Elsevier B.V. All rights reserved.

*Keywords:* Binary search tree; AVL tree; Suffix tree; Suffix array; String searching

## 1. Introduction

Given a string $\sigma = \sigma_1\sigma_2 \ldots \sigma_n$ of length $n$, a *suffix binary search tree* (or *SBST*) for $\sigma$ is a binary tree containing $n$ nodes, each labelled by a unique integer in the range $1 \ldots n$, the integer $i$ representing the $i$th suffix $\sigma^i = \sigma_i\sigma_{i+1} \ldots \sigma_n$ of $\sigma$. We refer to the node representing suffix $\sigma^i$ simply as node $i$ of the tree. Furthermore, the tree is structured so that, for each node $i$, $\sigma^i$ is lexicographically greater than $\sigma^j$ for every node $j$ in its left subtree, and lexicographically less than $\sigma^k$ for every node $k$ in its right subtree.

[*] Corresponding author.
*E-mail addresses:* rwi@dcs.gla.ac.uk (R.W. Irving), love@dcs.gla.ac.uk (L. Love).

1570-8667/$ – see front matter © 2003 Elsevier B.V. All rights reserved.
doi:10.1016/S1570-8667(03)00034-0

The concept of a suffix binary search tree is related to the suffix array, introduced by Manber and Myers [7] as an alternative to the widely applicable suffix tree [8,9,11]. See also [2] for an indication of suffix tree applications, and [4] for a detailed exposition of suffix trees and suffix arrays. Suffix arrays have some advantages over suffix trees, particularly in respect of space requirements, and we claim that suffix BSTs have their own potential advantages, at least in some circumstances. In Section 5, we present empirical evidence suggesting that, in practice, the suffix BST is broadly competitive with suffix trees and suffix arrays in indexing real data, such as plain text or DNA strings. A particular advantage is that a standard suffix BST can easily be constructed so as to represent a proper subset of the suffixes of a text. For example, if the text is natural language, it might be appropriate to represent in the tree only those suffixes that start on a word boundary, resulting in a saving in space and construction time by a factor of the order of $1 + w$, where $w$ is the average word length in the text.

Classical algorithms [8,9,11] construct a suffix tree for a string of length $n$ in $\mathrm{O}(n \log |\Sigma|)$ time and $\mathrm{O}(n)$ space, where $\Sigma$ is the alphabet, and a recent more involved algorithm described by Farach et al. [3] removes the dependence on alphabet size. Given a suffix tree for $\sigma$ and a pattern $\alpha$ of length $m$, an algorithm to determine whether the pattern appears in the string can be implemented to run in $\mathrm{O}(m \log |\Sigma|)$ time. The corresponding time bounds for construction and search in the case of a suffix array [7] are $\mathrm{O}(n \log n)$ and $\mathrm{O}(m + \log n)$, using $\mathrm{O}(n)$ space.

For a suitably implemented SBST, a search requires $\mathrm{O}(m+l)$ time, where $l$ is the length of the search path in the tree. This gives $\mathrm{O}(m + n)$ worst-case complexity, but typically in practice, all search paths will have $\mathrm{O}(\log n)$ length, and searching will be $\mathrm{O}(m + \log n)$ on average. In fact, this becomes a worst-case bound if we use AVL rotations to balance the tree on construction. (As we shall see, this is a feasible, but non-trivial extension.) The construction time for our standard SBST can be as bad as $\mathrm{O}(n^2)$ in the worst case, but for a refined version, it can be achieved in $\mathrm{O}(nh)$ time, where $h$ is the *height* of the tree, In the worst case, $h$ can be $\Theta(n)$, but for random strings, $h$ can be expected to be $\mathrm{O}(\log n)$, and in the case of the suffix AVL tree, construction can be accomplished in $\mathrm{O}(n \log n)$ time in the worst case.

Although both suffix trees and suffix arrays use linear space, the latter can be represented more compactly. This issue is explored in detail by Gusfield [4] and by Kurtz [6]. Traditional representations of a suffix tree [8] require $28n$ bytes, in the worst case, but more compact representations are possible. The most economical, due to Kurtz [6], has a worst-case requirement of $20n$ bytes, though empirical evidence suggests an actual requirement of around $10n$–$12n$ bytes in practical cases. For a suffix array, an implementation using just $5n$ bytes is feasible once the construction is complete, although $9n$ bytes are needed during construction.[1]

As we shall see, in the standard implementation of an SBST, each node contains two integers, two pointers and one additional bit. (Of course, the additional bit can easily be incorporated as a sign in one of these integers.) In fact, using an array to house the tree,

[1] In all cases, we exclude the space needed for the string itself, and we assume 4 bytes per integer or pointer value.

rather than dynamically created nodes, allows us to dispense with one of the integers. Hence the space requirement for an SBST representing a string of length $n$ is essentially $12n$ bytes. For the construction of the refined version, each node requires two additional pointers, and, in the case of the suffix AVL tree, two further bits to indicate its *balance factor*.

We refer again to the ease with which standard SBSTs can be used to represent a subset of the suffixes—we call these *partial* suffix SBSTs. For example, we can expect a saving of 80% or more in space (and time for construction) if only suffixes starting on a word boundary are included (when the string is plain text). Andersson et al. [1] describe a complex method of adapting suffix trees for this purpose, but no implementation of this method, or empirical evidence of its behaviour, have been reported. There appears to be no discussion in the literature of any corresponding variant of the suffix array.

The remainder of this paper is organised as follows. Section 2 contains a detailed description of the search algorithm for an SBST, together with proof of correctness, worst-case complexity analysis, and an easy extension to find all occurrences of a given search string. Section 3 contains a detailed description and analysis of algorithms for the construction of an SBST, both the standard version and the refined variant that significantly improves the worst-case performance (and indeed the performance in practice), together with a brief discussion of partial SBSTs. Section 4 describes the construction of suffix AVL-trees, and shows that this can be achieved in $\mathrm{O}(n \log n)$ time in the worst case. Finally, Section 5 contains empirical evidence comparing the performance, in practice, of SBSTs with that of suffix trees and suffix arrays.

## 2. The SBST search algorithm

### 2.1. A naive SBST

In the most basic form of an SBST, each node contains one suffix number together with pointers to its two children. However, in order to improve the performance of the search algorithm, we have to include some additional information in each node of the tree.

Suppose that we wish to find an occurrence, if one exists, of an $m$-long pattern $\alpha$ in an $n$-long string $\sigma$ by searching in a basic SBST $T_\sigma$ for $\sigma$. A naive search is potentially very inefficient, irrespective of the shape of the tree. If, at each node visited, comparisons begin with the first character of $\alpha$, then up to $m$ character comparisons may be required at each node, giving a worst-case complexity that is no better than $\mathrm{O}(mh)$, where $h$ is the height of $T_\sigma$.

### 2.2. Avoiding repeated comparisons

The key to a more efficient SBST search algorithm is the need to avoid repeated equal character comparisons. The number of unequal character comparisons during a search cannot exceed the length $l$ of the search path (at most one per node visited). It will be our aim to ensure that no character in the pattern can be involved in more than one equal comparison, so that the complexity of search will be $\mathrm{O}(h + m)$ in the worst case.

In order to establish how this can be achieved, we first require some terminology and notation. Given two strings $\alpha$ and $\beta$, we denote by $lcp(\alpha, \beta)$ the length of the *longest common prefix* of $\alpha$ and $\beta$. For a given node $i$ in an SBST, a *left* (respectively *right*) *ancestor* is any node $j$ such that $i$ is in the right (respectively left) subtree of $j$. The *closest left ancestor* $cla_i$ of $i$ is the left ancestor $j$ such that no descendant of $j$ is a left ancestor of $i$. The *closest right ancestor* $cra_i$ is defined similarly.

We also define two values associated with each node, namely

$$m_i = \begin{cases} 0 & \text{if node } i \text{ is the root,} \\ \max_j lcp(\sigma^i, \sigma^j) & \text{otherwise, where the maximum is taken} \\ & \text{over all ancestors } j \text{ of node } i, \end{cases}$$

and

$$d_i = \begin{cases} left & \text{if node } i \text{ is in the left subtree} \\ & \text{of the node } j \text{ for which } m_i = lcp(\sigma^i, \sigma^j), \\ right & \text{otherwise.} \end{cases}$$

Note that $d_i$ is undefined if $i$ is the root, but otherwise $m_i$ and $d_i$ are defined for all nodes (though there is a choice for the value of $d_i$ for those nodes $i$ for which $lcp(\sigma^i, \sigma^{cla_i}) = lcp(\sigma^i, \sigma^{cra_i})$, and that choice may be made arbitrarily).

