

# 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  $\{-1, \dots, N-1\}$ . We assume that all strings are terminated with the special character  $\$ = -1$  and that  $\$$  only appears as the last character of any string. In real applications, strings are not usually over the alphabet  $\{-1, \dots, N-1\}$ . Normally they would be ASCII or Unicode strings. However, the ASCII and Unicode alphabets can easily be translated into an alphabet of the form  $\{-1, \dots, N-1\}$ .

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, comparing two strings  $s_1$  and  $s_2$  takes time  $O(\min(|s_1|, |s_2|))$ , where  $|s|$  denotes the length of the string  $s$ .

### 7.1 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  $\text{weight}(v)$  and  $\text{size}(v)$ . The value of  $\text{weight}(v)$  is a number that is assigned to a node when it is created and which may be modified later. The value of  $\text{size}(v)$  is the sum of all the weight values stored in the subtree rooted at  $v$ , i.e.,

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

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

$$\text{size}(v) = \text{size}(\text{left}(v)) + \text{size}(\text{right}(v)) . \quad (7.1)$$

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 traversing  $T$  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

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

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 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  $x - \text{weight}(\text{left}(u)) + \text{weight}(u)$  in  $\text{right}(u)$ .

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  $O(\log M)$ .

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

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

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  $u_1, \dots, u_n, v_1, \dots, v_m$ .

Next, consider how we could use a prefix-tree to store a very long string  $s = c_1, \dots, c_M$  so that it supports the following operations.

1. **INSERT**( $i, s'$ ). Insert the string  $s'$  beginning at position  $s_i$  in the string  $S$ , to form a new string  $c_1, \dots, c_i \circ s' \circ c_{i+1}, \dots, c_M$ .<sup>1</sup>
2. **DELETE**( $i, l$ ). Delete the substring  $c_i, \dots, c_{i+l-1}$  from  $S$  to form a new string  $c_1, \dots, c_{i-1}, c_{i+l}, \dots, c_M$ .
3. **REPORT**( $i, l$ ). Output the string  $c_i, \dots, c_{i+l-1}$ .

To implement these operations, we use a prefix-tree in which each node  $u$  has an extra field,  $\text{string}(u)$ . In this implementation,  $\text{weight}(u)$  is always equal to the length of  $\text{string}(u)$  and we can reconstruct the string  $S$  by concatenating  $\text{string}(u_1) \circ \text{string}(u_2) \circ \dots \circ \text{string}(u_n)$ . 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  $c_i$ .

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

At this point the nodes  $v$  and  $w$  are not yet attached to  $T$ . To attach  $v$ , we find the leftmost descendant of  $\text{right}(u)$  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  $\text{right}(v)$ . The cost of an insertion is clearly proportional to the length of the search paths for  $v$  and  $w$ , which are  $O(\log M)$  in expectation.

To perform a deletion, we apply the **SPLIT** operation on treaps to make three trees. The tree  $T_1$  contains  $c_1, \dots, c_{i-1}$ , the tree  $T_2$  contains  $c_i, \dots, c_{i+l-1}$  and the treap  $T_3$  that contains  $c_{i+l}, \dots, c_M$ . We then use the **MERGE** operation of treaps to merge  $T_1$  and  $T_3$  and discard  $T_2$ . Since **SPLIT** and merge each take  $O(\log M)$  expected time, the entire delete operation takes in  $O(\log M)$  expected time.

To report the string  $s_i, \dots, s_{i+l-1}$  we first search for the node  $u$  that contains  $c_i$  and then traverse  $T$  starting at node  $u$ . We can then output  $c_i, \dots, c_{i+l-1}$  in  $O(l + \log M)$  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 17.** *Ropes support the operations **INSERT** and **DELETE** on a string of length  $M$  in  $O(\log M)$  expected time and **REPORT** in  $O(l + \log M)$  expected time.*

## 7.2 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  $N + 1$  children. All edges of  $T$  are assigned labels in  $\{-1, \dots, N - 1\}$  such that all the edges leading the children of a

<sup>1</sup>Here, and throughout,  $s_1 \circ s_2$  denotes the concatenation of strings  $s_1$  and  $s_2$ .

particular node receive different labels. Strings are stored as root-to-leaf paths in the trie so that, if the 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  $s$ .

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

Searching for the string  $s$  of length  $m$  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  $O(m)$ .

Insertion into a trie is not any more difficult. We simply 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  $O(N)$ , this insertion procedure runs in  $O(mN)$  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  $O(mN)$  time.

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

**Theorem 18.** *Let  $s$  be a string of length  $m$ . Tries support insertion or deletion of  $s$  in  $O(mN)$  time and searching for  $s$  in  $O(m)$  time. If  $M$  is the total length of all strings stored in a trie then the storage used is  $O(MN)$ .*

### 7.3 Patricia Trees

A *Patricia tree* (or 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 of the trie 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  $v$ .

