Chapter 7

Data Structures for Strings

In this chapter, we consider data structures for storing strings; sequences of characters taken from some alphabet. Data structures for strings are an important part of any system that does text processing, whether it be a text-editor, word-processor, or Perl interpreter.

Formally, we study data structures for storing sequences of symbols over the alphabet $\Sigma = \{0,\ldots,|\Sigma|-1\}$. We assume that all strings are terminated with the special character $\$ = |\Sigma|-1$ and that $\$$ only appears as the last character of any string. In real applications, Σ might be the ASCII alphabet ($|\Sigma| = 128$); extended ASCII or ANSI ($|\Sigma| = 256$), or even Unicode ($|\Sigma| = 95,221$ as of Version 3.2 of the Unicode standard).

Storing strings of this kind is very different from storing other types of comparable data. On the one hand, we have the advantage that, because the characters are integers, we can use them as indices into arrays. On the other hand, because strings have variable length, comparison of two strings is not a constant time operations. In fact, the only *a priori* upper bound on the cost of comparing two strings s_1 and s_2 is $O(\min(|s_1|,|s_2|\})$, where $|s|$ denotes the length of the string s.

7.1 Two Basic Representations

In most programming languages, strings are built in as part of the language and they take one of two basic representations, both of which involve storing the characters of the string in an array. Both representations are illustrated in Figure 7.1.

In the *null-terminated representation*, strings are represented as (a pointer to) an array of characters that ends with the special null terminator $\$$. This representation is used, for example, in the C and C++ programming languages. This representation is fairly straightforward. Any character of the string can be accessed by its index in constant time. Computing the length, $|s|$, of a string s takes $O(|s|)$ time since we have to walk through the array, one character at a time until we find the null terminator.

Less common is the *pointer/length representation* in which a string is represented as (a pointer

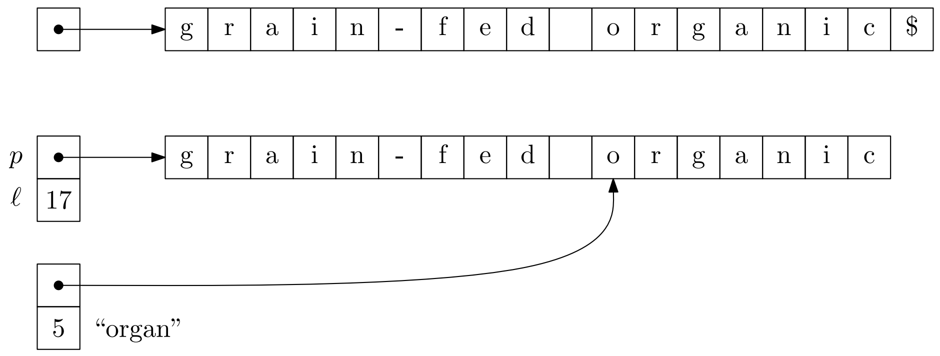
g r a i n - f e d o r g a n i c $
p
ℓ
17
g r a i n - f e d o r g a n i c
5
"organ"

Figure 7.1: The string "grain-fed organic" represented in both the null-terminated and the pointer/length representations. In the pointer/length representation we can extract the string "organ" in constant time.

to) an array of characters along with a integer that stores the length of the string. The pointer/length representation is a little more flexible and more efficient for some operations.

For example, in the pointer/length representation, determining the length of the string takes only constant time, since it is already stored. More importantly, it is possible to extract any substring in constant time: If a string s is represented as (p, ℓ) where p is the starting location and ℓ is the length, then we can extract the substring $s_i, s_{i+1}, \ldots, s_{i+m}$ by creating the pair $(p + i, m)$. For this reason several of the data structures in this chapter will use the pointer/length representation of strings.

7.2 Ropes

A common problem that occurs when developing, for example, a text editor is how to represent a very long string (text file) so that operations on the string (insertions, deletions, jumping to a particular point in the file) can be done efficiently. In this section we describe a data structure for just such a problem. However, we begin by describing a data structure for storing a sequence of weights.

A *prefix tree* T is a binary tree in which each node v stores two additional values $\mathrm{weight}(v)$ and $\mathrm{size}(v)$. The value of $\mathrm{weight}(v)$ is a number that is assigned to a node when it is created and which may be modified later. The value of $\mathrm{size}(v)$ is the sum of all the weight values stored in the subtree rooted at v, i.e.,

$$\mathrm{size}(v) = \sum_{u \in T(v)} \mathrm{weight}(u) \enspace .$$

It follows immediately that $\mathrm{size}(v)$ is equal to the sum of the sizes of v's two children, i.e.,

$$\mathrm{size}(v) = \mathrm{size}(\mathrm{left}(v)) + \mathrm{size}(\mathrm{right}(v)) \enspace . \tag{7.1}$$

When we insert a new node u by making it a child of some node already in T, the only size values that change are those on the path from u to the root of T. Therefore, we can perform such an insertion in time proportional to the depth of node u. Furthermore, because of identity (7.1), if all the size values in T are correct, it is not difficult to implement a left or right rotation so that the size values remain correct after the rotation. Therefore, by using the treap rebalancing scheme (Section 2.2), we can maintain a prefix-tree under insertions and deletions in $O(\log n)$ expected time per operation.

Notice that, just as a binary search tree represents its elements in sorted order, a prefix tree implicitly represents a sequence of weights that are the weights of nodes we encounter while travering T using an in-order (left-to-right) traversal. Let $u_1, \dots, u_n$ be the nodes of T in left-to-right order. We can use prefix-trees to perform searches on the set

$$W = \left\{ w_i : w_i = \sum_{j=1}^{i} \text{weight}(u_i) \right\} .$$

That is, we can find the smallest value of i such that $w_i \geq x$ for any query value x. To execute this kind of query we begin our search at the root of T. When the search reaches some node u, there are three cases

1. $x < \text{weight}(\text{left}(u))$. In this case, we continue the search for x in $\text{left}(u)$.
2. $\text{weight}(\text{left}(u)) \leq x < \text{weight}(\text{left}(u)) + \text{weight}(u)$. In this case, u is the node we are searching for, so we report it.
3. $\text{weight}(\text{left}(u)) + \text{weight}(u) \leq x$. In this case, we search for the value
$$x' = x - \text{size}(\text{left}(u)) - \text{weight}(u)$$
in the subtree rooted at $\text{right}(u)$.

Since each step of this search only takes constant time, the overall search time is proportional to the length of the path we follow. Therefore, if T is rebalanced as a treap then the expected search time is $O(\log n)$.

Furthermore, we can support SPLIT and JOIN operations in $O(\log n)$ time using prefix trees. Given a value x, we can split a prefix tree into two trees, where one tree contains all nodes u_i such that

$$\sum_{j=1}^{i} \text{weight}(u_i) \leq x$$

and the other tree contains all the remaining nodes. Given two prefix trees T_1 and T_2 whose nodes in left-to-right order are $u_1, \dots, u_n$ and $v_1, \dots, v_m$ we can create a new tree T' whose nodes in left-to-right order are $u_1, \dots, u_n, v_1, \dots, v_n$.

Next, consider how we could use a prefix-tree to store a very long string $t = t_1, \dots, t_n$ so that it supports the following operations.

1. INSERT(i, s). Insert the string s beginning at position t_i in the string t, to form a new string $t_1, \dots, t_{i-1} \circ s \circ t_{i+1}, \dots, t_n$.[1]
2. DELETE(i, l). Delete the substring $t_i, \dots, t_{i+l-1}$ from S to form a new string $t_1, \dots, t_{i-1}, t_{i+l}, \dots, t_n$.

[1]Here, and throughout, $s_1 \circ s_2$ denotes the concatenation of strings s_1 and s_2.