It turns out, as we will see, that inclusion in each node $i$ of the values $m_i$ and $d_i$ gives just enough information to enable repeated equal character comparisons in the search algorithm to be avoided.

The theorems that follow describe how the search for a string $\alpha$ should proceed on reaching a node $i$. At that point in the search, we need access to two values, namely

- $llcp = \max_j lcp(\alpha, \sigma^j)$ where the maximum is taken over all right ancestors $j$ of $i$;
- $rlcp = \max_j lcp(\alpha, \sigma^j)$ where the maximum is taken over all left ancestors $j$ of $i$.

Clearly, $llcp = lcp(\alpha, cra_i)$ and $rlcp = lcp(\alpha, cla_i)$. In addition, for brevity, we use $p$ to stand for node $cla_i$ and $q$ to stand for node $cra_i$.

We make substantial use of Lemma 1, which is trivial to verify.

**Lemma 1.** *If $\alpha$, $\beta$ and $\gamma$ are strings such that $\alpha < \beta < \gamma$, then $lcp(\alpha, \gamma) = \min(lcp(\alpha, \beta), lcp(\beta, \gamma))$.*

**Theorem 1.** *If $m_i > \max(llcp, rlcp)$ then the search for $\alpha$ should continue in the direction $d_i$ from node $i$. Furthermore the values of llcp and rlcp remain unchanged.*

**Proof.** We have $m_i = \max(lcp(\sigma^i, \sigma^p), lcp(\sigma^i, \sigma^q))$, $llcp = lcp(\alpha, \sigma^p)$, $rlcp = lcp(\alpha, \sigma^q)$. Suppose $\sigma^q < \sigma^i < \alpha < \sigma^p$. (A symmetrical argument applies if $\sigma^q < \alpha < \sigma^i < \sigma^p$.) Then from Lemma 1, we have

$$lcp(\sigma^i, \sigma^p) = \min(lcp(\sigma^i, \alpha), lcp(\alpha, \sigma^p)) \tag{1}$$

and so $lcp(\sigma^i, \sigma^p) \leqslant lcp(\alpha, \sigma^i)$. The fact that $m_i > \max(llcp, rlcp) \geqslant llcp$ therefore implies that $m_i = lcp(\sigma^i, \sigma^q)$, for otherwise $m_i = lcp(\sigma^i, \sigma^p) \leqslant lcp(\alpha, \sigma^p) = llcp \leqslant$

$\max(llcp, rlcp)$, which is a contradiction. It follows that $d_i = right$, as required. Hence,

$$lcp(\sigma^i, \sigma^q) = m_i > \max(llcp, rlcp) \geqslant rlcp = lcp(\alpha, \sigma^q), \tag{2}$$

so by Lemma 1

$$lcp(\alpha, \sigma^q) = \min(lcp(\sigma^i, \sigma^q), lcp(\sigma^i, \alpha)) = lcp(\sigma^i, \alpha). \tag{3}$$

It follows that the value of *rlcp* should remain unchanged as $rlcp = lcp(\sigma^i, \alpha) = lcp(\alpha, \sigma^q)$. It is immediate in this case that the value of *llcp* should remain unchanged since there is no new left branch to consider. □

Prior to the next theorem we require a further lemma.

**Lemma 2.** *At any node $i$ in the search tree,* $\max(llcp, rlcp) > m_i \Rightarrow llcp \neq rlcp$.

**Proof.** Suppose that $llcp = rlcp = t$, so that $\sigma^q(1..t) = \alpha(1..t) = \sigma^p(1..t)$. But because $\sigma^q < \sigma^i < \sigma^p$ it follows that $\sigma^q(1..t) = \sigma^i(1..t) = \sigma^p(1..t)$, so that $m_i \geqslant t = \max(llcp, rlcp)$, a contradiction. □

**Theorem 2.**

(a) *If $m_i < \max(llcp, rlcp)$ and $\max(llcp, rlcp) = llcp$ then the search for $\alpha$ should branch right from node $i$. Furthermore, if $d_i = right$ then the value of rlcp remains unchanged, otherwise rlcp should become $m_i$. In either case, the value of llcp remains unchanged.*
(b) *If $m_i < \max(llcp, rlcp)$ and $\max(llcp, rlcp) = rlcp$ then the search for $\alpha$ should branch left from node $i$. Furthermore, if $d_i = left$, then the value of llcp remains unchanged, otherwise llcp should become $m_i$. In either case, the value of rlcp remains unchanged.*

**Proof.** We prove only part (a), the proof of (b) being similar. If $\sigma^q < \alpha < \sigma^i < \sigma^p$ then, by Lemma 1,

$$lcp(\alpha, \sigma^p) = \min(lcp(\alpha, \sigma^i), lcp(\sigma^i, \sigma^p)) \leqslant lcp(\sigma^i, \sigma^p). \tag{4}$$

Also,

$$m_i < \max(llcp, rlcp) = llcp = lcp(\alpha, \sigma^p) \leqslant lcp(\sigma^i, \sigma^p). \tag{5}$$

But $m_i = \max(lcp(\sigma^i, \sigma^p), lcp(\sigma^i, \sigma^q)) \geqslant lcp(\sigma^i, \sigma^p)$, giving a contradiction. Hence, $\sigma^q < \sigma^i < \alpha < \sigma^p$, and the search for $\alpha$ should branch right from node $i$. It is immediate that the value of *llcp* should remain unchanged, since there is no new left branch to consider.

If $d_i$ = right then $lcp(\sigma^i, \sigma^q) \geqslant lcp(\sigma^i, \sigma^p)$. But, from Lemma 1 we have

$$lcp(\sigma^i, \sigma^p) = \min(lcp(\sigma^i, \alpha), lcp(\alpha, \sigma^p)) = lcp(\sigma^i, \alpha) \tag{6}$$

(since $lcp(\alpha, \sigma^p) = lcp(\sigma^i, \sigma^p) \Rightarrow llcp \leqslant m_i$). So $lcp(\sigma^i, \sigma^q) \geqslant lcp(\sigma^i, \alpha)$. It follows that

$$rlcp = lcp(\alpha, \sigma^q) = \min(lcp(\sigma^q, \sigma^i), lcp(\sigma^i, \alpha)) = lcp(\sigma^i, \alpha) \tag{7}$$

and hence the value of *rlcp* should remain unchanged.

If $d_i = \mathit{left}$, then $lcp(\sigma^i, \sigma^p) \geqslant lcp(\sigma^i, \sigma^q)$. But, by Lemma 1

$$lcp(\sigma^i, \sigma^p) = \min(lcp(\sigma^i, \alpha), lcp(\alpha, \sigma^p)). \tag{8}$$

If $lcp(\sigma^i, \sigma^p) = lcp(\alpha, \sigma^p)$ then $llcp = lcp(\alpha, \sigma^p) = lcp(\sigma^i, \sigma^p) = m_i$, contradicting the fact that $m_i < llcp$. Hence $lcp(\sigma^i, \sigma^p) = lcp(\sigma^i, \alpha)$, and $lcp(\sigma^i, \sigma^p) \geqslant lcp(\sigma^i, \sigma^q) \geqslant lcp(\alpha, \sigma^q)$. It follows that *rlcp* should become $m_i$, as claimed. □

There are a further two symmetric cases where, with the appropriate information, the decision to branch left or right can be made without performing any character comparisons.

