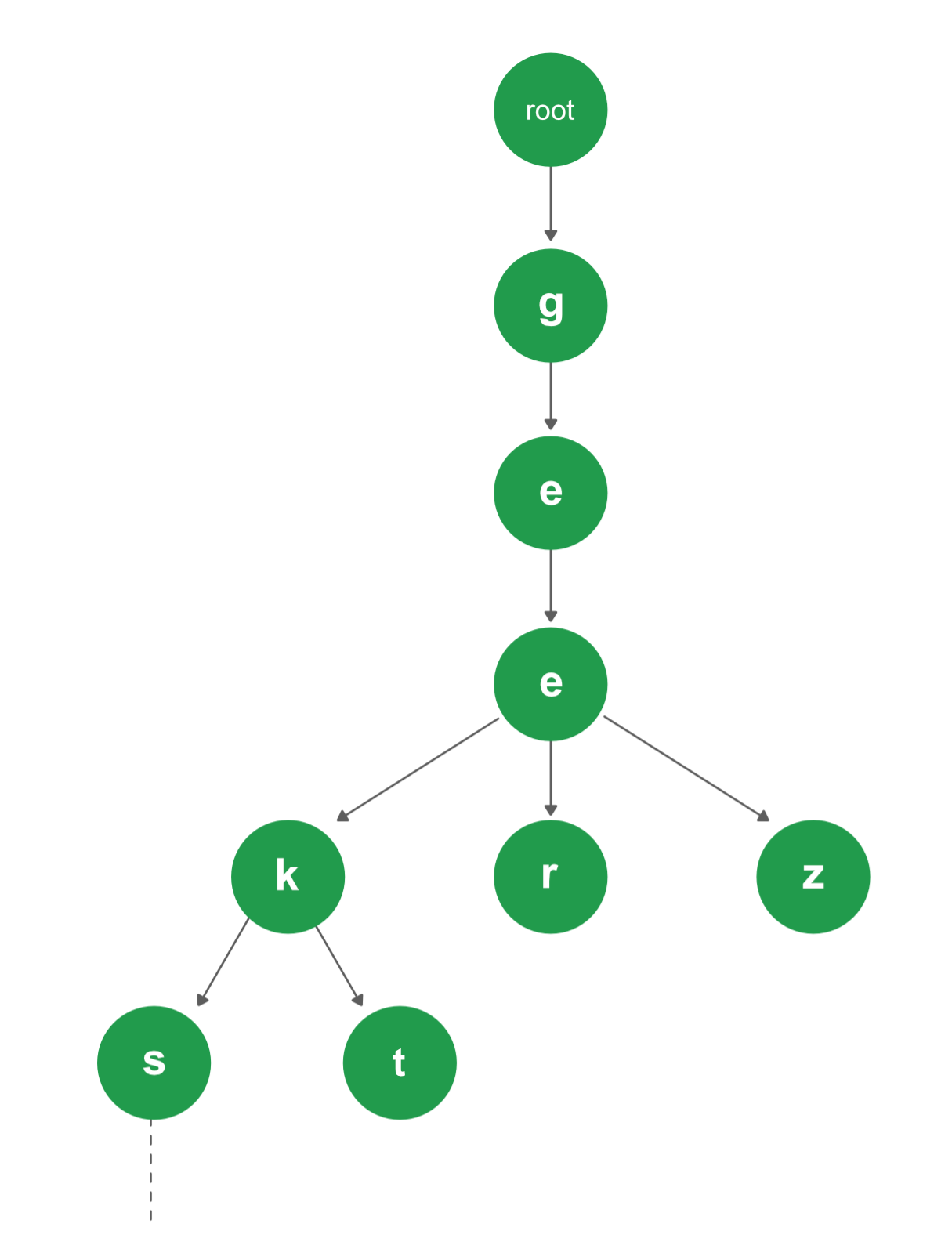
Trie

# 373. Construct a trie from scratch

[Trie](http://en.wikipedia.org/wiki/Trie) is an efficient information retrieval data structure. Using Trie, search complexities can be brought to optimal limit (key length). If we store keys in a binary search tree, a well balanced BST will need time proportional to **M \* log N**, where M is the maximum string length and N is the number of keys in the tree. Using Trie, we can search the key in O(M) time. However, the penalty is on Trie storage requirements (Please refer to [Applications of Trie](https://www.geeksforgeeks.org/advantages-trie-data-structure/) for more details)



