## Search Trees for Strings

A balanced binary search tree is a powerful data structure that stores a *set of objects* and supports many operations including:

**Insert and Delete.**

**Lookup:** Find if a given object is in the set, and if it is, possibly return some data associated with the object.

**Range query:** Find all objects in a given range.

The time complexity of the operations for a set of size $n$ is $\mathcal{O}(\log n)$ (plus the size of the result) assuming constant time comparisons.

There are also alternative data structures, particularly if we do not need to support all operations:

- A *hash table* supports operations in constant time but does not support range queries.
- An *ordered array* is simpler, faster and more space efficient in practice, but does not support insertions and deletions.

A data structure is called *dynamic* if it supports insertions and deletions and *static* if not.



When the objects are strings, operations slow down:

- Comparison are slower. For example, the average case time complexity is $\mathcal{O}(\log n \log_\sigma n)$ for operations in a binary search tree storing a random set of strings.
- Computing a hash function is slower too.

For a string set $\mathcal{R}$, there are also new types of queries:

**Lcp query:** What is the length of the longest prefix of the query string $S$ that is also a prefix of some string in $\mathcal{R}$.

**Prefix query:** Find all strings in $\mathcal{R}$ that have $S$ as a prefix.

The prefix query is a special type of range query.



## Trie

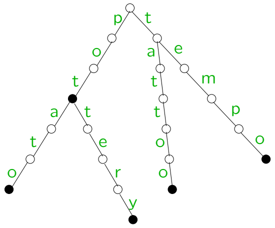
A trie is a *rooted tree* with the following properties:

- Edges are labelled with symbols from an alphabet $\Sigma$.
- The edges from any node $v$ to its children have all different labels.

Each node represents the string obtained by concatenating the symbols on the path from the root to that node.

The trie for a strings set $\mathcal{R}$, denoted by $trie(\mathcal{R})$, is the smallest trie that has nodes representing all the strings in $\mathcal{R}$.

**Example 3.23:** $trie(\mathcal{R})$ for $\mathcal{R} = \{\texttt{pot}, \texttt{potato}, \texttt{pottery}, \texttt{tattoo}, \texttt{tempo}\}$.





The time and space complexity of a trie depends on the implementation of the *child function*:

For a node $v$ and a symbol $c \in \Sigma$, *child*$(v, c)$ is $u$ if $u$ is a child of $v$ and the edge $(v, u)$ is labelled with $c$, and *child*$(v, c) = \perp$ if $v$ has no such child.

There are many implementation options including:

**Array:** Each node stores an array of size $\sigma$. The space complexity is $\mathcal{O}(\sigma||\mathcal{R}||)$, where $||\mathcal{R}||$ is the total length of the strings in $\mathcal{R}$. The time complexity of the child operation is $\mathcal{O}(1)$.

**Binary tree:** Replace the array with a binary tree. The space complexity is $\mathcal{O}(||\mathcal{R}||)$ and the time complexity $\mathcal{O}(\log \sigma)$.

**Hash table:** One hash table for the whole trie, storing the values *child*$(v, c) \neq \perp$. Space complexity $\mathcal{O}(||\mathcal{R}||)$, time complexity $\mathcal{O}(1)$.

Array and hash table implementations require an integer alphabet; the binary tree implementation works for an ordered alphabet.

A common simplification in the analysis of tries is to assume that $\sigma$ *is constant*. Then the implementation does not matter:

- Insertion, deletion, lookup and lcp query for a string $S$ take $\mathcal{O}(|S|)$ time.
- Prefix query takes $\mathcal{O}(|S| + \ell)$ time, $\ell$ is the total length of the strings in the answer.

The potential slowness of prefix (and range) queries is one of the main drawbacks of tries.

Note that a trie is a complete representation of the strings. There is no need to store the strings elsewhere. Most other data structures hold pointers to the strings, which are stored elsewhere.

