Suffix Binary Search Trees and Suffix Arrays

Robert W. Irving and Lorna Love

March 27, 2001

Abstract

Suffix arrays and suffix binary search trees are two data structures that have been proposed as alternatives to the classical suffix tree to facilitate efficient on-line string searching. Here, we explore the relationship between these two structures. In particular, we present an alternative view of a suffix array, with its auxiliary information, as a perfectly balanced suffix binary search tree, and describe an elegant and efficient algorithm to construct the suffix array and its auxiliaries from an arbitrary suffix binary search tree.

1 Introduction

A number of suffix-based data structures have been proposed to facilitate on-line string searching. The best known of these is the classical *suffix tree*, which dates back to the pioneering papers of Weiner [11] and McCreight [8], and which has an amazing variety of uses and applications — see Gusfield [3], for example. Manber and Myers [7] proposed the *suffix array* as a space-efficient alternative to the suffix tree, and Irving and Love [4] defined the *suffix binary search tree* (SBST), and its variant the *suffix AVL tree*, as further alternatives, competitive in practice with suffix trees and suffix arrays. Here, we are concerned with suffix binary search trees and suffix arrays; in particular with the relationship between them. We describe an alternative view of a suffix array as a perfectly balanced suffix binary search tree, and we show that a suffix array can be efficiently constructed from a suffix binary search tree, giving a competitive alternative construction method for suffix arrays.

The so-called *partial* suffix binary search tree is a useful variant of the SBST, particularly when the base string is plain text. This gives a representation of a proper subset of the suffixes of the string — for example, those beginning on a word boundary. Irving and Love show that a minor adaptation of the standard SBST construction algorithm can be used to build partial SBSTs, with a saving in space and time commensurate with the proportion of suffixes actually represented. Andersson et al [1] had previously described a complex method of adapting suffix trees to realise this objective, but no corresponding concept of a 'partial' suffix array has been described in the literature. However, a significant by-product of our SBST to suffix array conversion algorithm is a variant that constructs such a partial suffix array efficiently from a partial SBST — again giving a saving in space (and construction time) of a factor of around $1 + w$, where w is the average word length in the text represented.

Classical algorithms [8, 11, 10] construct a suffix tree for a string of length n in $O(n \log |\Sigma|)$ time, where Σ is the alphabet, and a recent more involved algorithm due to Farach [2] removes the dependence on alphabet size. Given a suffix tree for σ and a pattern α of length m, an algorithm to determine whether the pattern appears in the string can be implemented to run in $O(m \log |\Sigma|)$ time. The corresponding time bounds for construction and search in the case of a suffix array [7] are $O(n \log n)$ and $O(m + \log n)$.

For a suitably implemented SBST [4], a search requires $O(m + l)$ time, where l is the length of the search path in the tree. This gives $O(m + n)$ worst-case complexity, but typically in practice, all search paths will have $O(\log n)$ length, and searching will be $O(m + \log n)$ on average. In fact, this becomes a worst-case bound if we use AVL rotations to balance the tree on construction, which is a feasible, but non-trivial extension [4]. The construction time for our standard SBST can be as bad as $O(n^2)$ in the worst case, but for a refined version, it can be achieved in $O(nh)$ 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 $O(\log n)$, and in the case of the suffix AVL tree, construction can be accomplished in $O(n \log n)$ 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 [3] and by Kurtz [6]. Traditional representations of a suffix tree [8] require $28n$ bytes, in the worst case, but more compact representations are possible[1]. The most economical, due to Kurtz [6], has a worst-case requirement of $20n$ bytes, though empirical evidence suggests a lower requirement in practical cases. For a suffix array, an implementation realising

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

the $O(m + \log n)$ worst-case complexity for search requires $9n$ bytes, although Manber and Myers also describe a variant using just $5n$ bytes which appears to be faster in practice, though giving a worst-case guarantee that is no better than $O(m \log n)$.

The standard implementation of an SBST as an array of nodes uses $10n - 12n$ bytes. For the construction of the refined version, each node requires an additional integer, and, in the case of the suffix AVL tree, two further bits to indicate its *balance factor*.

Empirical evidence presented in [4] shows that the suffix binary search tree, at least in its refined form, is competitive with suffix trees and suffix arrays, both from the point of view of speed of construction and efficiency of on-line searching.