3. REPORT(i, l). Output the string $t_i, \ldots, t_{i+l-1}$.

To implement these operations, we use a prefix-tree in which each node u has an extra field, string(u). In this implementation, weight(u) is always equal to the length of string(u) and we can reconstruct the string S by concatenating string$(u_1) \circ$ string$(u_2) \circ \cdots \circ$ string(u_n). From this it follows that we can find the character at position i in S by searching for the value i in the prefix tree, which will give us the node u that contains the character t_i.

To perform an insertion we first create a new node v and set string$(v) = s'$. We then find the node u that contains t_i and we split string(u) into two parts at t_i; one part contains t_i and the characters that occur before it and the second part contains the remaining characters (that occur after t_i). We reset string(u) so that it contains the first part and create a new node w so that it string(w) contains the second part. Note that this split can be done in constant time if each string is represented using the pointer/length representation.

At this point the nodes v and w are not yet attached to T. To attach v, we find the leftmost descendant of right(u) and attach v as the left child of this node. We then update all nodes on the path from v to the root of T and perform rotations to rebalance T according to the treap rebalancing scheme. Once v is inserted, we insert w in the same way, i.e., by finding the leftmost descendant of right(v). The cost of an insertion is clearly proportional to the length of the search paths for v and w, which are $O(\log n)$ in expectation.

To perform a deletion, we apply the SPLIT operation on treaps to make three trees. The tree T_1 contains $t_1, \ldots, t_{i-1}$, the tree T_2 contains $t_i, \ldots, t_{i+l-1}$ and the treap T_3 that contains $t_{i+l}, \ldots, t_n$. This may require splitting the two substrings stored in nodes of T that contain the indices i and l, but the details are straightforward. We then use the MERGE operation of treaps to merge T_1 and T_3 and discard T_2. Since SPLIT and MERGE in treaps each take $O(\log n)$ expected time, the entire delete operation takes in $O(\log n)$ expected time.

To report the string $t_i, \ldots, t_{i+l-1}$ we first search for the node u that contains t_i and then traverse T starting at node u. We can then output $t_i, \ldots, t_{i+l-1}$ in $O(l + \log n)$ expected time by doing an in-order traversal of T starting at node u. The details of this traversal and its analysis are left as an exercise to the reader.

Theorem 18. *Ropes support the operations* INSERT *and* DELETE *on a string of length* n *in* $O(\log n)$ *expected time and* REPORT *in* $O(l + \log n)$ *expected time.*

7.3 Tries

Next, we consider the problem of storing a collection of strings so that we can quickly test if a query string is in the collection. The most obvious application of such a data structure is in a spell-checker.

A *trie* is a rooted tree T in which each node has somewhere between 0 and $|\Sigma|$ children. All edges of T are assigned labels in Σ such that all the edges leading to the children of a particular node receive different labels. Strings are stored as root-to-leaf paths in the trie so that, if the null-terminated string s is stored in T, then there is a leaf v in T such that the sequence of edge labels encountered on the path from the root of T to v is precisely the string s, including the null terminator. An example is

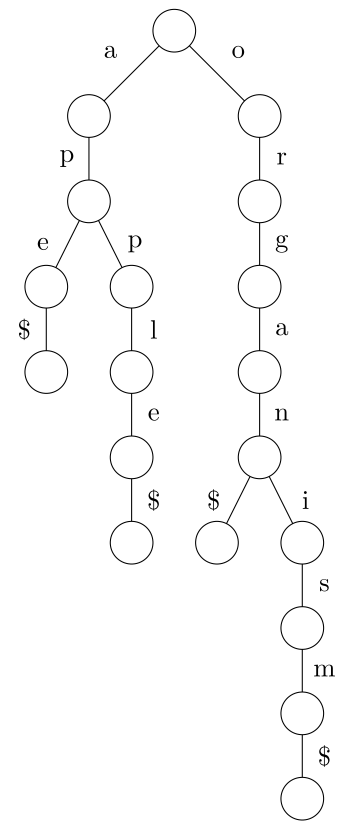

Figure 7.2: A trie containing the strings "ape", "apple", "organ", and "organism".

shown in Figure 7.2.

Notice that it is important that the strings stored in a trie are null-terminated, so that no string is the prefix of any other string. If this were not the case, it would be impossible to distinguish, for example, a trie that contained both "organism" and "organ" from a trie that contained just "organism".

Implementation-wise, a trie node is represented as an array of pointers of size $|\Sigma|$, which point to the children of the node. In this way, the labelling of edges is implicit, since the ith element of the array can represent the edge with label $i - 1$. When we create a new node, we initialize all of its $|\Sigma|$ pointers to nil.

Searching for the string s in a trie, T, is a simple operation. We examine each of the characters of s in turn and follow the appropriate pointers in the tree. If at any time we attempt to follow a pointer that is nil we conclude that s is not stored in T. Otherwise we reach a leaf v that represents s and we conclude that s is stored in T. Since the edges at each vertex are stored in an array and the individual characters of s are integers, we can follow each pointer in constant time. Thus, the cost of searching for s is $O(|s|)$.

Insertion into a trie is not any more difficult. We simply the follow the search path for s, and any time we encounter a nil pointer we create a new node. Since a trie node has size $O(|\Sigma|)$, this insertion

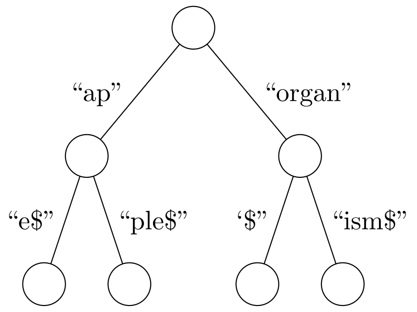

Figure 7.3: A Patricia tree containing the strings “ape”, “apple”, “organ”, and “organism”.

procedure runs in $O(|s| \cdot |\Sigma|)$ time.

Deletion from a trie is again very similar. We search for the leaf v that represents s. Once we have found it, we delete all nodes on the path from s to the root of T until we reach a node with more than 1 child. This algorithm is easily seen to run in $O(|s| \cdot |\Sigma|)$ time.

If a trie holds a set of n strings, S, which have a total length of N then the total size of the trie is $O(N \cdot |\Sigma|)$. This follows from the fact that each character of each string results in the creation of at most 1 trie node.

Theorem 19. *Tries support insertion or deletion of a string* s *in* $O(|s| \cdot |\Sigma|)$ *time and searching for* s *in* $O(|s|)$ *time. If* N *is the total length of all strings stored in a trie then the storage used by the trie is* $O(N \cdot |\Sigma|)$.

7.4 Patricia Trees

A *Patricia tree* (a.k.a. a *compressed trie*) is a simple variant on a trie in which any path whose interior vertices all have only one child is compressed into a single edge. For this to make sense, we now label the edges with strings, so that the string corresponding to a leaf v is the concatenation of all the edge labels we encounter on the path from the root of T to v. An example is given in Figure 7.3.

The edge labels of a Patricia tree are represented using the pointer/length representation for strings. As with tries, and for the same reason, it is important that the strings stored in a patricia tree are null-terminated. Implementation-wise, it makes sense to store the edge label of an edge uw directed from a parent u to it's child w in the node the w (since nodes can have many children, but at most one parent). See Figure 7.4 for an example.

Searching for a string s in a Patricia tree is similar to searching in a trie, except that when the search traverses an edge it checks the edge label against a whole substring of s, not just a single character. If the substring matches, the edge is traversed. If there is a mismatch, the search fails without finding s. If the search uses up all the characters of s, then it succeeds by reaching a leaf corresponding to s. In any case, this takes $O(|s|)$ time.