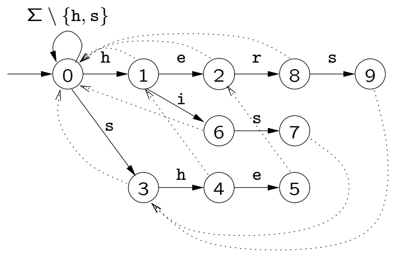


## Aho–Corasick Algorithm

Given a text $T$ and a set $\mathcal{P} = \{P_1.P_2, \ldots, P_k\}$ of patterns, the *multiple exact string matching* problem asks for the occurrences of all the patterns in the text. The Aho–Corasick algorithm is an extension of the Morris–Pratt algorithm for multiple exact string matching.

Aho–Corasick uses the trie $trie(\mathcal{P})$ as an *automaton* and augments it with a failure function similar to the Morris-Pratt failure function.

**Example 3.24:** Aho–Corasick automaton for $\mathcal{P} = \{\mathtt{he}, \mathtt{she}, \mathtt{his}, \mathtt{hers}\}$.






**Algorithm 3.25:** Aho–Corasick
Input: text $T$, pattern set $\mathcal{P} = \{P_1, P_2, \ldots, P_k\}$.
Output: all pairs $(i, j)$ such that $P_i$ occurs in $T$ ending at $j$.

Construct AC automaton
$v \leftarrow root$
*for* $j \leftarrow 0$ *to* $n - 1$ *do*
    *while child*$(v, T[j]) = \perp$ *do* $v \leftarrow$ *fail*$(v)$
    $v \leftarrow$ *child*$(v, T[j])$
    *for* $i \in$ *patterns*$(v)$ *do* output $(i, j)$

Let $S_v$ denote the string that node $v$ represents.

- $root$ is the root and *child*() is the child function of the trie.
- *fail*$(v) = u$ such that $S_u$ is the *longest proper suffix* of $S_v$ represented by any node.
- *patterns*$(v)$ is the set of pattern indices $i$ such that $P_i$ is a *suffix* of $S_v$.

At each stage, the algorithm computes the node $v$ such that $S_v$ is the longest suffix of $T[0..j]$ represented by any node.



**Algorithm 3.26:** Aho–Corasick trie construction
Input: pattern set $\mathcal{P} = \{P_1, P_2, \ldots, P_k\}$.
Output: AC trie: $root$, *child*() and *patterns*().

Create new node $root$
*for* $i \leftarrow 1$ *to* $k$ *do*
    $v \leftarrow root$; $j \leftarrow 0$
    *while child*$(v, P_i[j]) \neq \perp$ *do*
        $v \leftarrow$ *child*$(v, P_i[j])$; $j \leftarrow j + 1$
    *while* $j < |P_i|$ *do*
        Create new node $u$
        *child*$(v, P_i[j]) \leftarrow u$
        $v \leftarrow u$; $j \leftarrow j + 1$
    *patterns*$(v) \leftarrow \{i\}$

- The creation of a new node $v$ initializes *patterns*$(v)$ to $\emptyset$ and *child*$(v, c)$ to $\perp$ for all $c \in \Sigma$.
- After this algorithm, $i \in$ *patterns*$(v)$ iff $v$ represents $P_i$.

**Algorithm 3.27:** Aho–Corasick automaton construction
Input: AC trie: $root$, *child*() and *patterns*()
Output: AC automaton: *fail*() and updated *child*() and *patterns*()

(1) $queue \leftarrow \emptyset$
(2) *for* $c \in \Sigma$ *do*
(3) *if* *child*$(root, c) = \perp$ *then* *child*$(root, c) \leftarrow root$
(4) *else*
(5) $v \leftarrow$ *child*$(root, c)$
(6) $fail(v) \leftarrow root$
(7) pushback$(queue, v)$
(8) *while* $queue \neq \emptyset$ *do*
(9) $u \leftarrow$ popfront$(queue)$
(10) *for* $c \in \Sigma$ such that *child*$(u, c) \neq \perp$ *do*
(11) $v \leftarrow$ *child*$(u, c)$
(12) $w \leftarrow$ *fail*$(u)$
(13) *while* *child*$(w, c) = \perp$ *do* $w \leftarrow$ *fail*$(w)$
(14) $fail(v) \leftarrow$ *child*$(w, c)$
(15) *patterns*$(v) \leftarrow$ *patterns*$(v) \cup$ *patterns*(*fail*$(v)$)
(16) pushback$(queue, v)$

The algorithm does a *breath first traversal* of the trie. This ensures that correct values of *fail*() and *patterns*() are already computed when needed.



