Ternary Trie

Tries can be implemented in the general alphabet model but a bit awkwardly using a comparison-based child function. Ternary trie is a simpler data structure based on symbol comparisons.

Ternary trie is like a binary search tree except:
- Each internal node has three children: smaller, equal and larger.
- The branching is based on a single symbol at a given position as in a trie. The position is zero (first symbol) at the root and increases along the middle branches but not along side branches.

There are also compact ternary tries based on compacting branchless path segments.

A ternary trie is balanced if each left and right subtree contains at most half of the strings in its parent tree.

- The balance can be maintained by rotations similarly to binary search trees.
- We can also get reasonably close to a balance by inserting the strings in the tree in a random order.

Note that there is no restriction on the size of the middle subtree.

An important concept for sets of strings is the LCP (longest common prefix) array and its sum.

Definition 1.10: Let \( R = \{S_1, S_2, \ldots, S_n\} \) be a set of strings and assume \( S_1 < S_2 < \cdots < S_n \). Then the LCP array \( \text{LCP}(R) \) is defined so that \( \text{LCP}(R)_i = \max \{0, |S_i| - \text{lcp}(S_i, S_{1, i-1})\} \). Furthermore, the LCP array sum is

\[
\sum \text{LCP}(R) = \sum_{i=1}^{\infty} \text{LCP}(R)_i.
\]

Example 1.11: For \( R = \{ali$, alice$, anna$, elias$, eliza$\}$, \( \sum \text{LCP}(R) = 7 \) and the LCP array is:

\[
\begin{array}{c}
0 \\
1 \\
3 \\
0 \\
3 \\
3
\end{array}
\]

Ternary tries have the same asymptotic size as the corresponding (\( \sigma \)-ary) tries.

Longest Common Prefixes

The standard ordering for strings is the lexicographical order. It is induced by an order over the alphabet. We will use the same symbols (\( <, \leq, \geq, \), etc.) for both the alphabet order and the induced lexicographical order.

We can define the lexicographical order using the concept of the longest common prefix.

Definition 1.8: The length of the longest common prefix of two strings \( A[0..m] \) and \( B[0..n] \), denoted by \( \text{lcp}(A, B) \), is the largest integer \( \ell \leq \min(m, n) \) such that \( A[0..\ell] = B[0..\ell] \).

Definition 1.9: Let \( A \) and \( B \) be two strings over an alphabet with a total order \( \prec \), and let \( \ell = \text{lcp}(A, B) \). Then \( A \) is lexicographically smaller than or equal to \( B \), denoted by \( A \preceq B \), if and only if

1. either \( |A| = \ell \)
2. or \( |A| > \ell, |B| > \ell \) and \( A[\ell] < B[\ell] \).

A variant of the LCP array sum is sometimes useful.

Definition 1.12: For a string \( S \) and a string set \( R \), define

\[
\text{lcp}(S, R) = \max \{\text{lcp}(S, T) \mid T \in R\}
\]

\[
\Sigma \text{lcp}(R) = \sum_{S \in R} \text{lcp}(S, R \setminus \{S\})
\]

The relationship of the two measures is shown by the following two results:

Lemma 1.13: For \( i \in [2..n] \), \( \text{LCP}(R)_i = \text{lcp}(S_i, (S_1, \ldots, S_{i-1})) \).

Lemma 1.14: \( \Sigma \text{LCP}(R) \leq \Sigma \text{lcp}(R) \leq 2 \Sigma \text{LCP}(R) \).

The proofs are left as an exercise.

The concept of distinguishing prefix is closely related and often used in place of the longest common prefix for sets. The distinguishing prefix of a string is the shortest prefix that separates it from other strings in the set. It is easy to see that \( \text{dp}(S, R \setminus S) = \text{lcp}(S, R \setminus S) + 1 \) (at least for a prefix free \( R \)).

Example 1.15: For \( R = \{ali$, alice$, anna$, elias$, eliza$\} \), \( \Sigma \text{dp}(R) = 13 \) and \( \Sigma \text{lcp}(R) = 18 \).

Example 1.7: Ternary tries for \( \{ali$, alice$, anna$, elias$, eliza$\} \).

In a balanced ternary trie each step down either
- moves the position forward (middle branch), or
- halves the number of strings remaining in the subtree (side branch).

Thus, in a balanced ternary trie storing \( n \) strings, any downward traversal following a string \( S \) passes at most \( |S| \) middle edges and at most \( \log n \) side edges.

Thus the time complexity of insertion, deletion, lookup and lcp query is \( O(|S| + \log n) \).

In comparison based tries, where the child function is implemented using binary search trees, the time complexities could be \( O(|S| \log \sigma) \), a multiplicative factor \( O(\log \sigma) \) instead of an additive factor \( O(\log n) \).