The remainder of this report is structured as follows. In Section 2, we give the definitions associated with suffix arrays, and summarise their characteristics. In Section 3 we similarly summarise the salient features of suffix binary search trees. Section 4 describes how a suffix array can be viewed as a perfectly balanced suffix binary search tree, and Section 5 shows how the former can be efficiently constructed from the latter. Finally, Section 6 presents a summary and some concluding observations.

2 Suffix arrays

Given a finite string $\sigma = \sigma_1\sigma_2 \dots \sigma_n$ of length n over some alphabet Σ, a (non-empty) *substring* of σ is a string $\sigma_i\sigma_{i+1} \dots \sigma_j$ for some i, j $(1 \leq i \leq j \leq n)$. The ith *suffix* σ^i of σ is the substring $\sigma_i\sigma_{i+1} \dots \sigma_n$ of length $n-i+1$. A (non-empty) *prefix* of σ is a substring $\sigma_1\sigma_2 \dots \sigma_j$ for some j $(1 \leq j \leq n)$. It is trivial to observe that an arbitrary string α over Σ occurs as a substring of σ if and only if it occurs as a prefix of one or more suffixes of σ. We denote the length of the search string α by m.

The *suffix array* $A = A(\sigma)$ for σ is the unique permutation of the set $\{1, 2, \dots, n\}$ such that $\sigma^{A(1)} < \sigma^{A(2)} < \dots < \sigma^{A(n)}$. Given the suffix array $A(\sigma)$, it is easy to see that a search for occurrences of α as a substring of σ can be organised as a binary search of the sorted sequence $\sigma^{A(1)}\sigma^{A(2)} \cdots \sigma^{A(n)}$. If j and k denote, respectively, the smallest and largest values of i such that α is a prefix of $\sigma^{A(i)}$, then the values $A(j), A(j+1), \dots, A(k)$ represent all of the positions within σ where there is an occurrence of α. However, the efficiency of such a search can be adversely affected if α has significant prefixes in common with other suffixes of σ, since a 'single' comparison of

α with such a suffix during the binary search may in fact require many (equal) character comparisons prior to the unequal character comparison that determines the direction of branching.

Manber and Myers showed that the potential inefficiency due to equal character comparisons could be avoided, and a worst-case complexity for the search algorithm of $O(m + \log n)$ established, by maintaining two auxiliary arrays of so-called *lcp* (longest common prefix) values. For a given value of i, $llcp(i)$ represents the length of the longest common prefix of $\sigma^{A(i)}$ and $\sigma^{A(j)}$, where j is the left end point of the unique sub-interval of $1, \ldots, n$ with i at its centre, that could become the current subinterval during a binary search. An analogous definition applies for $rlcp(i)$, involving the right end point of this subinterval.

In fact, in a footnote in their paper, Manber and Myers observe that it suffices to maintain one such *lcp* array (of signed values), with a consequent space saving. However, their description of the structure and its search and construction algorithms, and subsequent expositions of the suffix array, as in [3] for example, are presented using the two auxiliary *lcp* arrays.

The Manber and Myers construction algorithm is based on a form of bucket sort. After the ith iteration of the algorithm, suffixes in the same bucket have identical prefixes of length 2^{i-1}, so that $O(\log n)$ iterations suffice to complete the sort. Let $p = 2^{i-1}$. Then each iteration can be achieved in $O(n)$ time by exploiting the fact that, when comparing suffixes σ^j and σ^k, the substrings in positions $p+1 \ldots 2p$ of these suffixes are just the prefixes of length p of suffixes σ^{j+p} and σ^{k+p}.

It is well known that a suffix array, and the appropriate auxiliary values, may be constructed directly from a suffix tree by an appropriate traversal. Details are described by Gusfield [3]. More recently, suffix array construction algorithms have been proposed by Sadakane [9] and Itoh and Tanaka [5]. Sadakane's algorithm has $O(n \log^2 n)$ complexity in the worst case, but appears to run faster than the Manber-Myers algorithm in practical cases, while the algorithm of Itoh and Tanaka is claimed to out-perform that of Sadakane.