**Theorem 3.**

(a) *If* $m_i = llcp > rlcp$ *and* $d_i = \mathit{right}$, *then the search path for* $\alpha$ *should branch right from node* $i$; *furthermore the values of rlcp and llcp should remain unchanged.*
(b) *If* $m_i = rlcp > llcp$ *and* $d_i = \mathit{left}$, *then the search path for* $\alpha$ *should branch left from node* $i$; *furthermore the values of rlcp and llcp should remain unchanged.*

**Proof.** We prove only part (a), the proof of (b) being similar. From $llcp = lcp(\alpha, \sigma^p)$ and $rlcp = lcp(\alpha, \sigma^q)$, we have

$$m_i = \max(lcp(\sigma^i, \sigma^p), lcp(\sigma^i, \sigma^q)) = lcp(\alpha, \sigma^p) > lcp(\alpha, \sigma^q). \tag{9}$$

From $d_i = \mathit{right}$ we have

$$lcp(\sigma^i, \sigma^q) \geqslant lcp(\sigma^i, \sigma^p). \tag{10}$$

If $\sigma^q < \alpha < \sigma^i < \sigma^p$, then by Lemma 1 we have

$$\begin{aligned} m_i = llcp = lcp(\alpha, \sigma^p) &= \min(lcp(\alpha, \sigma^i), (\sigma^i, \sigma^p)) \\ &\leqslant lcp(\sigma^i, \sigma^p) \leqslant lcp(\sigma^i, \sigma^q) = \min(lcp(\alpha, \sigma^q), lcp(\alpha, \sigma^i)). \end{aligned} \tag{11}$$

By Lemma 1 we also have $\min(lcp(\alpha, \sigma^i), lcp(\alpha, \sigma^q)) \leqslant lcp(\alpha, \sigma^q) = rlcp$. This is a contradiction. Hence $\sigma^q < \sigma^i < \alpha < \sigma^p$ and the search for $\alpha$ should branch right from node $i$. From $lcp(\alpha, \sigma^q) = rlcp < llcp = m_i = lcp(\sigma^i, \sigma^q)$ it follows that

$$lcp(\alpha, \sigma^q) = \min(lcp(\alpha, \sigma^i), lcp(\sigma^i, \sigma^q)) = lcp(\alpha, \sigma^i). \tag{12}$$

Hence the value of *rlcp* remains unchanged. It is immediate that the value of *llcp* remains unchanged, since there is no new left branch to consider. □

Of course there will be cases where these theorems do not apply. If none of the above theorems applies (e.g., in the initial case, when $m_i = llcp = rlcp = 0$) then character comparisons must be performed to determine the direction in which to branch. The remaining cases are covered by Theorem 4.

**Theorem 4.** (a) *If* $m_i = llcp = rlcp$, *or* (b) *if* $m_i = llcp > rlcp$ *and* $d_i = \mathit{left}$, *or* (c) *if* $m_i = rlcp > llcp$ *and* $d_i = \mathit{right}$, *then character comparisons must be performed to determine* *the direction of branching. If the search branches right from node $i$, say to node $j$, then the value of llcp remains unchanged and the value of rlcp becomes equal to lcp*$(\alpha, \sigma^i)$. *Otherwise* (*the search branches left*), *the value of rlcp remains unchanged, and the value of llcp becomes equal to lcp*$(\alpha, \sigma^i)$.

**Proof.** Suppose $m_i = \max(llcp, rlcp) = t$. In all of the above cases, we know that $\sigma^i$ and $\alpha$ have a common prefix of length $t$, but we have no information about the characters in position $t + 1$. Character comparisons are therefore necessary in these cases. Suppose that $\alpha < \sigma^i$, so that the search path branches left from node $i$ to node $j$. (The argument is similar if $\alpha > \sigma^i$ and the search branches right.) As there is no new right branch, it is immediate that the value of *rlcp* remains unchanged. Node $i$ is the last node on the path to $j$ from which the search branched left, so the value of *llcp* becomes *lcp*$(\alpha, \sigma^i)$. □

We can now use the preceding theorems to describe a more efficient algorithm for searching in an SBST. In so doing, we note that no actual reference is needed to the closest ancestor nodes $cla_i$ and $cra_i$, though the current *llcp* and *rlcp* values must be maintained throughout.

We refer to this improved search algorithm as the *standard search* algorithm. A pseudo-code description of the algorithm appears in Fig. 1. Here, the children of a node $i$ are represented as $lchild_i$ and $rchild_i$, which are assumed to be suffix numbers, with zero playing the role of a null child.

**Example.** Fig. 2 shows an example of a suffix binary search tree for the 15-long string `CAATCACGGTCGGAC`. Each node contains the suffix number $i$ together with the values of $m_i$ and $d_i$.

Consider searching this tree for the string `CGGA`.

- At the root, node 1, we make one equal and one unequal character comparison, branching right with $llcp = 0$ and $rlcp = 1$.
- At node 4, because $m_4 < \max(llcp, rlcp)$, we apply Theorem 2(b) to branch left with *llcp* and *rlcp* unchanged.
- At node 5, because $m_5 > \max(llcp, rlcp)$, we apply Theorem 1 to branch right with *llcp* and *rlcp* unchanged.
- At node 7 we make two equal and one unequal character comparisons, branching left with $llcp = 3$ and *rlcp* unchanged.
- Finally at node 11, one further equal character comparison reveals that the search pattern is present in the string beginning at position 11.

## *2.3. Analysis*

Each time the loop is iterated, at least one of the following occurs:

- the search descends one level in the tree;
- the value of *llcp* is increased;
- the value of *rlcp* is increased.

```
- - Algorithm to search for an occurrence of $\alpha$ in the SBST $T$;
- - returns its starting position in $\sigma$, or zero if there is none.
begin
  $i$ := Root of $T$; $llcp := 0$; $rlcp := 0$;
  while $i \neq$ null loop
    if $m_i > \max(llcp, rlcp)$ then
      $i$ := appropriate child of $i$; - - by Theorem 1
    elsif $m_i < \max(llcp, rlcp)$ then
      if $llcp > rlcp$ then - - by Theorem 2(a)
        $i := rchild_i$;
        if $d_i = left$ then
          $rlcp := m_i$;
        end if;
      elsif $rlcp > llcp$ then - - by Theorem 2(b)
        $i := lchild_i$;
        if $d_i = right$ then
          $llcp := m_i$;
        end if;
      end if;
    elsif $m_i = llcp$ and $llcp > rlcp$ and $d_i = right$ then - - by Theorem 3(a)
      $i := rchild_i$;
    elsif $m_i = rlcp$ and $rlcp > llcp$ and $d_i = left$ then - - by Theorem 3(b)
      $i := lchild_i$;
    else - - by Theorem 4
      $t := \max\{k{:}\ \alpha(m_i + 1 \ldots k) = \sigma(m_i + i \ldots k + i - 1)\}$;
      if $t = |\alpha|$ then
        return $i$;
      elsif $t + i - 1 = n$ or else $\alpha(t + 1) > \sigma(t + i)$ then
        $i := rchild_i$;
        $rlcp := t$;
      else
        $i := lchild_i$;
        $llcp := t$;
      end if;
    end if;
  end loop;
  return 0;
end;
```

Fig. 1. A standard search algorithm for an SBST.

Further, $\max(llcp, rlcp)$ never decreases in value. So the total number of iterations of the loop is at most $h + 2|\alpha|$. In addition, no character in $\alpha$ is ever involved more than once in an equality comparison, so the total number of such comparisons in all calls of the max function is bounded by $|\alpha|$, and the number of inequality comparisons is bounded by the number of loop iterations. Hence the overall complexity of the standard search algorithm is $O(|\alpha| + h)$, and we can expect $h$ to be $O(\log n)$, on average for random strings or on typical plain text, where $n$ is the number of nodes (i.e., the length of the string $\sigma$). In fact, as we shall see in Section 4, it is possible to maintain the SBST as an AVL tree during its construction, thereby enabling us to guarantee that $h = O(\log n)$.

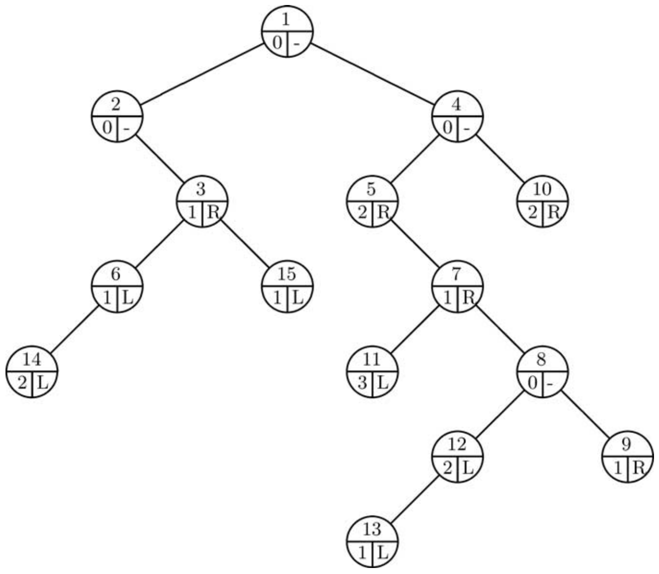

Fig. 2. An SBST for string `CAATCACGGTCGGAC`.