Descriptions of the insertion and deletion algorithms follow below. Figure 7.5 illustrates the

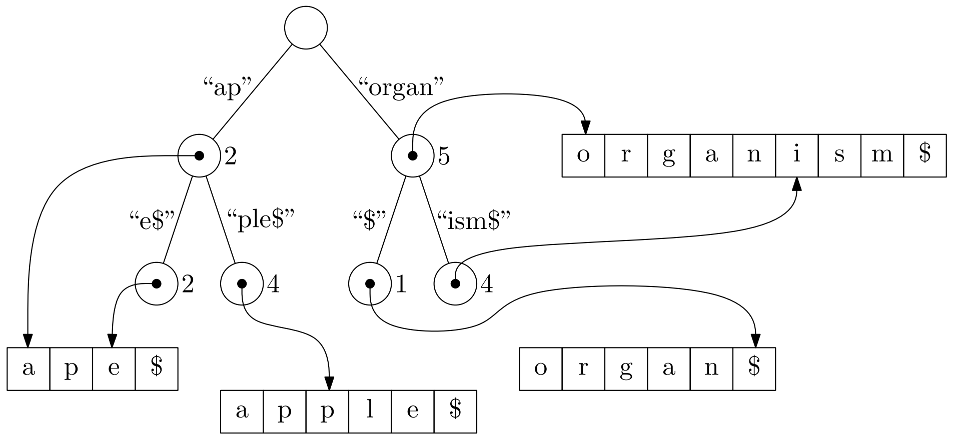

Figure 7.4: The edge labels of a Patricia tree use the pointer/length representation.

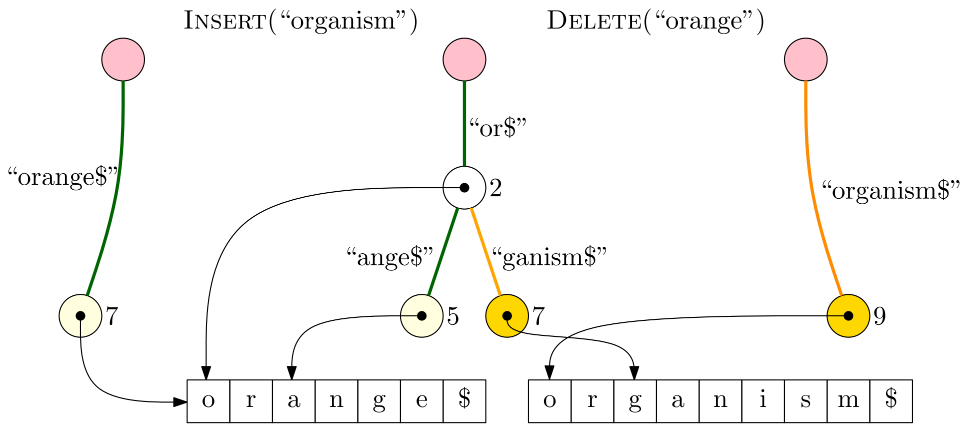

Figure 7.5: The evolution of a Patricia tree containing "orange$" as the string "organism$" is inserted and the string "orange$" is deleted.

actions of these algorithms. In this figure, a Patricia tree that initially stores only the string "orange" has the string "organism" added to it and then has the string "orange" deleted from it.

Inserting a string s into a Patricia tree is similar to searching right up until the point where the search gets stuck because s is not in the tree. If the search gets stuck in the middle of an edge, e, then e is split into two new edges joined by a new node, u, and the remainder of the string s becomes the edge label of the edge leading from u to a newly created leaf. If the search gets stuck at a node, u, then the remainder of s is used as an edge label for a new edge leading from u to a newly created leaf. Either case takes $O(|s| + |\Sigma|)$ time, since they each involve a search for s followed by the creation of at most two new nodes, each of size $O(|\Sigma|)$.

Removing a string s from a Patricia tree is the opposite of insertion. We first locate the leaf corresponding to s and remove it from the tree. If the parent, u, of this leaf is left with only one child, w, then we also remove u and replace it with a single edge, e, joining u's parent to w. The label for e is obtained in constant time by extending the edge label that was previously on the edge uw. How long this takes depends on how long it takes to delete two nodes of size $|\Sigma|$. If we consider this deletion to be a constant time operation, then running time is $O(|s|)$. If we consider this deletion to take $O(N)$ time, then the running time is $O(|s| + N)$.

An important difference between Patricia trees and tries is that Patricia trees contain no nodes with only one child. Every node is either a leaf or has at least two children. This immediately implies that the number of internal (non-leaf) nodes does not exceed the number of leaves. Now, recall that each leaf corresponds to a string that is stored in the Patricia tree so if the Patricia tree stores n strings, the total storage used by nodes is $O(n \cdot |\Sigma|)$. Of course, this requires that we also store the strings separately at a cost of $O(N)$. (As before, N is the total length of all strings stored in the Patricia tree.)

Theorem 20. *Patricia trees support insertion or deletion of any string s in $O(|s| + |\Sigma|)$ time and searching for s in $O(|s|)$ time. If N is the total length of all strings and n is the number of all strings stored in a Patricia tree then the storage used is $O(n \cdot |\Sigma| + N)$.*

In addition to the operations described in the preceding theorem, Patricia trees also support *prefix matches*; they can return a list of all strings that have some not-null-terminated string s as a prefix. This is done by searching for s in the usual way until running out of characters in s. At this point, every leaf in the subtree that the search ended at corresponds to a string that starts with s. Since every internal node has at least two children, this subtree can be traversed in $O(k)$ time, where k is the number of leaves in the subtree.

If we are only interested in reporting one string that has s as a prefix, we can look at the edge label of the last edge on the search path for s. This edge label is represented using the pointer/length representation and the pointer points to a longer string that has s as a prefix (consider, for example, the edge labelled “ple$” in Figure 7.4, whose pointer points to the second ‘p’ of “apple$”). This edge label can therefore be extended backwards to report the actual string that contains this label.

Theorem 21. *For a query string s, a Patricia tree can report one string that has s as a prefix in $O(|s|)$ time and can report all strings that have s as a prefix in $O(|s| + k)$ time, where k is the number of strings that have s as a prefix.*

7.5 Suffix Trees

Suppose we have a large body of text and we would like a data structure that allows us to query if particular strings occur in the text. Given a string t of length n, we can insert each of the n suffixes of t into a Patricia tree. We call the resulting tree the *suffix tree* for t. Now, if we want to know if some string s occurs in t we need only do a prefix search for s in the Patricia tree. Thus, we can test if s occurs in t in $O(|s|)$ time. In the same amount of time, we can locate some occurrence of s in t and in $O(|s| + k)$ time we can locate all occurrences of s in t, where k is the number of occurrences.

What is the storage required by the suffix tree for T? Since we only insert n suffixes, the storage required by tree nodes is $O(n \cdot |\Sigma|)$. Furthermore, recall that the labels on edges of T are represented as pointers into the strings that were inserted into T. However, every string that is inserted into T is a suffix of t, so all labels can be represented as pointers into a single copy of t, so the total spaced used to store t and all its edge labels is only $O(n)$. Thus, the total storage used by a suffix tree is $O(n \cdot |\Sigma|)$.

The cost of constructing the suffix tree for t can be split into two parts: The cost of creating and initializing new nodes, which is clearly $O(n \cdot |\Sigma|)$ because there are only $O(n)$ nodes; and the cost of following paths, which is clearly $O(n^2)$.

Theorem 22. *The suffix tree for a string* t *of length* n *can be constructed in* $O(n \cdot |\Sigma| + n^2)$ *time and uses* $O(n \cdot |\Sigma|)$ *storage. The suffix tree can be used to determine if any string* s *is a substring of* t *in* $O(|s|)$ *time. The suffix tree can also report the locations of all occurrences of* s *in* t *in* $O(m + k)$ *time, where* k *is the number of occurrences of* s *in* t.

The construction time in Theorem 22 is non-optimal. In particular, the $O(n^2)$ term is unnecessary. In the next few section we will develop the tools needed to construct a suffix tree in $O(n \cdot |\Sigma|)$ time.

