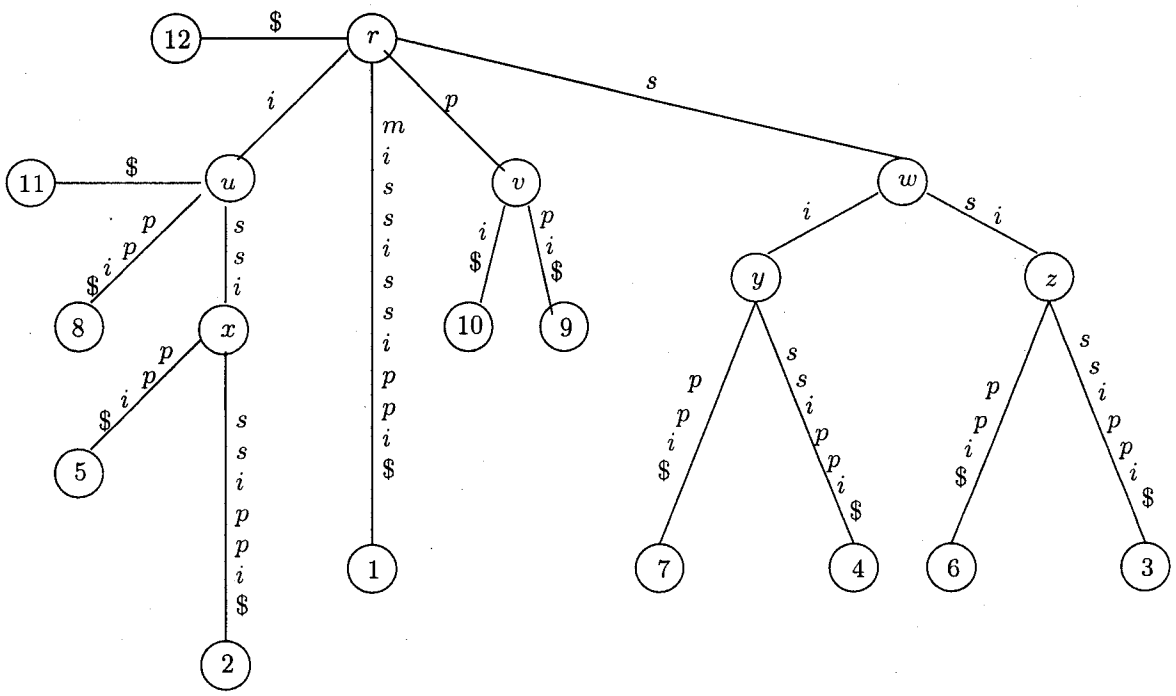

SA	12	11	8	5	2	1	10	9	7	4	6	3
Lcp	0	1	1	4	0	0	1	0	2	1	3	

FIGURE 1.1: Suffix tree, suffix array and *Lcp* array of the string *mississippi*. The suffix links in the tree are given by $x \rightarrow z \rightarrow y \rightarrow u \rightarrow r$, $v \rightarrow r$, and $w \rightarrow r$.

As each internal node has at least 2 children, an n-leaf suffix tree has at most $n-1$ internal nodes. Because of property (5), the maximum number of children per node is bounded by $|\Sigma|+1$. Except for the edge labels, the size of the tree is $O(n)$. In order to allow a linear space representation of the tree, each edge label is represented by a pair of integers denoting the starting and ending positions, respectively, of the substring describing the edge label. If the edge label corresponds to a repeat substring, the indices corresponding to any occurrence of the substring may be used. The suffix tree of the string *mississippi* is shown in Figure 1.1. For convenience of understanding, we show the actual edge labels.