## 2.4. Locating all occurrences

Given an SBST, $T_\sigma$, for a string $\sigma$, and a pattern $\alpha$, the function *Pos* determines whether $\alpha$ is a substring of $\sigma$, and if successful returns a position, say $k$, in $\sigma$ where $\alpha$ occurs. If we require *all* the positions in $\sigma$ where $\alpha$ occurs, then it suffices to partially traverse the subtree rooted at node $k$, since all occurrences will be represented in that subtree.

Suppose that we have reached a node $j$ in that subtree and we know whether $j$'s closest left and right ancestors represent occurrences of $\alpha$. The following two observations are immediate:

(a) if $j$'s closest left ancestor and $j$'s closest right ancestor represent occurrences of $\alpha$ then all nodes in the subtree rooted at $j$ also represent occurrences of $\alpha$;
(b) if neither $j$'s closest left ancestor nor $j$'s closest right ancestor represent occurrences of $\alpha$ then both represent strings $> \alpha$ or both represent strings $< \alpha$, so that no nodes in the subtree rooted at $j$ can represent an occurrence of $\alpha$.

Consider the case where $j$'s closest left ancestor represents an occurrence of $\alpha$ but its closest right ancestor does not (the case where only the right ancestor represents an occurrence of $\alpha$ may be treated analogously). If $m_j \geqslant |\alpha|$ and $d_j = \textit{left}$, then node $j$ represents an occurrence of $\alpha$. In this case, it follows from (a) that all nodes in $j$'s right subtree also represent occurrences of $\alpha$. The nodes in $j$'s left subtree can be resolved recursively. If $d_j = \textit{right}$, or if $m_j < |\alpha|$ then $j$ does not represent an occurrence of $\alpha$. In view of (b) then, it follows that no node in $j$'s left subtree can represent an occurrence of $\alpha$. The nodes in $j$'s right subtree can be resolved recursively.

These observations lead to a recursive algorithm to partially traverse the subtree in question, identifying those nodes that represent occurrences of $\alpha$. Furthermore, the traversal visits only those nodes that cannot, a priori, be eliminated from consideration, and is optimal in this sense, although, in the worst case, it may visit every node in the subtree even when there is only one occurrence of the pattern in the string.

## 3. Building an SBST

### 3.1. Using the standard search algorithm

Clearly there are many possible SBSTs for a given string. An SBST for $\sigma$ can be built in the same way as a binary search tree, namely by a sequence of insertions of all of the suffixes of $\sigma$, in any order, into an initially empty tree. We assume, however, that the suffixes are inserted in left to right order. We will see subsequently that this enables us to add a refinement to the construction algorithm. For the moment we will concentrate on the process of building the SBST with the correct $m$ and $d$ values stored at each node.

The process of repeated insertion of all suffixes of $\sigma$ begins with the creation of a root node representing $\sigma^1$, with $m_1 = 0$. Observe that the search algorithm described in the previous section requires little modification to perform the task of insertion. Instead of searching for a string $\alpha$ in $T_\sigma$, we ask it to search for $\sigma^{k+1}$ in a binary search tree containing the first $k$ suffixes of $\sigma$, and the search will terminate at the location where $\sigma^{k+1}$ should be inserted. Such a search will also make available, as a by-product, the values $m_{k+1}$ and $d_{k+1}$. To be precise, the former will be max(*llcp*, *rlcp*) and, by definition, the latter may be taken to be *left* if *llcp* $>$ *rlcp*, and *right* otherwise.

### 3.2. A 'partial' SBST

It is particularly straightforward to build an SBST that includes only a restricted set of the suffixes of a given string. The processes involved in constructing suffix trees and suffix arrays, differ from those involved in building SBSTs in this respect. The standard construction of an SBST by repeated insertion of suffixes is not dependent on the fact that all suffixes of the string are inserted.

This means that the standard construction algorithm requires little modification to build a structure holding only a proper subset of the suffixes of a given string. This could be appropriate, for example, in text processing where we may be interested only in suffixes marking the start of a new word.

We denote the set of characters of interest, the so-called *word set* by $\mathcal{C}$, and define the suffixes of interest to be those that begin with a character in $\mathcal{C}$ but are not immediately preceded by such a character. We denote the *partial* SBST for this set of suffixes by $T_\sigma(\mathcal{C})$. For a given string $\sigma$ and set of characters $\mathcal{C}$, $T_\sigma(\mathcal{C})$ will clearly require less space than $T_\sigma(\mathcal{C})$ by a factor of some $1 + w$, where $w$ is the average 'word length' in the text, and we can also expect a reduction in the time for construction by a similar factor.

## *3.3. A refined SBST build algorithm*

Empirical evidence (Section 5) suggests that the standard SBST construction algorithm performs well in practice for typical strings. However, regardless of the shape of the tree, insertion of the $i$th suffix of an $n$-long string may require as many as $\min(i, n+1-i)$ comparisons. The worst case complexity of this tree building algorithm is therefore no better than $O(n^2)$. For example, consider the use of this algorithm to construct an SBST for a string $\sigma$ of length $n$ that is a square (i.e., $n$ even, $\sigma_{n/2+i} = \sigma_i$ for all $i$, $1 \leqslant i \leqslant n/2$).

Fortunately, an improvement exploiting the relationship between the suffixes to be inserted is possible. This results in an algorithm whereby the tree is built in $O(nh)$ time in the worst case, where $h$ is the height of the tree.

We incorporate into our SBST, for each node $i$,[2]

- a suffix link, $s_i$, i.e., an explicit pointer from node $i$ to node $i+1$;
- a closest ancestor link $z_i$; i.e., an explicit pointer from node $i$ to the closest ancestor node $j$ such that $lcp(\sigma^i, \sigma^j) = m_i$ (and $i$ is in the subtree of node $j$ corresponding to the value of $d_i$, i.e., if $d_i = left$, then $z_i = cra_i$ and if $d_i = right$, then $z_i = cla_i$).

We define the *start node* for the insertion of suffix $\sigma^{i+1}$, denoted $st_{i+1}$, as follows:

$$st_{i+1} = \begin{cases} \text{the root} & \text{if } m_i \leqslant 1, \\ \text{node } s_{z_i} & \text{if } m_i > m_{s_{z_i}} + 1, \\ \text{node } k & \text{otherwise, where } k \text{ is the first node on a path of closest} \\ & \text{ancestor links from node } s_{z_i} \text{ for which } m_i > m_k + 1. \end{cases}$$

Such a node is guaranteed to exist, because in the worst case, the root can take on the role of node $k$. We now establish that suffix $\sigma^{i+1}$ must be inserted in the subtree rooted at its start node.

**Lemma 3.** *In all cases, $lcp(\sigma^{i+1}, \sigma^{st_{i+1}}) \geqslant m_i - 1$.*

**Proof.** If $m_i \leqslant 1$ then the result is trivial. Otherwise, node $st_{i+1}$ is reached from node $s_{z_i}$ by following a sequence of zero or more closest ancestor links, each of which is to a node for which the first $m_i - 1$ characters of the suffix are unchanged. Hence

$$\sigma^{st_{i+1}}(1 \ldots m_{i-1}) = \sigma^{s_{z_i}}(1 \ldots m_{i-1}) = \sigma^{i+1}(1 \ldots m_{i-1}). \qquad \square$$

**Lemma 4.** *The insertion point for suffix $\sigma^{i+1}$ is in the subtree rooted at node $st_{i+1}$.*

**Proof.** If $st_{i+1}$ is the root, then the lemma holds trivially. Otherwise, it suffices to show that there can be no ancestor node $j$ of $st_{i+1}$ such that $\sigma^{i+1} < \sigma^j < \sigma^{st_{i+1}}$ or $\sigma^{st_{i+1}} < \sigma^j < \sigma^{i+1}$.

If this were the case it would follow that $lcp(\sigma^{i+1}, \sigma^{st_{i+1}}) \leqslant lcp(\sigma^j, \sigma^{st_{i+1}})$. But $lcp(\sigma^j, \sigma^{st_{i+1}}) \leqslant m_{st_{i+1}} < m_i - 1$, and Lemma 3 gives a contradiction. $\square$

[2] Except node $n$, which has no suffix link.

Lemmas proved in Section 2 indicate how to branch from each node on the search path from the root to a leaf during the insertion of suffix $i+1$. We now describe how the search for the insertion point for $\sigma^{i+1}$ is initiated from the start node $st_{i+1}$. In so doing, we observe that, at any point during this search, we require only the larger of the current *llcp* and *rlcp* values, the value of the smaller being irrelevant. The refined algorithm for building an SBST therefore requires only a slight modification to the algorithm described in the previous section.