Assuming $\sigma$ is constant:

- The preprocessing time is $\mathcal{O}(m)$, where $m = ||\mathcal{P}||$.
  - The only non-trivial issue is the while-loop on line (13). Let $root, v_1, v_2, \ldots, v_\ell$ be the nodes on the path from root to a node representing a pattern $P_i$. Let $w_j =$ *fail*$(v_j)$ for all $j$. The depths in the sequence $w_1, w_2, \ldots, w_\ell$ increase by at most one in each step. Every round in the while-loop when computing $w_j$ reduces the depth of $w_j$ by at least one. Therefore, the total number of rounds when computing $w_1, w_2, \ldots, w_\ell$ is at most $\ell = |P_i|$. Thus, the while-loop is executed at most $||\mathcal{P}||$ times during the whole algorithm.
- The search time is $\mathcal{O}(n)$.
- The space complexity is $\mathcal{O}(m)$.

The analysis when $\sigma$ is not constant is left as an exercise.



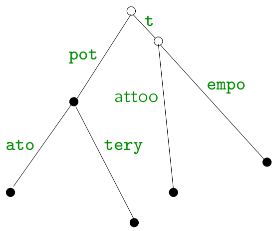
## Compact Trie

Tries suffer from a large number nodes, $\Omega(||\mathcal{R}||)$ in the worst case.

- The space requirement is large, since each node needs much more space than a single symbol.
- Traversal of a subtree is slow, which affects prefix and range queries.

The compact trie reduces the number of nodes by replacing *branchless path segments* with a single edge.

**Example 3.28:** Compact trie for $\mathcal{R} = \{\texttt{pot}, \texttt{potato}, \texttt{pottery}, \texttt{tattoo}, \texttt{tempo}\}$.






- The number of nodes and edges is $\mathcal{O}(|\mathcal{R}|)$.
- The egde labels are factors of the input strings. Thus they can be stored in constant space using pointers to the strings, which are stored separately.
- Any subtree can be traversed in linear time in the the number of leaves in the subtree. Thus a prefix query for a string $S$ can be answered in $\mathcal{O}(|S| + r)$ time, where $r$ is the *number* of strings in the answer.

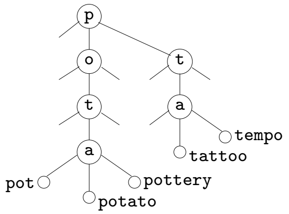
## Ternary Tree

The binary tree implementation of a trie supports ordered alphabets but awkwardly. Ternary tree is a simpler data structure based on symbol comparisons.

Ternary tree is like a binary tree except:

- Each internal node has three children: smaller, equal and larger.
- The branching is based on a single symbol at a given position. The position is zero at the root and increases along the middle branches.

**Example 3.29:** Ternary tree for $\mathcal{R} = \{\texttt{pot}, \texttt{potato}, \texttt{pottery}, \texttt{tattoo}, \texttt{tempo}\}$.






There is an analogy between *sorting algorithms* and *search trees* for strings.

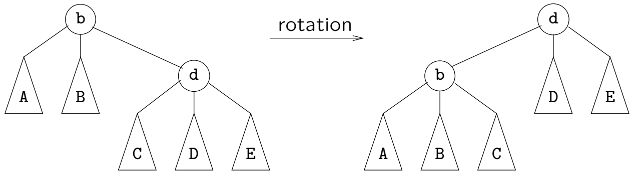
| sorting algorithm | search trees |
|---|---|
| standard binary quicksort | standard binary tree |
| string quicksort | ternary tree |
| radix sort | trie |

The ternary tree can be seen as the partitioning structure by string quicksort.



A ternary tree 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 trees.