To implement this idea we make a copy of each string  $s$  that is inserted into  $T$ . To label an edge with some substring of  $s$ , we use two pointers to the first and last characters of the label. Thus, ignoring the cost of storing the extra strings, the size of each edge-label is constant.

Searching, insertion and deletion in a Patricia tree is exactly the same as for tries except for some minor modifications required for handling the edge labels. The running times of these operations remain unchanged.

The 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 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  $O(nN)$ . Of course, this requires that we also store the strings separately at a cost of  $O(M)$ .

**Theorem 19.** *Let  $s$  be a string of length  $m$ . Patricia trees support insertion or deletion of  $s$  in  $O(mN)$  time and searching for  $s$  in  $O(m)$  time. If  $M$  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  $O(nN + M)$ .*

## 7.4 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 occurs in the text. One way to do this is to observe that tries and Patricia trees also allow us to perform *prefix searches*, i.e., given a string  $s$  that is not terminated with a  $\$$ , we can determine if  $s$  is the prefix of any string stored in the structure. To do a prefix search we simply follow the search path for  $s$  until we run into a nil pointer or  $s$  ends, leaving the search at some vertex  $v$ . In the former case,  $s$  is not a prefix of anything in the structure. In the latter case, each leaf-descendant of  $v$  that is leaf represents a string that has  $s$  as a prefix.

Given a string  $t$  of length  $M$ , we can insert each of the  $M$  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.

What is the storage required by the suffix tree for  $T$ ? Since we only insert  $M$  suffixes, the storage required by tree nodes is  $O(MN)$ . 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  $t$ . Thus, the total storage used by a suffix tree is  $O(MN)$ .

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  $O(MN)$  because there are only  $O(M)$  nodes and the cost of following paths, which is clearly  $O(M^2)$ .

**Theorem 20.** *The suffix tree for a string  $t$  of length  $M$  can be constructed in  $O(MN + M^2)$  time and uses  $O(MN)$  storage. The suffix tree can be used to determine if any string  $s$  of length  $m$  is a substring of  $t$  in  $O(m)$  time.*

We remark that the construction time in Theorem 20 is non-optimal. References to  $O(n)$  time suffix tree construction algorithms are given in Section ??.

## 7.5 Suffix Arrays\*

The suffix array  $A_1, \dots, A_M$  of a string  $t = t_1, \dots, t_M$  lists the suffixes of  $t$  in lexicographically increasing order. That is,  $A_i$  is the index such that  $t_{A_i}, \dots, t_m$  has rank  $i$  among all suffixes of  $A$ . Suffix arrays are a more space-efficient alternative to suffix trees.

Given a suffix-array  $A = A_1, \dots, A_M$  for  $t$  and a query string  $s$  one can do binary search in  $O(|s| \log M)$  time to determine whether  $s$  occurs in  $t$ ; binary search uses  $O(\log M)$  comparison and each comparison takes  $O(|s|)$  time. However, by being a little more sophisticated we can do better than this.

[Eventually we will describe the  $O(|s| + \log M)$  time search algorithm and Kärkkäinen and Sanders  $O(M + N)$  time construction algorithm.]

**Theorem 21.** *The suffix array for a string  $t$  of length  $M$  can be constructed in  $O(M + N)$  time and uses  $O(M)$  storage. The suffix array can be used to determine if any string  $s$  is a substring of  $t$  in  $O(|s| + \log M)$  time.*

## 7.6 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  $N$ , the size of the alphabet becomes a factor in the storage and running time.

To avoid having  $N$  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  $N$  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  $O(|s| \log N)$ . 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,  $\text{left}(v)$ , a right child,  $\text{right}(v)$ , a middle child,  $\text{mid}(v)$ , 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  $|\text{L}(\text{right}(v))| \leq |\text{L}(v)|/2$  and  $|\text{L}(\text{left}(v))| \leq |\text{L}(v)|/2$ , where  $\text{L}(v)$  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 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  $O(M)$ .

**Theorem 22.** *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.*

[Show how to make suffix trees dynamic, using randomization.]

## 7.7 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  $x_1, x_2, x_3, \dots$  and  $y_1, y_2, y_3, \dots$  where  $x_i$  (respectively  $y_i$ ) is the  $i$ th 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:

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

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  $(x_1, y_1)(x_2, y_2)(x_3, y_3), \dots$ , 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.

## 7.8 Discussion and References

Prefix-trees seem to 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 are now part 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 history 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  $O(M)$  time algorithm for their construction. Since then, several other simplified  $O(M)$  time construction algorithms for suffix trees have been presented [4, 8]. Recently, Farach [5] gave an  $O(M)$  time algorithm for the case where  $N$  is as large as  $M$ , 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 Sedgwick [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. Sedgwick. 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.