7.6 Suffix Arrays

The *suffix array* $A_1, \ldots, A_n$ of a string $t = t_1, \ldots, t_n$ lists the suffixes of t in lexicographically increasing order. That is, A_i is the index such that $t_{A_i}, \ldots, t_n$ has rank i among all suffixes of A.

For example, consider the string $t =$ "counterrevoluationary\$":

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
c	o	u	n	t	e	r	r	e	v	o	l	u	t	i	o	n	a	r	y	\$

The suffix array for t is $A = \langle 18, 1, 6, 9, 15, 12, 17, 4, 11, 16, 2, 8, 7, 19, 5, 14, 3, 13, 10, 20, 21 \rangle$. This is hard to verify, so here is a table that helps:

i	A_i	$t_{A_i}, \ldots, t_n$
1	18	$s_1 =$ "ary\$"
2	1	$s_2 =$ "counterrevolutionary\$"
3	6	$s_3 =$ "errevolutionary\$"
4	9	$s_4 =$ "evolutionary\$"
5	15	$s_4 =$ "ionary\$"
6	12	$s_5 =$ "lutionary\$"
7	17	$s_7 =$ "nary\$"
8	4	$s_8 =$ "nterrevolutionary\$"
9	11	$s_9 =$ "olutionary\$"
10	16	$s_{10} =$ "onary\$"
11	2	$s_{11} =$ "ounterrevolutionary\$"
12	8	$s_{12} =$ "revolutionary\$"
13	7	$s_{13} =$ "rrevolutionary\$"
14	19	$s_{14} =$ "ry\$"
15	5	$s_{15} =$ "terrevolutionary\$"
16	14	$s_{16} =$ "tionary\$"
17	3	$s_{17} =$ "unterrevolutionary\$"
18	13	$s_{18} =$ "utionary\$"
19	10	$s_{19} =$ "volutionary\$"
20	20	$s_{20} =$ "y\$"
21	21	$s_{21} =$ "\$"

Given a suffix-array $A = A_1, \ldots, A_n$ for t and a not-null-terminated query string s one can do binary search in $O(|s| \log n)$ time to determine whether s occurs in t; binary search uses $O(\log n)$ comparison and each comparison takes $O(|s|)$ time.

7.6.1 Faster Searching with an ℓ Matrix

We can do searches a little faster—in $O(|s| + \log n)$ time—if, in addition to the suffix array, we have a little more information. Let $s_i = t_{A_i}, t_{A_i+1}, \ldots, t_n$. In other words, s_i is the string in the ith row of the preceding table. In other other words the string s_i is the ith suffix of t when all suffixes of t are sorted in lexicographic order.

Suppose, that we have access to a table ℓ, where, for any pair of indices (i, j) with $1 \leq i \leq j \leq n$, $\ell_{i,j}$ is the length of the longest common prefix of s_i and s_j. In our running example, $\ell_{9,11} = 1$, since $s_9 =$ "olutionary\$$''$ and $s_{11} =$ "ounterrevolutionary" have the first character "o" in common but differ on their second character. Notice that this also implies that s_{10} starts with the same letter as s_9 and s_{11}, otherwise s_{10} would not be between s_9 and s_{11} in sorted order. More generally, if $\ell_{i,j} = r$, then the suffixes $s_i, s_{i+1}, \ldots, s_j$ all start with the same r characters.

The value $\ell_{i,j}$ is called the *longest common prefix (LCP)* of s_i and s_j since it is the length of the longest string s' that is a prefix of both s_i and s_j. With the extra LCP information provided by ℓ, binary search on the suffix array can be sped up. The sped-up binary search routine maintains four integer values i, j, a and b. At all times, the search has deduced that a matching string—if it is exists—is among $s_i, s_{i+1}, \ldots, s_j$ (because $s_i < s < s_j$) and that the longest prefixes of s_i and s_j that match s have length a and b respectively. The algorithm starts with $(i = 1, j = n, a, b = 0)$ where the value of a is obtained by comparing s with s_1.

Suppose, without loss of generality, that $a \geq b$ (the case where $b > a$ is symmetric). Now, consider the entry $\ell_{i,m}$ which tells us how many characters of s_m match s_i. As an example, suppose we are searching for the string $s =$ "programmer", and we have reached a state where $s_i =$ "$\underline{\text{programm}}$able. . . \$", $s_j =$ "$\underline{\text{pro}}$tectionism", so $a = 8$ and $b = 3$.:

$$\begin{aligned} s_i &= \text{"}\underline{\text{programm}}\text{able...\$"} \\ &\vdots \\ s_m &= \text{"pro...\$"} \\ &\vdots \\ s_j &= \text{"}\underline{\text{pro}}\text{tectionism...\$"} \end{aligned}$$

Since s_i and s_j share the common prefix "pro" of length $\min\{a, b\} = 3$, we are certain that s_m must also have "pro" as a prefix. There are now three cases to consider:

1. If $\ell_{i,m} < a$, then we can immediately conclude that $s_m > s$. This is because $s_m > s_i$, so the

character at position $\ell_{i,m}+1$ in s_m is greater than the corresponding character in s_i and s. An example of this occurs when $s_m =$ "$\underline{\text{progr}}\text{ess}\ldots\$$", so that $\ell_{i,m} = 5$ and character at position 6 in s_m (an 'e') is greater than the character at position 6 in s_i and s (an 'a').

Therefore, our search can now be restricted to $s_i, \ldots, s_m$. Furthermore, we know that the longest prefix of s_m that matches s has length $b' = \ell_{i,m}$. Therefore, we can recursively search $s_i, \ldots, s_m$ using the values $a' = a$ and $b' = \ell_{i,m}$. That is, we recurse with the values $(i, m, a, \ell_{i,m})$

2. If $\ell_{i,m} > a$, then we can immediately conclude that $s_m < s$, for the same reason that $s_i < s$; namely s_i and s differ in character $a+1$ and this character is greater in s than it is in s_i and s_m. In this case, we recurse with the values (m, j, a, b). An example of this occurs when $s_m =$ "$\underline{\text{programma}}\text{tically}\ldots\$$", so that $\ell_{i,m} = 9$. In this case s_i and s_m both have a 'a' at position $a+1 = 9$ while s has an 'e'.

3. If $\ell_{i,m} = a$, then we can begin comparing s_m and s starting at the $(a+1)$th character position. After matching $k \geq 0$ additional characters of s and s_m this will eventually result in one of three possible outcomes:

   (a) If $s < s_m$, then the search can recursive on the set $s_i, \ldots s_m$ using the values $(i, m, a, a+k)$. An example of this occurs when $s_m =$ "$\underline{\text{programme}}\text{s}\ldots\$$", so $k = 1$ and we recurse on $(i, m, 8, 9)$.

   (b) If $s > s_m$, then the search can recurse on the set $s_m, \ldots, s_j$ using the values $(m, j, a+k, b)$. An example of this occurs when $s_m =$ "$\underline{\text{programme}}\text{d}\ldots\$$", so $k = 1$ and we recurse on $(m, j, 9, 3)$.

   (c) If all characters of s match the first $|s|$ characters of s_m, then s_m represents an occurrence of s in the text. An example of this occurs when $s_m =$ "$\underline{\text{programmer}}\text{s}\ldots\$$".

The search can proceed in this manner, until the string is found at some s_m or until $j = i+1$. In the latter case, the algorithm concludes that s does not occur as a substring of t since $s_i < s < s_{i+1}$.

To see that this search runs in $O(|s| + \log n)$ time, notice that each stage of the search reduces $j - i$ by roughly a factor of 2, so there are only $O(\log n)$ stages. Now, the time spent during a stage is proportional to k, the number of additional characters of s_m that match s. Observe that, at the end of the phase $\max\{a, b\}$ is increased by at least k. Since a and b are indices into s, $a \leq |s|$ and $b \leq |s|$. Therefore, the total sum of all k values during all phases is at most $|s|$. Thus, the total time spent searching for s is $O(|s| + \log n)$.

