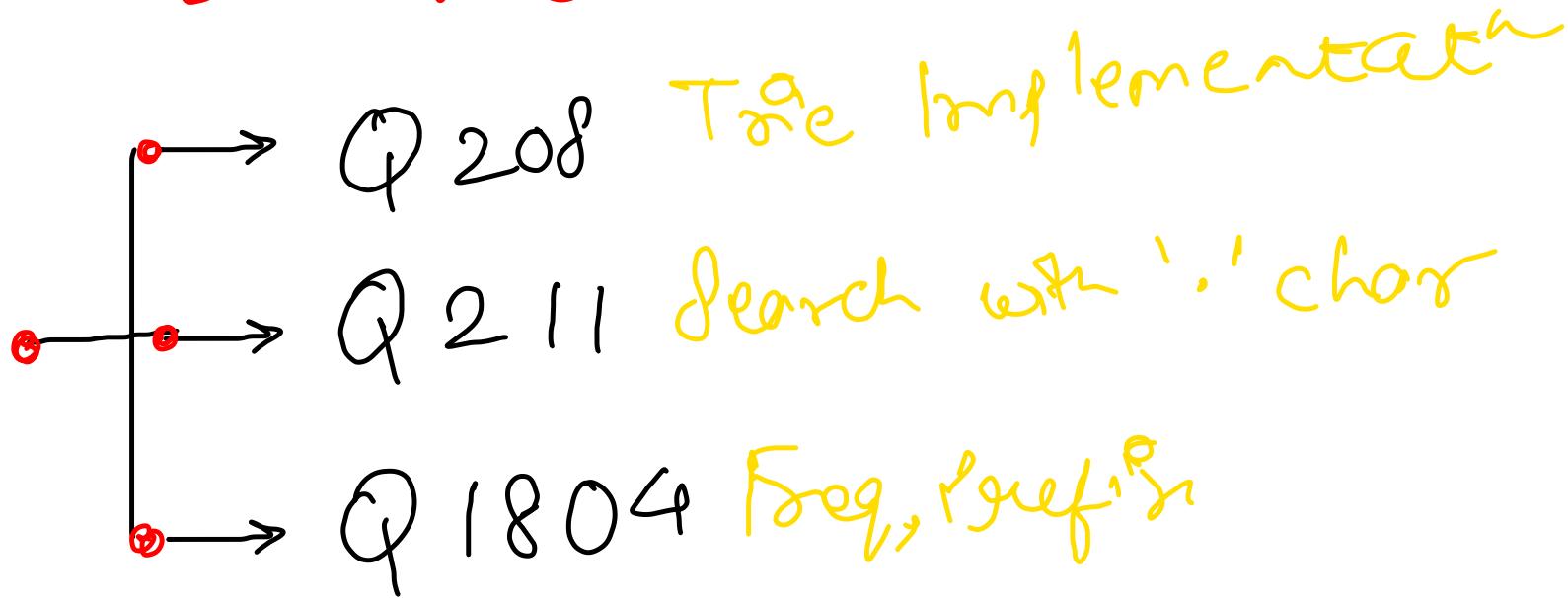


TRIE IMPLEMENTATION



Advantages & Disadvantages
of Trie over Hashing
Applications of Tries

Autocomplete System
Word Suggestion System
Spellchecker / Autocorrect
Prefix Matching
Searching { Phone Directory, Dictionary }

Dictionary Problems

~~Q720~~ longest Word in Dictionary

~~Q677~~ weighted Prefix search

~~Q648~~ Replace Word with Prefix

~~Q14~~ longest Common Prefix

Q676 Magic Dictionary

shortest Unique Prefix

~~GFG~~ (Q1698) Count Distinct Substrings

~~HN~~ GFG Most Frequent Word

Q1268 Search Suggestion Systems (DS)

