① Spell checker

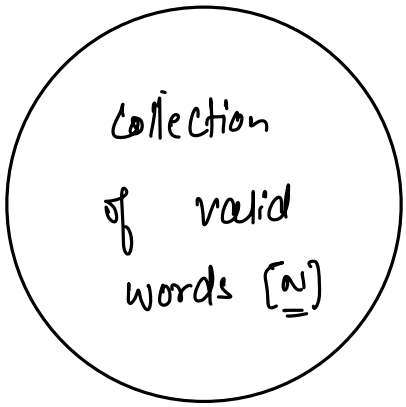
playground

loyal

friend

lyfe

length → l.



HashSet<String>

T.C → O(l)

because of hashcode
+
char by char comparison.

Trie:

↳ hierarchical data structure

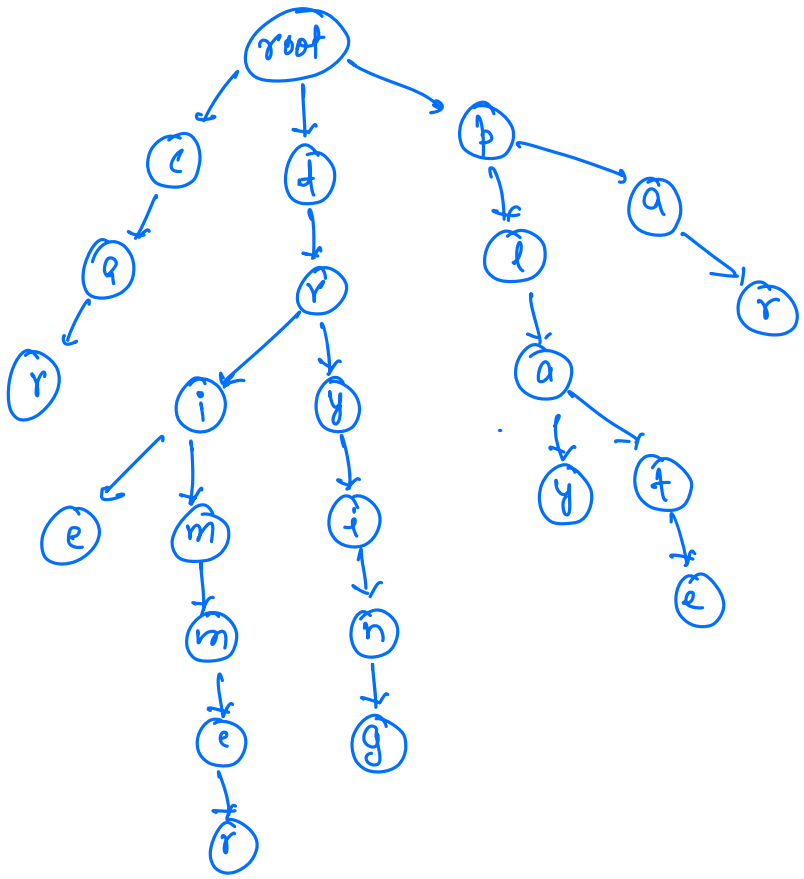
↳ prefix-tree

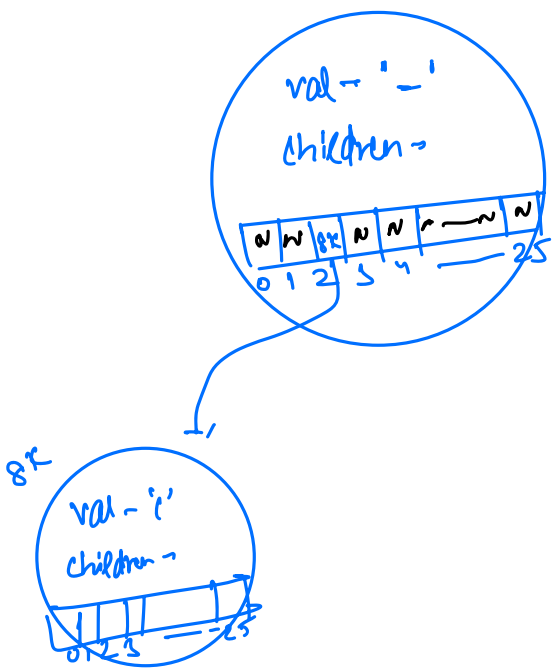
↳ It is used for information retrieval.

