<u>CS201c: Programming Evaluation 4 solutions</u>

We maintain two tries T and U. The description of these tries is given below. Every insert or remove call is made on both tries as follows:

<u>insert(bit string b):</u>

```
val=insert_T(b)
if (val == 1):
    insert_U(b)
return val
```

<u>remove(bit string b):</u>

```
val=remove_T(b)
if (val == 1):
    remove_U(b)
return val
```

<u>Part 1: Trie T</u>

We will maintain a **trie** T of bit strings currently in set S.

Every node v of trie T will have three fields:

- *v.n* - the total number of leaf nodes in v's subtree
- *v.zero* - pointer to be followed from v if next bit in the string is 0
- *v.one* - pointer to be followed from v if next bit in the string is 1

For a leaf node, both pointers are NULL.

<u>insert_T(bit string b).</u> (coded like the usual insert in a trie):

(i) Follow the path labeled by bit string b in trie T.
(ii) If you reach a leaf node at depth 32 :- string b already exists in set S, return 0.
(iii) If you come across a NULL link at a depth d < 32, insert b into the trie T at that link.
(You will need to create 32-d new nodes for the unmatched part of bit string b. Initial value of v.n field for these nodes is 0.)
(iv) Increase the *v.n* field of every node in the insertion path by 1.

<u>remove_T(bit string b)</u> (coded like the usual delete in a trie):

(i) Follow the path labeled by bit string b in trie T.
(ii) If you come across a NULL link at a depth d < 32 :- b does not exist in set S, return 0.
(iii) If you reach a leaf node w at depth 32. Decrease the v.n field of every node v on the path from w to root node by 1.

(iv) Walk towards the root from the leaf node w, keep removing nodes till you reach the first node v with *v.n* > 0.

First, we define an auxiliary function ``search2_new" of three arguments:

search2_new(p, i, v):

```
 if (i == |p|+1):
        return ( (i-1) * v.n )

z = i-th bit of pattern p
v1 = v.zero
v2 = v.one
L1 = number of leaf nodes in subtree of v1 in trie T
L2 = number of leaf nodes in subtree of v2 in trie T

if (z == '?'):
        return [ search2_new(p, i+1, v1) + search2_new(p, i+1, v2) ]
else if (z == '0'):
        return [ search2_new(p, i+1, v1) + ( (i-1) * L2) ]
else if (z=='1'):
        return [ ( (i-1) * L1) + (search2_new(p, i+1, v2) ]
```

search2(pattern p):

The output of this function is the return value of the function call:

```
search2_new(p, 1, r)  // here r is the root of trie T
```

Part 2: Trie U

For the set S of current strings, let suffix(S) denote the *multiset* of all suffixes of strings in S. (Every string b in S has 32 bits, and hence has exactly 32 suffixes of lengths 1, 2, …, 32 respectively.)

Note that, unlike S, suffix(S) is a multiset - the same string is allowed to occur multiple times. Thus, suffix(S) will contain exactly 32*|S| elements, where |S| is the cardinality of set S.

Every node v of trie U will have three fields:

- *v.num_suffixes* - the total number of strings in suffix(S) in the subtree rooted at v.
- *v.zero* - pointer to be followed from v if next bit in the string is 0
- *v.one* - pointer to be followed from v if next bit in the string is 1

insert_single_suffix_U(bit string s).:

(i) Follow the path labeled by bit string s in trie U.
(ii) If you reach a node w at depth |s| :- string s already exists in set suffix(S). Increment *v.num_suffixes* by 1 for all nodes v on the path from root to w. Return 1.
(iii) If you come across a NULL link at a depth d < |s|, insert s into the trie U at that link. (You will need to create |s|-d new nodes for the unmatched part of bit string s.)
(iv) Increment *v.num_suffixes* by 1 for all nodes on the path from the new leaf node created to the root of U.
(v) Return 1.
[Since U maintains a multiset, *insert_single_suffix_U* is always successful.]

remove_single_suffix_U(bit string s):

(i) Follow the path labeled by bit string s in trie U.
(ii) If you come across a NULL link at a depth d < |s| :- s does not exist in set suffix(S), return 0.
(iii) If you reach a node w at depth |s|. Decrease the *v.num_suffixes* field for all nodes on the path from w to root by 1.
(iv) If there are no nodes in the subtree of w (except itself), walk towards the root from w and keep removing nodes till you reach the first node v with *v.num_suffixes* > 0.
(v) Return 1.

insert_U(bit string b):

For every suffix s of b:
    Call insert_single_suffix_U(s)

remove_U(bit string b):

Check if b exists in trie T. If not, return 0.

Otherwise, for every suffix s of b:
    Call remove_single_suffix_U(s)

We now describe the function search1:

search1(pattern p):

Search for pattern p in trie U.
If p is not found, return 0.
Else, suppose pattern p is found at node v.
Return *v.num_suffixes*