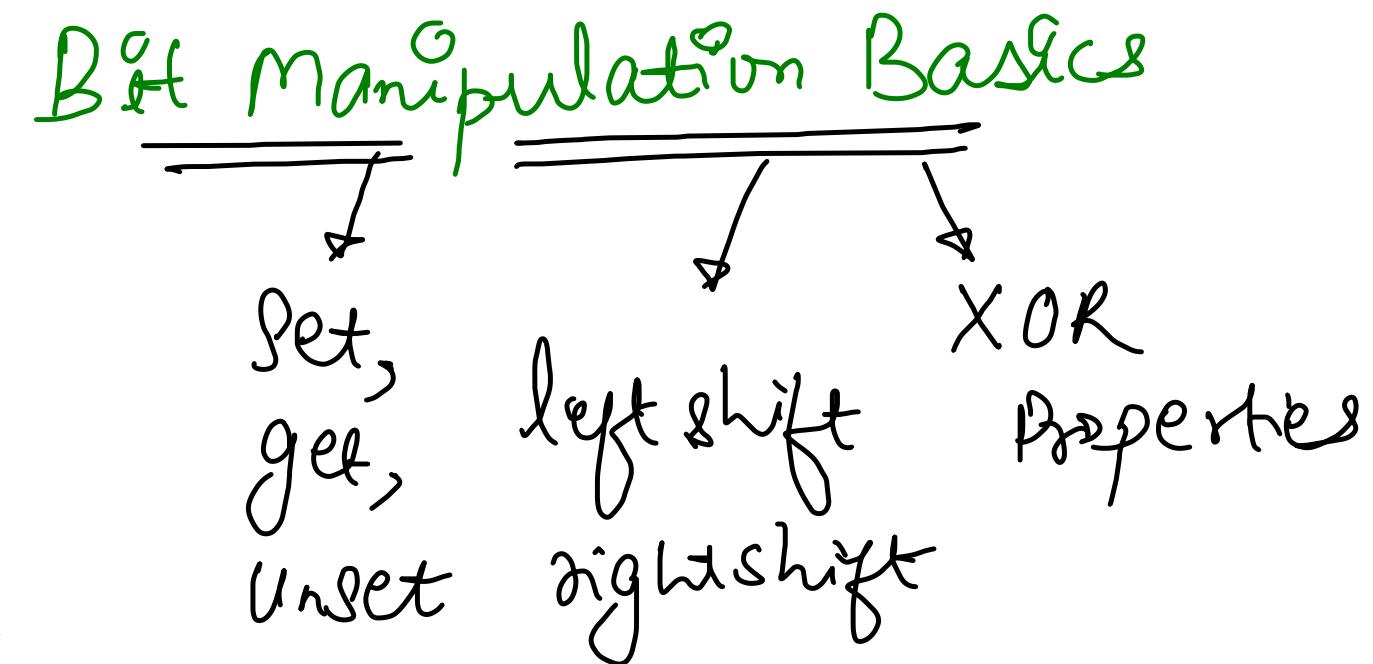
Q1032 Stream of Characters → Hard

~~Q212~~ Word Search - II

Q336 Palindrome Pairs → Hard

Q745 Prefix & Suffix Search → Hard

XOR Problems



Q421 Maximum XOR Pair - I

Q1707 Maximum XOR Pair - II

Q1803 XOR Pairs in Range

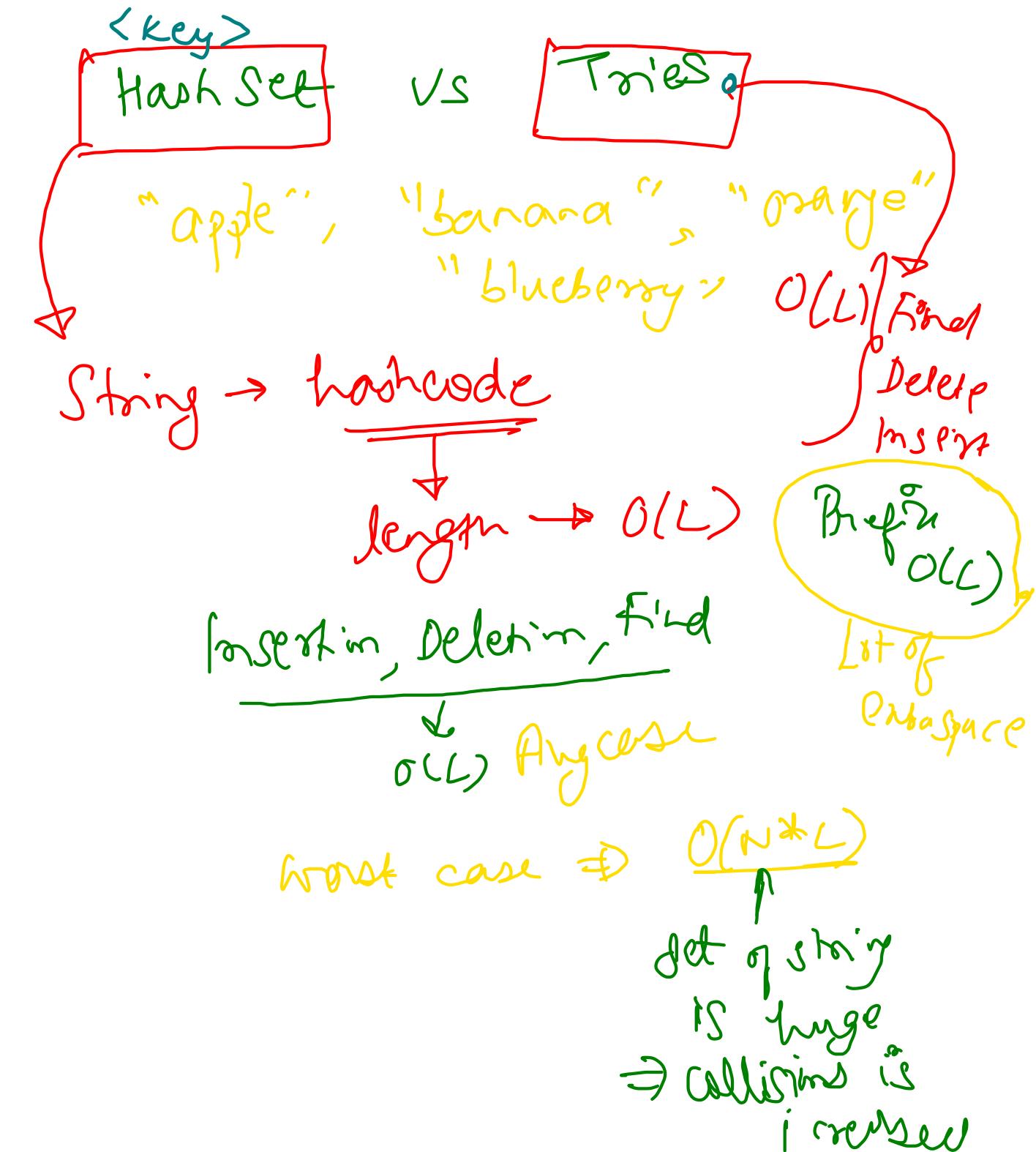
QFG Subarrays with XOR < K

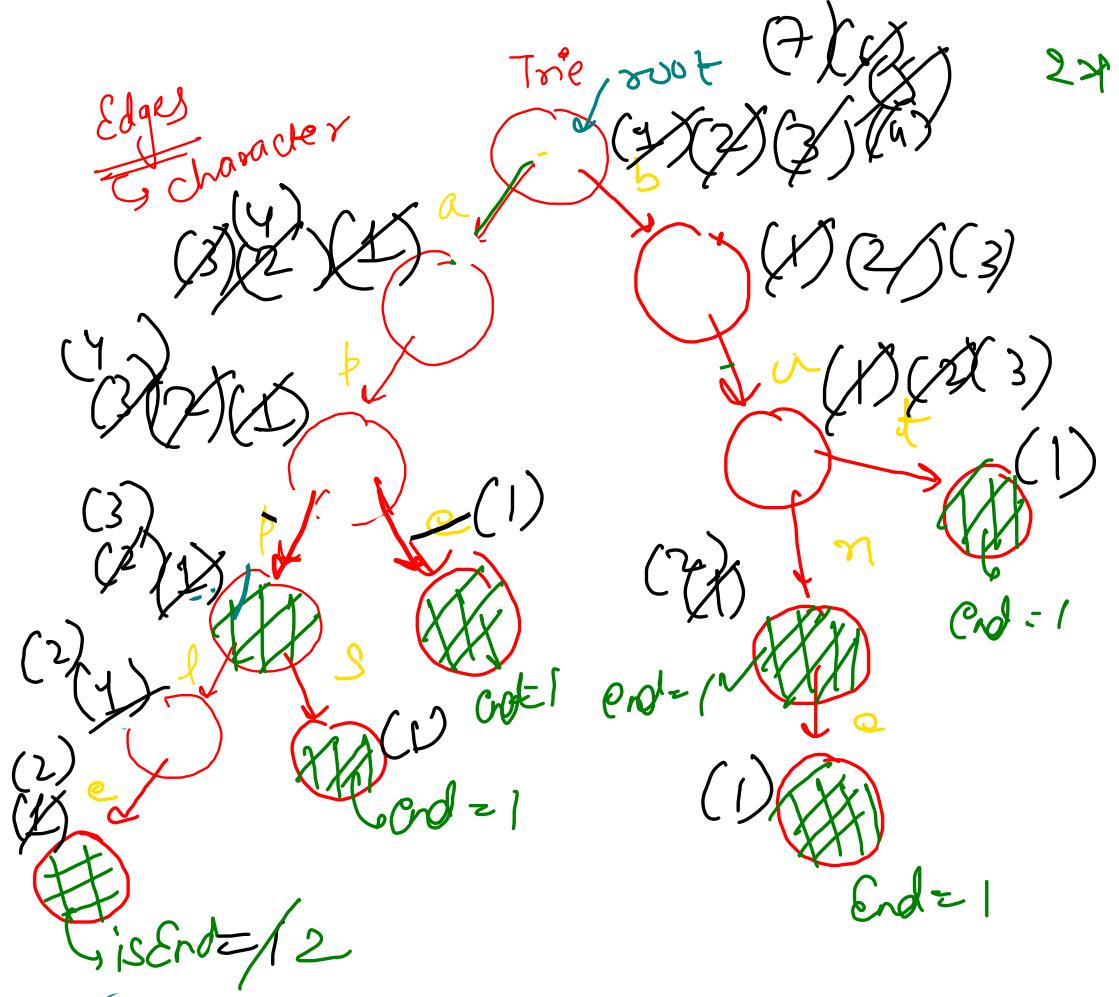
QFG Unique Rows in Boolean Matrix

A **trie** (pronounced as "try") or **prefix tree** is a tree data structure used to efficiently store and retrieve keys in a dataset of strings. There are various applications of this data structure, such as autocomplete and spellchecker.

Implement the Trie class:

- `Trie()` Initializes the trie object.
- `void insert(String word)` Inserts the string `word` into the trie.
- `boolean search(String word)` Returns true if the string `word` is in the trie (i.e., was inserted before), and false otherwise.
- `boolean startsWith(String prefix)` Returns true if there is a previously inserted string `word` that has the prefix `prefix`, and false otherwise.





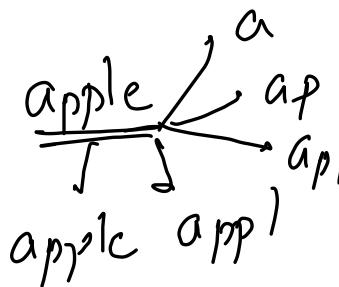
printpref("ap")

insert("apple")
insert("bank")
insert("bad")
insert("ban")
insert("app")
insert("apps")

search("apple") True
search("bank") False
search("ap") False

prefix("apple") True
prefix("bank") False
prefix("ap") True

islast("ape")



delete ("ham")
End 7, 2 :> decrease End

End = 1

The diagram illustrates a search tree structure for a Postorder traversal. The root node is labeled "Postorder". A red arrow points from the label "Postorder" to the root. The root has two children: "Left" and "Right". The "Left" child has a red arrow pointing to it from the handwritten word "distress". The "Right" child has a red arrow pointing to it from the handwritten word "Befind". The "Left" child further has two children: "Postfix" and "End". The "Postfix" child has a red arrow pointing to it from the handwritten word "Postfix". The "End" child has a red arrow pointing to it from the handwritten word "decrease". The "Right" child has two children: "Node" and "Postorder". The "Node" child has a red arrow pointing to it from the handwritten word "Nsde". The "Postorder" child has a red arrow pointing to it from the handwritten word "(End : 0)".

```

public static class Node{
    private Node[] children = new Node[26];
    private boolean isEnd = false;

    public boolean contains(char ch){
        return (children[ch - 'a'] != null);
    }

    public Node get(char ch){
        return children[ch - 'a'];
    }

    public void set(char ch){
        children[ch - 'a'] = new Node();
    }

    public boolean getEnd(){
        return isEnd;
    }

    public void setEnd(){
        isEnd = true;
    }
}

```

```

public void insert(String word) {
    Node curr = root;
    for(int i=0; i<word.length(); i++){
        char ch = word.charAt(i);

        if(curr.contains(ch) == false)
            curr.set(ch);

        curr = curr.get(ch);
    }

    curr.setEnd();
}

public boolean search(String word) {
    Node curr = root;
    for(int i=0; i<word.length(); i++){
        char ch = word.charAt(i);

        if(curr.contains(ch) == false)
            return false;

        curr = curr.get(ch);
    }

    return curr.getEnd();
}

public boolean startswith(String word) {
    Node curr = root;
    for(int i=0; i<word.length(); i++){
        char ch = word.charAt(i);

        if(curr.contains(ch) == false)
            return false;

        curr = curr.get(ch);
    }

    return true;
}

```

Worst case

$T_{\text{ave}} \rightarrow$
 Insert
 Delete
 Find
 Begin
 For 1 String

$\Rightarrow O(L)$

Space for entire Trie
 $\Rightarrow O(NKL)$

~~Q21~~ ~~Leetcode~~

```
public boolean search(String word, int idx, Node curr){  
    if(idx == word.length())  
        return curr.getEnd();  
  
    char ch = word.charAt(idx);  
  
    if(ch != '.'){  
        if(curr.contains(ch) == false) return false;  
        return search(word, idx + 1, curr.get(ch));  
    }  
  
    for(char chn = 'a'; chn <= 'z'; chn++){  
        if(curr.contains(chn) == false) continue;  
  
        if(search(word, idx + 1, curr.get(chn)))  
            return true;  
    }  
  
    return false;  
}  
  
public boolean search(String word) {  
    return search(word, 0, root);  
}
```

