Compact Tries

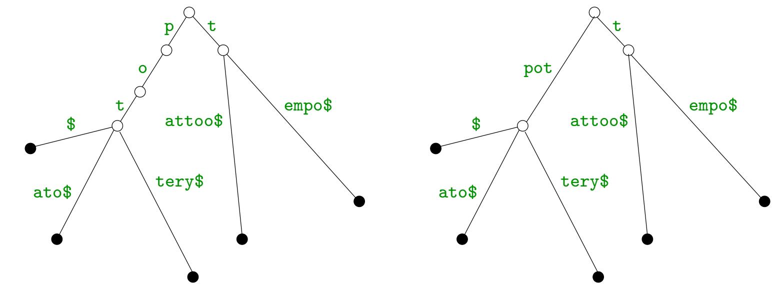
Tries suffer from a large number nodes, $\Omega(||\mathcal{R}||)$ in the worst case.

- 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.

Compact tries reduce the number of nodes by replacing *branchless path segments* with a single edge.

- Leaf path compaction applies this to path segments leading to a leaf. The number of nodes is now $\mathcal{O}(dp(\mathcal{R}))$.
- Full path compaction applies this to all path segments. Then every internal node has at least two children. In such a tree, there is always more leaves than internal nodes. Thus the number of nodes is $\mathcal{O}(|\mathcal{R}|)$.

Example 2.4: Compact tries for
$\mathcal{R} = \{\texttt{pot\$}, \texttt{potato\$}, \texttt{pottery\$}, \texttt{tattoo\$}, \texttt{tempo\$}\}$.



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.

In a fully compact trie, any subtree can be traversed in linear time in 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 Tries

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

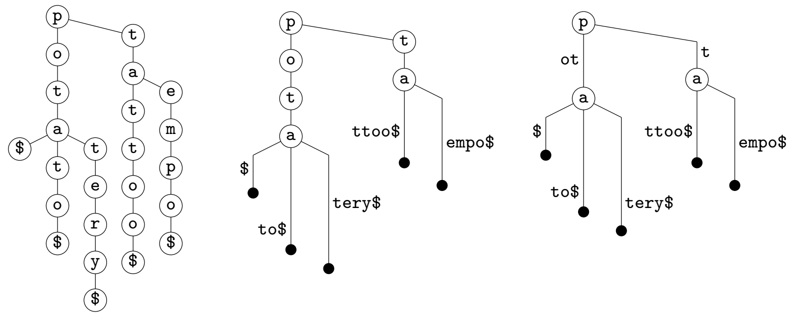
Ternary trie is like a binary search tree except:

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

Ternary tries have variants similar to σ-ary tries:

- A basic ternary trie is a full representation of the strings.
- Compact ternary tries reduce space by compacting branchless path segments.

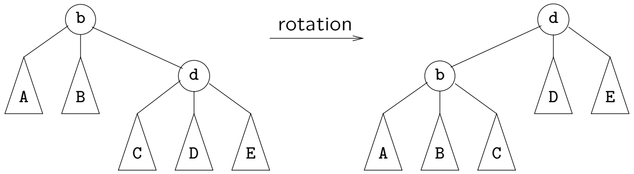
Example 2.5: Ternary tries for
$\mathcal{R} = \{\texttt{pot\$}, \texttt{potato\$}, \texttt{pottery\$}, \texttt{tattoo\$}, \texttt{tempo\$}\}$.



The sizes of ternary tries have the same asymptotic bounds as the corresponding tries: $\mathcal{O}(||\mathcal{R}||)$, $\mathcal{O}(dp(\mathcal{R}))$ and $\mathcal{O}(|\mathcal{R}|)$.

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 balance by inserting the strings in the tree in a random order.

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

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

The time complexities of operations in balanced ternary tries are the same as in the correspoding tries except an *additional* $\mathcal{O}(\log n)$:

- Insertion, deletion, lookup and lcp query for a string S takes $\mathcal{O}(|S| + \log n)$ time.