The suffix array of $s = s'\$$, denoted $SA(s)$ or simply SA, is a lexicographically sorted array of all suffixes of s. Each suffix is represented by its starting position in s. $SA[i] = j$ iff $suff_j$ is the i^{th} lexicographically smallest suffix of s. The suffix array is often used in conjunction with an array termed *Lcp* array, containing the lengths of the longest common prefixes between every consecutive pair of suffixes in SA. We use $lcp(\alpha, \beta)$ to denote the longest common prefix between strings α and β. We also use the term *lcp* as an abbreviation for the term *longest common prefix*. $Lcp[i]$ contains the length of the *lcp* between $suff_{SA[i]}$ and $suff_{SA[i+1]}$, i.e., $Lcp[i] = lcp(suff_{SA[i]}, suff_{SA[i+1]})$. As with suffix trees, we use the notation $SA(s')$ to denote the suffix array of the string obtained by appending \$ to s'. The suffix and *Lcp* arrays of the string *mississippi* are shown in Figure 1.1.

continue as described in constructing the tree for one string.

This procedure immediately gives an algorithm to maintain the generalized suffix tree of a set of strings in the presence of insertions and deletions of strings. Insertion of a string is the same as executing McCreight's algorithm on the current tree, and takes time proportional to the length of the string being inserted. To delete a string, we must locate the leaves corresponding to all the suffixes of the string. By mimicking the process of inserting the string in GST using McCreight's algorithm, all the corresponding leaf nodes can be reached in time linear in the size of the string to be deleted. To delete a suffix, examine the corresponding leaf. If it is multiply labelled, it is enough to remove the label corresponding to the suffix. It it has only one label, the leaf and edge leading to it must be deleted. If the parent of the leaf is left with only one child after deletion, the parent and its two incident edges are deleted by connecting the surviving child directly to its grandparent with an edge labelled with the concatenation of the labels of the two edges deleted. As the adjustment at each leaf takes $O(1)$ time, the string can be deleted in time proportional to its length.

Suffix trees were invented by Weiner [23], who also presented the first linear time algorithm to construct them for a constant sized alphabet. McCreight's algorithm is a more space-economical linear time construction algorithm [19]. A linear time on-line construction algorithm for suffix trees was invented by Ukkonen [22]. In fact, our presentation of McCreight's algorithm also draws from ideas developed by Ukkonen. A unified view of these three suffix tree construction algorithms is studied by Giegerich and Kurtz [10]. Farach [7] presented the first linear time algorithm for strings over integer alphabets. The algorithm recursively constructs suffix trees for all odd and all even suffixes, respectively, and uses a clever strategy for merging them. The complexity of suffix tree construction algorithms for various types of alphabets is explored in [8].

1.2.3 Linear Time Construction of Suffix Arrays

Suffix arrays were proposed by Manber and Myers [18] as a space-efficient alternative to suffix trees. While suffix arrays can be deduced from suffix trees, which immediately implies any of the linear time suffix tree construction algorithms can be used for suffix arrays, it would not achieve the purpose of economy of space. Until recently, the fastest known direct construction algorithms for suffix arrays all required $O(n \log n)$ time, leaving a frustrating gap between asymptotically faster construction algorithms for suffix trees, and asymptotically slower construction algorithms for suffix arrays, despite the fact that suffix trees contain all the information in suffix arrays. This gap is successfully closed by a number of researchers in 2003, including Kärkkäinen and Sanders [13], Kim *et al.* [15], and Ko and Aluru [16]. All three algorithms work for the case of integer alphabet. Given the simplicity and/or space efficiency of some of these algorithms, it is now preferable to construct suffix trees via the construction of suffix arrays.

Kärkkäinen and Sanders' Algorithm

Kärkkäinen and Sanders' algorithm is the simplest and most elegant algorithm to date to construct suffix arrays, and by implication suffix trees, in linear time. The algorithm also works for the case of an integer alphabet. Let s be a string of length n over the alphabet $\Sigma = \{1, 2, \ldots, n\}$. For convenience, assume n is a multiple of three and $s[n+1] = s[n+2] = 0$. The algorithm has the following steps:

1. Recursively sort the $\frac{2}{3}n$ suffixes $suff_i$ with $i \ mod \ 3 \neq 0$.
2. Sort the $\frac{1}{3}n$ suffixes $suff_i$ with $i \ mod \ 3 = 0$ using the result of step (1).
3. Merge the two sorted arrays.