worst case

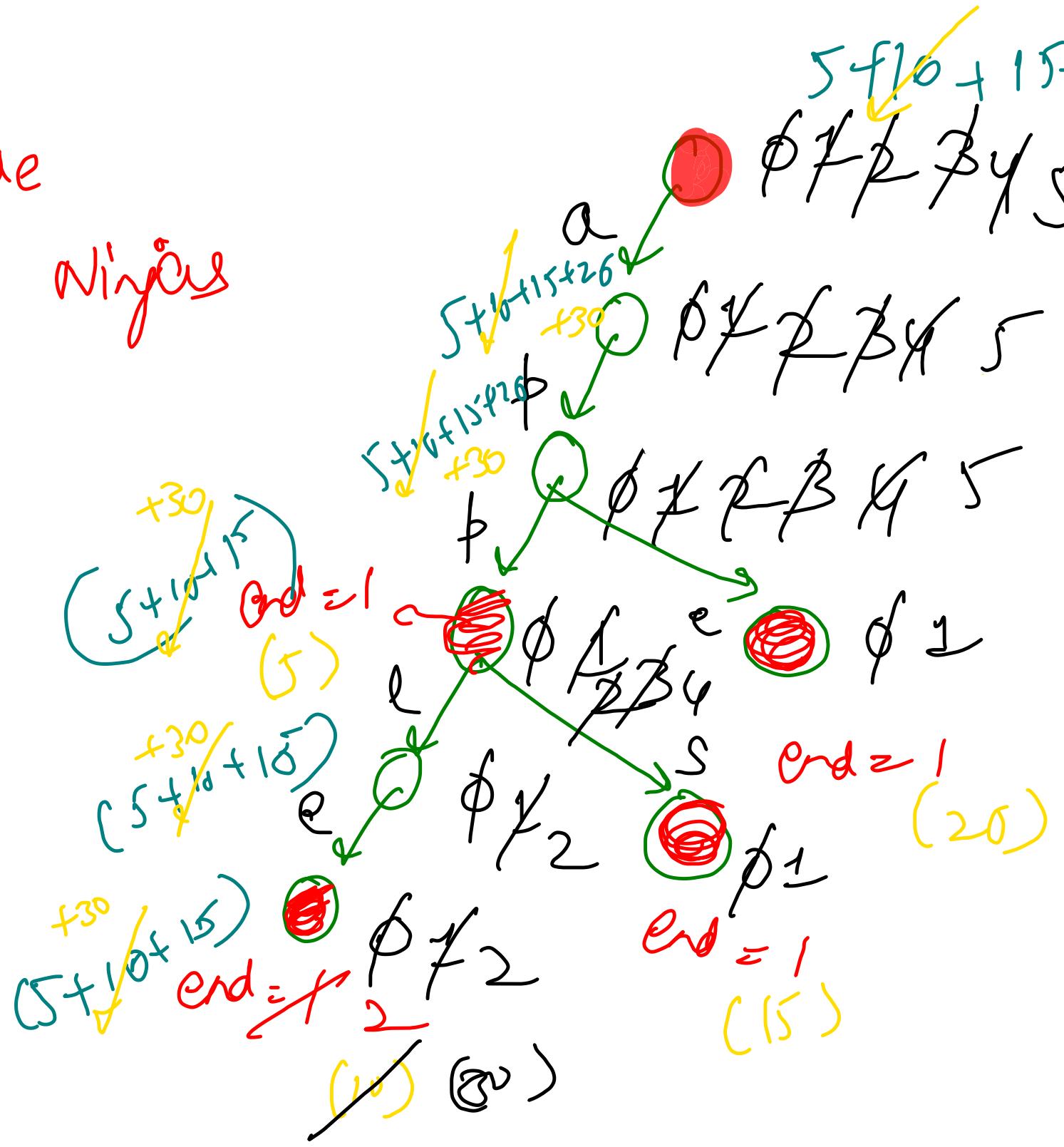
(26)
—

All are dots ('.')

Q1804

LeetCode

→ Coding Nigéria



insert ("apple") → 2
insert ("ape") → 30

insert ("app's")

insert ("app")

freq ("apple") → 2
freq ("apples") → 0
freq ("ap") → 1
post ("apple") → 2
post ("app") → 4

```
public static class Node{
    private Node[] children = new Node[26];
    private int isEnd = 0;
    private int prefixCount = 0;

    public boolean contains(char ch){
        return (children[ch - 'a'] != null);
    }

    public Node get(char ch){
        return children[ch - 'a'];
    }

    public void set(char ch){
        children[ch - 'a'] = new Node();
    }

    public int getFreq(){
        return isEnd;
    }

    public int getPref(){
        return prefixCount;
    }

    public void increaseFreq(){
        isEnd++;
    }
}
```

```
public void decreaseFreq(){
    isEnd--;
}

public void increasePref(){
    prefixCount++;
}

public void decreasePref(){
    prefixCount--;
}
```

```
Node root;
public Trie() {
    root = new Node();
}

public void insert(String word) {
    Node curr = root;
    for(int i=0; i<word.length(); i++){
        curr.increasePref();

        char ch = word.charAt(i);

        if(curr.contains(ch) == false)
            curr.set(ch);

        curr = curr.get(ch);
    }
    curr.increasePref();
    curr.increaseFreq();
}
```

↗(L)

```
public int countWordsEqualTo(String word) {
    Node curr = root;

    for(int i=0; i<word.length(); i++){
        char ch = word.charAt(i);

        if(curr.contains(ch) == false)
            return 0;

        curr = curr.get(ch);
    }

    return curr.getFreq();
}
```

↗(L)

```
public int countWordsStartingwith(String word) {
    Node curr = root;

    for(int i=0; i<word.length(); i++){
        char ch = word.charAt(i);

        if(curr.contains(ch) == false)
            return 0;

        curr = curr.get(ch);
    }

    return curr.getPref();
}
```

```
public void erase(String word) {
    if(countWordsEqualTo(word) == 0){
        return;
    }

    Node curr = root;
    for(int i=0; i<word.length(); i++){
        curr.decreasePref();
        char ch = word.charAt(i);
        curr = curr.get(ch);
    }

    curr.decreasePref();
    curr.decreaseFreq();
}
```

↗(L)

Q(?) MapSum Pair

```

public static class Node{
    private Node[] children = new Node[26];
    private int val = 0;
    public int pref = 0;

    public Node get(char ch){
        return children[ch - 'a'];
    }

    public int getVal(){
        return val;
    }

    public void add(char ch){
        children[ch - 'a'] = new Node();
    }

    public void setVal(int val){
        this.val = val;
    }

    public boolean contains(char ch){
        return (children[ch - 'a'] != null);
    }
}

```

```

Node root;
public MapSum() {
    root = new Node();
}

public int search(String word){
    Node curr = root;
    for(int i=0; i<word.length(); i++){
        char ch = word.charAt(i);

        if(curr.contains(ch) == false)
            return 0;

        curr = curr.get(ch);
    }
    return curr.getVal();
}

```

```

public void insert(String word, int val) {
    int oldVal = search(word); O(L)

    Node curr = root;
    for(int i=0; i<word.length(); i++){
        curr.pref += (val - oldVal);

        char ch = word.charAt(i);

        if(curr.contains(ch) == false)
            curr.add(ch);

        curr = curr.get(ch);
    }
    curr.pref += (val - oldVal);
    curr.setVal(val);
}

```

SL

Weighted Prefix Search

```

public int sum(String word) {
    Node curr = root;
    for(int i=0; i<word.length(); i++){
        char ch = word.charAt(i);

        if(curr.contains(ch) == false)
            return 0;

        curr = curr.get(ch);
    }
    return curr.pref;
}

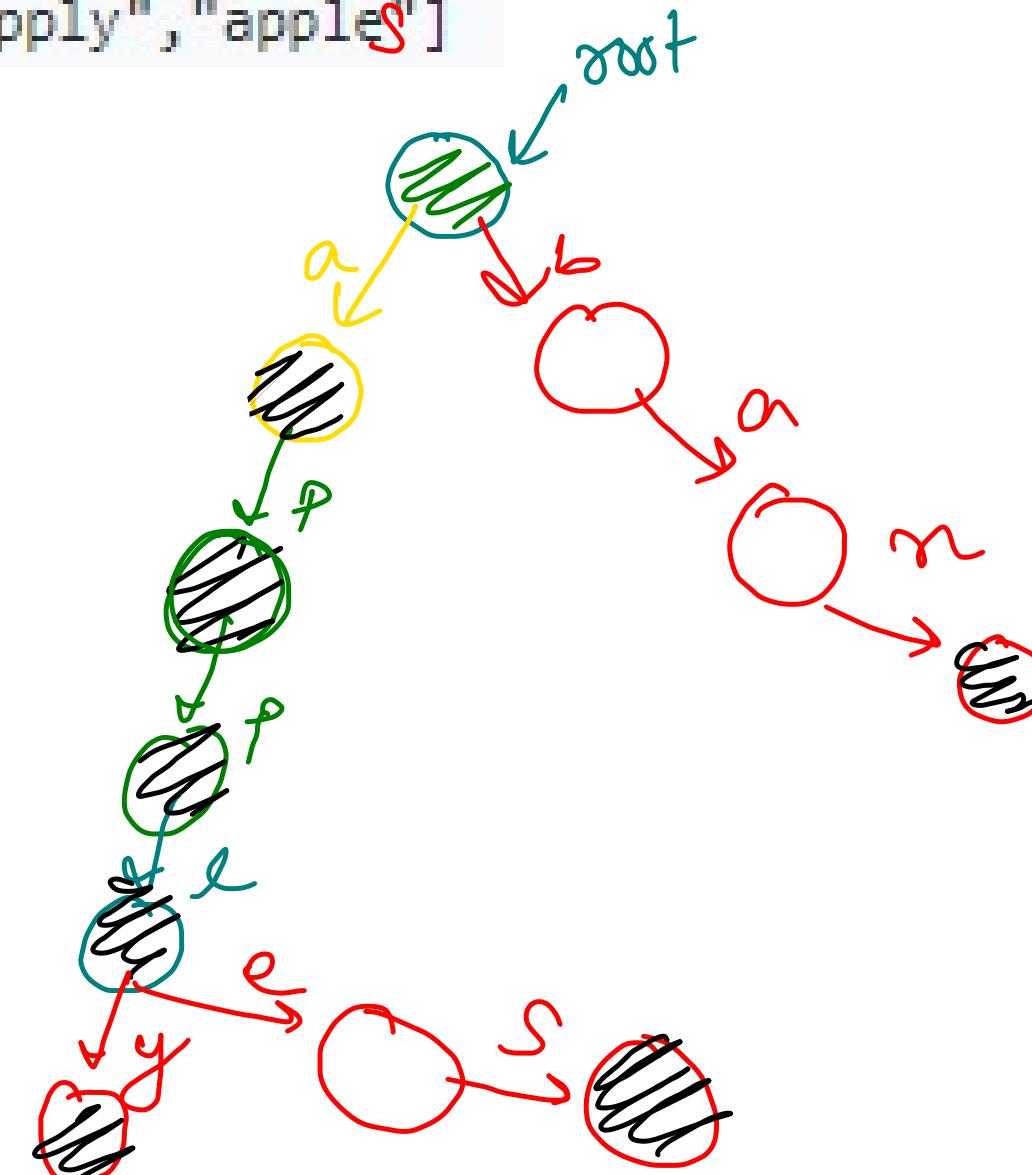
```

Given an array of strings `words` representing an English Dictionary, return the longest word in `words` that can be built one character at a time by other words in `words`.

If there is more than one possible answer, return the longest word with the smallest lexicographical order. If there is no answer, return the empty string.