Every node of Trie consists of multiple branches. Each branch represents a possible character of keys. We need to mark the last node of every key as the end of the word node. A Trie node field *isEndOfWord* is used to distinguish the node as the end of the word node. A simple structure to represent nodes of the English alphabet can be as follows,   
// Trie node   
struct TrieNode   
{   
     struct TrieNode \*children[ALPHABET\_SIZE];  
     // isEndOfWord is true if the node   
     // represents end of a word   
     bool isEndOfWord;   
};   
Inserting a key into Trie is a simple approach. Every character of the input key is inserted as an individual Trie node. Note that the *children* is an array of pointers (or references) to next level trie nodes. The key character acts as an index to the array *children*. If the input key is new or an extension of the existing key, we need to construct non-existing nodes of the key, and mark the end of the word for the last node. If the input key is a prefix of the existing key in Trie, we simply mark the last node of the key as the end of a word. The key length determines Trie depth. 

Searching for a key is similar to an insert operation, however, we only compare the characters and move down. The search can terminate due to the end of a string or lack of key in the trie. In the former case, if the *isEndofWord* field of the last node is true, then the key exists in the trie. In the second case, the search terminates without examining all the characters of the key, since the key is not present in the trie.  
The following picture explains the construction of trie using keys given in the example below, 

root

/ \ \

t a b

| | |

h n y

| | \ |

e s y e

/ | |

i r w

| | |

r e e

|

r

In the picture, every character is of type *trie\_node\_t*. For example, the *root* is of type trie\_node\_t, and it’s children *a*, *b* and *t* are filled, all other nodes of root will be NULL. Similarly, “a” at the next level is having only one child (“n”), all other children are NULL. The leaf nodes are in blue.

## Solution:

Insert and search costs **O(key\_length)**, however, the memory requirements of Trie is **O(ALPHABET\_SIZE \* key\_length \* N)** where N is the number of keys in Trie. There are efficient representations of trie nodes (e.g. compressed trie, [ternary search tree](http://en.wikipedia.org/wiki/Ternary_search_tree), etc.) to minimize the memory requirements of the trie.

// C++ implementation of search and insert

// operations on Trie

#include <bits/stdc++.h>

using namespace std;

const int ALPHABET\_SIZE = 26;

// trie node

struct TrieNode

{

struct TrieNode \*children[ALPHABET\_SIZE];

// isEndOfWord is true if the node represents

// end of a word

bool isEndOfWord;

};

// Returns new trie node (initialized to NULLs)

struct TrieNode \*getNode(void)

{

struct TrieNode \*pNode = new TrieNode;

pNode->isEndOfWord = false;

for (int i = 0; i < ALPHABET\_SIZE; i++)

pNode->children[i] = NULL;

return pNode;

}

// If not present, inserts key into trie

// If the key is prefix of trie node, just

// marks leaf node

void insert(struct TrieNode \*root, string key)

{

struct TrieNode \*pCrawl = root;

for (int i = 0; i < key.length(); i++)

{

int index = key[i] - 'a';

if (!pCrawl->children[index])

pCrawl->children[index] = getNode();

pCrawl = pCrawl->children[index];

}

// mark last node as leaf

pCrawl->isEndOfWord = true;

}

// Returns true if key presents in trie, else

// false

bool search(struct TrieNode \*root, string key)

{

struct TrieNode \*pCrawl = root;

for (int i = 0; i < key.length(); i++)

{

int index = key[i] - 'a';

if (!pCrawl->children[index])

return false;

pCrawl = pCrawl->children[index];

}

return (pCrawl->isEndOfWord);

}

// Driver

int main()

{

// Input keys (use only 'a' through 'z'

// and lower case)

string keys[] = {"the", "a", "there",

"answer", "any", "by",

"bye", "their" };

int n = sizeof(keys)/sizeof(keys[0]);

struct TrieNode \*root = getNode();

// Construct trie

for (int i = 0; i < n; i++)

insert(root, keys[i]);

// Search for different keys

search(root, "the")? cout << "Yes\n" :

cout << "No\n";

search(root, "these")? cout << "Yes\n" :

cout << "No\n";

search(root, "their")? cout << "Yes\n" :

cout << "No\n";

search(root, "thaw")? cout << "Yes\n" :

cout << "No\n";

return 0;