**Lemma 5.**

(a) *If $st_{i+1}$ is the root, then the search begins as in the case of the standard SBST, with $llcp = rlcp = 0$, and no characters matched*;
(b) *if $st_{i+1} = s_{z_i}$ and $d_i = left$, then we branch left from node $st_{i+1}$, set $rlcp = 0$, and $llcp = m_i - 1$*;
(c) *if $st_{i+1} = s_{z_i}$ and $d_i = right$, then we branch right from node $st_{i+1}$, set $llcp = 0$, and $rlcp = m_i - 1$*;
(d) *otherwise, if $st_{i+1} = k$, so that $\sigma^{i+1}(1 \ldots m_i - 1) = \sigma^k(1 \ldots m_i - 1)$, then comparison of characters from position $m_i$ in these 2 suffixes will reveal whether to branch left or right, and the appropriate value of llcp or rlcp.*

**Proof.** We prove only (b) and (d), the proof of (a) being trivial, and the proof of (c) similar to that of (b).

(b) Because $d_i = left$, we have $\sigma^i < \sigma^{z_i}$. Since $\sigma_i = \sigma_{z_i}$ it follows that $\sigma^{i+1} < \sigma^{z_i+1} = \sigma^{st_{i+1}}$, and so the search should branch left from node $st_{i+1}$. In addition, we know that $lcp(\sigma^{i+1}, \sigma^{st_{i+1}}) = lcp(\sigma^i, \sigma^{z_i}) - 1 = m_i - 1$, so that *llcp* should be set to this value, and *rlcp*, the true value of which cannot be larger, can remain as zero.

(d) Because we know that $\sigma^{i+1}(1 \ldots m_i - 1) = \sigma^k(1 \ldots m_i - 1)$, we need only compare the substrings $\sigma^{i+1}(m_i \ldots |\sigma|)$ and $\sigma^k(m_i \ldots |\sigma|)$ to decide the direction in which to branch. Suppose we match $m$ characters of these two substrings, and we find that $\sigma^{i+1}(m_i \ldots |\sigma|) < \sigma^k(m_i \ldots |\sigma|)$ (and similarly if the inequality is the other way). Then we branch left from node $k$, with *llcp* set to $m_i + m - 1$, and *rlcp* set to zero. □

## *3.4. Analysis*

Since the search paths for the insertion of many suffixes are likely to be shorter than in the standard algorithm, this refined algorithm can be expected to reduce the average time taken to build a suffix BST in practice. Indeed, the empirical results in Section 5 seem to indicate a significant improvement. What has been achieved, though, in terms of the worst case time complexity? The following lemmas allow us to show that the refined construction algorithm also gives an improvement in this respect.

**Lemma 6.** *During the entire execution of the refined construction algorithm, no more than $O(L)$ unequal character comparisons are made, where $L$ is the path length of the final tree.*

**Proof.** This follows at once from the observation that, during the insertion of each suffix, at most one unequal character comparison takes place at each node on the path. □

**Lemma 7.** *During the entire execution of the refined construction algorithm, no more than* $\mathrm{O}(n)$ *equal character comparisons are made, where* $n$ *is the length of the string.*

**Proof.** During the insertion of suffix $i$, no equality comparisons involving character $\sigma_{i+r}$ are made, for any $r > 0$, if that character was involved in an equal character comparison during the insertion of any previous suffix. Suppose, on the contrary, that an equality comparison involving $\sigma_{i+r}$ was made during the insertion of suffix $i - t$, for some $t \geqslant 1$. Then it is immediate that $m_{i-t} \geqslant r + t + 1$. Hence, during the insertion of suffix $i - t + 1$, that suffix $i$ and suffix $st_{i-t+1}$ had a common prefix of length $m_{i-t}$, and hence no comparisons involving $\sigma_{i+r}$ would be made. The argument extends inductively to the insertion of suffix $i$, giving a contradiction.

It follows that, during the refined construction, each character in $\sigma$ is involved in at most one equality comparison with a character that precedes it in $\sigma$, and so the total number of equality comparisons is $\mathrm{O}(n)$, as claimed. □

**Theorem 5.** *Using the refined algorithm, an SBST* $T_\sigma$ *for an* $n$*-long string* $\sigma$ *can be constructed in* $\mathrm{O}(nh)$ *time in the worst case, where* $h$ *is the height of the tree.*

**Proof.** The complexity of the algorithm is determined by two factors, namely the number of character comparisons and the number of node-to-node steps taken in the tree. Lemmas 6 and 7 together establish that the total number of character comparisons is $\mathrm{O}(L) = \mathrm{O}(nh)$, where $L$ is the *path length* of the tree (since, for the latter, it is immediate that $n = \mathrm{O}(L)$). As far as steps in the tree are concerned, consider the insertion of any particular node $i + 1$. The number of downward steps taken during the insertion of this node cannot exceed the distance of the node from the root, while the number of upward steps cannot exceed the height of the tree. Hence the total number of steps, summed over all insertions, is $\mathrm{O}(nh)$.[3] □

## 4. The suffix AVL tree

On average, an SBST will be reasonably well balanced, and the expected height will be $\mathrm{O}(\log n)$, but will inevitably be no better than $\mathrm{O}(n)$ in the worst case. So the question arises whether some standard tree balancing technique can be used to guarantee that the tree has logarithmic height, while not adversely affecting the complexity of tree construction. In this section, we explore the *suffix AVL tree*, i.e., the suffix binary search tree balanced using rotations as in classical AVL trees [10].

Recall that, in an AVL tree, the heights of the left and right subtrees of every node differ by at most one. If the tree becomes unbalanced by the insertion of a new node, a *rotation* is

[3] In fact, we conjecture that the appropriate worst case time bound is $\mathrm{O}(L)$, but we lack a proof that the total number of upward steps in the tree satisfies this bound.

Table 1
The updated values of $m$, $d$, and $z$ after a single left rotation

| $d_a$ | $d_b$ | $lca_a$ | $lca_b$ | $m'_a$ | $m'_b$ | $d'_a$ | $d'_b$ | $z'_a$ | $z'_b$ |
|---|---|---|---|---|---|---|---|---|---|
| $l$ | $l$ | $f$ | $f$ | $m_a$ | $m_b$ | $d_a$ | $d_b$ | $b$ | $f$ |
| $l$ | $r$ | $f$ | $a$ | $m_b$ | $m_a$ | $d_a$ | $\neg d_b$ | $b$ | $f$ |
| $r$ | $l$ | $g$ | $f$ | $m_a$ | $m_b$ | $d_a$ | $d_b$ | $g$ | $f$ |
| $r$ | $r$ | $g$ | $a$ | $\max(m_a, m_b)$ | $\min(m_a, m_b)$ | $\begin{cases} d_a & \text{if } m_a \geqslant m_b \\ \neg d_a & \text{otherwise} \end{cases}$ | $d_b$ | $\begin{cases} g & \text{if } m_a \geqslant m_b \\ b & \text{otherwise} \end{cases}$ | $g$ |

performed, and the balance property is restored. There are essentially four possible kinds of rotations, a *single left* rotation, a *double left* rotation, and the mirror images of these two cases a *single right* and a *double right* rotation. In fact, a double rotation can be envisaged as the composition of two single rotations, a fact that we exploit in what follows. After an insertion has been performed, at most one (single or double) rotation is required to restore the AVL balance property.

It is well known that the sparsest possible AVL trees are Fibonacci trees, which are of height approximately $1.44\log_2 n$, for a tree with $n$ nodes, so that every AVL tree has height $O(\log n)$.

AVL rotations can easily be applied to balance a naive SBST in which only suffix numbers are stored at the nodes. However, in our standard SBSTs, each node contains two other values that are tightly coupled to the structure of the tree, and in the refined version there are a further two such values. Some or all of the $m_i$, $d_i$, and $z_i$ values may change as a result of a rotation that affects the ancestors of node $i$. (It should be clear however, that the $s_i$ values do not pose a problem in this respect.) Furthermore, it is not immediately obvious whether enough information is available to enable the correct $m$, $d$, and $z$ values for affected nodes to be recalculated without significantly increasing the time complexity.