`["a", "banana", "app", "appl", "ap", "apply", "apple"]`

DFS on
shaded nodes only
(end of word = true)
↓
Deepest node



longest word = " " ⑥

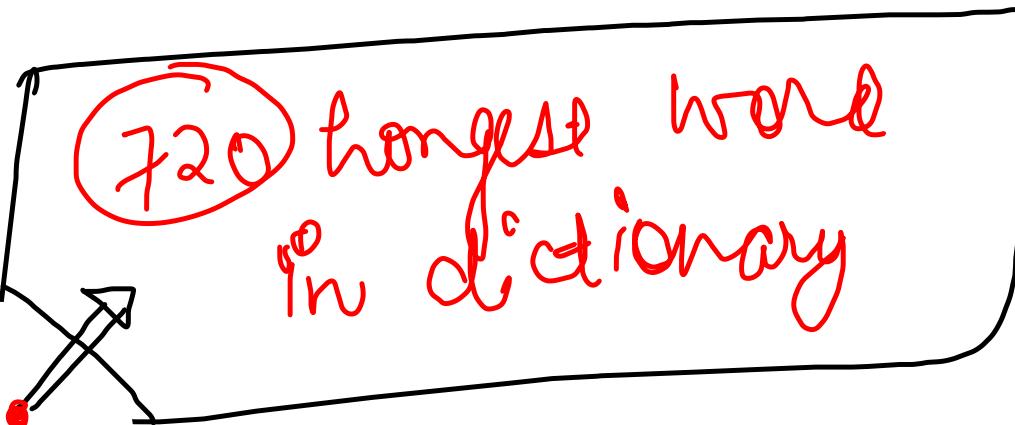
= "a" ①

= "ap" ②

= "app" ③

= "appl" ④

= "apply" ⑤



720 longest word in Dictionary

```
String res = "";
public void DFS(Node root, String anssofar){
    if(root.isTerminal() == false) return;

    if(anssofar.length() > res.length())
        res = anssofar;

    for(char ch = 'a'; ch <= 'z'; ch++){
        if(root.contains(ch) == true){
            DFS(root.get(ch), anssofar + ch);
        }
    }
}

public String longestWord(String[] words) {
    Node root = new Node();
    for(String word: words)
        insert(word, root);

    root.setTerminal();
    DFS(root, "");
    return res;
}
```

```
public static class Node{
    private Node[] children = new Node[26];
    private boolean isTerminal = false;

    public Node get(char ch){
        return children[ch - 'a'];
    }

    public boolean isTerminal(){
        return isTerminal;
    }

    public void setTerminal(){
        isTerminal = true;
    }

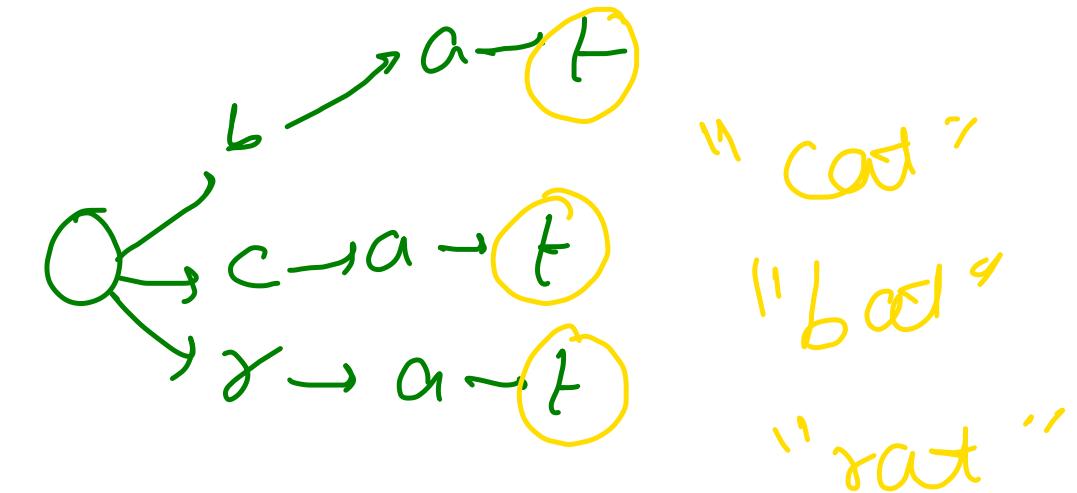
    public void add(char ch){
        children[ch - 'a'] = new Node();
    }

    public boolean contains(char ch){
        return (children[ch - 'a'] != null);
    }
}
```



Replace words with shortest prefix

Input: dictionary = ["cat", "bat", "rat"], sentence = "the cattle was rattled by the battery"
 Output: "the cat was rat by the bat"



"the cattle was rattled by the battery"
 ↓ ↓ ↓ ↓ ↓
 "the — cat — was — rat — by — the — bat "

Every two consecutive words in sentence will be separated by exactly one space.
 sentence does not have leading or trailing spaces.

Assumptions

```
public String replaceWords(List<String> dictionary, String str) {
    Node root = new Node();
    for(String word: dictionary)
        insert(root, word);

    StringBuilder res = new StringBuilder("");
    for(String word: str.split(" ")){
        if(res.length() > 0) res.append(" ");
        res.append(search(root, word));
    }

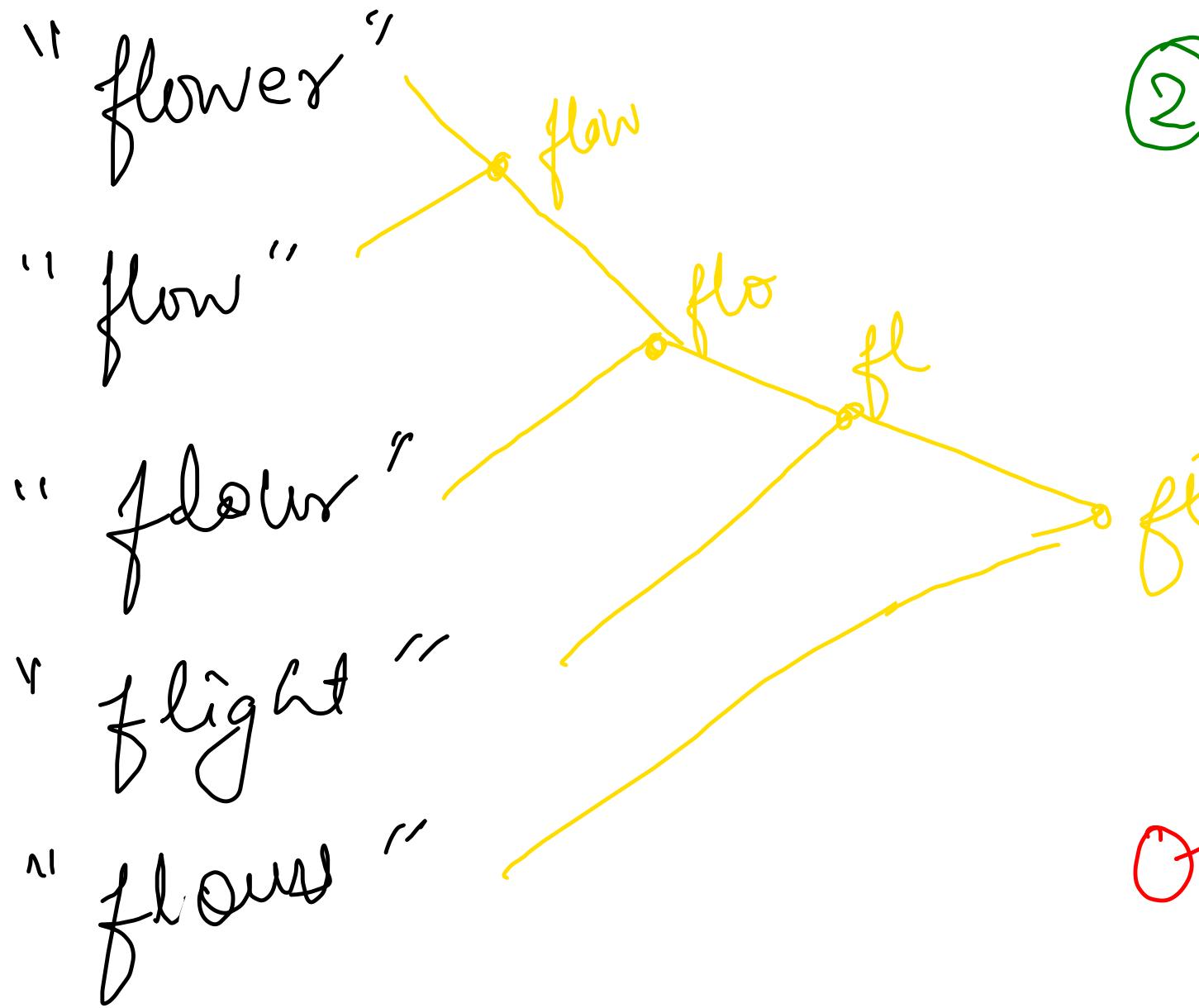
    return res.toString();
}
```

```
public String search(Node curr, String word){
    for(int i=0; i<word.length(); i++){
        char ch = word.charAt(i);

        if(curr.isTerminal() == true){
            return word.substring(0, i);
        }

        if(curr.contains(ch) == false)
            return word;
        curr = curr.get(ch);
    }
    return word;
}
```