To execute step (1), first perform a radix sort of the $\frac{2}{3}n$ triples $(s[i], s[i+1], s[i+2])$ for each $i \ mod \ 3 \neq 0$ and associate with each distinct triple its rank $\in \{1, 2, \ldots, \frac{2}{3}n\}$ in sorted order. If all triples are distinct, the suffixes are already sorted. Otherwise, let $suff'_i$ denote the string obtained by taking $suff_i$ and replacing each consecutive triplet with its corresponding rank. Create a new string s' by concatenating $suff'_1$ with $suff'_2$. Note that all $suff'_i$ with $i \ mod \ 3 = 1$ ($i \ mod \ 3 = 2$, respectively) are suffixes of $suff'_1$ ($suff'_2$, respectively). A lexicographic comparison of two suffixes in s' never crosses the boundary between $suff'_1$ and $suff'_2$ because the corresponding suffixes in the original string can be lexicographically distinguished. Thus, sorting s' recursively gives the sorted order of $suff_i$ with $i \ mod \ 3 \neq 0$.

Step (2) can be accomplished by performing a radix sort on tuples $(s[i], rank(suff_{i+1}))$ for all $i \ mod \ 3 = 0$, where $rank(suff_{i+1})$ denotes the rank of $suff_{i+1}$ in sorted order obtained in step (1).

Merging of the sorted arrays created in steps (1) and (2) is done in linear time, aided by the fact that the lexicographic order of a pair of suffixes, one from each array, can be determined in constant time. To compare $suff_i$ ($i \ mod \ 3 = 1$) with $suff_j$ ($i \ mod \ 3 = 0$), compare $s[i]$ with $s[j]$. If they are unequal, the answer is clear. If they are identical, the ranks of $suff_{i+1}$ and $suff_{j+1}$ in the sorted order obtained in step (1) determines the answer. To compare $suff_i$ ($i \ mod \ 3 = 2$) with $suff_j$ ($i \ mod \ 3 = 0$), compare the first two characters of the two suffixes. If they are both identical, the ranks of $suff_{i+2}$ and $suff_{j+2}$ in the sorted order obtained in step (1) determines the answer.

The run-time of this algorithm is given by the recurrence $T(n) = T\left(\lceil \frac{2n}{3} \rceil\right) + O(n)$, which results in $O(n)$ run-time. Note that the $\frac{2}{3} : \frac{1}{3}$ split is designed to make the merging step easy. A $\frac{1}{2} : \frac{1}{2}$ split does not allow easy merging because when comparing two suffixes for merging, no matter how many characters are compared, the remaining suffixes will not fall in the same sorted array, where ranking determines the result without need for further comparisons. Kim *et al.*'s linear time suffix array construction algorithm is based on a $\frac{1}{2} : \frac{1}{2}$ split, and the merging phase is handled in a clever way so as to run in linear time. This is much like Farach's algorithm for constructing suffix trees [7] by constructing suffix trees for even and odd positions separately and merging them. Both the above linear time suffix array construction algorithms partition the suffixes based on their starting positions in the string. A completely different way of partitioning suffixes based on the lexicographic ordering of a suffix with its right neighboring suffix in the string is used by Ko and Aluru to derive a linear time algorithm [16]. This reduces solving a problem of size n to that of solving a problem of size no more than $\lceil \frac{n}{2} \rceil$, while eliminating the complex merging step. The algorithm can be made to run in only $2n$ words plus $1.25n$ bits for strings over constant alphabet. Algorithmically, Kärkkäinen and Sanders' algorithm is akin to mergesort and Ko and Aluru's algorithm is akin to quicksort. Algorithms for constructing suffix arrays in external memory are investigated by Crauser and Ferragina [5].

It may be more space efficient to construct a suffix tree by first constructing the corresponding suffix array, deriving the *Lcp* array from it, and using both to construct the suffix tree. For example, while all direct linear time suffix tree construction algorithms depend on constructing and using suffix links, these are completely avoided in the indirect approach. Furthermore, the resulting algorithms have an alphabet independent run-time of $O(n)$ while using only the $O(n)$ space representation of suffix trees. This should be contrasted with the $O(|\Sigma|n)$ run-time of either McCreight's or Ukkonen's algorithms.

1.2.4 Space Issues

Suffix trees and suffix arrays are space efficient in an asymptotic sense because the memory required grows linearly with input size. However, the actual space usage is of significant