- Prefix query for a string S takes $\mathcal{O}(|S| + \log n + ||\mathcal{Q}||)$ time in an uncompact ternary trie and $\mathcal{O}(|S| + \log n + |\mathcal{Q}|)$ time in a compact ternary trie, where $\mathcal{Q}$ is the set of strings given as the result of the query.

In tries, where the *child* function is implemented using binary search tree, the time complexities could be $\mathcal{O}(|S| \log \sigma)$, a multiplicative factor instead of an additive factor.

String Binary Search

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

Algorithm 2.6: 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 + 1        // final answer is in the range [left..right)
while right − left > 1 do
    mid ← left + ⌊(right − left)/2⌋
    if k_mid < x then left ← mid
    else right ← mid
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 2.7: Let $A \leq B, B' \leq C$ be strings. Then $lcp(B, B') \geq lcp(A, C)$.

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

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

$\square$

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}, and we know that $S_{left} \le P, S_{mid} \le S_{right}$.
- If we are using LcpCompare, we know $lcp(S_{left}, P)$ and $lcp(P, S_{right})$.

By Lemmas 1.18 and 2.7,

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

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

Algorithm 2.8: 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.

$left \leftarrow 0$; $right \leftarrow n + 1$
$llcp \leftarrow 0$; $rlcp \leftarrow 0$
while $right - left > 1$ *do*
  $mid \leftarrow left + \lfloor (right - left)/2 \rfloor$
  $mlcp \leftarrow \min\{llcp, rlcp\}$
  $(x, mlcp) \leftarrow \text{LcpCompare}(S_{mid}, P, mlcp)$
  if $x =$ "$<$" *then* $left \leftarrow mid$; $llcp \leftarrow mclp$
  else $right \leftarrow mid$; $rlcp \leftarrow mclp$
return $left$

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

We can further improve string binary search using precomputed information about the lcp's between the strings in $\mathcal{R}$.

Consider again the basic situation during string binary search:

- We want to compare P and S_{mid}.
- We have already compared P against S_{left} and S_{right}, and we know $lcp(S_{left}, P)$ and $lcp(P, S_{right})$.

The values $left$ and $right$ depend only on mid. In particular, they do not depend on P. Thus, we can precompute and store the values

$$LLCP[mid] = lcp(S_{left}, S_{mid})$$
$$RLCP[mid] = lcp(S_{mid}, S_{right})$$

Now we know all lcp values between P, S_{left}, S_{mid}, S_{right} except $lcp(P, S_{mid})$. The following lemma shows how to utilize this.

Lemma 2.9: Let $A \le B, B' \le C$ be strings.
(a) If $lcp(A, B) > lcp(A, B')$, then $B < B'$ and $lcp(B, B') = lcp(A, B')$.
(b) If $lcp(A, B) < lcp(A, B')$, then $B > B'$ and $lcp(B, B') = lcp(A, B)$.
(c) If $lcp(B, C) > lcp(B', C)$, then $B > B'$ and $lcp(B, B') = lcp(B', C)$.
(d) If $lcp(B, C) < lcp(B', C)$, then $B < B'$ and $lcp(B, B') = lcp(B, C)$.
(e) If $lcp(A, B) = lcp(A, B')$ and $lcp(B, C) = lcp(B', C)$, then $lcp(B, B') \ge \max\{lcp(A, B), lcp(B, C)\}$.

Proof. Cases (a)–(d) are symmetrical, we show (a). $B < B'$ follows directly from Lemma 1.19. Then by Lemma 1.18, $lcp(A, B') = \min\{lcp(A, B), lcp(B, B')\}$. Since $lcp(A, B') < lcp(A, B)$, we must have $lcp(A, B') = lcp(B, B')$.