In case the search is successful, it is even possible to find all occurences of s in time proportional to the number of occurrences. This is done by the finding the maximal integers x and y such that $\ell_{m-x,m+y} \geq |s|$. Finding x is easily done by trying $x = 1, 2, 3, \ldots$ until finding a value, $x+1$, such that $\ell_{m-(x+1),m}$ is less than $|s|$. Finding y is done in the same way. When this happens, all strings $s_{m-x}, \ldots, s_{m+y}$ have s as a prefix.

Unfortunately, the assumption that we have an access to a longest common prefix matrix $\ell_{\cdot,\cdot}$ is unrealistic and impractical, since this matrix has $\binom{n}{2}$ entries. Note, however, that we do not need to store $\ell_{i,j}$ for every value $i, j \in \{1, \ldots, n\}$. For example, when n is one more than a power of 2, then we only need to know $\ell_{i,j}$ for values in which $i = 1 + q2^p$ and $j = i + 2^p$, with $p \in \{0, \ldots, \lfloor \log n \rfloor\}$ and $q \in \{0, \ldots, \lfloor n/2^p \rfloor\}$. This means that the longest common prefix information stored in ℓ consists of no more than

$$\sum_{i=0}^{\lfloor \log n \rfloor} n/2^i < 2n$$

values. In the next two sections we will show how the suffix array and the longest common prefix information can be computed efficiently given the string t.

7.6.2 Constructing the Suffix Array in Linear Time

Next, we show that a suffix array can be constructed in linear time from the input string, t. The algorithm we present makes use of the *radix-sort* algorithm, which allows us to sort an array of n integers whose values are in the set $\{0,\dots,n^c-1\}$ in $O(cn)$ time. In addition to being a sorting algorithm, we can also think of radix-sort as a compression algorithm. It can be used to convert a sequence, X, of n integers in the range $\{0,\dots,n^c-1\}$ into a sequence, Y, of n integers in the range $\{0,\dots,n-1\}$. The resulting sequence, Y, preserves the ordering of X, so that $Y_i < Y_j$ if and only if $X_i < X_j$ for all $i,j \in \{1,\dots,n\}$.

The suffix-array construction algorithm, which is called the *skew algorithm* is recursive. It takes as input a string s whose length is n, which is terminated with *two* \$ characters, and whose characters come from the alphabet $\{0,\dots n\}$.

Let S denote the set of all suffixes of t, and let S_r, for $r \in \{0,1,2\}$ denote the set of suffixes $t_i, t_{i+1}, \dots, t_n$ where $i \equiv r \pmod 3$. That is, we partition the sets of suffixes into three sets, each having roughly $n/3$ suffixes in them. The outline of the algorithm is as follows:

1. Recursively sort the suffixes in $S_1 \cup S_2$.
2. Use radix-sort to sort the suffixes in S_0.
3. Merge the two sorted sets resulting from steps 1 and 2 to obtain the sorted order of the suffixes in $S_0 \cup S_1 \cup S_2 = S$.

If the overall running time of the algorithm is denoted by $T(n)$, then the first step takes $T(2n/3)$ time and we will show, shortly, that steps 2 and 3 each take $O(n)$ time. Thus, the total running time of the algorithm is given by the recurrence

$$T(n) \le cn + T(2n/3) \le cn \sum_{i=0}^{\infty} (2/3)^i \le 3cn \ .$$

Next we describe how to implement each of the three steps efficiently.

Step 1. Assume $n = 3k+1$ for some integer k. If not, then append at most two additional \$ characters to the end of t to make it so. To implement Step 1, we transform the string t into the sequence of triples:

$$t' = \underbrace{(t_1,t_2,t_3),(t_4,t_5,t_6),\dots,(t_{n-3},t_{n-2},t_{n-1})}_{\text{suffixes in } S_1},\underbrace{(t_2,t_3,t_4),(t_5,t_6,t_7),\dots,(t_{n-2},t_{n-1},t_n)}_{\text{suffixes in } S_2}$$

Observe that each suffix in $S_1 \cup S_2$ is represented somewhere in this sequence. That, is, for any $t_i, t_{i+1}, \dots, t_n$ with $i \not\equiv 0 \pmod 3$, the suffix

$$(t_i,t_{i+1},t_{i+2}),(t_{i+3},t_{i+4},t_{i+5}),\dots,(t_{i+3\lceil(n-i)/3\rceil},t_{i+3\lceil(n-i)/3\rceil+1},t_{i+3\lceil(n-i)/3\rceil+2})$$

appears in our transformed string. Therefore, if we can sort the suffixes of the transformed string, we will have sorted all the suffixes in $S_1 \cup S_2$. The string t' contains about $2n/3$ characters, as required, but each character is a triple of characters in the range $(1, \ldots, n)$, so we cannot immediately recurse on t'. Instead, we first apply radix sort to relabel each triple with an integer in the range $1, \ldots, 2n/3$. We can now recursively sort the relabelled transformed string—which contains $2n/3$ characters in the range $1, \ldots, 2n/3$—and undo the relabelling to obtain the sorted order of $S_1 \cup S_2$. Aside from the recursive call, which takes $T(2n/3)$ time, all of the sorting and relabelling takes only $O(n)$ time.

When Step 1 is done, we know, for each index $i \not\equiv 0 \pmod 3$, the rank of the suffix $t_i, \ldots, t_n$ in the set $S_1 \cup S_2$. This means that for any pair of indices $i, j \not\equiv 0 \pmod 3$, we can determine—in constant time, by comparing ranks—if the suffix $t_i, \ldots, t_n$ comes before or after the suffix $t_j, \ldots, t_n$.

Step 2. To implement Step 2, we observe that every suffix in S_0 starts with a single character and is the followed by a suffix in S_1. That is, each suffix $t_i, \ldots, t_n$ with $i \equiv 0 \pmod 3$ can be represented as a pair (t_i, ℓ) where $\ell \in \{1, \ldots, 2n/3\}$ is the rank of the suffix $t_{i+1}, \ldots, t_n$ obtained during Step 1. Applying radix sort to all of these pairs allows us to sort S_0 in $O(n)$ time.

Step 3. To implement Step 3, we have to merge two sorted lists, one of length $2n/3$ and one of length $n/3$ in $O(n)$ time. Merging two sorted lists using $O(n)$ comparisons is straightforward. Therefore, the only trick here is to ensure that we can compare two elements—one from the first list and one from the second list—in constant time.

To see why this is possible, suppose we are comparing $t_i, \ldots, t_n \in S_0$ with $t_j, \ldots, t_n \in S_1$. Then we can write these as comparing

$$t_i, \underbrace{t_{i+1}, \ldots, t_n}_{\text{a suffix in } S_1}$$

and

$$t_j, \underbrace{t_{j+1}, \ldots, t_n}_{\text{a suffix in } S_2} \; .$$

Now this can certainly be done in constant time, by comparing t_i and t_j and, in case of a tie, comparing two suffixes whose relative order was already determined in Step 1.

Similarly, if we are comparing $t_i, \ldots, t_n \in S_0$ with $t_j, \ldots, t_n \in S_2$, then we can treat this as comparing

$$t_i, t_{i+1}, \underbrace{t_{i+2}, \ldots, t_n}_{\text{a suffix in } S_2}$$

and

$$t_j, t_{j+1}, \underbrace{t_{j+2}, \ldots, t_n}_{\text{a suffix in } S_1} \; .$$

This can also be done in constant time, by comparing at most two pairs of characters in t and, if necessary an suffix in S_1 with a suffix in S_2. The relative order of the suffixes in S_1 and S_2 were determined in Step 1 of the algorithm.

This completes the description of all the steps required to construct a suffix array in $O(n)$ time.