It is a data-structure which stores the information from top to down.

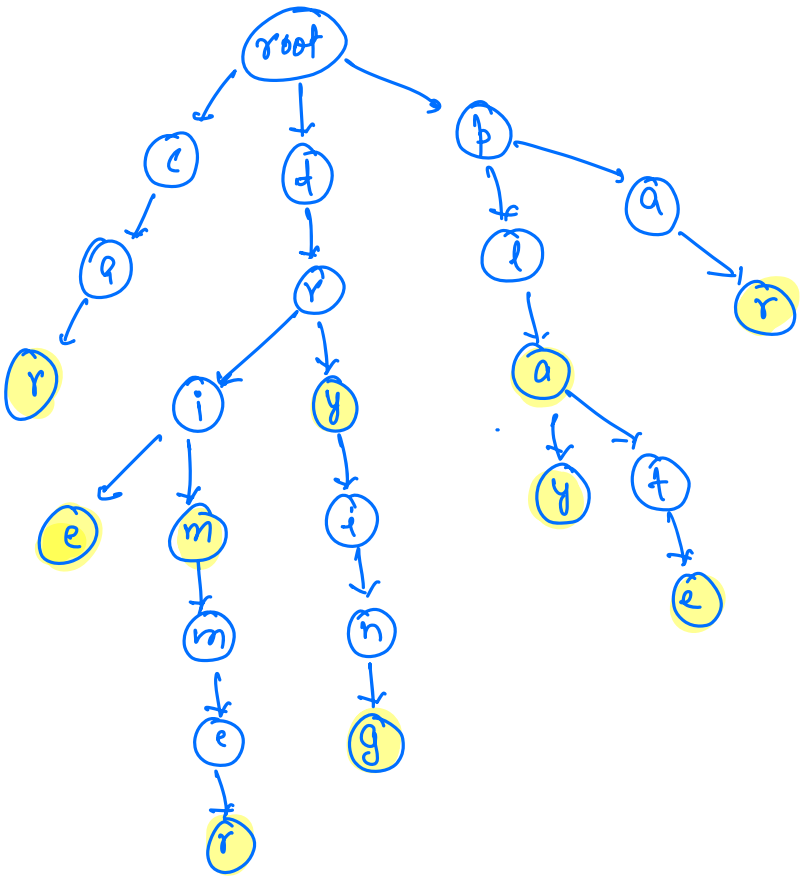
dict →

try	trim	trie	play	trying
plate	car	par	trimmer	pla






```
class Node {
    char val;
    Node children[];
    public Node (char v){
        val = v;
        children = new Node[26];
    }
}
```



search(trie) → true

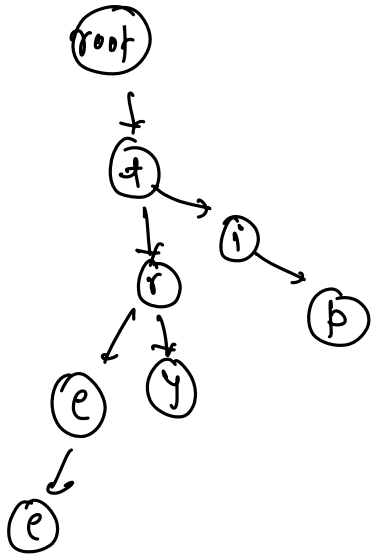
search(trim) → true.