Highest Common Prefix = {lowest common ancestor of shaded nodes}

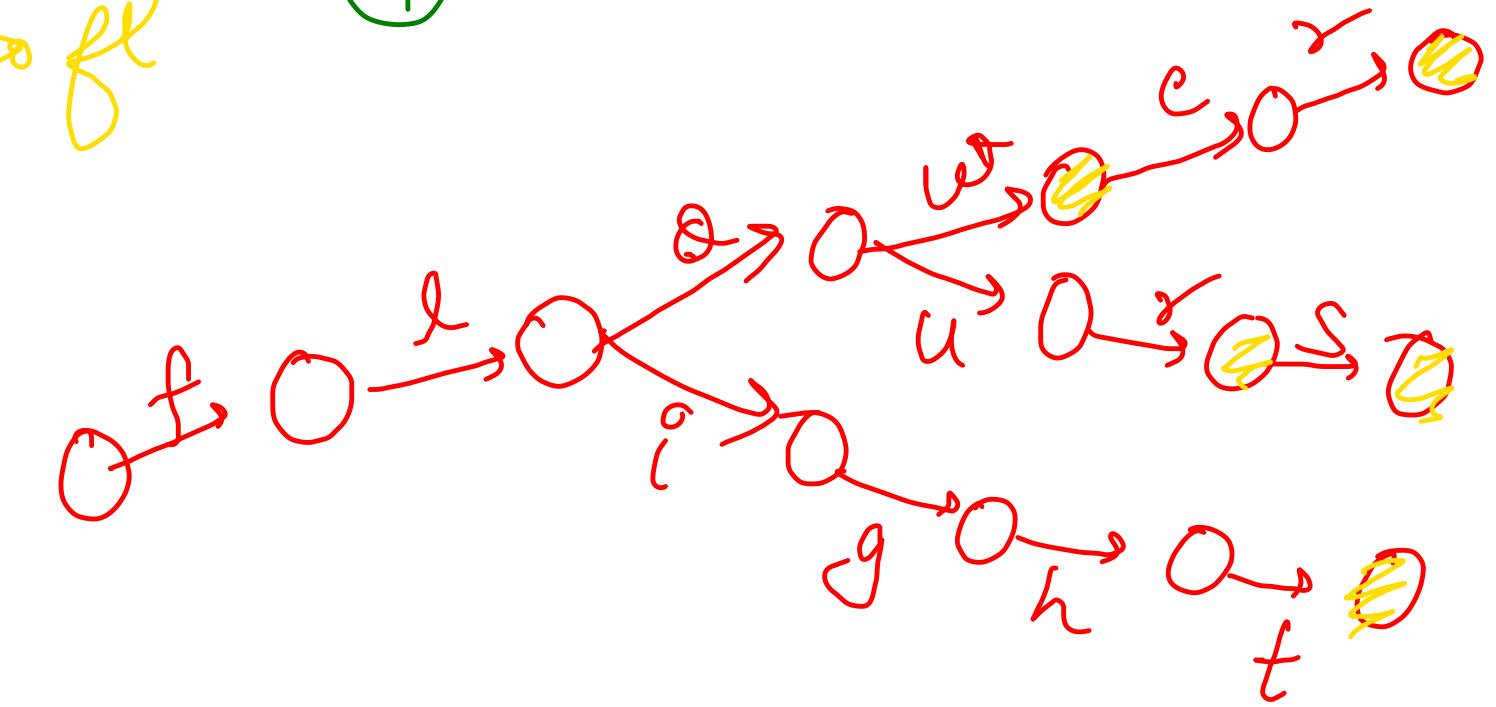


① Vertical scanning

② Horizontal scanning

③ Divide & Conquer

④ Trie \Rightarrow Queries



```

String res = "";
public void DFS(Node curr, String ans){
    int count = 0;

    if(ans.length() > res.length()){
        res = ans;
    }

    if(curr.isTerminal == true){
        return;
    }

    char child = 'a';
    for(char ch = 'a'; ch <= 'z'; ch++){
        if(curr.contains(ch) == true){
            count++;
            child = ch;
        }
    }

    if(count == 1) DFS(curr.get(child), ans + child);
}

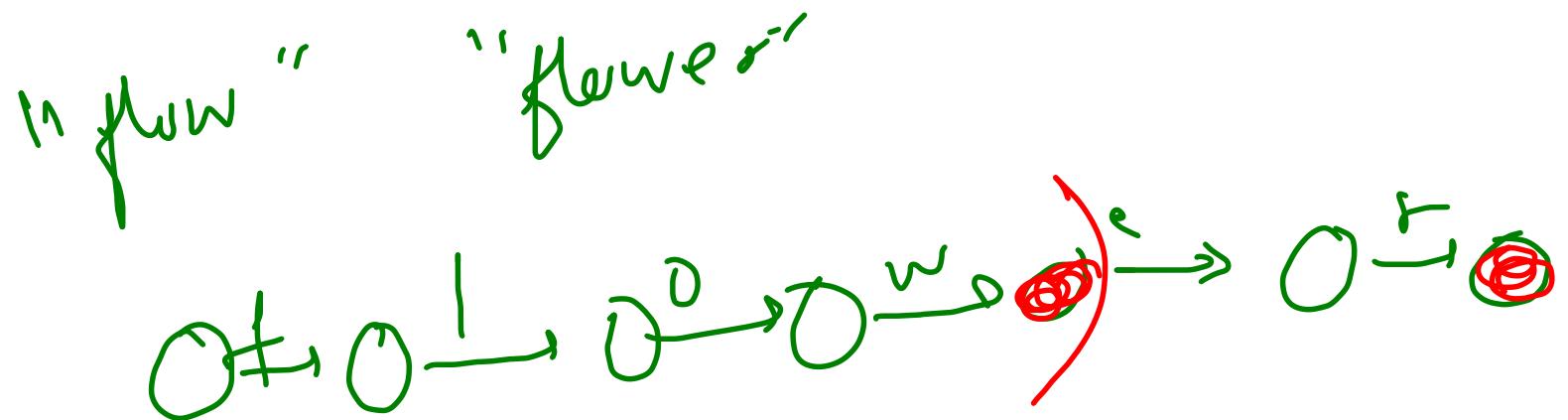
```

```

public String longestCommonPrefix(String[] strs) {
    Node root = new Node();
    for(String str: strs){
        insert(root, str);
    }

    DFS(root, "");
    return res;
}

```



(B)

Shortest Unique Prefix

[zebra, dog, duck, dove]

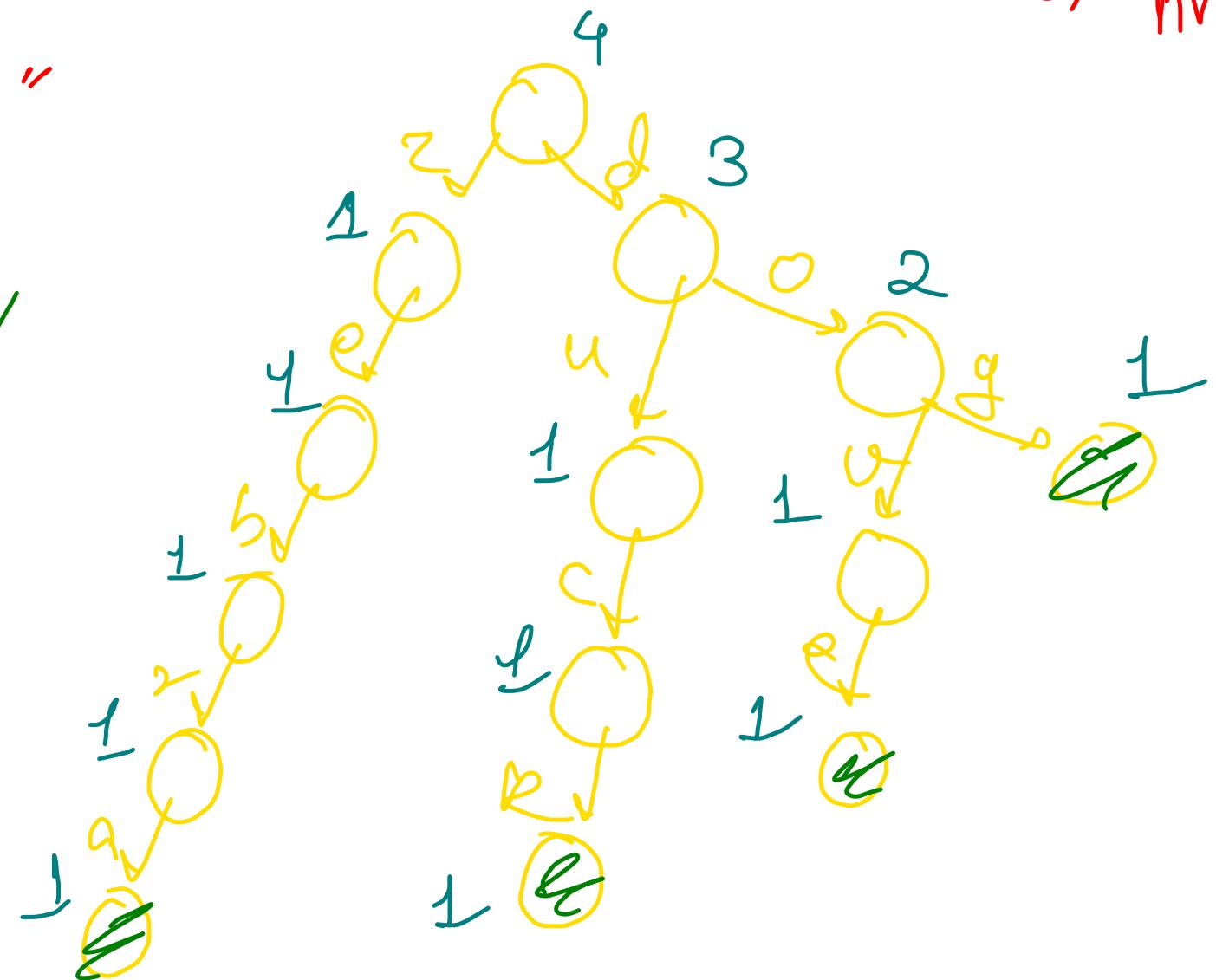
"z"
"d"
"du"
"dov"

Approach
Find first character
which have prefix
Count = 1

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

"dog" vs "dogs"

Answer will
always
exist



```

public static class Node{
    Node[] children = new Node[26];
    int prefCount = 0;
    int end = 0;

    public Node get(char ch){
        return children[ch - 'a'];
    }

    public void set(char ch){
        children[ch - 'a'] = new Node();
    }
}

```

```

public String search(Node root, String word){
    for(int i=0; i<word.length(); i++){
        char ch = word.charAt(i);

        if(root.prefCount == 1)
            return word.substring(0, i);

        root = root.get(ch);
    }

    return word;
}

```

```

public void insert(Node root, String word){
    for(int i=0; i<word.length(); i++){
        char ch = word.charAt(i);

        if(root.get(ch) == null){
            root.set(ch);
        }

        root.prefCount++;
        root = root.get(ch);
    }

    root.prefCount++;
    root.end++;
}