}

**Output :**

the --- Present in trie

these --- Not present in trie

their --- Present in trie

thaw --- Not present in trie

**Recursive approach:**

//Function to insert string into TRIE.

void insert(struct TrieNode \*root, string key)

{

// code here

if(key.size()==0){

root->isLeaf = true;

}

else if(root->children[key[0]-'a']!=NULL){

insert(root->children[key[0]-'a'], key.substr(1));

}

else{

//struct TrieNode\* nd = (struct TrieNode\*)malloc(sizeof(struct TrieNode));

struct TrieNode\* nd = getNode();

root->children[key[0]-'a'] = nd;

insert(nd, key.substr(1));

}

}

//Function to use TRIE data structure and search the given string.

bool search(struct TrieNode \*root, string key)

{

// code here

if(key.size()==0 && root->isLeaf==true)

return true;

else{

if(root->children[key[0]-'a']==NULL)

return false;

return search(root->children[key[0]-'a'], key.substr(1));

}

}

# 374. Find shortest unique prefix for every word in a given list

Given an array of words, find all shortest unique prefixes to represent each word in the given array. Assume that no word is prefix of another.

**Example 1:**

**Input:**

N = 4

arr[] = {"zebra", "dog", "duck", "dove"}

**Output:** z dog du dov

**Explanation:**

z => zebra

dog => dog

duck => du

dove => dov

**Example 2:**

**Input:**

N = 3

arr[] = {"geeksgeeks", "geeksquiz",

"geeksforgeeks"};

**Output:** geeksg geeksq geeksf

**Explanation:**

geeksgeeks => geeksg

geeksquiz => geeksq

geeksforgeeks => geeksf

**Your task:**

You don't have to read input or print anything. Your task is to complete the function **findPrefixes()** which takes the array of strings and it's size N as input and returns a list of shortest unique prefix for each word

**Expected Time Complexity:** O(N\*length of each word)

**Expected Auxiliary Space:**O(N\*length of each word)

**Constraints:**

1 ≤ N, Length of each word ≤ 1000

## Solution:

A **Simple Solution** is to consider every prefix of every word (starting from the shortest to largest), and if a prefix is not prefix of any other string, then print it.   
An **Efficient Solution** is to use [Trie](https://www.geeksforgeeks.org/trie-insert-and-search/). The idea is to maintain a count in every node. Below are steps.  
1) Construct a [Trie](https://www.geeksforgeeks.org/trie-insert-and-search/) of all words. Also maintain frequency of every node (Here frequency is number of times node is visited during insertion). Time complexity of this step is O(N) where N is total number of characters in all words.   
2) Now, for every word, we find the character nearest to the root with frequency as 1. The prefix of the word is path from root to this character. To do this, we can traverse Trie starting from root. For every node being traversed, we check its frequency. If frequency is one, we print all characters from root to this node and don’t traverse down this node.  
Time complexity if this step also is O(N) where N is total number of characters in all words. 

root

/ \

(d, 3)/ \(z, 1)

/ \

Node1 Node2

/ \ \

(o,2)/ \(u,1) \(e,1)

/ \ \

Node1.1 Node1.2 Node2.1

/ \ \ \

(g,1)/ \ (t,1) \(c,1) \(b,1)

/ \ \ \

Leaf Leaf Node1.2.1 Node2.1.1

(dog) (dot) \ \

\(k, 1) \(r, 1)

\ \

Leaf Node2.1.1.1

(duck) \

\(a,1)

\

Leaf

(zebra)

Below is the implementation of above idea.

// C++ program to print all prefixes that

// uniquely represent words.

#include<bits/stdc++.h>

using namespace std;

#define MAX 256

// Maximum length of an input word

#define MAX\_WORD\_LEN 500

// Trie Node.

struct trieNode

{

struct trieNode \*child[MAX];

int freq; // To store frequency

};

// Function to create a new trie node.

struct trieNode \*newTrieNode(void)

{

struct trieNode \*newNode = new trieNode;

newNode->freq = 1;

for (int i = 0; i<MAX; i++)

newNode->child[i] = NULL;

return newNode;

}

// Method to insert a new string into Trie

void insert(struct trieNode \*root, string str)

{

// Length of the URL

int len = str.length();

struct trieNode \*pCrawl = root;

// Traversing over the length of given str.

for (int level = 0; level<len; level++)

{

// Get index of child node from current character

// in str.

int index = str[level];

// Create a new child if not exist already

if (!pCrawl->child[index])

pCrawl->child[index] = newTrieNode();

else

(pCrawl->child[index]->freq)++;

// Move to the child

pCrawl = pCrawl->child[index];

}

}

// This function prints unique prefix for every word stored

// in Trie. Prefixes one by one are stored in prefix[].