3 Suffix binary search trees

A *suffix binary search tree* (SBST for short) for string σ is a binary tree in which each node represents a suffix of σ — we think of the node as containing the suffix number. Furthermore, for each node, every suffix represented in the left subtree precedes, lexicographically, the suffix contained in that node, which in turn precedes every suffix represented in the right subtree. This corresponds to the standard notion of a binary search tree, and leads to a search algorithm, for a string α say, that follows a well-defined path from the root, branching left or right at each node according to the outcome of a comparison with the suffix represented at that node. The search succeeds if α is a prefix of the suffix represented at the current node, and fails if an empty subtree is reached.

As in the case of the suffix array, the efficiency of the SBST search can be adversely affected by multiple equal character comparisons. Irving and Love [4] show how this can be avoided by storing, at each node, one *lcp* value and one direction value, defined as follows. For a node x, $l(x)$ is the maximum length of a longest common prefix of the suffix represented by x and that represented by any of its ancestor nodes. If the value of $l(x)$ comes from an ancestor y such that x is in y's left subtree (we say that y is a *right ancestor* of x), then the direction value $d(x)$ is set to $left$, otherwise it is set to $right$. (For some nodes, the l value might arise from either a left or a right ancestor, in which case the value of $d(x)$ can be set arbitrarily, while if $l(x)$ is zero, $d(x)$ is undefined.)

By exploiting the l and d values during the search for an occurrence of the m-long string α, Irving and Love show that the total number of operations carried out is $O(m + h)$, where h is the length of the search path.

The search algorithm is summarised in Figure 1. A key feature of the algorithm is the maintenance of two values during the search, the so-called *llcp* and *rlcp* values, representing the length of a longest common prefix of α with the suffix represented by the last node from which the search branched left and right respectively. A proof of the correctness of this algorithm, and an analysis of its complexity, appears in [4]. On reaching a first node x that represents an occurrence of α in σ, all occurrences can be identified by a suitably pruned traversal of the subtree rooted at x — see [4].

Irving and Love [4] also describe several algorithms for building a suffix binary search tree. The so-called *standard* construction uses the search algorithm of Figure 1 to locate the insertion position for each suffix in turn in the growing tree. However, in the worst case, this can result in complexity that is quadratic in the length of the string, even if the tree is well balanced. An alternative *refined* construction algorithm exploits suffix links in the tree (i.e., an explicit link from the node representing σ^i to the node representing σ^{i+1} for each i), to reduce this worst-case complexity to $O(nh)$, where h is the height of the tree. Classical AVL rotations can be used to

```
- - Algorithm to search for an occurrence of α in the SBST T;
- - returns its starting position in σ, or zero if there is none.
begin
   i := Root of T;   llcp := 0;   rlcp := 0;
   while i ≠ null loop
     if l(i) > max(llcp, rlcp) then
       i := d(i) child of i;
     elsif l(i) < max(llcp, rlcp) then
       if llcp > rlcp then
         i := rchild_i;
         if d(i) = Left then rlcp := l(i); end if;
       elsif rlcp > llcp then
         i := lchild_i;
         if d(i) = Right then llcp := l(i); end if;
       end if;
     elsif l(i) = llcp and llcp > rlcp and d(i) = Right then
       i := rchild_i;
     elsif l(i) = rlcp and rlcp > llcp and d(i) = Left then
       i := lchild_i;
     else
       t := max{k : α(l(i) + 1 ... k) = σ(l(i) + i ... k + i − 1)};
       if t = |α| then
         return i;
       elsif t + i − 1 = n or else α(t + 1) > σ(t + i) then
         i := rchild_i;   rlcp := t;
       else
         i := lchild_i;   llcp := t;
       end if;
     end if;
   end loop;
   return 0;
end;
```

Figure 1: The search algorithm for an SBST

balance the tree during construction, ensuring that $h = O(\log n)$, and giving a worst-case construction time of $O(n \log n)$.

4 An alternative view of a suffix array

Our suffix array is equivalent to that of Manber and Myers, at least to their tuned version that includes just a single signed *lcp* value for each array entry. However, we view the suffix array from a somewhat different perspective, interpreting it as a perfectly balanced suffix binary search tree. Of course, an important practical benefit of the balance property is that the explicit child pointers, necessary in a normal SBST, can be omitted. We start with the relevant definitions.