In case (e), we use Lemma 1.18:

$$lcp(B, B') \ge \min\{lcp(A, B), lcp(A, B')\} = lcp(A, B)$$
$$lcp(B, B') \ge \min\{lcp(B, C), lcp(B', C)\} = lcp(B, C)$$

Thus $lcp(B, B') \ge \max\{lcp(A, B), lcp(B, C)\}$. $\square$

Algorithm 2.10: String binary search (with precomputed lcps)
Input: Ordered string set $\mathcal{R} = \{S_1, S_2, \ldots, S_n\}$, arrays LLCP and RLCP, query string P.
Output: The number of strings in $\mathcal{R}$ that are smaller than P.

(1) $left \leftarrow 0$; $right \leftarrow n+1$
(2) $llcp \leftarrow 0$; $rlcp \leftarrow 0$
(3) *while* $right - left > 1$ *do*
(4) $\quad mid \leftarrow left + \lfloor (right - left)/2 \rfloor$
(5) $\quad$ *if* $LLCP[mid] > llcp$ *then* $left \leftarrow mid$
(6) $\quad$ *else if* $LLCP[mid] < llcp$ *then* $right \leftarrow mid$; $rlcp \leftarrow LLCP[mid]$
(7) $\quad$ *else if* $RLCP[mid] > rlcp$ *then* $right \leftarrow mid$
(8) $\quad$ *else if* $RLCP[mid] < rlcp$ *then* $left \leftarrow mid$; $llcp \leftarrow RLCP[mid]$
(9) $\quad$ *else*
(10) $\qquad mlcp \leftarrow \max\{llcp, rlcp\}$
(11) $\qquad (x, mlcp) \leftarrow$ LcpCompare$(S_{mid}, P, mlcp)$
(12) $\qquad$ *if* $x =$ "$<$" *then* $left \leftarrow mid$; $llcp \leftarrow mclp$
(13) $\qquad$ *else* $right \leftarrow mid$; $rlcp \leftarrow mclp$
(14) *return* $left$

Theorem 2.11: An ordered string set $\mathcal{R} = \{S_1, S_2, \ldots, S_n\}$ can be preprocessed in $\mathcal{O}(dp(\mathcal{R}))$ time and $\mathcal{O}(n)$ space so that a binary search with a query string P can be executed in $\mathcal{O}(|P| + \log n)$ time.

Proof. The values $LLCP[mid]$ and $RLCP[mid]$ can be computed in $\mathcal{O}(dp_{\mathcal{R}}(S_{mid}))$ time. Thus the arrays $LLCP$ and $RLCP$ can be computed in $\mathcal{O}(dp(\mathcal{R}))$ time and stored in $\mathcal{O}(n)$ space.

The main while loop in Algorithm 2.10 is executed $\mathcal{O}(\log n)$ times and everything except LcpCompare on line (11) needs constant time.

If a given LcpCompare call performs $1 + t$ symbol comparisons, $mclp$ increases by t on line (11). Then on lines (12)–(13), either $llcp$ or $rlcp$ increases by at least t, since $mlcp$ was $\max\{llcp, rlcp\}$ before LcpCompare. Since $llcp$ and $rlcp$ never decrease and never grow larger than $|P|$, the total number of extra symbol comparisons in LcpCompare during the binary search is $\mathcal{O}(|P|)$. □

Binary search can be seen as a search on an *implicit* binary search tree, where the middle element is the root, the middle elements of the first and second half are the children of the root, etc.. The string binary search technique can be extended for arbitrary *binary search trees.*

- Let S_v be the string stored at a node v in a binary search tree. Let $S_<$ and $S_>$ be the closest lexicographically smaller and larger strings stored at *ancestors* of v.

- The comparison of a query string P and the string S_v is done the same way as the comparison of P and S_{mid} in string binary search. The roles of S_{left} and S_{right} are taken by $S_<$ and $S_>$.