// 'ind' is current index of prefix[]

void findPrefixesUtil(struct trieNode \*root, char prefix[],

int ind)

{

// Corner case

if (root == NULL)

return;

// Base case

if (root->freq == 1)

{

prefix[ind] = '\0';

cout << prefix << " ";

return;

}

for (int i=0; i<MAX; i++)

{

if (root->child[i] != NULL)

{

prefix[ind] = i;

findPrefixesUtil(root->child[i], prefix, ind+1);

}

}

}

// Function to print all prefixes that uniquely

// represent all words in arr[0..n-1]

void findPrefixes(string arr[], int n)

{

// Construct a Trie of all words

struct trieNode \*root = newTrieNode();

root->freq = 0;

for (int i = 0; i<n; i++)

insert(root, arr[i]);

// Create an array to store all prefixes

char prefix[MAX\_WORD\_LEN];

// Print all prefixes using Trie Traversal

findPrefixesUtil(root, prefix, 0);

}

// Driver function.

int main()

{

string arr[] = {"zebra", "dog", "duck", "dove"};

int n = sizeof(arr)/sizeof(arr[0]);

findPrefixes(arr, n);

return 0;

}

**Output:**

dog dov du z

**My approach:**

struct Trie{

struct Trie\* ch[26];

bool isLeaf;

};

struct Trie\* getNode(){

struct Trie\* nd = (struct Trie\*)malloc(sizeof(struct Trie));

if(nd){

nd->isLeaf = false;

for(int i=0;i<26;i++)

nd->ch[i] = NULL;

}

return nd;

}

class Solution

{

public:

vector<string> findPrefixes(string arr[], int n)

{

//code here

struct Trie\* root = getNode();

struct Trie\* curr = root;

for(int i=0;i<n;i++){

curr = root;

for(int j=0;j<arr[i].size();j++){

int ind = arr[i][j] - 'a';

if(curr->ch[ind]==NULL){

struct Trie\* nd = getNode();

curr->ch[ind] = nd;

curr = nd;

}

else

curr = curr->ch[ind];

}

curr->isLeaf = true;

}

vector<string> res;

for(int i=0;i<n;i++){

int len=1;

curr = root;

for(int j=0;j<arr[i].size();j++){

int ind = arr[i][j] - 'a', count=0;

for(int k=0;k<26;k++){

if(curr->ch[k]!=NULL)

count++;

}

if(count>1)

len = j+1;

curr = curr->ch[ind];

// if(curr==NULL){

// cout<<"break"<<j<<endl;

// break;

// }

}

res.push\_back(arr[i].substr(0, len));

}

//delete root;

return res;

}

};

# 375. Word Break Problem | (Trie solution)

Given a string **A**and a dictionary of n words **B**, find out if A can be segmented into a space-separated sequence of dictionary words.

**Example 1:**

**Input:**

n = 12

B = { "i", "like", "sam", "sung", "samsung",

"mobile","ice","cream", "icecream", "man",

"go", "mango" }, A = "ilike"

**Output:** 1

**Explanation:** The string can be segmented as

"i like".

**Example 2:**

**Input**:

n = 12

B = { "i", "like", "sam", "sung", "samsung",

"mobile","ice","cream", "icecream", "man",

"go", "mango" }, A = "ilikesamsung"

**Output:** 1

**Explanation**: The string can be segmented as

"i like samsung" or "i like sam sung".

**Your Task:**  
Complete **wordBreak()** function which takes a string and list of strings as a parameter and returns 1 if it is possible to break words, else return 0. You don't need to read any input or print any output, it is done by driver code.

**Expected time complexity:**O(n\*l) wwhere l is the leght of longest string present in dictionary  
**Expected auxiliary space:** O(|A| + k) , where k = sum of length of all strings present in B

**Constraints**:  
1 <= N <= 12  
1 <= s <=1000 , where s = length of string A  
 The length of each word is less than 15.