search(tri) → ✗

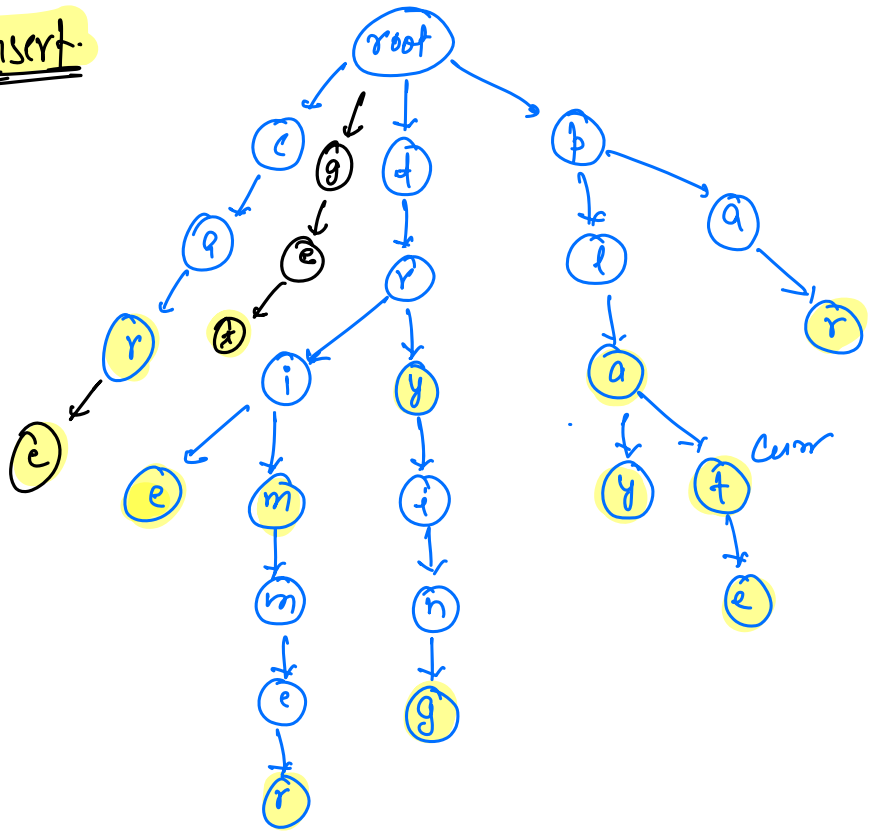
search(pla) → true.

```
class Node {
    char val;
    Node children[26];
    boolean eow;
    public Node(char v){
        val = v;
        children = new Node[26];
        eow = false;
    }
}
```

trie, try, tip



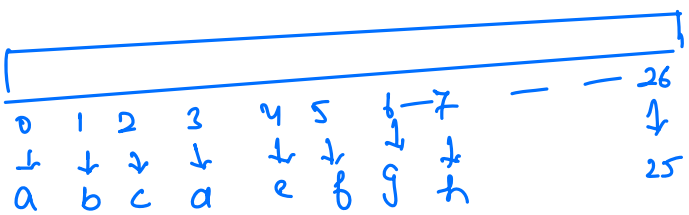
insert.



✓ insert("care");

✓ insert("get");

✓ insert("plat");



code.

```
void insert(Node root, String word){
    Node curr = root;
    for(int i=0; i<len(word); i++){
        char ch = word[i];
        if(curr.children[ch-'a'] == NULL){
            curr.children[ch-'a'] = new Node(ch);
        }
        curr = curr.children[ch-'a'];
    }
    curr.eow = true;
}
```