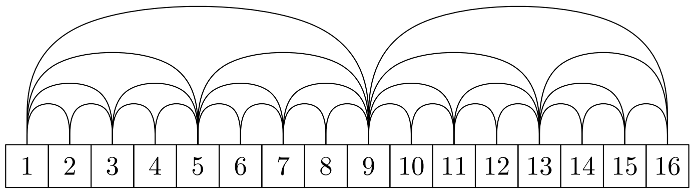

Figure 7.6: To implement binary search using a suffix array, only $O(n)$ LCP values are needed.

Theorem 23. *Given a string $t = t_1, \ldots, t_n$ over the alphabet $\{1, \ldots, n\}$, there exists an $O(n)$ time algorithm that can construct the suffix array for t.*

Of course, the preceding theorem can be generalized to larger alphabets. In particular, if the alphabet has size n^c, then the string t can be preprocessed in $O(cn)$ time using radix sort to reduce the alphabet size to n. The suffix array of the preprocessed string is then also a suffix array for the original string.

7.6.3 Constructing the Longest-Common-Prefix Array in Linear Time

As we have seen, $O(|s| + \log n)$ time searches in suffix arrays make use of an auxiliary data structure to determine the length of the longest common prefix between two suffixes s_i and s_j.

The *longest common prefix array* (*LCP array*) of a suffix array $A = A_1, \ldots, A_n$ is an array $L = L_1, \ldots, L_{n-1}$ where L_i is the length of the longest common prefix between $s_i = t_{A_i}, \ldots, t_n$ and $s_{i+1} = t_{A_{i+1}}, \ldots, t_n$. That is, L contains information about lexicographically consecutive suffixes of the string t. In terms of the notation in the previous section, $L_i = \ell_{i,i+1}$.

Before showing how to construct the LCP array, we recall that, in order to implement efficient prefix searches in a suffix array, we need more than an LCP array. When n is a power of two, the binary search algorithm needs to know the values $\ell_{i,j}$ for all pairs i, j of form $i = 1 + q2^p$ and $j = \min\{n, i + 2^p\}$, with $p \in \{1, \ldots, \lceil \log n \rceil - 1\}$ and $q \in \{0, \ldots, \lceil n/2^p \rceil\}$. See Figure 7.6 for an illustration.

To see that these can be efficiently obtained from the longest common prefix array, L, we first note that, for any $1 \le i \le j \le n$, $\ell_{i,j} = \min\{L_i, L_{i+1}, \ldots, L_{j-1}\}$. Next, we observe that, the pairs (i, j) needed by the binary search algorithm are nicely nested so that $\ell_{i,j}$ can be computed as the minimum of two smaller values. Specifically,

$$\ell_{q2^p,q2^p+2^p} = \min\{\ell_{2q2^{p-1},2q2^{p-1}+2^{p-1}}, \ell_{(2q+1)2^{p-1},(2q+1)2^{p-1}+2^{p-1}}\} \ .$$

(This is really easier to understand by looking at Figure 7.6 which shows that, for example, $\ell_{9,13}$ can be obtained as $\ell_{9,13} = \min\{\ell_{9,11}, \ell_{11,13}\}$.) This allows for an easy algorithm that constructs the required values in a bottom up fashion, starting with $p = 1$ (which is directly available from the LCP array, L) and working up to $p = \lfloor \log n \rfloor$. Each value can be computed in constant time and there are only $O(n)$ values to compute, so converting the LCP array, L into a structure useable for binary search can be done in $O(n)$ time. By doing this carefully, all the values needed to implement the $O(|s| + \log n)$ time search algorithm can be packed into a single array of length less than $2n$.

All that remains is to show how to compute the LCP array $L = L_1, \ldots, L_{n-1}$. The algorithm to do this uses an auxilliary array, R, which is the inverse of the suffix array, A, so that $R_{A_i} = i$ for all $i \in \{1, \ldots, n\}$. Another way to think of this is that R_i gives the location of A_i. Yet another way to think of this is that R_i is the rank of the suffix $t_{A_i}, \ldots, t_n$ in the lexicographically sorted list of all suffixes of t.

The algorithm processes the suffixes in order of their occurrence in t. That is, $t_1, \ldots, t_n$ is processed first. In our notation, this is suffix s_{R_1} and we want to compare it to the suffix s_{R_1-1} in order to determine the value of L_{R_1-1}. We can do this by comparing letter by letter to determine that the length of the longest common prefix of s_{R_1-1} and s_{R_1} is h.

Things get more interesting when we move onto the string $s_{R_2} = t_2, \ldots, t_n$ and we want to compare it to s_{R_2-1}. Notice that the strings $s_{R_1} = t_1, \ldots, t_n$ that we just considered and the string $s_{R_2} = t_2, \ldots, t_n$ that we are now considering have a lot in common. In particular, if $h > 0$, then we can be guaranteed that s_{R_2} matches the first $h-1$ characters of s_{R_2-1}. Stop now and think about why this is so, with the help of the following pictogram:

$$\begin{array}{rccccc}
s_{R_1} = & t_1, & t_2, & t_3, & \ldots, & t_h, & \ldots \\
 & \| & \| & \| & \ldots & \| & \ldots \\
s_{R_1-1} = & t_i, & t_{i+1}, & t_{i+2}, & \ldots, & t_{i+h-1}, & \ldots \\
 & & \| & \| & \ldots & \| & \ldots \\
s_{R_2} = & , & t_2, & t_3, & \ldots, & t_h, & \ldots
\end{array}$$

The string s_{R_1-1} is a proof that t has some suffix—namely $t_{i+1}, \ldots, t_n$—that comes before s_{R_2} in sorted order and matches the first $h-1$ characters of s_{R_2}. Therefore, the suffix s_{R_2-1} that immediately precedes s_{R_2} in sorted order must match at least the first $h-1$ characters of s_{R_2}. This means that the comparison between s_{R_2} and s_{R_2-1} can start at the hth character position.

More generally, if the longest common prefix of s_{R_i} and s_{R_i-1} has length h, then the longest common prefix of $S_{R_{i+1}}$ and $s_{R_{i+1}-1}$ has length at least $h-1$. This gives the following very slick algorithm for computing the LCP array, L.

```
BuildLCP($A, t$)
  for $i \leftarrow 1, \ldots, n$ do
    $R_{A_i} \leftarrow i$
  $h \leftarrow 0$
  for $i \leftarrow 1, \ldots, n$ do
    if $R_i > 1$ then
      $k \leftarrow A_{R_i-1}$
      while $t_{i+h} = t_{A_k+h}$ do
        $h \leftarrow h+1$
      $L_{R_i-1} \leftarrow h$
      $h \leftarrow \max\{h-1, 0\}$
  return $L$
```

The correctness of BuildLCP(A, t) follows from the preceding discussion. To bound the running time, we note that, with the exception of code inside the while loop, the algorithm clearly runs in $O(n)$ time. To bound the total time spent in the while loop, we notice that the value of h is initially 0, never exceeds n, is incremented each time the body of the while loop executes, and is decremented at most n times. It follows that the body of the while loop can not execute more than $2n$ times during the entire execution of the algorithm.

Theorem 24. *Given a suffix array A for a string t of length n, the LCP array, L, can be constructed in $O(n)$ time.*

To summarize everything we know about suffix arrays: Given a string t, the suffix array, A for t can be constructed in $O(n)$ time. From A and t, the LCP array, L can be constructed in $O(n)$ time. From the LCP array, L, an additional array L' of length $2n$ can be constructed that, in combination with A and t makes it possible to locate any string s in t in $O(|s| + \log n)$ time or report all k occurrences of s in t in $O(|s| + \log n + k)$ time..

In addition to storing the original text, t, this data structure needs the suffix array A, of length n and the auxiliary array L' of length $2n$. The values in A and L' are integers in the range $1, \ldots, n$, so they can be represented using $\lceil \log_2 n \rceil$ words. Thus, the entire structure for indexing t requires only $3n$ words of space.