## Solution:

The solution discussed here is mainly an extension of below DP based solution.   
[Dynamic Programming | Set 32 (Word Break Problem)](https://www.geeksforgeeks.org/dynamic-programming-set-32-word-break-problem/)

In the above post, a simple array is used to store and search words in a dictionary. Here we use [Trie](https://www.geeksforgeeks.org/trie-insert-and-search/) to do these tasks quickly.

// A DP and Trie based program to test whether

// a given string can be segmented into

// space separated words in dictionary

#include <iostream>

using namespace std;

const int ALPHABET\_SIZE = 26;

// trie node

struct TrieNode

{

struct TrieNode \*children[ALPHABET\_SIZE];

// isEndOfWord is true if the node represents

// end of a word

bool isEndOfWord;

};

// Returns new trie node (initialized to NULLs)

struct TrieNode \*getNode(void)

{

struct TrieNode \*pNode = new TrieNode;

pNode->isEndOfWord = false;

for (int i = 0; i < ALPHABET\_SIZE; i++)

pNode->children[i] = NULL;

return pNode;

}

// If not present, inserts key into trie

// If the key is prefix of trie node, just

// marks leaf node

void insert(struct TrieNode \*root, string key)

{

struct TrieNode \*pCrawl = root;

for (int i = 0; i < key.length(); i++)

{

int index = key[i] - 'a';

if (!pCrawl->children[index])

pCrawl->children[index] = getNode();

pCrawl = pCrawl->children[index];

}

// mark last node as leaf

pCrawl->isEndOfWord = true;

}

// Returns true if key presents in trie, else

// false

bool search(struct TrieNode \*root, string key)

{

struct TrieNode \*pCrawl = root;

for (int i = 0; i < key.length(); i++)

{

int index = key[i] - 'a';

if (!pCrawl->children[index])

return false;

pCrawl = pCrawl->children[index];

}

return (pCrawl != NULL && pCrawl->isEndOfWord);

}

// returns true if string can be segmented into

// space separated words, otherwise returns false

bool wordBreak(string str, TrieNode \*root)

{

int size = str.size();

// Base case

if (size == 0) return true;

// Try all prefixes of lengths from 1 to size

for (int i=1; i<=size; i++)

{

// The parameter for search is str.substr(0, i)

// str.substr(0, i) which is prefix (of input

// string) of length 'i'. We first check whether

// current prefix is in dictionary. Then we

// recursively check for remaining string

// str.substr(i, size-i) which is suffix of

// length size-i

if (search(root, str.substr(0, i)) &&

wordBreak(str.substr(i, size-i), root))

return true;

}

// If we have tried all prefixes and none

// of them worked

return false;

}

// Driver program to test above functions

int main()

{

string dictionary[] = {"mobile","samsung","sam",

"sung","ma\n","mango",

"icecream","and","go","i",

"like","ice","cream"};

int n = sizeof(dictionary)/sizeof(dictionary[0]);

struct TrieNode \*root = getNode();

// Construct trie

for (int i = 0; i < n; i++)

insert(root, dictionary[i]);

wordBreak("ilikesamsung", root)? cout <<"Yes\n": cout << "No\n";

wordBreak("iiiiiiii", root)? cout <<"Yes\n": cout << "No\n";

wordBreak("", root)? cout <<"Yes\n": cout << "No\n";

wordBreak("ilikelikeimangoiii", root)? cout <<"Yes\n": cout << "No\n";

wordBreak("samsungandmango", root)? cout <<"Yes\n": cout << "No\n";

wordBreak("samsungandmangok", root)? cout <<"Yes\n": cout << "No\n";

return 0;

}

**Output:**

Yes

Yes

Yes

Yes

Yes

No

**Output:**

Yes

Yes

Yes

Yes

Yes

No

**My approach:**

class Solution{

public:

// A : given string to search

// B : vector of available strings

Trie\* start;

int search(Trie\* root, string str, int ind){

if(ind==str.size() && root->isLeaf==true)

return 1;

if(ind==str.size())

return 0;

int chr = str[ind] - 'a';

if(root->ch[chr]==NULL){

if(root->isLeaf==true)

return search(start, str, ind);

return 0;