For any ordered sequence $X = x_1, x_2, \ldots, x_n$, define the *perfectly balanced* binary search tree for X to be the binary tree which is empty if X is empty, and otherwise has $x_{\lfloor \frac{n+1}{2} \rfloor}$ in its root, and has the perfectly balanced binary search trees for $x_1, \ldots, x_{\lfloor \frac{n-1}{2} \rfloor}$ and $x_{\lfloor \frac{n+3}{2} \rfloor}, \ldots, x_n$ as its left and right subtrees, respectively.

It is easy to verify that the search algorithm in a perfectly balanced binary search tree for a sequence X exactly mirrors binary search in an array representation of X (provided the leftmost of the two mid-points is chosen whenever the current sequence has even length).

For a given string σ, we furnish the suffix array $A(\sigma)$ with two auxiliary arrays, denoted L and D. As always, the array A contains the permutation of $1, 2, \ldots, n$ such that $\sigma^{A(1)} < \sigma^{A(2)} < \cdots < \sigma^{A(n)}$. (In practice, for space efficiency, each D value can be incorporated as a sign bit in the corresponding L value.)

Consider the perfectly balanced binary search tree that represents the ordered sequence $\sigma^{A(1)}, \sigma^{A(2)}, \ldots, \sigma^{A(n)}$. We use the term node i, or the ith node, to mean the node containing $\sigma^{A(i)}$, in other words the node corresponding to position i in the suffix array, or equivalently, the node that would occupy position i in an inorder traversal of the tree.

We define $L(i)$ $(1 \le i \le n, i \ne \lfloor \frac{n+1}{2} \rfloor)$ to be

$$L(i) = \begin{cases} lcp(i, cra(i)) \text{ if node } i \text{ has no left ancestor,} \\ lcp(i, cla(i)) \text{ if node } i \text{ has no right ancestor,} \\ \max\{lcp(i, cla(i)), lcp(i, cra(i))\} \text{ otherwise,} \end{cases}$$

where $cla(i)$ (respectively $cra(i)$) denotes the closest left (respectively right) ancestor of node i, and $lcp(i, j)$ denotes the length of the longest common prefix of the suffixes represented by nodes i and j of the tree. (Note that $L(i)$ is undefined for $i = \lfloor \frac{n+1}{2} \rfloor$, corresponding to the root of the tree.)

The D array is defined by

$$D(i) = \begin{cases} Right, \text{ if node } i \text{ has no left ancestor,} \\ Left, \text{ if node } i \text{ has no right ancestor,} \end{cases}$$

and otherwise

$$D(i) = \begin{cases} Left \text{ if } lcp(i, cra(i)) > lcp(i, cla(i)), \\ Right \text{ if } lcp(i, cla(i)) > lcp(i, cra(i)), \end{cases}$$

with any tie in the last case broken arbitrarily, and the D value for the root undefined.

It is immediate from these definitions that the L and D components of the suffix array for σ correspond exactly to the l and d values for the corresponding nodes in the perfectly balanced binary search tree for σ.

Searching the suffix array

It is immediate that a search for an occurrence of a particular target string α as a substring of σ reduces to a binary search of the suffix array. This can be summarised as in Figure 2.

Of course, the comparison between α and $\sigma^{A(j)}$ is done exactly as in the SBST search algorithm, using $llcp$ and $rlcp$ values together with the L and D arrays to avoid repeating equal character comparisons. Just as in that context, the complexity of the search is $O(m + h)$ where m is the length of the target string and h, the height of the tree, is $O(\log n)$ in this case.

5 Constructing the suffix array from an SBST

Given an arbitrary suffix binary search tree $T = T(\sigma)$ for a string σ, we now show how we can construct the suffix array $A(\sigma)$, together with the auxiliary L and D arrays, in linear time. In fact, linear time construction of the array A is trivial via an inorder traversal of T. However, derivation of the L and D arrays from the l and d values contained in the nodes of T is not so straightforward, but can be achieved rather elegantly by an inorder traversal of T followed by a postorder traversal of (the perfectly balanced tree corresponding to) A.

```
i:= 1;
k:= n;
Found := False;
while not Found and k ≥ i loop
  j := ⌊(i+k)/2⌋;
  if α is a prefix of σ^A(j) then
    Found := True;
  elsif α < σ^A(j) then
    k := j − 1;
  else
    i := j + 1;
  end if;
end loop;
```

Figure 2: Binary search using the suffix array

In the first stage of this process, we assign the final values of the array A and transfer the l values from the nodes of T into the array L to form what we call the *intermediate* auxiliary array. In the second stage, we rearrange the values in L and assign the values in D to give the final auxiliary array values.