## 4.1. Balancing the SBST subtree

Suppose that we have a suffix AVL tree containing the first $i$ suffixes of $\sigma$, and we are about to use the refined insertion algorithm to insert the suffix $\sigma^{i+1}$ into the subtree rooted at node $st_{i+1}$. We concentrate only on the subtree rooted at $st_{i+1}$ for the moment, and in the next subsection we describe how to ensure that the entire tree retains the AVL property.

It turns out that, for our proposed suffix AVL subtree,

- after a single left or single right rotation, at most one $d$ value, two $z$ values, and two $m$ values need to be updated, and this can be achieved in constant time;
- after a double left or double right rotation, at most two $d$ values, three $z$ values and three $m$ values need to be updated, and this can also be achieved in constant time.

We will prove in detail the results for a single rotation. Because a double rotation can be viewed as a sequence of two single rotations, it follows at once that a double rotation can also be achieved in constant time. However, although we state the rules for updating the $d$, $z$ and $m$ values, we will omit the details of the proof.

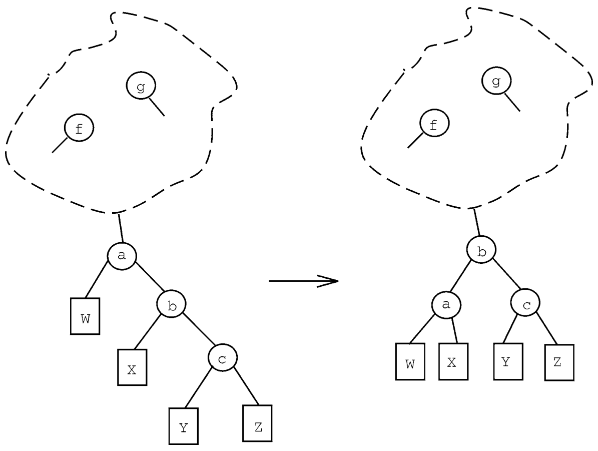

Fig. 3. A single left AVL rotation.

In the following, we consider the effect of some particular rotation in $T_\sigma$. We use the symbol $'$ to indicate the (possibly altered) value of a parameter after the rotation has been carried out; for example we refer to $m'_i$, $d'_i$, $cla'_i$, $cra'_i$, etc. We represent the opposite of direction $d_i$ by $\neg d_i$, i.e., $\neg right = left$ and $\neg left = right$.

The following lemma is trivial to verify (although it does depend on our assumption that, when $lcp(\sigma^i, \sigma^{cla_i}) = lcp(\sigma^i, \sigma^{cra_i})$, we can choose $d_i$ to be either *left* or *right*.

**Lemma 8.** *If $cla'_i = cla_i$ and $cra'_i = cra_i$ then $m'_i = m_i$ and $d'_i = d_i$.*

The next theorem characterises the alterations required to accomplish a single rotation. The context is given in Fig. 3.

**Theorem 6.** *Consider a single left rotation pivoted at node $a$, and let $b$ be the right child of node $a$. Then*

(i) *the values of $m_i$, $z_i$, and $d_i$ are unchanged for all nodes $i$ other than $a$ and $b$*;
(ii) *the new $m$, $z$, and $d$ values for nodes $a$ and $b$ are as presented in Table* 1.

**Proof.** (i) For all nodes $i$ in the tree, excluding nodes $a$ and $b$, $cla'_i = cla_i$ and $cra'_i = cra_i$. It follows from Lemma 8 that for these nodes, $d'_i = d_i$ and $m'_i = m_i$. It follows also that for these nodes, $z'_i = z_i$.

(ii) Let the closest left and right ancestors of node $a$ be nodes $g$ and $f$ respectively. (It is easy to verify that the results of the theorem continue to hold in the special cases in which either or both of these do not exist.)

We first observe that, once the values of $d'_a$ and $d'_b$ are established, the values of $z'_a$ and $z'_b$ follow immediately. For example, $z'_a$ is equal to $b$ or $g$ according as $d'_a$ is *left* or *right*, and similarly for $z'_b$.

Within the binary search tree we have the lexicographic ordering

$$\sigma^g < \sigma^a < \sigma^b < \sigma^f. \tag{13}$$

Subcase ii(a) Suppose $d_b = \mathit{left}$ (as in lines 1 and 3 of Table 1); then

$$m_b = lcp(\sigma^b, \sigma^f) \geqslant lcp(\sigma^b, \sigma^a). \tag{14}$$

From Lemma 1, (13) and (14), it follows that

$$lcp(\sigma^a, \sigma^f) = \min(lcp(\sigma^a, \sigma^b), lcp(\sigma^b, \sigma^f)) = lcp(\sigma^b, \sigma^a). \tag{15}$$

It can be seen from this, and the definitions of $m'_a$ and $m_a$, that

$$\begin{aligned} m'_a &= \max(lcp(\sigma^a, \sigma^b), lcp(\sigma^a, \sigma^g)) \\ &= \max(lcp(\sigma^a, \sigma^f), lcp(\sigma^a, \sigma^g)) = m_a. \end{aligned} \tag{16}$$

It is immediate from (16) that $d'_a = d_a$. From (14), the definitions of $m'_b$ and $m_b$ and the knowledge from (13) that $lcp(\sigma^b, \sigma^a) \geqslant lcp(\sigma^b, \sigma^g)$, we have

$$m'_b = \max(lcp(\sigma^b, \sigma^f), lcp(\sigma^b, \sigma^g)) = lcp(\sigma^b, \sigma^f) = m_b. \tag{17}$$

From this, it is immediate that $d'_b = d_b$.

Subcase ii(b) Suppose $d_a = \mathit{left}$ and $d_b = \mathit{right}$ (as in line 2 of Table 1); then

$$m_a = lcp(\sigma^a, \sigma^f) \geqslant lcp(\sigma^a, \sigma^g) \tag{18}$$

and

$$m_b = lcp(\sigma^b, \sigma^a) \geqslant lcp(\sigma^b, \sigma^f). \tag{19}$$

From (19), (13) and (18), it follows that

$$lcp(\sigma^a, \sigma^b) \geqslant lcp(\sigma^b, \sigma^f) \geqslant lcp(\sigma^a, \sigma^f) \geqslant lcp(\sigma^a, \sigma^g). \tag{20}$$

From (20) and the definitions of $m_b$ and $m'_a$, we obtain

$$m'_a = \max(lcp(\sigma^a, \sigma^b), lcp(\sigma^a, \sigma^g)) = lcp(\sigma^a, \sigma^b) = m_b. \tag{21}$$

It is immediate from (21) that $d'_a = \mathit{left} = d_a$. It follows from (13), Lemma 1, and (19) that

$$lcp(\sigma^a, \sigma^f) = \min(lcp(\sigma^a, \sigma^b), lcp(\sigma^b, \sigma^f)) = lcp(\sigma^b, \sigma^f). \tag{22}$$

It is immediate from (13), Lemma 1, and (16) that

$$lcp(\sigma^b, \sigma^g) = \min(lcp(\sigma^a, \sigma^g), lcp(\sigma^a, \sigma^b)) = lcp(\sigma^a, \sigma^g). \tag{23}$$

From (22), (23) and the definitions of $m_a$ and $m'_b$, we obtain

$$\begin{aligned} m'_b &= \max(lcp(\sigma^b, \sigma^f), lcp(\sigma^b, \sigma^g)) \\ &= \max(lcp(\sigma^a, \sigma^f), lcp(\sigma^a, \sigma^g)) = m_a. \end{aligned} \tag{24}$$

From this it is immediate that $d'_b = d_a = \neg d_b$.