```

```

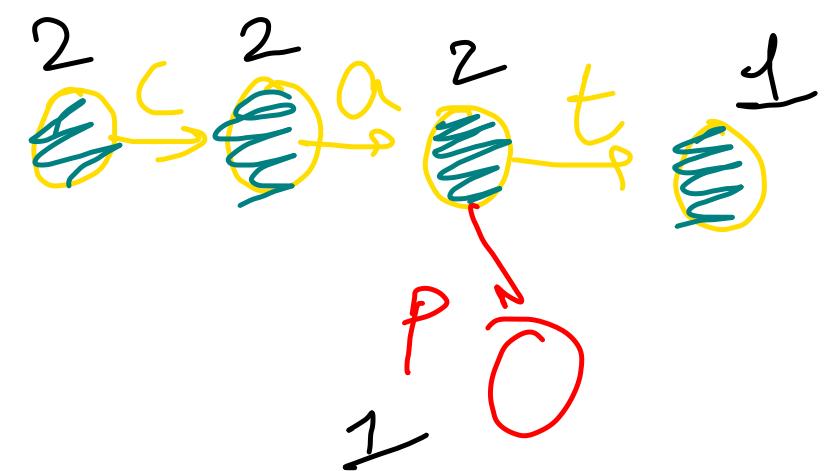
public String[] prefix(String[] A) {
    Node root = new Node();
    for(String str: A){
        insert(root, str);
    }

    String[] res = new String[A.length];
    for(int i=0; i<res.length; i++){
        res[i] = search(root, A[i]);
    }

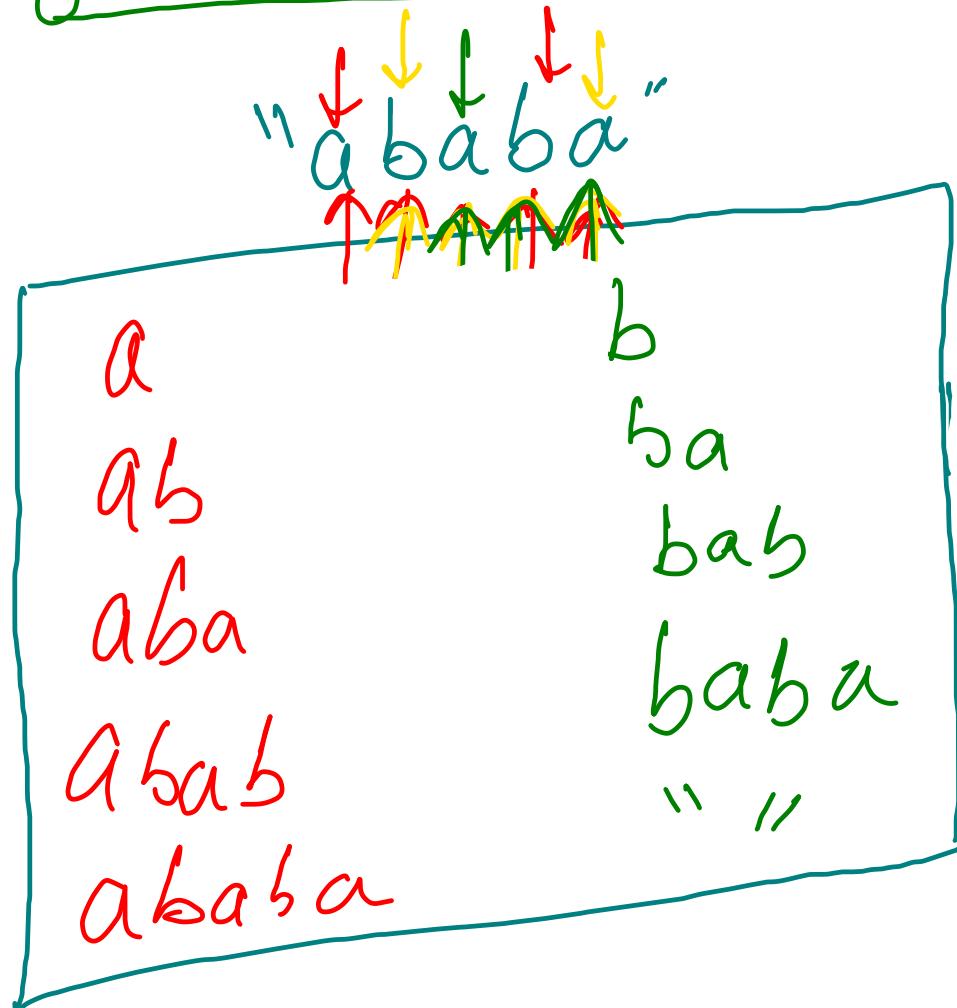
    return res;
}

```

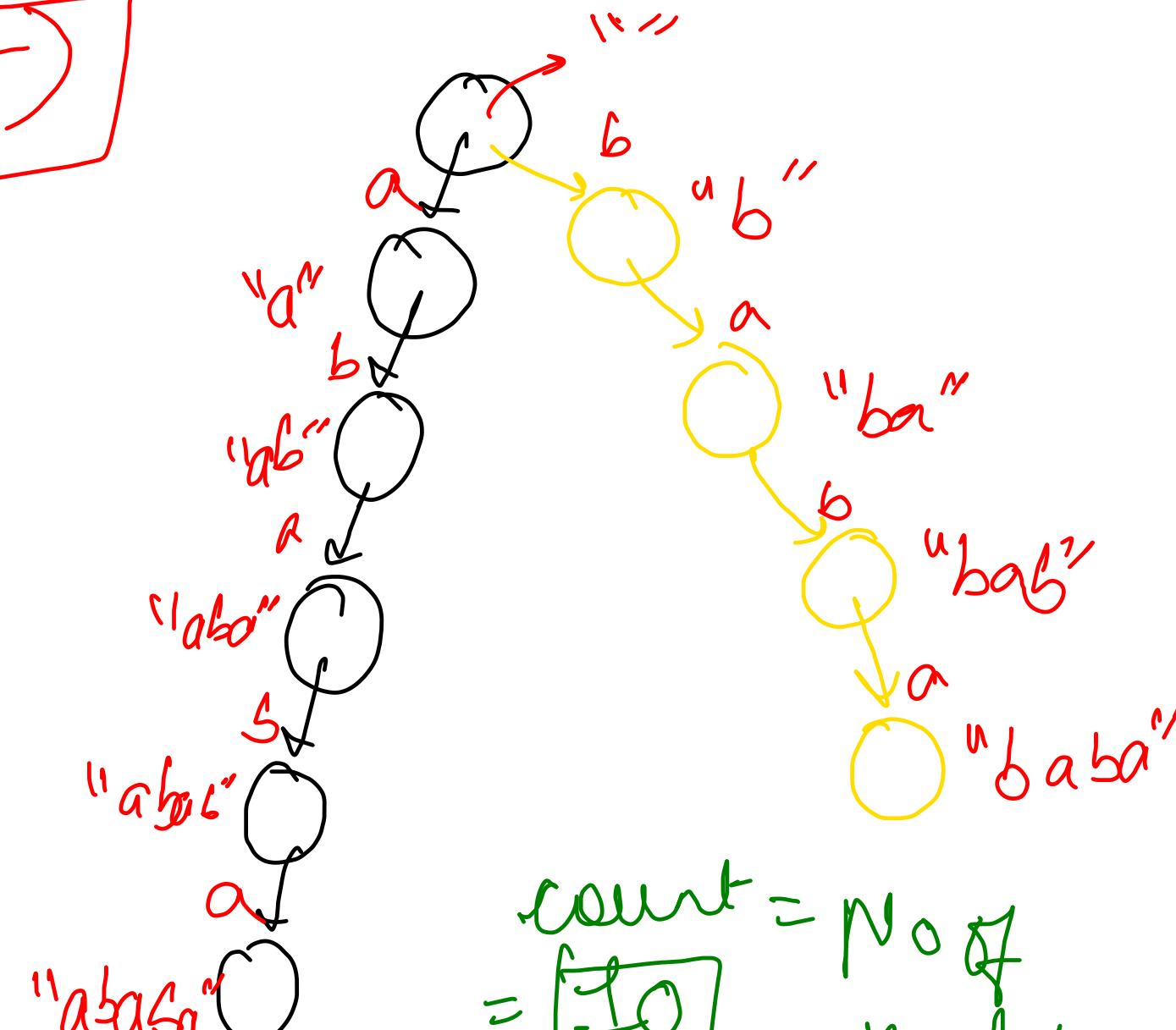
*Shortest Unique
Prefin*



Count Distinct substrings



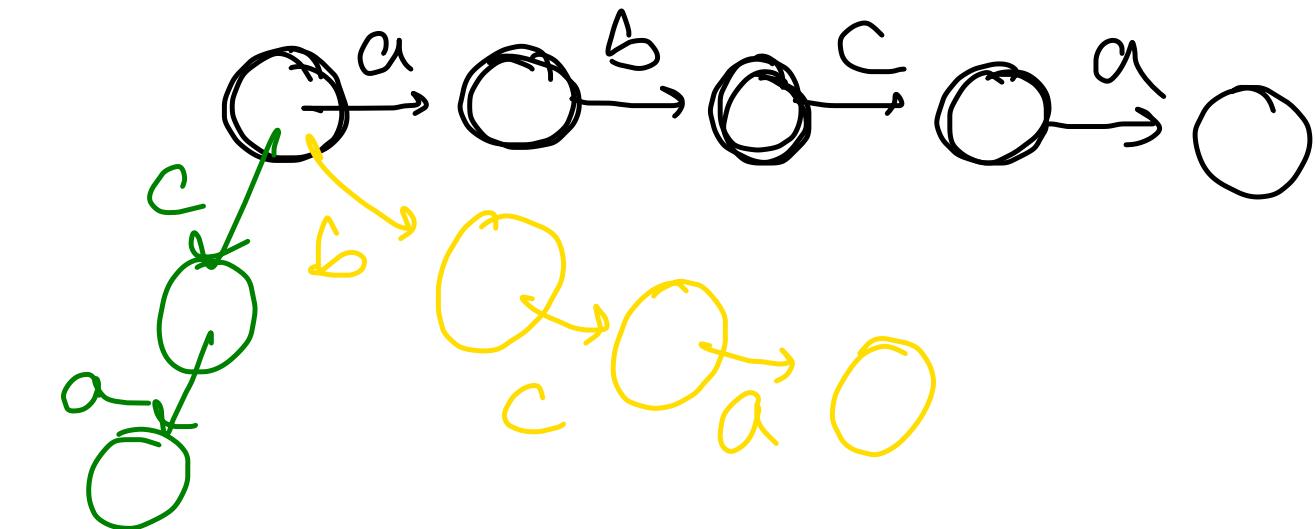
$O(N^2)$



count = No of
nodes
in
tree

$$= \boxed{10}$$

"abca"
 count = 7
~~7~~
~~8~~
~~10~~



$O(N^2)$

```

public static class Node{
    Node[] children = new Node[26];

    public Node get(char ch){
        return children[ch - 'a'];
    }

    public void set(char ch){
        children[ch - 'a'] = new Node();
    }
}
    
```

```

public static int countDistinctSubstring(String s)
{
    Node root = new Node();
    int count = 1;

    for(int i=0; i<s.length(); i++){
        Node curr = root;
        for(int j=i; j<s.length(); j++){
            if(curr.get(s.charAt(j)) == null){
                curr.set(s.charAt(j));
                count++;
            }
            curr = curr.get(s.charAt(j));
        }
    }
    return count;
}
    
```

