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 $S_{right+1}$, and we know $lcp(S_{left}, P)$ and $lcp(P, S_{right+1})$.

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

$$LLCP[mid] = lcp(S_{left}, S_{mid})$$
$$RLCP[mid] = lcp(S_{mid}, S_{right+1})$$

Now we know all lcp values between P, S_{left}, S_{mid}, $S_{right+1}$ except $lcp(P, S_{mid})$. The following lemma shows how to utilize this.

Lemma 3.33: Let $A \leq B, B' \leq C$ 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 $lcp(B, B') \geq \max\{lcp(A, B), lcp(B, C)\}$.

Proof. Cases (a)–(d) are symmetrical, we show (a). $B < B'$ follows directly from Lemma 3.18. Then by Lemma 3.17, $lcp(A, B') = \min\{lcp(A, B), lcp(B, B')\}$. Since $lcp(A, B') < lcp(A, B)$, we must have $lcp(A, B') = lcp(B, B')$.

In case (e), we use Lemma 3.17:

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

Thus $lcp(B, B') \geq \max\{lcp(A, B), lcp(B, C)\}$. □

Algorithm 3.34: 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) $left \leftarrow 0$; $right \leftarrow n$
(2) $llcp \leftarrow 0$; $rlcp \leftarrow 0$
(3) *while* $left < right$ *do*
(4) $\quad mid \leftarrow \lceil (left + right)/2 \rceil$
(5) $\quad$ *if* $LLCP[mid] > llcp$ *then* $left \leftarrow mid$
(6) $\quad$ *else if* $RLCP[mid] > rlcp$ *then* $right \leftarrow mid - 1$
(7) $\quad$ *else if* $llcp > LLCP[mid]$ *then* $right \leftarrow mid - 1$; $rlcp \leftarrow LLCP[mid]$
(8) $\quad$ *else if* $rlcp > RLCP[mid]$ *then* $left \leftarrow mid$; $llcp \leftarrow RLCP[mid]$
(9) $\quad$ *else*
(10) $\quad\quad mlcp \leftarrow \max\{llcp, rlcp\}$
(11) $\quad\quad (x, mlcp) \leftarrow \text{LcpCompare}(S_{mid}, P, mlcp)$
(12) $\quad\quad$ *if* $x =$ "$<$" *then* $left \leftarrow mid$; $llcp \leftarrow mclp$
(13) $\quad\quad$ *else* $right \leftarrow mid - 1$; $rlcp \leftarrow mclp$
(14) *return* $left$

Theorem 3.35: 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 3.7.7 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 $rlcp$ increase 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 $S_{right+1}$ 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.

4. Suffix Trees and Arrays

Let $T = T[0..n)$ be the text. For $i \in [0..n]$, let T_i denote the *suffix* $T[i..n)$. Furthermore, for any subset $C \in [0..n]$, we write $T_C = \{T_i \mid i \in C\}$. In particular, $T_{[0..n]}$ is the *set of all suffixes* of T.

Suffix tree and suffix array are *search data structures* for the set $T_{[0..n]}$.

- Suffix tree is a *compact trie* for $T_{[0..n]}$.
- Suffix array is a *ordered array* for $T_{[0..n]}$.

They support fast *exact string matching* on T:

- A pattern P has an occurrence starting at position i if and only if P is a *prefix* of T_i.
- Thus we can find all occurrences of P by a *prefix search* in $T_{[0..n]}$.

There are numerous other applications too, as we will see later.

The set $T_{[0..n]}$ contains $|T_{[0..n]}| = n + 1$ strings of total length $||T_{[0..n]}|| = \Theta(n^2)$. It is also possible that $DP(T_{[0..n]}) = \Theta(n^2)$, for example, when $T = \mathtt{a}^n$ or $T = XX$ for any string X.

- Trie with $||T_{[0..n]}||$ nodes and ternary tree with $DP(T_{[0..n]})$ nodes would be too large.

- Compact trie with $\mathcal{O}(n)$ nodes and an ordered array with $n + 1$ entries have linear size.

- Binary search tree with $\mathcal{O}(n)$ nodes would be an option too, but an ordered array is a better choice for a static text. We do not cover the case of dynamic, changing text on this course: it a non-trivial problem because changing a single symbol can affect a large number of suffixes.

Even for a compact trie or an ordered array, we need a *specialized construction algorithm*, because any general construction algorithm would need $\Omega(DP(T_{[0..n]}))$ time.

Suffix Tree

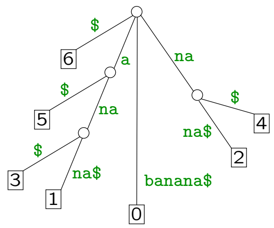
The suffix tree of a text T is the compact trie of the set $T_{[0..n]}$ of all suffixes of T.

We assume that there is an extra character $\$ \notin \Sigma$ at the end of the text. That is, $T[n] = \$$ and $T_i = T[i..n]$ for all $i \in [0..n]$. Then:

- No suffix is a prefix of another suffix, i.e., the set $T_{[0..n]}$ is *prefix free*.
- All nodes in the suffix tree representing a suffix are leaves.

This simplifies algorithms.

Example 4.1: $T = \mathtt{banana\$}$.



As with tries, there are many possibilities for implementing the child operation. We again avoid this complication by assuming that σ *is constant*. Then the size of the suffix tree is $\mathcal{O}(n)$:

- There are exactly $n+1$ leaves and at most n internal nodes.

- There are at most $2n$ edges. The edge labels are factors of the text and can be represented by pointers to the text.

Given the suffix tree of T, all occurrences of P in T can be found in time $\mathcal{O}(|P| + occ)$, where occ is the number of occurrences.