Prefix Matching

Yufei Tao

KAIST

May 1, 2013

Let us now consider the prefix matching problem on strings:

Problem

Let S be a set of strings, each of which has a unique integer id. Given a query string q, a query reports all the ids of the strings $s \in S$ such that q is a prefix of s.

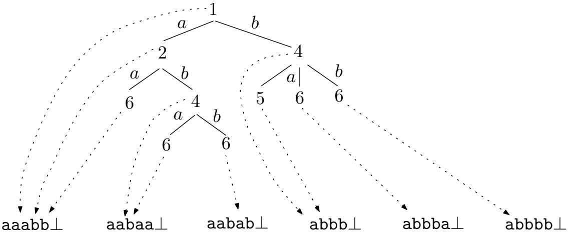
Example

Let $S = \{\texttt{abbba}\bot, \texttt{aabaa}\bot, \texttt{aaabb}\bot, \texttt{abbb}\bot, \texttt{aabab}\bot, \texttt{abbbb}\bot\}$, where the strings have ids 1, 2, ..., 6, respectively. Then:

- for $q = \texttt{ab}$, we should return ids 1, 4, 6.
- for $q = \texttt{aab}$, return 2, 5.
- for $q = \texttt{ba}$, return nothing.

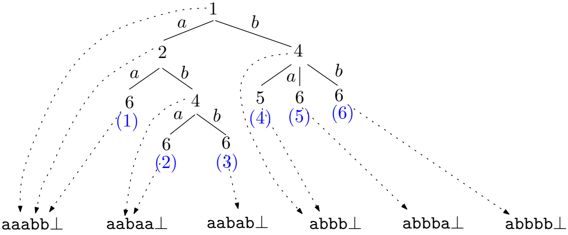
We will show how to augment the Patricia trie to answer prefix matching queries efficiently.

Here is the Patricia trie for our example in the previous slide:



For each string s, define its rank as 1 greater than the number of strings in S (alphabetically) less than s. In other words, the smallest string in S has rank 1, while the largest string in S has rank $|S|$. We store the rank of s at its leaf.

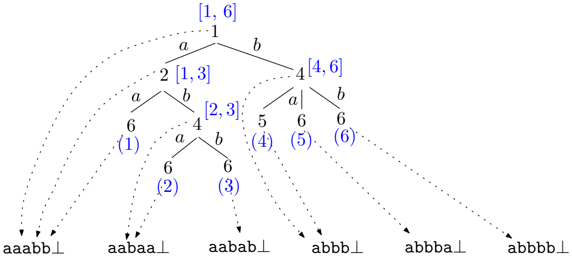
The ranks are written in blue numbers in brackets:



Augmentation 2

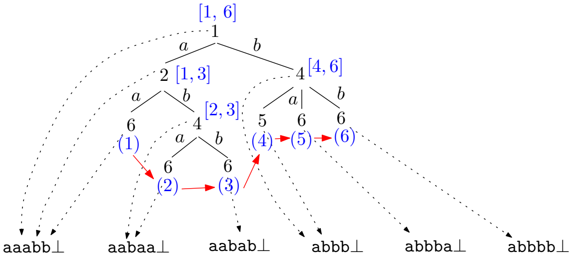
For each internal node u in the trie, store a rank interval $[l, r]$, where l (r) is the smallest (largest) rank of the leaves in the subtree of u.

The rank intervals are given in blue:



Chain up all leaf nodes as follows: for each $i \in [1, n-1]$, store at the leaf with rank i a pointer to the leaf with rank $i+1$. Call these pointers the bottom pointers. The bottom pointer of the leaf with rank n is nil.

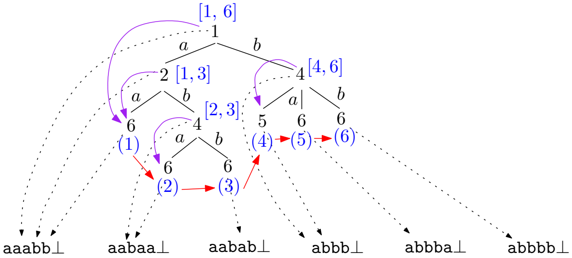
The bottom pointers are shown in red.