Prefix and range queries behave similarly (exercise).

An important concept for sets of strings is the LCP (longest common prefix) array and its sum.

Definition 1.10: Let \( R = \{S_1, S_2, \ldots, S_n\} \) be a set of strings and assume \( S_1 < S_2 < \cdots < S_n \). Then the LCP array \( \text{LCP}(R) \) is defined so that \( \text{LCP}(R)_i = 0 \) and for \( i \in [2..n] \), \( \text{LCP}(R)_i = \text{lcp}(S_i, S_{1, i-1}) \).

Thus the time complexity of insertion, deletion, lookup and lcp query is \( O(|S| + \log n) \).

The relationship of the two measures is shown by the following two results:

Lemma 1.13: For \( i \in [2..n] \), \( \text{LCP}(R)_i = \text{lcp}(S_i, (S_1, \ldots, S_{i-1})) \).

Lemma 1.14: \( \Sigma \text{LCP}(R) \leq \Sigma \text{lcp}(R) \leq 2 \Sigma \text{LCP}(R) \).

The proofs are left as an exercise.

The concept of distinguishing prefix is closely related and often used in place of the longest common prefix for sets. The distinguishing prefix of a string is the shortest prefix that separates it from other strings in the set. It is easy to see that \( \text{dp}(S, R \setminus S) = \text{lcp}(S, R \setminus S) + 1 \) (at least for a prefix free \( R \)).

Example 1.15: For \( R = \{ali$, alice$, anna$, elias$, eliza$\} \), \( \Sigma \text{dp}(R) = 13 \) and \( \Sigma \text{lcp}(R) = 18 \).

Theorem 1.16: The number of nodes in \( \text{trie}(R) \) is exactly \( |R| - \Sigma \text{LCP}(R) + 1 \), where \( |R| \) is the total length of the strings in \( R \).

Proof. Consider the construction of \( \text{trie}(R) \) by inserting the strings one by one in the lexicographical order using Algorithm 1.2. Initially, the trie has just one node, the root. When inserting a string \( S \), the algorithm executes exactly \( |S| \) rounds of the two while loops, because each round moves one step forward in \( S \). The first loop follows existing edges as long as possible and thus the number of rounds is \( \text{LCP}(R)_i = \text{lcp}(S_i, (S_1, \ldots, S_{i-1})) \). This leaves \( |S_i| - \text{LCP}(R)_i \) rounds for the second loop, each of which adds one new node to the trie. Thus the total number of nodes in the trie at the end is

\[
1 + \sum_{i=1}^{|R|} (|S_i| - \text{LCP}(R)_i) = |R| - \Sigma \text{LCP}(R) + 1.
\]

The proof reveals a close connection between \( \text{LCP}(R) \) and the structure of the trie. We will later see that \( \text{LCP}(R) \) is useful as an actual data structure in its own right.
String Sorting

Ω(n log n) is a well known lower bound for the number of comparisons needed for sorting a set of n objects by any comparison based algorithm. This lower bound holds both in the worst case and in the average case.

There are many algorithms that match the lower bound, i.e., sort using O(n log n) comparisons (worst or average case). Examples include quicksort, heapsort and mergesort.

If we use one of these algorithms for sorting a set of n strings, it is clear that the number of symbol comparisons can be more than O(n log n) in the worst case. Determining the order of A and B needs at least lcp(A, B) symbol comparisons and lcp(A, B) can be arbitrarily large in general.

On the other hand, the average number of symbol comparisons for two random strings in O(n log n) time using a standard sorting algorithm?

The following theorem shows that we cannot achieve O(n log n) symbol comparisons for any set of strings (when σ = n^2).

**Theorem 1.17:** Let A be an algorithm that sorts a set of objects using only comparisons between the objects. Let R = {S_1, S_2, ... , S_n} be a set of n strings over an ordered alphabet Σ of size σ. Sorting R using A requires Ω(n log n) symbol comparisons on average, where the average is taken over the initial orders of R.

- If R is considered to be a constant, the lower bound is Ω(n log n).
- Note that the theorem holds for any comparison based sorting algorithm A and any string set R. In other words, we can choose A and R to minimize the number of comparisons and still not get below the bound.
- Only the initial order is random rather than “any”. Otherwise, we could pick the correct order and use an algorithm that first checks if the order is correct, needing only O(n + Σ LCP(R)) symbol comparisons.

An intuitive explanation for this result is that the comparisons made by a sorting algorithm are not random. In the later stages, the algorithm tends to compare strings that are close to each other in lexicographical order and thus are likely to have long common prefixes.