- If each node v stores the values $lcp(S_<, S_v)$ and $lcp(S_v, S_>)$, then a search in a balanced search tree can be executed in $\mathcal{O}(|P| + \log n)$ time. Other operations including insertions and deletions take $\mathcal{O}(|P| + \log n)$ time too.

Automata

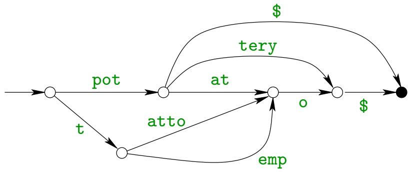
Finite automata are a well known way of representing sets of strings. In this case, the set is often called a **language**.

A trie is a special type of an automaton.

- Trie is generally not a *minimal* automaton.

- Trie techniques including path compaction and ternary branching can be applied to automata.

Example 2.12: Compacted minimal automaton for $\mathcal{R} = \{$pot\$, potato\$, pottery\$, tattoo\$, tempo\$$\}$.



Automata are much more powerful than tries in representing languages:

- Infinite languages
- Nondeterministic automata
- Even an acyclic, deterministic automaton can represent a language of exponential size.

Automata do not support all operations of tries:

- Insertions and deletions
- Satellite data, i.e., data associated to each string.

3. Exact String Matching

Let $T = T[0..n)$ be the *text* and $P = P[0..m)$ the *pattern*. We say that P *occurs* in T at position j if $T[j..j+m) = P$.

Example: $P =$ `aine` occurs at position 6 in $T =$ `karjalainen`.

In this part, we will describe algorithms that solve the following problem.

Problem 3.1: Given text $T[0..n)$ and pattern $P[0..m)$, report the first position in T where P occurs, or n if P does not occur in T.

The algorithms can be easily modified to solve the following problems too.

- Is P a factor of T?
- Count the number of occurrences of P in T.
- Report all occurrences of P in T.

The naive, brute force algorithm compares P against $T[0..m)$, then against $T[1..m+1)$, then against $T[2..m+2)$ etc. until an occurrence is found or the end of the text is reached.

Algorithm 3.2: Brute force
Input: text $T = T[0 \ldots n)$, pattern $P = P[0 \ldots m)$
Output: position of the first occurrence of P in T
(1) $i \leftarrow 0; j \leftarrow 0$
(2) *while* $i < m$ *and* $j < n$ *do*
(3) *if* $P[i] = T[j]$ *then* $i \leftarrow i + 1; j \leftarrow j + 1$
(4) *else* $j \leftarrow j - i + 1; i \leftarrow 0$
(5) *if* $i = m$ *then* output $j - m$ *else* output n

The worst case time complexity is $\mathcal{O}(mn)$. This happens, for example, when $P = \mathtt{a}^{m-1}\mathtt{b} = \mathtt{aaa..ab}$ and $T = \mathtt{a}^n = \mathtt{aaaaaa..aa}$.

Knuth–Morris–Pratt

The Brute force algorithm forgets everything when it moves to the next text position.

The Morris–Pratt (MP) algorithm remembers matches. It never goes back to a text character that already matched.

The Knuth–Morris–Pratt (KMP) algorithm remembers mismatches too.

Example 3.3:

Brute force	Morris–Pratt	Knuth–Morris–Pratt
`ainaisesti-ainainen`	`ainaisesti-ainainen`	`ainaisesti-ainainen`
ainai~~n~~en (6 comp.)	ainai~~n~~en (6)	ainai~~n~~en (6)
~~a~~inainen (1)	ai~~n~~ainen (1)	~~a~~inainen (1)
~~a~~inainen (1)	~~a~~inainen (1)	
ai~~n~~ainen (3)		
~~a~~inainen (1)		
~~a~~inainen (1)		

MP and KMP algorithms never go backwards in the text. When they encounter a mismatch, they find another pattern position to compare against the same text position. If the mismatch occurs at pattern position i, then $fail[i]$ is the next pattern position to compare.