Augmentation 4

At each internal node u, store a down pointer to the smallest leaf (in the alphabetic order) in the subtree of u.

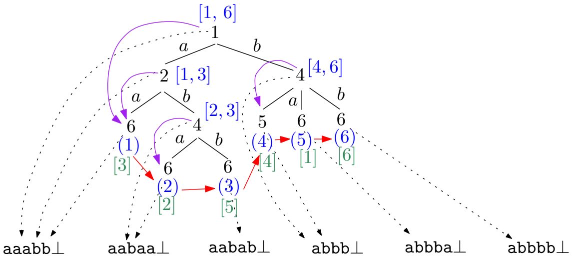
The down pointers are shown in purple.



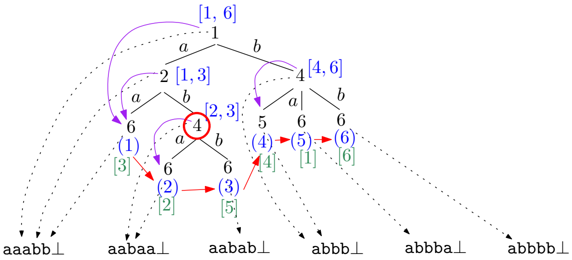
Augmentation 5

Finally, for each string $s \in S$, store its id at the leaf corresponding to s.

The ids are given in green.



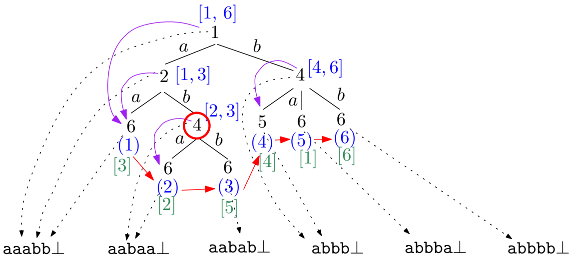
To answer a prefix matching query with string q, first find the highest node u such that q is a prefix of the possible prefix represented by u.



Suppose that $q =$ aab. In the above figure, u is the node in the red circle (whose possible prefix is aaba). We know that all the leaves in the subtree of u correspond to strings that have q as a prefix.

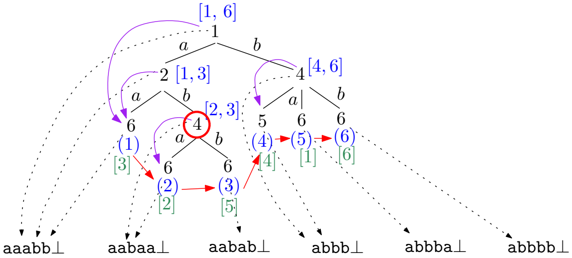
Query

Follow the down pointer of u to the left most leaf z in its subtree.



In the above, we get to the leaf node with rank 2.

Report the id of z. Then, follow the bottom pointer of z to the leaf z' with the next rank. If the rank of z' is in the rank interval of u, it means that z' is still in the subtree of u. In that case, we report the id of z', and repeat the above. Otherwise (i.e., z' is outside the subtree of u), we stop.



In the above, we visit the leaf nodes with ranks 2, 3, and 4, but report only ids 2 and 5. Note that rank 4 is outside the rank interval $[2, 3]$ of node u (which is the node in the red circle).

The above structure has the following performance:

Theorem

For the prefix matching problem, our structure consumes $O(|S|)$ space, and answers a query with string q in $O(|q||A| + k)$ time, where k is the number of ids reported, and A is the alphabet.

For $|A| = O(1)$, the query time of the above theorem becomes $O(|q| + k)$. For large alphabets, we can combine the above structure with the balanced trie to obtain:

Theorem

For the prefix matching problem, there is a structure that consumes $O(|S|)$ space, and answers a query with string q in $O(|q| + \log |S| + k)$ time.