}

int x = search(root->ch[chr], str, ind+1);

if(root->isLeaf==false || x==1)

return x;

int y = search(start, str, ind);

return y;

}

int wordBreak(string A, vector<string> &B) {

//code here

int n = B.size();

Trie\* root = getNode();

start = root;

Trie\* curr = root;

for(int i=0;i<n;i++){

curr = root;

for(int j=0;j<B[i].size();j++){

int ind = B[i][j] - 'a';

if(curr->ch[ind]!=NULL){

curr = curr->ch[ind];

}

else{

Trie\* nd = getNode();

curr->ch[ind] = nd;

curr = nd;

}

}

curr->isLeaf = true;

}

return search(root, A, 0);

}

};

# 376. Given a sequence of words, print all anagrams together

Given an array of strings, return all groups of strings that are anagrams. The groups must be created in order of their appearance in the original array. Look at the sample case for clarification.

**Note: The finial output will be in lexicographic order.**

**Example 1:**

**Input:**

N = 5

words[] = {act,god,cat,dog,tac}

**Output:**

act cat tac

god dog

**Explanation:**

There are 2 groups of

anagrams "god", "dog" make group 1.

"act", "cat", "tac" make group 2.

**Example 2:**

**Input:**

N = 3

words[] = {no,on,is}

**Output:**

no on

is

**Explanation:**

There are 2 groups of

anagrams "no", "on" make group 1.

"is" makes group 2.

**Your Task:**  
The task is to complete the function **Anagrams()** that takes a list of strings as input and returns a list of groups such that each group consists of all the strings that are anagrams.

**Expected Time Complexity:** O(N\*|S|\*log|S|), where |S| is the length of the strings.  
**Expected Auxiliary Space:** O(N\*|S|), where |S| is the length of the strings.

**Constraints:**  
1<=N<=100

1<=|S|<=10

## Solution:

We have discussed two different methods in the [previous post](https://www.geeksforgeeks.org/given-a-sequence-of-words-print-all-anagrams-together/). In this post, a more efficient solution is discussed.  
Trie data structure can be used for a more efficient solution. Insert the sorted order of each word in the trie. Since all the anagrams will end at the same leaf node. We can start a linked list at the leaf nodes where each node represents the index of the original array of words. Finally, traverse the Trie. While traversing the Trie, traverse each linked list one line at a time. Following are the detailed steps.  
**1)** Create an empty Trie   
**2)** One by one take all words of input sequence. Do following for each word   
…**a)** Copy the word to a buffer.   
…**b)** Sort the buffer   
…**c)** Insert the sorted buffer and index of this word to Trie. Each leaf node of Trie is head of a Index list. The Index list stores index of words in original sequence. If sorted buffer is already present, we insert index of this word to the index list.   
**3)**Traverse Trie. While traversing, if you reach a leaf node, traverse the index list. And print all words using the index obtained from Index list.

Below is the implementation of the above approach:

// An efficient program to print all anagrams together

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#define NO\_OF\_CHARS 26

// Structure to represent list node for indexes of words in

// the given sequence. The list nodes are used to connect

// anagrams at leaf nodes of Trie

struct IndexNode

{

int index;

struct IndexNode\* next;

};

// Structure to represent a Trie Node

struct TrieNode

{

bool isEnd; // indicates end of word

struct TrieNode\* child[NO\_OF\_CHARS]; // 26 slots each for 'a' to 'z'

struct IndexNode\* head; // head of the index list

};

// A utility function to create a new Trie node

struct TrieNode\* newTrieNode()

{

struct TrieNode\* temp = new TrieNode;

temp->isEnd = 0;

temp->head = NULL;

for (int i = 0; i < NO\_OF\_CHARS; ++i)

temp->child[i] = NULL;

return temp;

}

/\* Following function is needed for library function qsort(). Refer

http://www.cplusplus.com/reference/clibrary/cstdlib/qsort/ \*/

int compare(const void\* a, const void\* b)

{ return \*(char\*)a - \*(char\*)b; }

/\* A utility function to create a new linked list node \*/

struct IndexNode\* newIndexNode(int index)

{

struct IndexNode\* temp = new IndexNode;

temp->index = index;

temp->next = NULL;

return temp;

}

// A utility function to insert a word to Trie

void insert(struct TrieNode\*\* root, char\* word, int index)