Creating the suffix array and intermediate auxiliary array from the SBST

The intermediate values in the array L are defined by

$$L(i) = lcp(i, i+1) \quad (1 \leq i < n)$$

$$L(n) = 0.$$

As indicated earlier, we construct the intermediate auxiliary array from the SBST T by an inorder traversal of T. During this traversal, when we 'process' node i, we assign a value to $A(i)$ — simply the suffix number contained in that node — and to $L(i-1)$ — an l value chosen from an appropriate node of T.

For a given node i in T, define an ancestor node k to be *good* if

1. $l(k) = 0$ or is undefined, or

2. $d(k) = Left$ and node i is in the right subtree of node k, or

3. $d(k) = Right$ and node i is in the left subtree of node k.

The *best* ancestor $b(i)$ of node i is its nearest good ancestor. It is immediate that every node apart from the root has a best ancestor (since the root is a good ancestor of every other node).

We first give the algorithm for the construction of the suffix array and intermediate auxiliary array, and then prove its correctness. Linear time complexity is immediate, as it is a tree traversal in which the number of operations carried out at each node is bounded by a constant. The algorithm is outlined in Figure 3.

```
A(1) := suffix number from node 1;
for i in 2 .. n loop
  if node i has no left child then
    if d(i) = Right then
      L(i − 1) := l(i);
    else
      L(i − 1) := l(b(i));
    end if;
  else
    if d(i − 1) = Left then
      L(i − 1) := l(i − 1);
    else
      L(i − 1) := l(b(i − 1));
    end if;
  end if;
  A(i) := suffix from node i;
end loop;
```

Figure 3: Constructing the suffix array and intermediate auxiliary array

For the correctness of this algorithm, there are four cases to consider, two straightforward and two rather less so. We deal with the straightforward cases first.

Lemma 5.1 *If node i has no left child and $d(i) = Right$ then the intermediate value $L(i-1)$ is given by $l(i)$.*

Proof The inorder predecessor $i-1$ of node i is its closest left ancestor. Since $d(i) = Right$, the value $l(i)$, by definition, gives $lcp(i, i-1)$, and this is the value required for the intermediate $L(i-1)$. □

Lemma 5.2 *If node i has a left child and $d(i) = Left$ then the intermediate value $L(i-1)$ is given by $l(i-1)$.*

Proof The inorder predecessor $i-1$ of node i is its rightmost left child. Since $d(i) = Left$, the value $l(i-1)$, by definition, gives $lcp(i-1, i)$, and this is the value required for the intermediate $L(i-1)$. □

To establish the remaining cases, we require an additional piece of notation and a couple of subsidiary lemmas.

For a non-root node j in an SBST, define c_j to have the value Left or Right according as the node is a left or a right child, respectively, of its parent.

Lemma 5.3 *If, for a given node j in an SBST, (i) $c_j = d_j = Left$ and a is an arbitrary left ancestor of j, or (ii) $c_j = d_j = Right$ and a is an arbitrary right ancestor of j, then*

$$lcp(j, a) = lcp(p, a),$$

where p is the parent of node j.

Proof (i) It is immediate that $\sigma_a < \sigma_j < \sigma_p$. Hence

$$lcp(a, p) = \min\{lcp(a, j), lcp(j, p)\},$$

and because $d(j) = Left$, this is $lcp(a, j)$.
(ii) Similar. □

Lemma 5.4 *If, for a given node j in an SBST, (i) $c_j = Left$, $d_j = Right$, and a is the closest left ancestor of j, or (ii) $c_j = Right$, $d_j = Left$, and a is the closest right ancestor of j, then*

$$lcp(j, p) = lcp(p, a),$$

where p is the parent of node j.