a -'a'-> 0
b -'a'-> 1
c -'a'-> 2
d -'a'-> 3
z -'a'-> 25

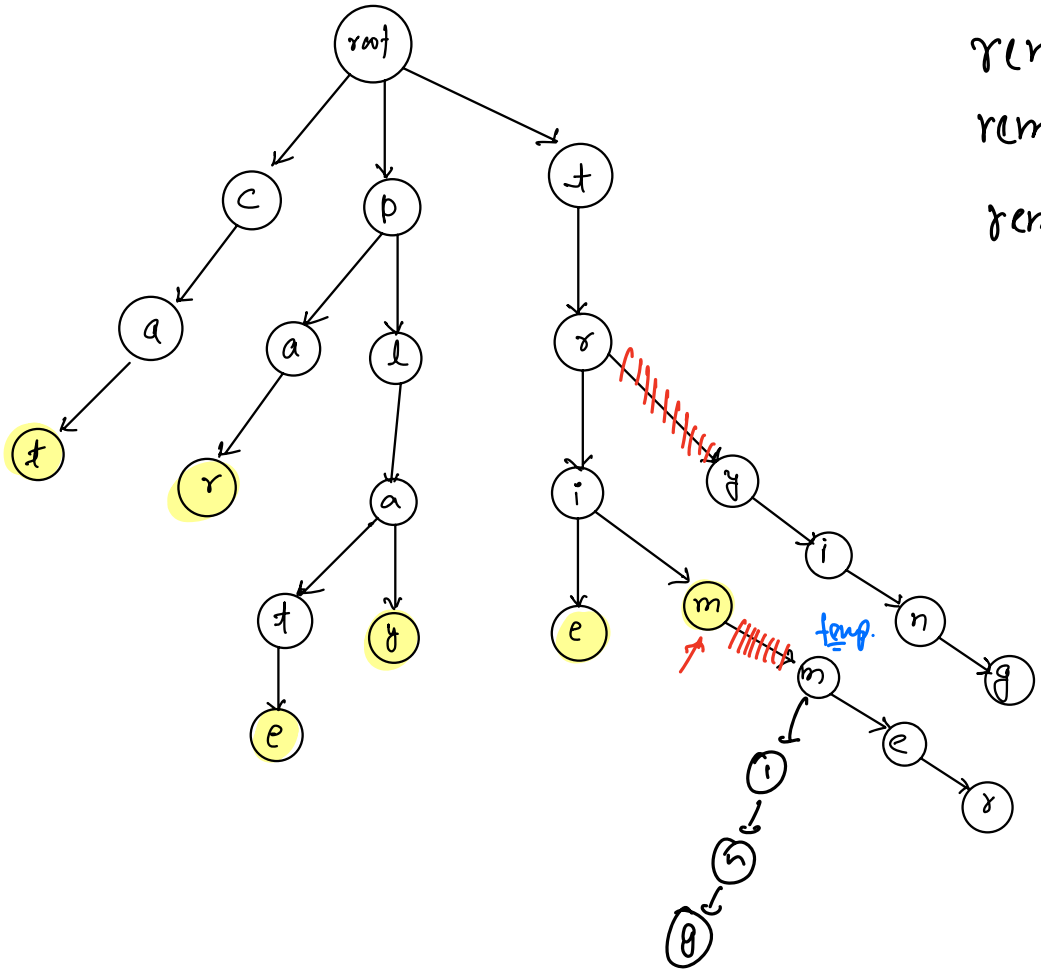
T.C → O(l)
S.C → O(l)

Search.

```
boolean search ( Node root, String word) {
    Node curr = root;
    for ( int i = 0; i < len(word); i++) {
        char ch = word[i];
        if ( curr.children [ch - 'a'] == NULL) {
            return false;
        }
        curr = curr.children [ch - 'a'];
    }
    return curr.eow;
}
```

T.C → O(L)
S.C → O(1)

deletion.



remove (try)
remove (trimmer)
remove (trying)

n(→y

① if word, that we want to remove, is a prefix to another word, then we can't remove any no. of nodes.

② Nodes that can't be removed from trie →

→ node where eow is marked as true;

→ node which have more than 1 children.

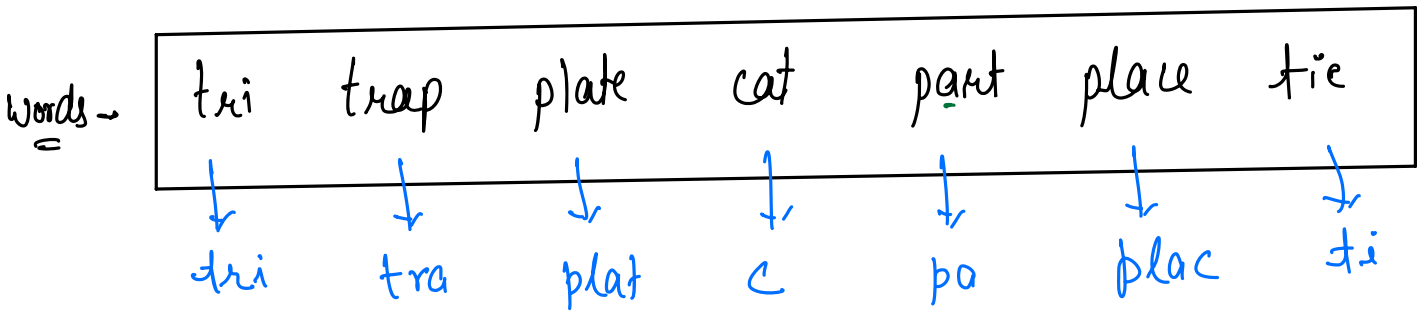
pseudo-code.