{

// Base case

if (\*root == NULL)

\*root = newTrieNode();

if (\*word != '\0')

insert( &( (\*root)->child[tolower(\*word) - 'a'] ), word+1, index );

else // If end of the word reached

{

// Insert index of this word to end of index linked list

if ((\*root)->isEnd)

{

IndexNode\* pCrawl = (\*root)->head;

while( pCrawl->next )

pCrawl = pCrawl->next;

pCrawl->next = newIndexNode(index);

}

else // If Index list is empty

{

(\*root)->isEnd = 1;

(\*root)->head = newIndexNode(index);

}

}

}

// This function traverses the built trie. When a leaf node is reached,

// all words connected at that leaf node are anagrams. So it traverses

// the list at leaf node and uses stored index to print original words

void printAnagramsUtil(struct TrieNode\* root, char \*wordArr[])

{

if (root == NULL)

return;

// If a lead node is reached, print all anagrams using the indexes

// stored in index linked list

if (root->isEnd)

{

// traverse the list

IndexNode\* pCrawl = root->head;

while (pCrawl != NULL)

{

printf( "%s ", wordArr[ pCrawl->index ] );

pCrawl = pCrawl->next;

}

}

for (int i = 0; i < NO\_OF\_CHARS; ++i)

printAnagramsUtil(root->child[i], wordArr);

}

// The main function that prints all anagrams together. wordArr[] is input

// sequence of words.

void printAnagramsTogether(char\* wordArr[], int size)

{

// Create an empty Trie

struct TrieNode\* root = NULL;

// Iterate through all input words

for (int i = 0; i < size; ++i)

{

// Create a buffer for this word and copy the word to buffer

int len = strlen(wordArr[i]);

char \*buffer = new char[len+1];

strcpy(buffer, wordArr[i]);

// Sort the buffer

qsort( (void\*)buffer, strlen(buffer), sizeof(char), compare );

// Insert the sorted buffer and its original index to Trie

insert(&root, buffer, i);

}

// Traverse the built Trie and print all anagrams together

printAnagramsUtil(root, wordArr);

}

// Driver program to test above functions

int main()

{

char\* wordArr[] = {"cat", "dog", "tac", "god", "act", "gdo"};

int size = sizeof(wordArr) / sizeof(wordArr[0]);

printAnagramsTogether(wordArr, size);

return 0;

}

**Output:**

cat

tac

act

dog

god

gdo

**My approach:**

truct Trie{

Trie\* child[26];

int index;

};

Trie\* getNode(){

Trie\* nd = (Trie\*)malloc(sizeof(Trie));

if(nd){

nd->index = -1;

for(int i=0;i<26;i++)

nd->child[i] = NULL;

}

return nd;

}

class Solution{

public:

vector<vector<string> > Anagrams(vector<string>& string\_list) {

//code here

Trie\* root= getNode(), \*curr = root;

int n = string\_list.size();

vector<vector<string>> res;

int ind = 0;

for(string str:string\_list){

curr = root;

string temp(str);

sort(temp.begin(), temp.end());

for(int i=0;i<temp.size();i++){

int ch = temp[i]-'a';

if(curr->child[ch]!=NULL){

curr = curr->child[ch];

}

else{

Trie\* nd = getNode();

curr->child[ch] = nd;

curr = nd;

}

}

if(curr->index!=-1){

res[curr->index].push\_back(str);

}

else{

curr->index = ind;

vector<string> vec;

vec.push\_back(str);

res.push\_back(vec);

ind++;

}

}

return res;

}

};

# 377. Implement a Phone Directory

Given a list of contacts **contact[]** of length **n** where each contact is a string which exist in a phone directory and a query string **s**. The task is to implement a search query for the phone directory. Run a search query for each prefix **p** of the query string **s**(*i.e.* from  index 1 to |s|) that prints all the distinct contacts which have the same prefix as p in **lexicographical increasing order**. Please refer the explanation part for better understanding.  
**Note:**If there is no match between query and contacts, print "0".

**Example 1:**

**Input:**

n = 3

contact[] = {"geeikistest", "geeksforgeeks",

"geeksfortest"}

s = "geeips"

**Output:**

geeikistest geeksforgeeks geeksfortest

geeikistest geeksforgeeks geeksfortest

geeikistest geeksforgeeks geeksfortest

geeikistest

0

0

**Explaination:** By running the search query on

contact list for "g" we get: "geeikistest",

"geeksforgeeks" and "geeksfortest".