Proof (i) Again it is immediate that $\sigma_a < \sigma_j < \sigma_p$. Hence

$$lcp(a,p) = \min\{lcp(a,j), lcp(j,p)\},$$

and because $d(j) = Right$, this is $lcp(j,p)$.
(ii) Similar. $\Box$

Now we can complete the proof of the correctness of the algorithm for constructing the intermediate suffix array.

Lemma 5.5 *If node i has no left child and $d(i) = Left$ (or is undefined) then $L(i-1) = l(b(i))$.*

Proof The inorder predecessor, node $i-1$, of node i, is its closest left ancestor. Let $b = b(i)$ be the best ancestor of node i.
Case (i) Node b is a descendant of node $i-1$. The result follows by a sequence of applications of Lemma 5.3.
Case (ii) Node b is node $i-1$. The result follows by a sequence of applications of Lemma 5.3 and a single application of Lemma 5.4.
Case (iii) Node b is an ancestor of node $i-1$. The result follows by a sequence of applications of Lemmas 5.3 and 5.4. $\Box$

Lemma 5.6 *If node i has a left child and $d(i-1) = Right$ (or is undefined) then $L(i-1) = l(b(i-1))$.*

Proof The inorder predecessor, node $i-1$, of node i, is its rightmost left child. Node i is the closest left ancestor of node $i-1$, and we seek $lcp(i-1, i)$. The result follows at once by analogy with Lemma 5.5, with the roles of nodes $i-1$ and i interchanged. $\Box$

Creating the final auxiliary arrays

As previously, we exploit heavily the interpretation of our version of the suffix array as a perfectly balanced suffix binary search tree. Wherever appropriate, we refer to array entries as nodes in the tree, and reason in terms of closest left and right ancestors.

The algorithm for converting the intermediate suffix array to the final suffix array permutes the values in the L array and establishes the values in the D array. It can be viewed as a postorder traversal of the implicit binary tree. We first describe the algorithm and then establish its correctness.

```
procedure Convert(i, k : Integer) is
begin
  if k − i > 1 then
    j := (i + k)/2;
    Convert(i, j);
    Convert(j, k);
    if i = 0 then
      D(j) := Left;
    elsif k = n + 1 or else L(i) > L(j) then
      Swap(L(i), L(j));
      D(j) := Right;
    else
      D(j) := Left;
    end if
  end if
end Convert;
```

Figure 4: Constructing the final auxiliary arrays

Its linear time complexity follows at once from its interpretation as a tree traversal during which the work done at each node is bounded by a constant.

The algorithm is set out in procedure *Convert* in Figure 4. This procedure is called initially as $Convert(0,\, n+1)$. In terms of the implicit balanced binary tree, a given call of the procedure is designed to 'process' the node whose inorder value is j — i.e., to assign final L and D values for that node. It is clear that, if $i = 0$ the node has no left ancestor, and if $k = n + 1$ then the node has no right ancestor. Otherwise, nodes i and k are the closest left and right ancestors, respectively, of node j.

We establish an invariant for the recursive procedure. At any given point during its execution, let S denote the set of nodes yet to be processed, and for any node m in S, denote by $s(m)$ the node in S that is the inorder successor of node m, i.e.,

$$s(m) = \min_{r>m}\{r : r \in S\}.$$

The invariant that we wish to establish is that

$$L(m) = lcp(m, s(m))$$

for all nodes in S.

It is immediate from the definition of the intermediate suffix array that this invariant holds initially.

To see that the invariant is preserved during the processing of node j, we consider three cases:

Case 1) $i = 0$: in other words, node j has no left ancestor. Here, there is no node m in S for which $j = s(m)$. Hence, when j is removed from S, none of the nodes that remain in S has its s value changed, and because none of the L values change, the invariant is indeed preserved.

Case 2) $i > 0, k = n + 1$: in other words, node j has a left ancestor but no right ancestor. Here, let r be such that $j = s(r)$. It is immediate that r is the parent, and closest left ancestor of j, so in fact $r = i$. So just prior to processing node j we have $L(r) = L(i) = lcp(i, j)$ and $L(j)$ is undefined. So swapping $L(i)$ and $L(j)$ places the undefined value in $L(i)$, correctly representing the fact that, after processing node j and removing it from S, node i has no inorder successor in S. So again the invariant is preserved.