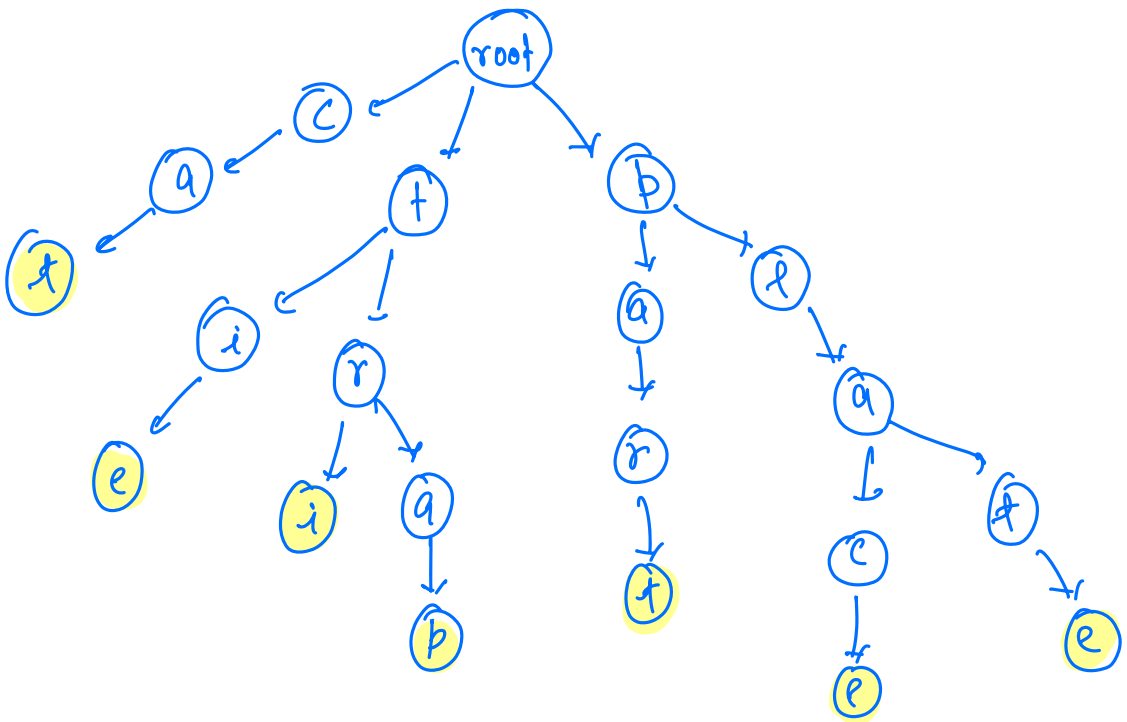
```
void deleteNode ( Node root, String word) {
    Node curr = root , temp → root. , nc → word[0];
    for( i=0; i < len(word); i++){
        char ch = word[i];
        int count = 0;   // no. of children of curr node
        for( j=0; j<26; j++){
            if( curr.children[j] != NULL){
                count++;
            }
        }
        if( count > 1 || curr.eow == true){
            temp = curr; nc = word[i];
        }
        curr = curr.children[ch-'a'];
    }
    curr.eow = false;
    count = 0;
    for( j=0; j<26; j++){
        if( curr.children[j] != NULL){
            count++;
        }
    }
    if( count == 0){ temp.children[nc-'a'] == NULL }
}
```

T.C → O(l)
S.C → O(1)

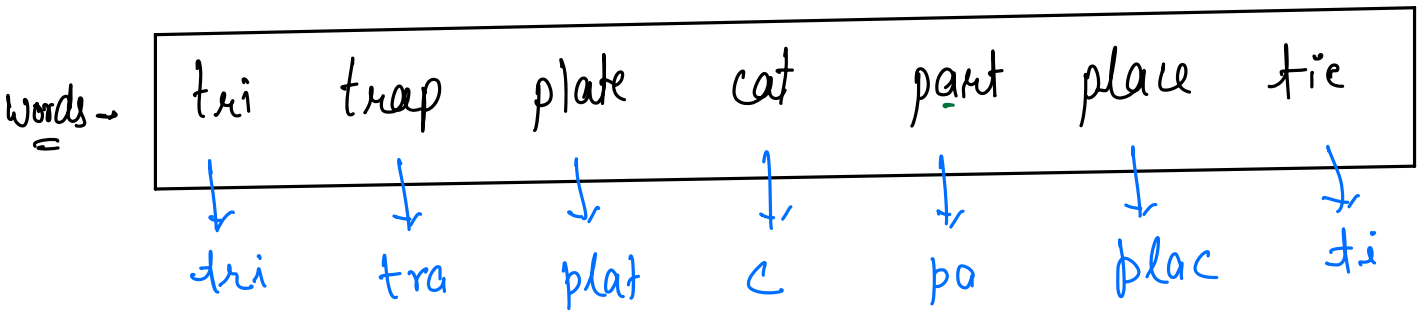
Q.1 Find shortest unique prefix to represent each word.