Subcase ii(c) Suppose $d_a = d_b = right$ (as in line 4 of Table 1); then

$$m_a = lcp(\sigma^a, \sigma^g) \geqslant lcp(\sigma^a, \sigma^f) \tag{25}$$

and

$$m_b = lcp(\sigma^b, \sigma^a) \geqslant lcp(\sigma^b, \sigma^f). \tag{26}$$

From (25), (26) and the definition of $m'_a$, it follows that

$$m'_a = \max(lcp(\sigma^a, \sigma^b), lcp(\sigma^a, \sigma^g)) = \max(m_a, m_b). \tag{27}$$

From (27), it follows that $d'_a = right = d_a$ if $m_a \geqslant m_b$, and $d'_a = left = \neg d_a$ otherwise. From (13), Lemma 1, (25) and (26), we obtain

$$lcp(\sigma^b, \sigma^g) = \min(lcp(\sigma^a, \sigma^g), lcp(\sigma^a, \sigma^b)) = \min(m_a, m_b). \tag{28}$$

Also by (13), Lemma 1, and (26), it follows that

$$lcp(\sigma^a, \sigma^f) = \min(lcp(\sigma^a, \sigma^b), lcp(\sigma^b, \sigma^f)) = lcp(\sigma^b, \sigma^f). \tag{29}$$

Eqs. (28) and (29) and the definition of $m'_b$ give us

$$m'_b = \max(lcp(\sigma^b, \sigma^f), lcp(\sigma^b, \sigma^g)) = \max(lcp(\sigma^b, \sigma^f), \min(m_a, m_b)). \tag{30}$$

From (25) and (29), we obtain

$$m_a = lcp(\sigma^a, \sigma^g) \geqslant lcp(\sigma^a, \sigma^f) = lcp(\sigma^b, \sigma^f). \tag{31}$$

From (26) we know that $m_b \geqslant lcp(\sigma^b, \sigma^f)$. This, together with (31), gives us

$$lcp(\sigma^b, \sigma^f) \leqslant \min(m_a, m_b). \tag{32}$$

So, from (30) and (32), we obtain

$$m'_b = \min(m_a, m_b). \tag{33}$$

From (28) and (33), we obtain

$$m'_b = \min(m_a, m_b) = lcp(\sigma^b, \sigma^g), \tag{34}$$

and from this it follows that $d'_b = right = d_b$. $\square$

Corresponding to Theorem 6 and Table 1 there is, of course, an exactly analogous theorem and corresponding table for the case of a single right rotation. We omit the details. The next theorem characterises the alterations required to accomplish a double rotation. The context is given in Fig. 4.

**Theorem 7.** *Consider a double left rotation pivoted first at node b, then at node a, and let c be the left child of b. Then,*

(i) *the values of $m_i$, $z_i$, and $d_i$ are unchanged for all nodes $i$ other than $a$, $b$ and $c$;*
(ii) *the new $m$, $z$, and $d$ values for nodes $a$, $b$ and $c$ are as presented in Tables* 2 *and* 3.

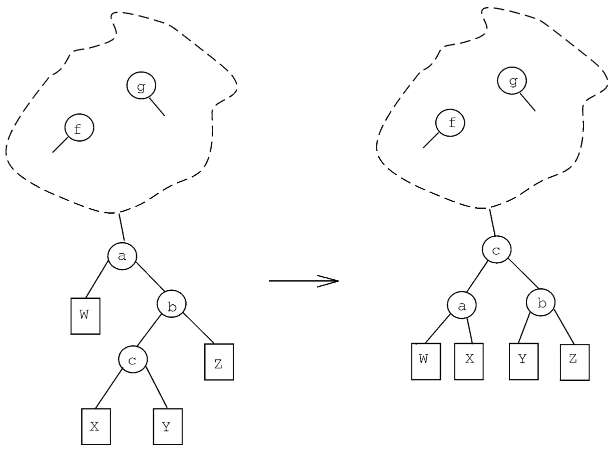

Fig. 4. A double left AVL rotation.

Table 2
The updated values of $m$ and $d$ after a double left rotation

| $d_a$ | $d_b$ | $d_c$ | $m'_a$ | $m'_b$ | $m'_c$ | $d'_a$ | $d'_b$ | $d'_c$ |
|---|---|---|---|---|---|---|---|---|
| $l$ | $l$ | $l$ | $m_a$ | max $(m_b, m_c)$ | min $(m_b, m_c)$ | $d_a$ | $\begin{cases} d_b & \text{if } m_b \geqslant m_c \\ \neg d_b & \text{otherwise} \end{cases}$ | $d_c$ |
| $l$ | $l$ | $r$ | $m_c$ | $m_b$ | $m_a$ | $d_a$ | $d_b$ | $\neg d_c$ |
| $l$ | $r$ | $l$ | $m_b$ | $m_c$ | $m_a$ | $d_a$ | $d_b$ | $d_c$ |
| $l$ | $r$ | $r$ | $m_c$ | $m_b$ | $m_a$ | $d_a$ | $d_b$ | $\neg d_c$ |
| $r$ | $l$ | $l$ | $m_a$ | max $(m_b, m_c)$ | min $(m_b, m_c)$ | $d_a$ | $\begin{cases} d_b & \text{if } m_b \geqslant m_c \\ \neg d_b & \text{otherwise} \end{cases}$ | $d_c$ |
| $r$ | $l$ | $r$ | max $(m_a, m_c)$ | $m_b$ | min $(m_a, m_c)$ | $\begin{cases} d_a & \text{if } m_a \geqslant m_c \\ \neg d_a & \text{otherwise} \end{cases}$ | $d_b$ | $d_c$ |
| $r$ | $r$ | $l$ | max $(m_a, m_b)$ | $m_c$ | min $(m_a, m_b)$ | $\begin{cases} d_a & \text{if } m_a \geqslant m_b \\ \neg d_a & \text{otherwise} \end{cases}$ | $d_b$ | $\neg d_c$ |
| $r$ | $r$ | $r$ | max $(m_a, m_c)$ | $m_b$ | min $(m_a, m_c)$ | $\begin{cases} d_a & \text{if } m_a \geqslant m_c \\ \neg d_a & \text{otherwise} \end{cases}$ | $d_b$ | $d_c$ |

As observed earlier, we omit the proof of this theorem for the sake of brevity. Full details can be found in [5].

Once again, there are analogues corresponding to Theorem 7 and Tables 2 and 3 for the case of a double right rotation.

## 4.2. Balancing the entire tree

We now show that, in the worst case, the balance property of the entire tree can be restored in $O(h)$ time, where $h = O(\log n)$ is the height of the tree.

Table 3
The updated values of $z$ after a double left rotation

| $d_a$ | $d_b$ | $d_c$ | $z_a$ | $z_b$ | $z_c$ | $z'_a$ | $z'_b$ | $z'_c$ |
|---|---|---|---|---|---|---|---|---|
| $l$ | $l$ | $l$ | $f$ | $f$ | $b$ | $c$ | $\begin{cases} f & \text{if } m_b \geqslant m_c \\ c & \text{otherwise} \end{cases}$ | $f$ |
| $l$ | $l$ | $r$ | $f$ | $f$ | $a$ | $c$ | $f$ | $f$ |
| $l$ | $r$ | $l$ | $f$ | $a$ | $b$ | $c$ | $c$ | $f$ |
| $l$ | $r$ | $r$ | $f$ | $a$ | $a$ | $c$ | $c$ | $f$ |
| $r$ | $l$ | $l$ | $g$ | $f$ | $b$ | $c$ | $\begin{cases} f & \text{if } m_b \geqslant m_c \\ c & \text{otherwise} \end{cases}$ | $f$ |
| $r$ | $l$ | $r$ | $g$ | $f$ | $a$ | $\begin{cases} g & \text{if } m_b \geqslant m_c \\ c & \text{otherwise} \end{cases}$ | $f$ | $g$ |
| $r$ | $r$ | $l$ | $g$ | $a$ | $b$ | $\begin{cases} g & \text{if } m_a \geqslant m_b \\ c & \text{otherwise} \end{cases}$ | $c$ | $g$ |
| $r$ | $r$ | $r$ | $g$ | $a$ | $a$ | $\begin{cases} g & \text{if } m_a \geqslant m_c \\ c & \text{otherwise} \end{cases}$ | $c$ | $g$ |

By proceeding as in the previous subsection, we can be sure that the subtree rooted at $st_{i+1}$ is balanced, but this does not necessarily extend to the entire tree. If the height of that subtree is unchanged as a result of the insertion (possibly following a rotation) then the entire tree will also be balanced, and no ancestors of node $st_{i+1}$ need be considered. But if the height of the subtree has increased then the balance factor of one or more ancestor nodes may have to be updated, and a rotation pivoted at some ancestor node may be necessary. The nodes that may have to be considered are those on the path from $st_{i+1}$ to the root. As soon as we reach a node on this path that is the root of a subtree whose height is unchanged, whether or not a rotation has been carried out to achieve this, we can stop.

So the question arises as to how we access the relevant nodes, starting from node $st_{i+1}$. Suppose we refer to this node as node $j$. We cannot step up the path directly, but we can immediately access the closest ancestor node $z_j$, and knowing the value of $d_j$ enables us to locate the path from $z_j$ to $j$, and therefore the reverse of this, in constant time per node. Hence we can adjust the balance factors of nodes on that path, as necessary, and identify and apply a rotation at one of these nodes should it be required. Even after so doing, if the height of the subtree rooted at $z_j$ has increased, we can apply the same process to that node, and can continue iteratively all the way back to the root should this be necessary. In the event that a rotation is required at whatever stage, the $m$, $z$, and $d$ values can be updated (in constant time) exactly as described previously.