7.7 Linear Time Suffix Tree Construction

Although a suffix array, in combination with an LCP array, is a viable substitute for a suffix tree in many applications, there are still times when one wants a suffix tree. In this section, we show that, given the suffix array, A, and the LCP array, L, for the string t, it is possible to compute the suffix tree, T, for t in $O(n \cdot |\Sigma|)$ time. Indeed, the tree structure can be constructed in $O(n)$ time, the factor of $|\Sigma|$ comes only from creating an array of length $|\Sigma|$ at each of the nodes in the suffix tree.

The key observation in this algorithm is that the suffix array and LCP array allow us to perform a traversal of the suffix tree—creating nodes as we go. To see how this works, recall that the first node traversed in an in-order traversal is lexicographically smallest. This node corresponds to the first suffix s_1 corresponding to the index, A_1 in the suffix array. We join this node, u_1, to a sibling, u_2 that corresponds to s_2. The sibling leaves u_1 and u_2 are joined to a common parent, x, and the labels on the edges xu_1 and xu_2 are suffixes of s_1 and s_2, respectively. The lengths of these labels can be computed by the length of the longest common prefix between s_1 and s_2, which is given by the value L_1. And now we can continue this way using s_3 and L_2 to determine how the leaf, u_3, corresponding to s_3 fits into the picture. The length of s_2 and the value of L_2 tells us whether

1. The parent of u_3 should subdivide the edge xu_2; or
2. The parent of u_3 should be x; or
3. The parent of u_3 should be a parent of x.

In general, when we process the suffix s_i, the length of s_{i-1} and L_{i-1} are used to determine how the new leaf, u_i should be attached to the partial tree constructed so far. Specifically, these values allow the algorithm to walk upward from u_{i-1} to find an appropriate location to attach u_i. This will either result in

1. A new parent for u_3 being created by splitting an existing edge;
2. Attaching u_3 as the child of an existing node; or
3. A new parent for u_3 being created and becoming the root of the current tree.

Processing u_i may require walking up many, say k, steps from u_{i-1}, which takes $O(k)$ time. However, when this process walks past a node, this node is never visited again by the algorithm. Since the final result is a suffix tree that has at most $2n-1$ nodes, the total running time is therefore $O(n)$. The first few steps of running this algorithm on the suffix array for "counterrevolutionary\$" are illustrated in Figure 7.7.

Theorem 25. *Given a suffix array and LCP array for a string* t*, the structure of the suffix tree,* T*, for* t *can be constructed in* $O(n)$ *time. If one wants to also allocate and populate the arrays associated with the nodes of* T *then this can be done in an additional*

7.8 Ternary Tries

The last three sections discussed data structures for storing strings where the running times of the operations were independent of the number of strings actually stored in the data structure. This is not to be underestimated. Consider that, if we store the collected works of Shakespeare in a suffix tree, it is possible to test if the word "warble" appears anywhere in the text by following 7 pointers.

This result is possible because the individual characters of strings are used as indices into arrays. Unfortunately, this approach has two drawbacks: The first is that it requires that the characters be integers. The second is that $|\Sigma|$, the size of the alphabet becomes a factor in the storage and running time.

To avoid having $|\Sigma|$ play a role in the running time, we can restrict ourselves to algorithms that only perform comparisons between characters. One way to do this would be to simply store our strings in a binary search tree, in lexicographic order. Then a search for the string s could be done with $O(\log n)$ string comparison, each of which takes $O(|s|)$ time, for a total of $O(|s| \log n)$.

Another way to reduce the dependence on $|\Sigma|$ is to store child pointers in a binary search tree. A *ternary* trie is a trie in which pointers to the children of each of node v are stored in a binary search tree. If we use a balanced binary search trees for this, then it is clear that the insertion, deletion and search times for the string s are $O(|s| \log |\Sigma|$. If N is large, we can do significantly better than this, by using a different method of balancing our search trees.

Note that a ternary trie can be viewed as a single tree in which each node has up to 3 children (hence the name). Each node v has a left child, left(v), a right child, right(v), a middle child, mid(v), and is labelled with a character, denoted $m(v)$. A root-to-leaf path P in the tree corresponds to exactly one string, which is obtained by concatenating the characters $m(v)$ for each node v whose successor in P is a middle child.

Now, suppose that every node v of our ternary trie has the balance property $|L(\text{right}(v))| \leq |L(v)|/2$ and $|L(\text{left}(v))| \leq |L(v)|/2$, where $L(v)$ denotes the set of leaves in the subtree rooted at v. Then the length of any root-to-leaf path P is at most $O(|s| + \log n)$ where $|s|$ is the length of the string

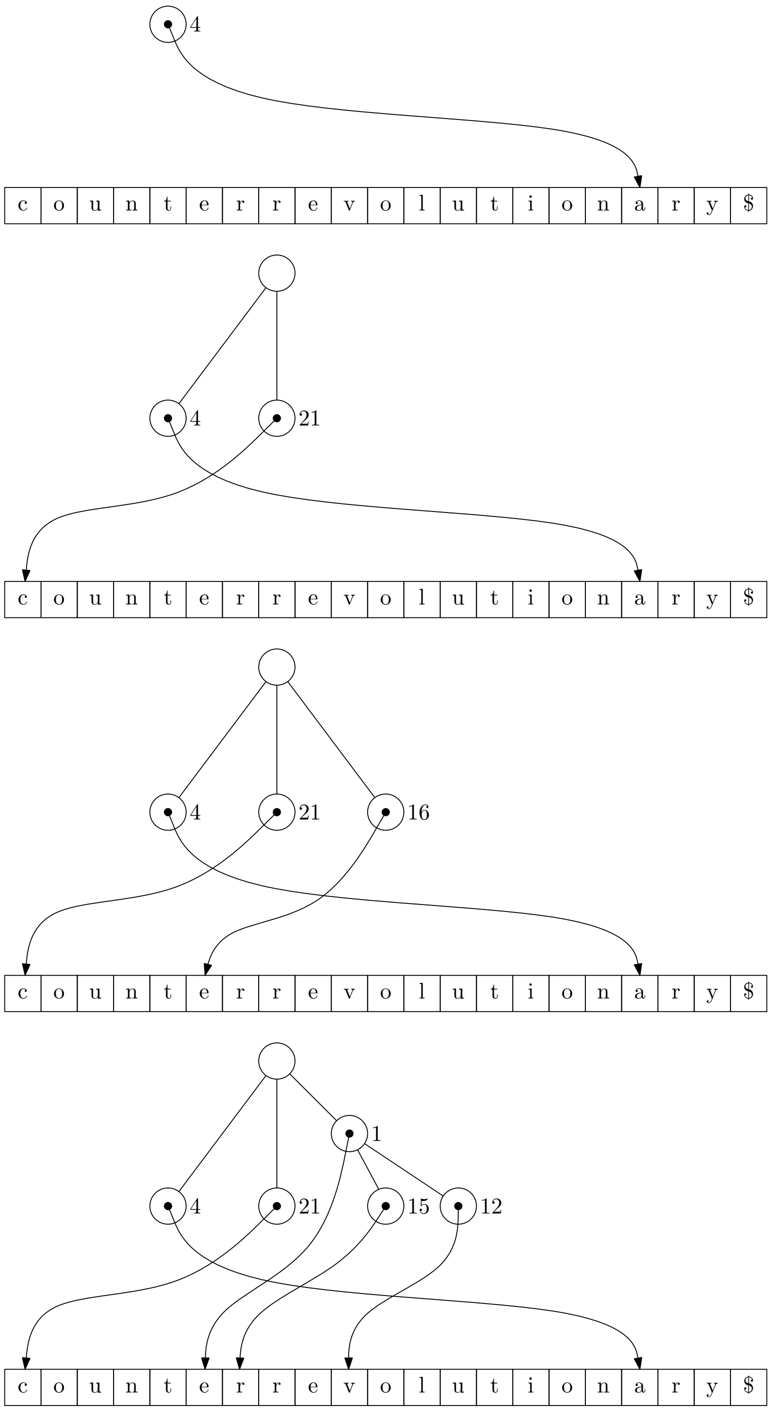

Figure 7.7: Building a suffix tree for "counterrevoluationary$" in a bottom-up fashion starting with the suffixes "ary$", "counterrevoluationary$", "errevoluationary$", and "evoluationary$".

represented by P. This is easy to see, because every time the path traverses an edge represented by a mid pointer the number of symbols in s that are unmatched decreases by one and every time the path traverses an edge represented by a left or right pointer the number of leaves in the current subtree decreases a factor of at least $1/2$.