String Quick sort (Multikey Quicksort)

Quicksort is one of the fastest general purpose sorting algorithms in practice. Here is a variant of quicksort that partitions the input into three parts instead of the usual two parts.

**Algorithm 1.19:** TernaryQuickSort(R)

Input: (Multi)set R in arbitrary order.

Output: R in ascending order.

1. If |R| ≤ 1 then return R
2. Select a pivot x ∈ R
3. R< ← {s ∈ R | s < x}
4. R= ← {s ∈ R | s = x}
5. R> ← {s ∈ R | s > x}
6. R< ← TernaryQuickSort(R<)
7. R= ← TernaryQuickSort(R=)
8. return R< · R= · R>

**Theorem 2.22:** String quicksort sorts a set R of n strings in O(Σ LCP(R) + n log n) time.

- Thus string quicksort is an optimal symbol comparison based algorithm.
- String quicksort is also fast in practice.
Proof of Theorem 1.22. The time complexity is dominated by the symbol comparisons on lines (4)–(6). We charge the cost of each comparison either on a single symbol or on a string depending on the result of the comparison:

\[ S[i] = X[i] \] to charge the comparison on the symbol \( S[i] \).

- Now the string \( S \) is placed in the set \( R_m \). The recursive call on \( R_m \) increases the common prefix length to \( i + 1 \). Thus \( S[i] \) cannot be involved in any future comparison and the total charge on \( S[i] \) is 1.

- Only \( lcp(S, R \setminus \{S\}) \) symbols in \( S \) can be involved in these comparisons. Thus the total number of symbol comparisons resulting equality is at most \( \Sigma lcp(R) = \Theta(\Sigma LCP(R)) \).

\[ S[i] \neq X[i] \] to charge the comparison on the string \( S \).

- Now the string \( S \) is placed in the set \( R_i \) or \( R_{i+1} \). The size of either set is at most \( |R|/2 \) assuming an optimal choice of the pivot \( X \).

- Every comparison charged on \( S \) halves the size of the set containing \( S \), and hence the total charge accumulated by \( S \) is at most \( \log n \).

Thus the total number of symbol comparisons resulting inequality is at most \( O(n \log n) \). □

Radix Sort

The \( \Omega(n \log n) \) sorting lower bound does not apply to algorithms that use stronger operations than comparisons. A basic example is counting sort for sorting integers.

Algorithm 1.23: CountingSort\( (R) \)

Input: (Multi)set \( R = \{k_1, k_2, \ldots, k_n\} \) of integers from the range \([0..n]\).

Output: \( R \) in nondecreasing order in array \( f(0..n) \).

(1) for \( i \leftarrow 0 \) to \( n - 1 \) do \( C[j] = 0 \)

(2) for \( i \leftarrow 1 \) to \( n \) do \( C[k[i]] = C[k[i]] + 1 \)

(3) \( \text{sum} \leftarrow 0 \)

(4) for \( i \leftarrow 0 \) to \( n - 1 \) do // cumulative sums

\( \text{tmp} = C[i]; \quad C[i] = \text{sum} = \text{sum} + \text{tmp} \)

(6) for \( i \leftarrow 1 \) to \( n \) do // distribute

(7) \( j[C[k[i]]] \leftarrow k[i]; \quad C[k[i]] = C[k[i]] + 1 \)

(8) return \( J \)

The time complexity is \( O(n + \sigma) \).

- Counting sort is a stable sorting algorithm, i.e., the relative order of equal elements stays the same.

There are two types of radix sorting:

- **MSD radix sort** starts sorting from the beginning of strings (most significant digit).

- **LSD radix sort** starts sorting from the end of strings (least significant digit).

The algorithm assumes that all strings have the same length \( m \), but it can be modified to handle strings of different lengths (exercise).

**Theorem 1.26:** LSD radix sort sorts a set \( R \) of strings over the alphabet \([0..\sigma]\) in \( O(|R| n + m \sigma) \) time, where \(|R|\) is the total length of the strings in \( R \) and \( m \) is the length of the longest string in \( R \).

**Proof.** Assume all strings have length \( m \). The LSD radix sort performs \( m \) rounds with each round taking \( O(n + \sigma) \) time. The total time is \( O(n m + m \sigma) = O(|R| n + m \sigma) \).

The case of variable length is left as an exercise.

- The weakness of LSD radix sort is that it uses \( O(|R|) \) time even when \( \Sigma LCP(R) \) is much smaller than \( |R| \).
- It is best suited for sorting short strings and integers.

**Algorithm 1.28:** MSDRadixSort\( (R, \ell) \)

Input: \( R \) in ascending lexicographical order.