- We can also get reasonably close to balance by inserting the strings in the tree in a random order.

In a balanced ternary tree, each step down either

- moves the position forward (middle branch), or
- halves the number of strings remaining the the subtree.

Thus, in a ternary tree storing $n$ strings, any downward traversal following a string $S$ takes at most $\mathcal{O}(|S| + \log n)$ time.



For the ternary tree of a string set $\mathcal{R}$ of size $n$:

- The number of nodes is $\mathcal{O}(DP(\mathcal{R}))$.
- Insertion, deletion, lookup and lcp query for a string $S$ takes $\mathcal{O}(|S| + \log n)$ time.
- Prefix search for a string $S$ takes $\mathcal{O}(|S| + \log n + DP(\mathcal{Q}))$, where $\mathcal{Q}$ is the set of strings given as the result of the query. With some additional data structures, this can be reduced to $\mathcal{O}(|S| + \log n + |\mathcal{Q}|)$

## String Binary Search

An *ordered array* is a simple static data structure supporting queries in $\mathcal{O}(\log n)$ time using binary search.

**Algorithm 3.30:** Binary search
Input: Ordered set $R = \{k_1, k_2, \ldots, k_n\}$, query value $x$.
Output: The number of elements in $R$ that are smaller than $x$.

```
left ← 0; right ← n        // final answer is in the range [left..right]
while left < right do
    mid ← ⌈(left + right)/2⌉
    if k_mid < x then left ← mid
    else right ← mid − 1
return left
```

With strings as elements, however, the query time is

- $\mathcal{O}(m \log n)$ in the worst case for a query string of length $m$
- $\mathcal{O}(m + \log n \log_\sigma n)$ on average for a random set of strings.



We can use the *lcp comparison technique* to improve binary search for strings. The following is a key result.

**Lemma 3.31:** Let $A \le B, B' \le C$ be strings. Then $lcp(B, B') \ge lcp(A, C)$.

**Proof.** Let $B_{min} = \min\{B, B'\}$ and $B_{max} = \max\{B, B'\}$. By Lemma 3.17,

$$\begin{aligned} lcp(A, C) &= \min(lcp(A, B_{max}), lcp(B_{max}, C)) \\ &\le lcp(A, B_{max}) = \min(lcp(A, B_{min}), lcp(B_{min}, B_{max})) \\ &\le lcp(B_{min}, B_{max}) = lcp(B, B') \end{aligned}$$

□



During the binary search of $P$ in $\{S_1, S_2, \ldots, S_n\}$, the basic situation is the following:

- We want to compare $P$ and $S_{mid}$.
- We have already compared $P$ against $S_{left}$ and $S_{right+1}$, and we know that $S_{left} \le P, S_{mid} \le S_{right+1}$.
- If we are using LcpCompare, we know $lcp(S_{left}, P)$ and $lcp(P, S_{right+1})$.

By Lemmas 3.17 and 3.31,

$$lcp(P, S_{mid}) \ge lcp(S_{left}, S_{right+1}) = \min\{lcp(S_{left}, P), lcp(P, S_{right+1})\}$$

Thus we can skip $\min\{lcp(S_{left}, P), lcp(P, S_{right+1})\}$ first characters when comparing $P$ and $S_{mid}$.



**Algorithm 3.32:** String binary search (without precomputed lcps)
Input: Ordered string set $\mathcal{R} = \{S_1, S_2, \ldots, S_n\}$, query string $P$.
Output: The number of strings in $\mathcal{R}$ that are smaller than $P$.

```
left ← 0; right ← n
llcp ← 0; rlcp ← 0
while left < right do
    mid ← ⌈(left + right)/2⌉
    mlcp ← min{llcp, rlcp}
    (x, mlcp) ← LcpCompare(S_mid, P, mlcp)
    if x = "<" then left ← mid; llcp ← mclp
    else right ← mid − 1; rlcp ← mclp
return left
```

- The average case query time is now $\mathcal{O}(m + \log n)$.
- The worst case query time is still $\mathcal{O}(m \log n)$.