The total number of operations carried out, even in the worst case, during the insertion of a new node and any subsequent updating and rebalancing is bounded by a constant times the distance from the root of the new node. This clearly applies even if we have to step our way back up the tree towards the root by following a sequence of closest ancestor links.

## 4.3. Analysis of suffix AVL tree construction

We have shown that, when a new node is inserted during the construction of a suffix AVL tree, the number of $m$, $z$, and $d$ values that may have to updated is bounded by a constant, and each update can be achieved in constant time. Furthermore adjustments to

Table 4
Construction times using strings of length 1 000 000

| File type | $\|\Sigma\|$ | Construction time | | | | | |
|---|---|---|---|---|---|---|---|
| | | SBSTS | SBSTA | SBSTR | SBSTP | ST | SA |
| Text | 79 | 8.7 | 11.2 | 3.0 | 1.4 | 3.5 | 15.5 |
| DNA | 4 | 8.9 | 11.5 | 2.9 | – | 3.4 | 23.7 |
| Protein | 21 | 9.8 | 12.2 | 4.1 | – | 3.4 | 25.6 |
| Code | 98 | 10.4 | 12.6 | 2.8 | – | 3.1 | 35.8 |
| Random | 4 | 9.1 | 11.6 | 3.1 | – | 3.5 | 8.1 |
| Random | 64 | 9.0 | 11.4 | 8.1 | – | 3.2 | 8.3 |

balance factors of nodes, and any necessary rotation, can be identified and carried out in $O(h)$ time, where $h$ is the height of the tree (even though, in the case of the refined version, the algorithm for achieving this is a little more complicated than for a standard AVL tree).

Since, as for a standard AVL tree, the height of a suffix AVL tree is $O(\log n)$, it follows that a suffix AVL tree can be constructed in $O(n \log n)$ time.

## 5. Empirical results

To evaluate the practical utility of SBSTs, we carried out computational experiments similar to those used in [7] to compare the performance of suffix arrays with that of suffix trees. All programs were compiled with the highest level of optimisation, and were run under Solaris on a 450 Mhz workstation. All cpu times recorded in Tables 4 and 7 are in seconds.

Table 4 summarises the results obtained for the various construction algorithms using strings of 1 000 000 characters. Suffix trees (ST in the tables) were constructed using Kurtz's tightly coded implementations [6], choosing in each case the list or hash-table version, whichever was faster (the list version for DNA and random text with alphabet size 4, the hash-table version in the other case). The suffix array implementation (SA in the tables) was the one used in the experiments of Manber and Myers [7].[4]

Four variants of the SBST were included, namely

- SBSTS—the standard construction algorithm;
- SBSTA—standard construction with AVL balancing;
- SBSTR—the refined construction algorithm;
- SBSTP—the standard construction algorithm for a partial SBST (for text only).

A variety of files were used, namely

- ordinary English plain text (the first million characters of 'War and Peace');
- a DNA sequence;

[4] The authors are grateful to Gene Myers for providing source code for this implementation.

Table 5
Construction statistics using a plain text string of length 1 000 000

| | Construction statistics | | | |
|---|---|---|---|---|
| | SBSTS | SBSTR | SBSTP | ST |
| Nodes created | 1 000 000 | 1 000 000 | 175 454 | 1 518 457 |
| Nodes accessed | 67 047 855 | 8 316 402 | 4 077 277 | 21 265 311 |
| Character comparisons | 44 740 736 | 5 486 249 | 5 886 192 | 18 525 149 |

Table 6
Construction statistics using a DNA string of length 1 000 000

| | Construction statistics | | |
|---|---|---|---|
| | SBSTS | SBSTR | ST |
| Nodes created | 1 000 000 | 1 000 000 | 1 661 657 |
| Nodes accessed | 26 653 063 | 6 751 230 | 12 510 875 |
| Character comparisons | 39 994 578 | 4 379 745 | 11 560 423 |

- a concatenation of protein sequences (with separators);
- program code;
- random strings over alphabets of sizes 4 and 64.

From the table, it is clear that the construction refinement has a significant impact on average performance as well as on worst-case complexity. On the other hand, in spite of the worst-case guarantee provided by suffix AVL-trees, the empirical evidence strongly suggests that the overheads of maintaining balance substantially outweigh the benefits in practice. As expected, the partial SBST is constructed in a fraction of the time required for the full standard SBST.

Tables 5 and 6 give an alternative comparison of the various tree construction algorithms based on counting certain key operations. As well as recording the number of nodes in each structure, this table also indicates the number of nodes accessed and the number of individual character comparisons made during the construction. Table 5 covers the construction of standard, refined, and partial SBSTs, and suffix trees with the children of each node represented as a list, for a plain text file of 1 000 000 characters, and Table 6 covers all but the partial case for a DNA text file of the same length.

Of course, these are not the only operations that affect the running times of the various algorithms—integer and direction comparisons, for example, are also significant in SBST construction. However, the results show the expected significant reduction in nodes accessed and characters compared in the refined algorithm relative to the standard algorithm for SBSTs. The suffix tree has, of course, more nodes, and in terms of node accesses and character comparisons appears to lie intermediate between the standard and refined SBSTs.

Table 7 summarises the results obtained for the various search algorithms. In each case, searches were conducted for all substrings of length 50 of the original string of length 1 000 000. In this table, we include just a single column representing the standard and refined SBSTs, since these two construction algorithms build structurally identical trees.

Table 7
Search times for all substrings of length 50

| File type | $\|\Sigma\|$ | Search time | | |
|---|---|---|---|---|
| | | SBSTS | ST | SA |
| Text | 79 | 9.0 | 12.5 | 8.2 |
| DNA | 4 | 9.3 | 9.6 | 6.2 |
| Protein | 21 | 9.9 | 12.9 | 6.7 |
| Code | 97 | 9.2 | 14.2 | 7.3 |
| Random | 4 | 9.7 | 9.8 | 6.2 |
| Random | 64 | 9.6 | 25.7 | 7.0 |

In this case, the suffix tree implementation is our own tightly coded version, using a list of children at each node.

The table confirms the speed advantage of suffix arrays for on-line string searching, but also shows that the SBST is competitive with the suffix tree in this respect, at least with the version represented using a list of children at each node.

Overall, at least in the particular experiments that were carried out, the SBST performed creditably in comparison with suffix trees and suffix arrays. The results show the refined and partial versions to be particularly competitive on real data sets.

## References

[1] A. Andersson, N.J. Larsson, K. Swanson, Suffix trees on words, Algorithmica 23 (1999) 246–260.
[2] A. Apostolico, The myriad virtues of subword trees, in: A. Apostolico, Z. Galil (Eds.), Combinatorial Algorithms on Words, in: NATO ASI Ser., Vol. F12, Springer, Berlin, 1985, pp. 85–96.
[3] M. Farach, P. Ferragina, S. Muthukrishnan, On the sorting complexity of suffix tree construction, J. Assoc. Comput. Mach. 47 (6) (2000) 987–1011.
[4] D. Gusfield, Algorithms on Strings, Trees, and Sequences, Computer Science and Computational Biology, Cambridge University Press, Cambridge, 1997.
[5] R.W. Irving, L. Love, The suffix binary search tree and suffix AVL tree, Technical Report, Computing Science Department, University of Glasgow, Technical Report TR-2000-54, 2000.
[6] S. Kurtz, Reducing the space requirement of suffix trees, Software Practice Experience 29 (1999) 1149–1171.
[7] U. Manber, G. Myers, Suffix arrays: A new method for on-line string searches, SIAM J. Comput. 22 (5) (1993) 935–948.
[8] E. McCreight, A space-economical suffix tree construction algorithm, J. Assoc. Comput. Mach. 23 (2) (1976) 262–272.
[9] E. Ukkonen, On-line construction of suffix trees, Algorithmica 14 (3) (1995) 249–260.
[10] G. Adel'son Velskii, E. Landis, An algorithm for the organisation of information, Dokl. Akad. Nauk SSSR 146 (1962) 263–266; English translation in Soviet Math. Dokl. 3.
[11] P. Weiner, Linear pattern matching algorithms, in: Proceedings of the IEEE 14th Annual Symposium on Switching and Automata Theory, 1973, pp. 1–11.