Given a set S of strings, constructing a ternary trie with the above balance property is easily done. We first sort all our input strings and then look at the first character, m, of the string whose rank is $n/2$ in the sorted order. We then create a new node u with label m to be the root of the ternary trie. We collect all the strings that begin with m, strip off their first character, and recursively insert these into the middle child of u. Finally, we recursively insert all strings beginning with characters less than u in the left child of u and all strings beginning with characters greater than u in the right child of u.

Ignoring the cost of sorting and recursive calls, the above algorithm can easily be implemented to run in $O(n')$ time, where n' is the number of strings beginning with m. However, during this operation, we strip off n' characters from strings and never have to look at these again. Therefore the total running time, including recursive calls, is not more than $O(M)$, where M is the total length of all strings stored in the ternary trie.

Theorem 26. *After sorting the input strings, a ternary trie can be constructed in* $O(M)$ *time and can search for a string* s *in* $O(|s| + \log n)$ *time.*

TODO: Talk about how ternary quicksort can build a ternary search tree in $O(M + n \log n)$.

TODO: Talk about how to make ternary tries dynamic using treaps, splaying, weight-balancing, or partial rebuilding.

7.9 Quad-Trees

An interesting generalization of tries occurs when we want to store two (or more) dimensional data. Imagine that we want to store real values in the unit square $[0, 1)^2$, where each point (x, y) is represented by two binary strings $x_1, x_2, x_3, \ldots$ and $y_1, y_2, y_3, \ldots$ where x_i (respectively y_i) is the ith bit in the binary expansion of x (respectively, y). When we consider the most-significant bits of the binary expansion of x and y, four cases can occur:

$$(x, y) = \begin{array}{|c|c|}\hline (.0\ldots, .1\ldots) & (.1\ldots, .1\ldots) \\ \hline (.0\ldots, .0\ldots) & (.1\ldots, .0\ldots) \\ \hline \end{array}$$

Thus, it is natural that we store our points in a tree where each node has up to four children. In fact, if we treat (x, y) as the string $(x_1, y_1)(x_2, y_2)(x_3, y_3), \ldots,$ over the alphabet $\Sigma = \{(0, 0), (0, 1), (1, 0), (1, 1)\}$ and store (x, y) in a trie then this is exactly what happens. The tree that we obtain is called a *quad tree*.

Quad trees have many applications because they preserve spatial relationships very well, much in the way that tries preserve prefix relationships between strings. As a simple example, we can consider queries of the form: report all points in the rectangle with bottom left corner $[.0101010, .1111110]$

and top right corner [.0101011, .1111111]. To answer such a query, we simply follow the path for $(0,1)(1,1)(0,1)(1,1)(0,1)(1,1)$ in the quadtree (trie) and report all leaves of the subtree we find.

Quadtrees can be generalized in many ways. Instead of considering binary expansions of the x and y coordinates we can consider m-ary expansions, in which case we obtain a tree in which each node has up to m^2 children. If, instead of points in the unit square, we have points in the unit hypercube in $\mathbb{R}^d$ then we obtain a tree in which each node has 2^d children. If we use a Patricia tree in place of a trie then we obtain a *compressed quadtree* which, like the Patricia tree uses only $O(n + M)$ storage where M is the total size of (the binary expansion of) all points stored in the quadtree.

7.10 Discussion and References

Prefix-trees seem be part of the computer science folklore, though they are not always implemented using treaps. The first documented use of a prefix-tree is unclear.

Ropes (sometimes called cords) are described by Boehm *et al* [3] as an alternative to strings represented as arrays. They are so useful that they have made it into the SGI implementation of the C++ Standard Template Library [?].

Tries, also known as digital search trees, have a very long history. Knuth [7] is a good resource to help unravel some of their history. Patricia trees were described and given their name by Morrison [10]. PATRICIA is an acronym for Practical Algorithm To Retrieve Information Coded In Alphanumeric.

Suffix trees were introduced by Weiner [14], who also showed that, if N, the size of the alphabet, is constant then there is an $O(n)$ time algorithm for their construction. Since then, several other simplified $O(n)$ time construction algorithms for suffix trees have been presented [4, 8, 5].

Suffix arrays were introduced by Manber and Myers [?], who gave an $O(n \log n)$ time algorithm for their construction. The $O(n)$ time suffix array construction algorithm given here is due to Karkainnen and Sanders [?], though several other $O(n)$ time construction algorithms are known [?]. The LCP array construction algorithm given here is due to Kasai *et al* [?].

Karwere viewed as an way of compressing suffix trees [?]

Recently, Farach [5] gave an $O(n)$ time algorithm for the case where N is as large as n, the length of the string.

Ternary tries also have a long history, dating back at least until 1964, when Clampett [6] suggested storing the children of trie nodes in a binary search tree. Mehlhorn [9] shows that ternary tries can be rebalanced so that insertions and deletions can be done in $O(|s| + \log n)$ time. Sleator and Tarjan [12] showed that, if the splay heuristic (Section 6.1) is applied to ternary tries, then the cost of a sequence of n operations involving a set of strings whose total length is M is $O(M + n \log n)$. Furthermore, with splaying, a sequence of n searches can be executed in $O(M+nH)$ time, where H is the empirical entropy (Section 5.1) of the access sequence. Vaishnavi [13] and Bentley and Saxe [1] arrived at ternary tries through the geometric problem of storing a set of vectors in d-dimensions. Finally, ternary tries were recently revisited by Bentley and Sedgewick [2].

Samet's books [11, ?, ?] provide an encyclopedic treatment of quadtrees and their applications.

Bibliography

[1] J. L. Bentley and J. B. Saxe. Algorithms on vector sets. *SIGACT News*, 11(9):36–39, 1979.

[2] J. L. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA2000)*, pages 360–369, 1997.

[3] Hans-Juergen Boehm, Russ Atkinson, and Michael F. Plass. Ropes: An alternative to strings. *Software—Practice and Experience*, 25(12):1315–1330, 1995.

[4] M. T. Chen and J. Seiferas. Efficient and elegant subword tree construction. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, NATO ASI Series F: Computer and System Sciences, chapter 12, pages 97–107. NATO, 1985.

[5] M. Farach. Optimal suffix tree construction with large alphabets. In *38th Annual Symposium on Foundations of Computer Science (FOCS'97*, pages 137–143, 1997.

[6] H. H. Clampett Jr. Randomized binary searching with tree structures. *Communications of the ACM*, 7(3):163–165, 1964.

[7] Donald E. Knuth. *The Art of Computer Programming. Volume III: Sorting and Searching.* Addison-Wesley, 1973.

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

[9] K. Mehlhorn. Dynamic binary searching. *SIAM Journal on Computing*, 8(2):175–198, 1979.

[10] D. R. Morrison. PATRICIA - practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, October 1968.

[11] Hanan Samet. *The Design and Analysis of Spatial Data Structures.* Series in Computer Science. Addison-Wesley, Reading, Massachusetts, U.S.A., 1990.

[12] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(2):652–686, 1985.

[13] V. K. Vaishnavi. Multidimensional height-balanced trees. *IEEE Transactions on Computers*, C-33(4):334–343, 1984.

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