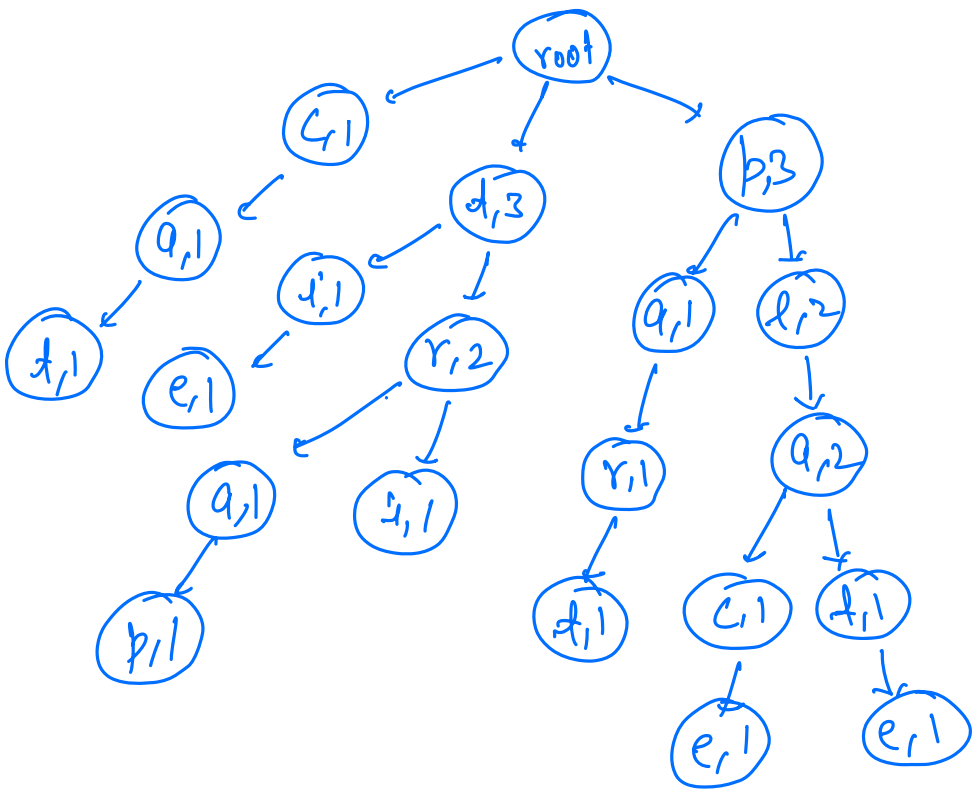
Note: Assume that no word is prefix of another word.
In other words, the representation is always possible.





→ For every word, find the last node with no. of children > 1. ans → all characters till this node + next character



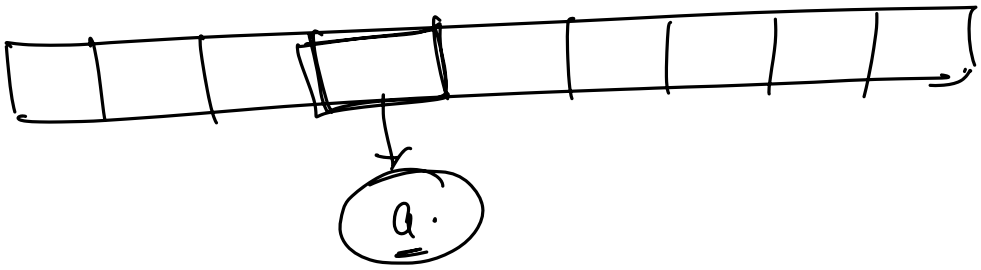


plac

insert words in the trie by maintaining the frequency of prefix-characters.

Strings Pattern Matching [Rabin Karp]

String a ⟶ hashCode() → [0 to N-1]



Str → 0 1 2 3 4 — — N-1 (N)

↑

$S[0] \times 31^{N-1} + S[1] \times 31^{N-2} + \cdots + S[N-1] \times 31^{0}$

100% PSP ⇒ Target