The only difference between MP and KMP is how they compute the *failure function* $fail$.

Algorithm 3.4: Knuth–Morris–Pratt / Morris–Pratt
Input: text $T = T[0 \dots n)$, pattern $P = P[0 \dots m)$
Output: position of the first occurrence of P in T
(1) compute $fail[0..m]$
(2) $i \leftarrow 0; j \leftarrow 0$
(3) *while* $i < m$ *and* $j < n$ *do*
(4)   *if* $i = -1$ *or* $P[i] = T[j]$ *then* $i \leftarrow i + 1; j \leftarrow j + 1$
(5)   *else* $i \leftarrow fail[i]$
(6) *if* $i = m$ *then* output $j - m$ *else* output n

- $fail[i] = -1$ means that there is no more pattern positions to compare against this text positions and we should move to the next text position.
- $fail[m]$ is never needed here, but if we wanted to find all occurrences, it would tell how to continue after a full match.

We will describe the MP failure function here. The KMP failure function is left for the exercises.

- When the algorithm finds a mismatch between $P[i]$ and $T[j]$, we know that $P[0..i) = T[j - i..j)$.
- Now we want to find a new $i' < i$ such that $P[0..i') = T[j - i'..j)$. Specifically, we want the largest such i'.
- This means that $P[0..i') = T[j - i'..j) = P[i - i'..i)$. In other words, $P[0..i')$ is the *longest proper border* of $P[0..i)$.

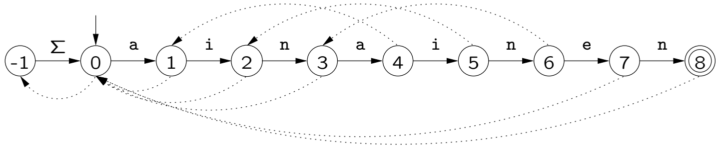
Example: `ai` is the longest proper border of `ainai`.

- Thus $fail[i]$ is the length of the longest proper border of $P[0..i)$.
- $P[0..0) = \varepsilon$ has no proper border. We set $fail[0] = -1$.

Example 3.5: Let $P =$ `ainainen`.

i	$P[0..i)$	border	$fail[i]$
0	ε	–	-1
1	`a`	ε	0
2	`ai`	ε	0
3	`ain`	ε	0
4	`aina`	`a`	1
5	`ainai`	`ai`	2
6	`ainain`	`ain`	3
7	`ainaine`	ε	0
8	`ainainen`	ε	0

The (K)MP algorithm operates like an automaton, since it never moves backwards in the text. Indeed, it can be described by an automaton that has a special *failure transition*, which is an ε-transition that can be taken only when there is no other transition to take.



An efficient algorithm for computing the failure function is very similar to the search algorithm itself!

- In the MP algorithm, when we find a match $P[i] = T[j]$, we know that $P[0..i] = T[j - i..j]$. More specifically, $P[0..i]$ is the longest prefix of P that matches a suffix of $T[0..j]$.
- Suppose $T = \#P[1..m)$, where $\#$ is a symbol that does not occur in P. Finding a match $P[i] = T[j]$, we know that $P[0..i]$ is the longest prefix of P that is a proper suffix of $P[0..j]$. Thus $fail[j + 1] = i + 1$.

Algorithm 3.6: Morris–Pratt failure function computation
Input: pattern $P = P[0 \dots m)$
Output: array $fail[0..m]$ for P
(1) $i \leftarrow -1; j \leftarrow 0;\ fail[j] \leftarrow i$
(2) *while* $j < m$ *do*
(3)   *if* $i = -1$ *or* $P[i] = P[j]$ *then* $i \leftarrow i + 1; j \leftarrow j + 1;\ fail[j] \leftarrow i$
(4)   *else* $i \leftarrow fail[i]$
(5) output $fail$

- When the algorithm reads $fail[i]$ on line 4, $fail[i]$ has already been computed.