(1) if \( |R| < \sigma \) then return StringQuickSort\( (R, \ell) \)

(2) \( R_0 \leftarrow \{\ell \in R \mid |S[\ell]| \leq \ell\}; \quad R \leftarrow R \setminus R_0 \)

(3) \( R_0, R_1, \ldots, R_{\ell-1} \) \( \Leftarrow \) CountingSort\( (R, \ell) \)

(4) for \( i \leftarrow 0 \) to \( \ell - 1 \) do \( R_i \leftarrow MSDRadixSort\( (R_i, \ell + 1) \) \)

(5) return \( R_0, R_1, \ldots, R_{\ell-1} \)

- Here CountingSort\( (R, \ell) \) not only sorts but also returns the partitioning based on symbols at position \( \ell \). The time complexity is still \( O(|R| n + \sigma) \).

- The recursive calls eventually lead to a large number of very small sets, but counting sort needs \( O(\alpha) \) time no matter how small the set is. To avoid the potentially high cost, the algorithm switches to string quicksort for small sets.

Similarly, the \( O(\Sigma LCP(R) n + \sigma \log n) \) lower bound does not apply to string sorting algorithms that use stronger operations than symbol comparisons.

Radix sort was developed for sorting large integers, but it treats an integer as a string of digits, so it is really a string sorting algorithm.

The LSD radix sort algorithm is very simple.

**Algorithm 1.24:** LSDRadixSort\( (R) \)

Input: (Multi)set \( R = \{S_1, S_2, \ldots, S_n\} \) of strings of length \( m \) over alphabet \([0..\sigma]\).

Output: \( R \) in ascending lexicographical order.

(1) for \( \ell \leftarrow m - 1 \) to 0 do CountingSort\( (R, \ell) \)

(2) return \( R \)

- CountingSort\( (R, \ell) \) sorts the strings in \( R \) by the symbols at position \( \ell \) using counting sort with \( k[i] \) replaced by \( S_i[\ell] \). The time complexity is \( O(|R| n + \sigma) \).
- The stability of counting sort is essential.

**Example 1.25:** \( R = \{\text{cat}, \text{ham}, \text{bat}\} \)

\[ \begin{align*}
\text{cat} & \quad \Rightarrow \quad \text{ha} \quad \Rightarrow \quad b \\
\text{ham} & \quad \Rightarrow \quad \text{ca} \quad \Rightarrow \quad c \\
\text{bat} & \quad \Rightarrow \quad \text{ba} \quad \Rightarrow \quad h
\end{align*} \]

It is easy to show that after \( i \) rounds, the strings are sorted by suffix of length \( i \). Thus, they are fully sorted at the end.

The LSD radix sort resembles string quicksort but partitions the strings into \( \sigma \) parts instead of three parts.

**Example 1.27:** MSD radix sort partitioning.

\[ \begin{align*}
\text{al} & \quad \Rightarrow \quad \text{habet} \\
\text{al} & \quad \Rightarrow \quad \text{gument} \\
\text{al} & \quad \Rightarrow \quad \text{ocate} \\
\text{al} & \quad \Rightarrow \quad \text{ernative} \\
\text{al} & \quad \Rightarrow \quad \text{as} \\
\text{al} & \quad \Rightarrow \quad \text{ernate} \\
\text{al} & \quad \Rightarrow \quad \text{ernate} \\
\text{al} & \quad \Rightarrow \quad \text{ernate}
\end{align*} \]

**Theorem 1.29:** MSD radix sort sorts a set \( R \) of \( n \) strings over the alphabet \([0..\sigma]\) in \( O(\Sigma LCP(R) + n \log n) \) time.

**Proof.** Consider a call processing a subset of size \( k \geq \sigma \):

- The time excluding the recursive calls but including the call to counting sort is \( O(k + \sigma) = O(k) \). The \( k \) symbols accessed here will not be accessed again.
- At most \( dp(|S|, S \setminus \{S\}) \leq \sigma lcp(S, R \setminus \{S\}) + 1 \) symbols in \( S \) will be accessed by the algorithm. Thus the total time spent in this kind of calls is \( O(|\Sigma p(R)|) = O(\Sigma p(R) n + n) = O(\Sigma LCP(R) + n) \).

The calls for a subsets of size \( k < \sigma \) are handled by string quicksort. Each string is involved in at most one such call. Therefore, the total time over all calls to string quicksort is \( O(\Sigma LCP(R) n + n \log n) \).

- There exists a more complicated variant of MSD radix sort with time complexity \( O(\Sigma LCP(R) n) \) in general for any algorithm that must access symbols one at a time (simple string model).
- In practice, MSD radix sort is very fast, but it is sensitive to implementation details.