Case 3) $0 < k \leq n$: in other words, node j has both a left and a right ancestor. Again let r be such that $j = s(r)$. Node r cannot be a descendant of node j, since no descendants of node j are in S, so it follows that node r is the closest left ancestor (though this time not necessarily the parent) of node j. So just prior to processing node j, the invariant tells us that $L(r) = L(i) = lcp(i, j)$. Also, by similar reasoning we have $k = s(j)$, and therefore $L(j) = lcp(j, k)$.

To show that the invariant is preserved, we require that, after processing node j, $L(i)$ should be equal to $lcp(i, k)$, since on removing j from S we obtain $s(i) = k$.

If $L(i) \leq L(j)$ then $lcp(i, j) \leq lcp(j, k)$, so that

$$lcp(i, k) = \min\{lcp(i, j), (j, k)\} = lcp(i, j),$$

whereas if $L(i) > L(j)$ then $lcp(i, j) > lcp(j, k)$, so that

$$lcp(i, k) = \min\{lcp(i, j), (j, k)\} = lcp(j, k).$$

Hence, in either case, we see that the correct value is given to $L(i)$ when node j is processed, completing the proof that the invariant is preserved.

It is now straightforward to verify that, after application of the algorithm of Figure 4, the L array satisfies the required property, namely

$$L(j) = \begin{cases} lcp(j, cla(j)) \text{ if node } j \text{ has no left ancestor,} \\ lcp(j, cra(j)) \text{ if node } j \text{ has no right ancestor,} \\ \max\{lcp(j, cla(j)), lcp(j, cra(j))\} \text{ otherwise.} \end{cases}$$

This is because the final value is assigned to $L(j)$ when node j is processed, and this value is $lcp(j, k)$ in the first case, $lcp(i, j)$ in the second case, and the larger of $lcp(i, j)$ and $lcp(j, k)$ in the general case, and $i = cla(j)$, $k = cra(j)$.

It then follows immediately that the value of $D(j)$, also assigned when node j is processed, contains the required direction value for node j.

6 Conclusion

In this report, we have explored the relationship between suffix binary search trees and suffix arrays. We have seen that a suffix array, with the auxiliary arrays that facilitate an efficient search algorithm, may be viewed as an implicit representation of a suffix binary search tree. This correspondence gives a more tangible interpretation than hitherto of the compact version of the suffix array auxiliaries.

We have also shown that the suffix array and its auxiliaries may be created in linear time from an arbitrary suffix binary search tree, and therefore that efficient construction algorithms for the latter may be used as the basis of efficient construction of the former.

References

[1] A. Andersson, N. J. Larsson, and K. Swanson. Suffix trees on words. *Algorithmica*, 23:246 – 260, 1999.

[2] M. Farach. Optimal suffix tree construction with large alphabets. Technical report, Rutgers University, April 1997.

[3] D. Gusfield. *Algorithms on strings, trees, and sequences. Computer Science and Computational Biology.* Cambridge University Press, 1997.

[4] R. W. Irving and L. Love. The suffix binary search tree and suffix AVL tree. Technical report, Computing Science Department, University of Glasgow, 2000. Technical report TR-2000-54.

[5] H. Itoh and H. Tanaka. An efficient method for in-memory construction of suffix arrays. In *Proceedings of the sixth Symposium on String Processing and Information Retrieval, SPIRE'99*, pages 81–87. IEEE Computer Society Press, 1999.

[6] S. Kurtz. Reducing the space requirement of suffix trees. *Software Practice and Experience*, 29:1149 – 1171, 1999.

[7] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal of Computing*, 22(5):935 – 948, October 1993.

[8] E. McCreight. A space - economical suffix tree construction algorithm. *Journal of the Association for Computing Machinery*, 23(2):262 – 272, April 1976.

[9] K. Sadakane. A fast algorithm for making suffix arrays and for the burrows-wheeler transformation. In *Proceedings of the Data Compression Conference, DCC'98*, pages 129–138, 1998.

[10] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249 – 260, 1995.

[11] P. Weiner. Linear pattern matching algorithms. *Proceedings of the IEEE 14th Annual Symposium on Switching and Automata Theory*, pages 1 – 11, 1973.