Word search - II

① Word exists \Rightarrow getEnd()

② Word Prefix \Rightarrow root != null

```

public void exist(int r, int c, char[][] board, String ssof, Node root){
    if(root == null) return; // if word is prefix of at least one word
    if(root.getEnd() == true){
        // if word exists in dictionary or not
        res.add(ssof);
    }

    if(r < 0 || c < 0 || r >= board.length || c >= board[0].length
    || board[r][c] == '0')
        return;

    for(int call=0; call<4; call++){
        char ch = board[r][c];
        board[r][c] = '0';
        exist(r + x[call], c + y[call], board, ssof + ch, root.get(ch));
        board[r][c] = ch;
    }
}

```

```

public static class Node{ }

public void insert(Node curr, String word) {
    for(int i=0; i<word.length(); i++){
        char ch = word.charAt(i);

        if(curr.contains(ch) == false)
            curr.set(ch);

        curr = curr.get(ch);
    }

    curr.setEnd();
}

int[] x = {-1, 1, 0, 0};
int[] y = {0, 0, -1, 1};
HashSet<String> res;

```

```

public List<String> findWords(char[][] board, String[] words) {
    res = new HashSet<>();

    Node root = new Node();
    for(String word: words){
        // insert in trie
        insert(root, word);
    }

    for(int i=0; i<board.length; i++){
        for(int j=0; j<board[0].length; j++){
            exist(i, j, board, "", root);
        }
    }

    List<String> ans = new ArrayList<>();
    for(String str: res){
        ans.add(str);
    }
    return ans;
}

```

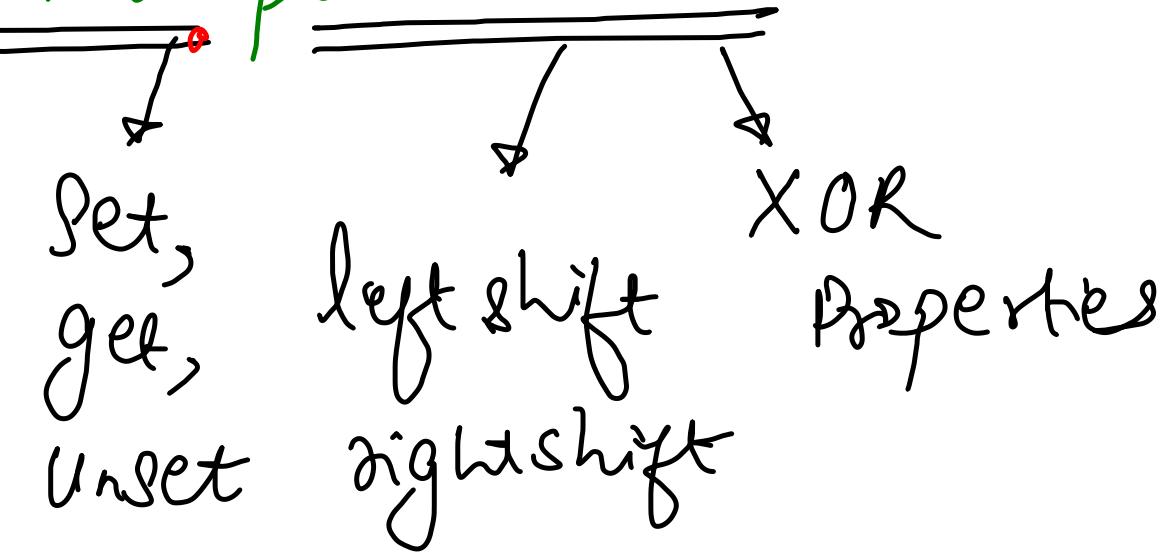
Magic Dictionary

Search suggestion System (SD)

Stream of Characters

Prefix & Suffix Search → Manacher's Algo

Bit Manipulation Basics



Magic Dictionary

Design a data structure that is initialized with a list of **different** words. Provided a string, you should determine if you can change **exactly one character** in this string to match any word in the data structure.

(*)

Implement the MagicDictionary class:

- `MagicDictionary()` Initializes the object.
- `void buildDict(String[] dictionary)` Sets the data structure with an array of distinct strings `dictionary`.
- `bool search(String searchWord)` Returns `true` if you can change **exactly one character** in `searchWord` to match any string in the data structure, otherwise returns `false`.

Input

```
["MagicDictionary", "buildDict", "search", "search", "search", "search"]
[], [[["hello", "leetcode"]], ["hello"], ["hhllo"], ["hell"], ["leetcoded"]]
Output
[null, null, false, true, false, false]
```



hello → FALSE

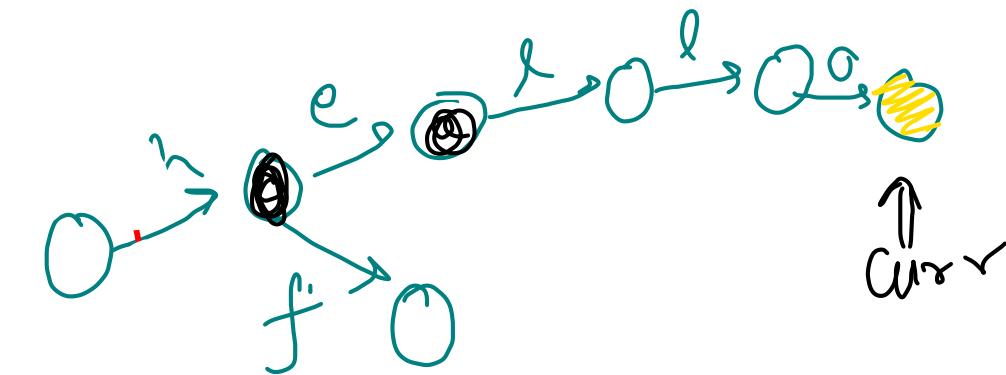
hh~~l~~lo → hello TRUE

hell → FALSE

leetcoded → FALSE

Reencod~~e~~z → FALSE

Search parameters { String word, int id, Node curr, boolean change }



hello
01234
0, false

hello
01234
1, false

hello
01234
2, true

hello
01234
3, true

hello
01234
4, true

for all char at \geq
if char \rightarrow replaces { curr \rightarrow contains(ch)
if change == true
return false
else { id++ , curr.get(ch) , true } }

base case $idx = str.length$
change \leftarrow curr \leftarrow curr + ch

"Hello"
0,f \rightarrow 1,f \rightarrow 2,f \rightarrow 3,f \rightarrow 4,f \rightarrow 5, f \Rightarrow return false

"~~he~~~~ll~~o"
0,f \rightarrow 1,f \rightarrow 2,f \rightarrow 3,t \rightarrow
return false

hell - false
end of word
== false

```
public boolean search(String word, int idx, Node curr, boolean change){
    if(idx == word.length()){
        if(change == true && curr.isTerminal() == true) return true;
        return false;
    }

    char ch = word.charAt(idx);
    if(curr.contains(ch) == true && search(word, idx + 1, curr.get(ch), change))
        return true;

    if(change == true) return false;

    for(char chn = 'a'; chn <= 'z'; chn++){
        if(chn == ch) continue;

        if(curr.contains(chn) && search(word, idx + 1, curr.get(chn), true)){
            return true;
        }
    }
    return false;
}

public boolean search(String searchWord) {
    return search(searchWord, 0, root, false);
}
```

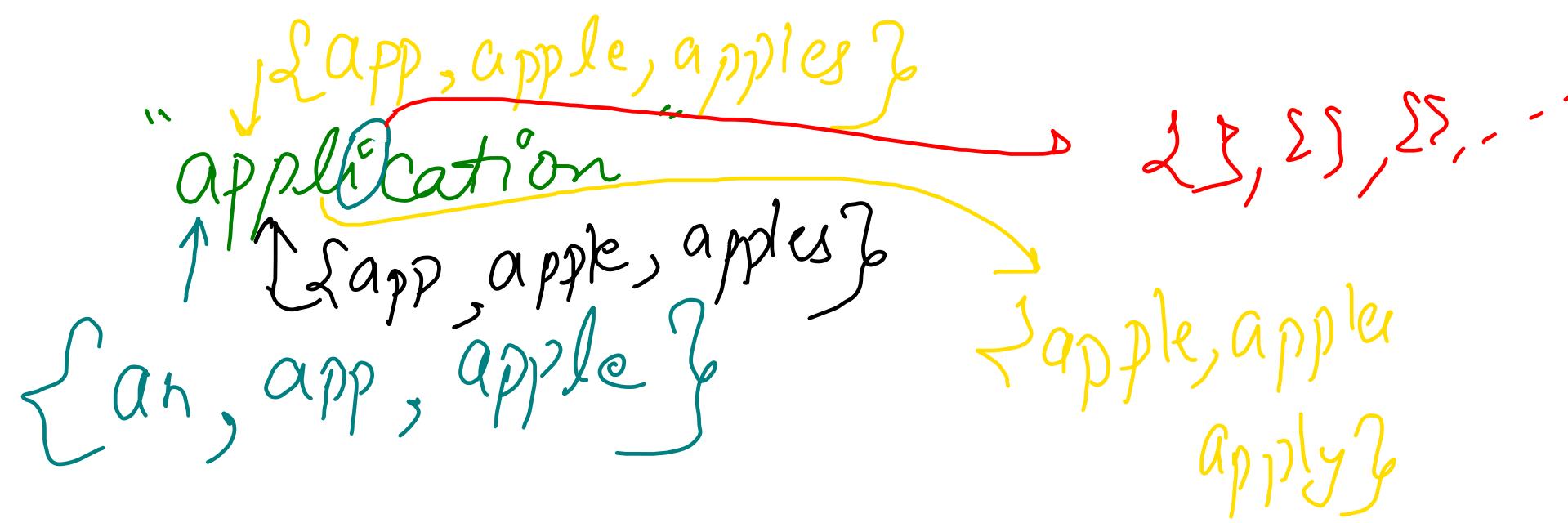
Search suggestion system { System Design }

You are given an array of strings `products` and a string `searchWord`.

Design a system that suggests at most three product names from products after each character of searchWord is typed.

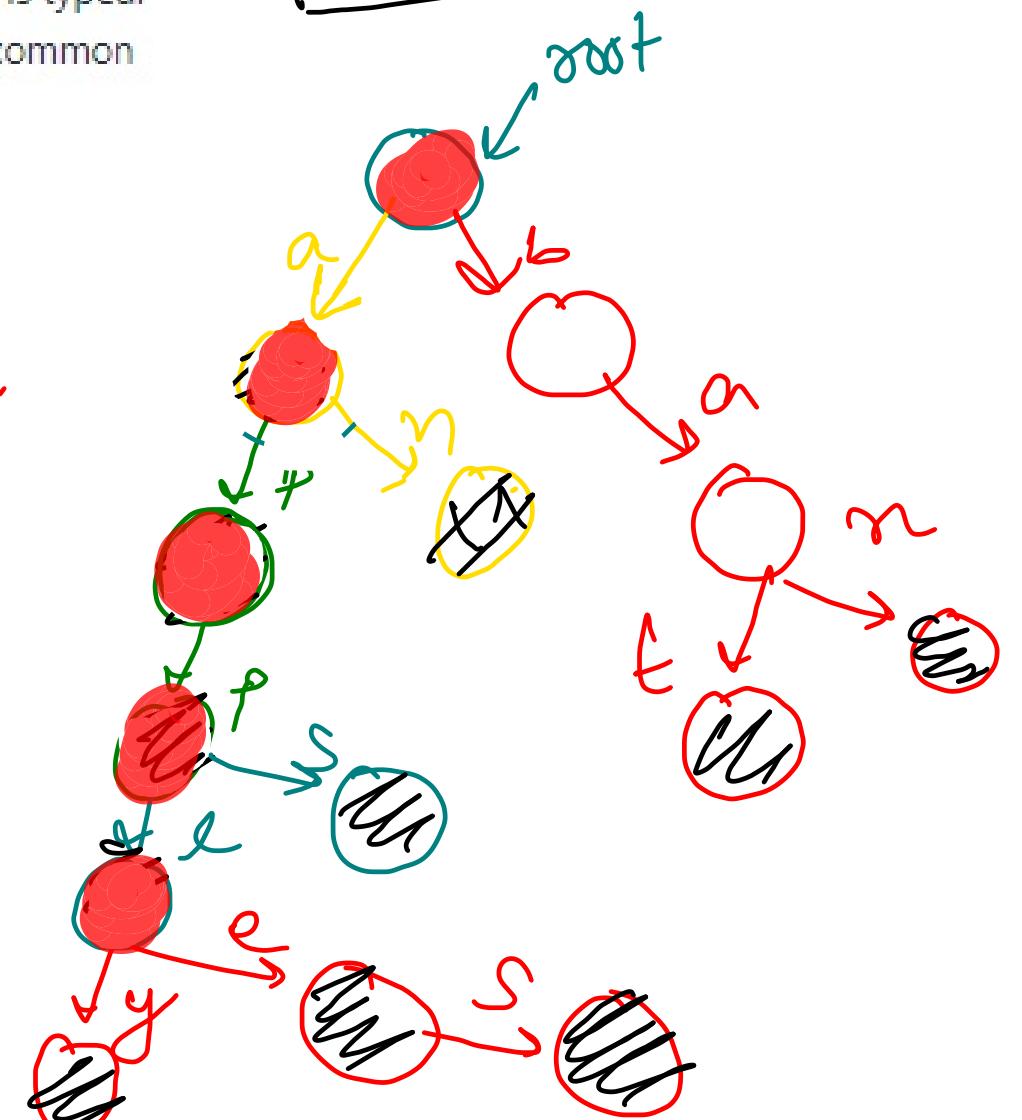
Suggested products should have common prefix with `searchWord`. If there are more than three products with a common prefix return the three lexicographically minimums products.

Return a list of lists of the suggested products after each character of `searchWord` is typed.



DFS → Preorder → At most 3 words

↑
semigraphical



app, apple, apples,
apps, apply, ban, bat,
an,

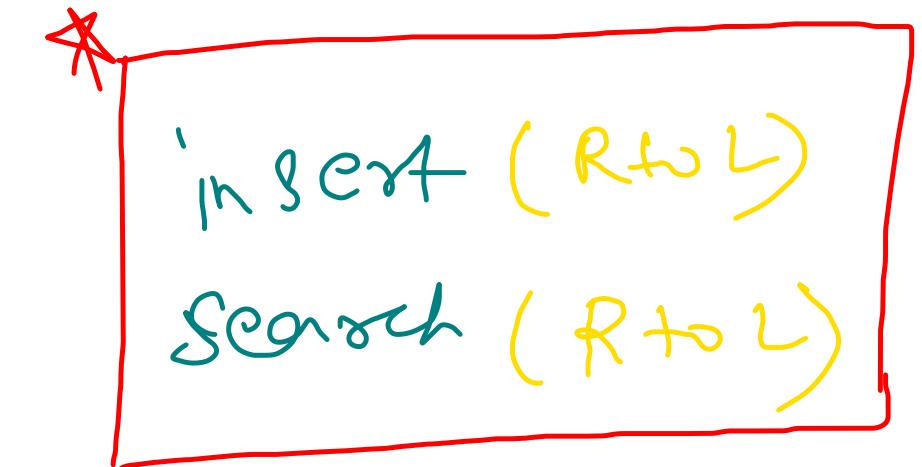
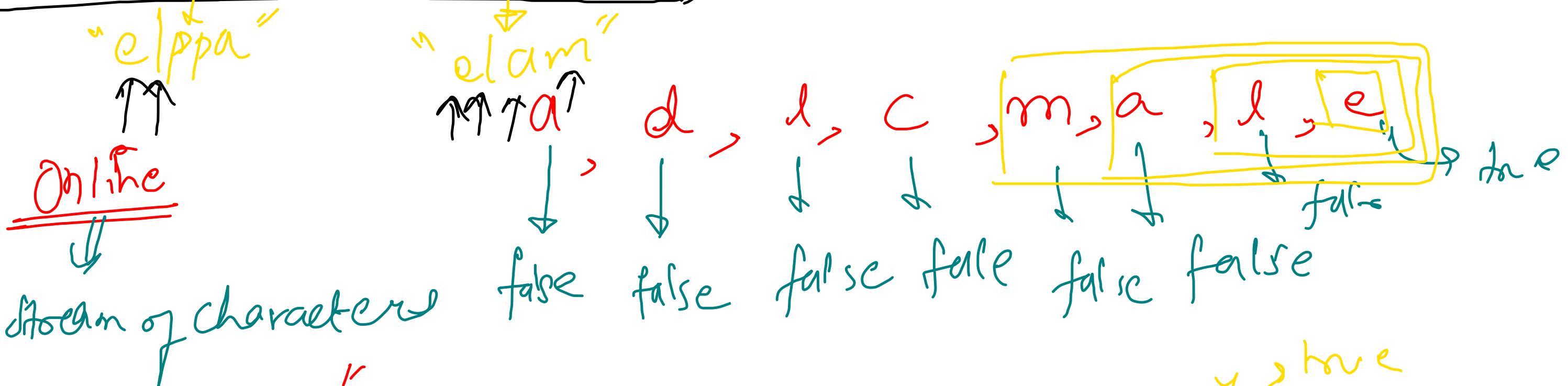
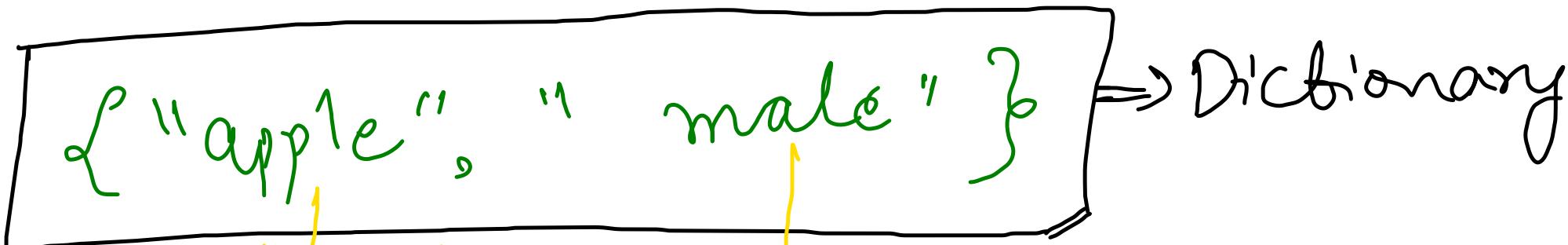
```
public void DFS(Node root, String ssf, List<String> ans, int k){  
    if(ans.size() == k) return;  
  
    if(root.getEnd() == true) {  
        ans.add(ssf);  
    }  
  
    for(char ch='a'; ch<='z'; ch++){  
        if(root.contains(ch) == true){  
            DFS(root.get(ch), ssf + ch, ans, k);  
        }  
    }  
}
```

```
public List<List<String>> suggestedProducts(String[] products, String searchWord) {  
    Node root = new Node();  
    for(String word: products)  
        insert(root, word);  
  
    List<List<String>> res = new ArrayList<>();  
  
    for(int i=0; i<searchWord.length(); i++){  
        char ch = searchWord.charAt(i);  
  
        if(root.contains(ch) == true){  
            root = root.get(ch);  
            List<String> ans = new ArrayList<>();  
            DFS(root, searchWord.substring(0, i + 1), ans, 3);  
            res.add(ans);  
        } else break;  
    }  
  
    while(res.size() < searchWord.length())  
        res.add(new ArrayList<>());  
    return res;  
}
```

3 Almost lexicographically minimum Products

Q(03)

Stream of characters



Given a list of **unique** words, return all the pairs of the **distinct** indices (i, j) in the given list, so that the concatenation of the two words $\text{words}[i] + \text{words}[j]$ is a palindrome.

Input: $\text{words} = ["abcd", "dcba", "lls", "s", "sssll"]$

Output: $[[0,1], [1,0], [3,2], [2,4]]$

Explanation: The palindromes are $["dcbaabcd", "abcddcba", "slls", "llssssll"]$

$$\checkmark \\ 0+1 \\ abcd + dcba \\ abcd dcba$$

$$\checkmark \\ 1+0 \\ dcba + abcd \\ dcba abcd$$

$$2 \rightarrow 4 \\ llS + sSSll \\ llSSSSll$$

$$3+2 \\ s + llS \\ llS$$

$x+y \Rightarrow \text{palindrome}$

x is reverse of y
 Suffix of y = reverse of x & remaining
 pref. of x = reverse of y
 & remaining x should be
 a palindrome