By running the search query on contact list

for "ge" we get: "geeikistest" "geeksforgeeks"

and "geeksfortest".

By running the search query on contact list

for "gee" we get: "geeikistest" "geeksforgeeks"

and "geeksfortest".

By running the search query on contact list

for "geei" we get: "geeikistest".

No results found for "geeip", so print "0".

No results found for "geeips", so print "0".

**Your Task:**  
Youd do not need to read input or print anything. Your task is to complete the function **displayContacts()** which takes **n, contact[ ]**and**s** as input parameters and returns a list of list of strings for required prefixes. If some prefix has no matching contact return "0" on that list.

**Expected Time Complexity:** O(|s| \* n \* max|contact[i]|)  
**Expected Auxiliary Space:** O(n \* max|contact[i]|)

**Constraints:**  
1 ≤ n ≤ 50  
1 ≤ |contact[i]| ≤ 50  
1 ≤ |s| ≤ 6

## Solution:

struct Trie{

Trie\* child[26];

bool isLeaf;

};

Trie\* getNode(){

Trie\* nd = (Trie\*)malloc(sizeof(Trie));

if(nd){

nd->isLeaf = false;

for(int i=0;i<26;i++)

nd->child[i] = NULL;

}

return nd;

}

class Solution{

public:

void printSuggestions(Trie\* curr, vector<string> &temp, string prefix){

if(curr->isLeaf){

temp.push\_back(prefix);

}

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

Trie\* nxt = curr->child[ch-'a'];

if(nxt!=NULL){

prefix.push\_back(ch);

printSuggestions(nxt, temp, prefix);

prefix.pop\_back();

}

}

}

vector<vector<string>> displayContacts(int n, string contact[], string s)

{

// code here

Trie\* root = getNode(), \*curr = root;

for(int i=0;i<n;i++){

curr = root;

for(int j=0;j<contact[i].size();j++){

int ind = contact[i][j] - 'a';

if(curr->child[ind]==NULL)

curr->child[ind] = getNode();

curr = curr->child[ind];

}

curr->isLeaf = true;

}

Trie\* prev = root;

vector<vector<string>> res;

string prefix = "";

for(int i=0;i<s.size();i++){

char lastch = s[i];

prefix.push\_back(lastch);

if(prev!=NULL)

curr = prev->child[lastch-'a'];

vector<string> temp;

if(curr!=NULL){

printSuggestions(curr, temp, prefix);

}

else{

temp.push\_back("0");

}

res.push\_back(temp);

temp.clear();

prev = curr;

}

return res;

}

};

# 378. Print unique rows in a given boolean matrix

Given a binary matrix your task is to find all unique rows of the given matrix.

**Example 1:**

**Input:**

row = 3, col = 4

M[][] = {{1 1 0 1},{1 0 0 1},{1 1 0 1}}

**Output:** 1 1 0 1 $1 0 0 1 $

**Explanation:** Above the matrix of size 3x4

looks like

1 1 0 1

1 0 0 1

1 1 0 1

The two unique rows are 1 1 0 1 and

1 0 0 1 .

**Your Task:**  
You only need to implement the given function **uniqueRow()**. The function takes three arguments the first argument is a matrix **M** and the next two arguments are **row** and **col** denoting the rows and columns of the matrix. The function should **return** the list of the unique row of the martrix. Do not read input, instead use the arguments given in the function.

**Note:**The drivers code print the rows "$" separated.

**Expected Time Complexity:** O(row \* col)  
**Expected Auxiliary Space:** O(row \* col)

**Constraints:**  
1<=row,col<=40  
0<=M[][]<=1

## Solution:

**My code:**

vector<vector<int>> uniqueRow(int M[MAX][MAX],int row,int col)

{

//Your code here

Trie\* root = getNode(), \*curr = root;

vector<vector<int>> res;

for(int i=0;i<row;i++){

curr = root;

vector<int> temp;

for(int j=0;j<col;j++){

if(curr->child[M[i][j]]==NULL)

curr->child[M[i][j]] = getNode();

curr = curr->child[M[i][j]];

temp.push\_back(M[i][j]);

}

if(curr->isLeaf==false){

curr->isLeaf = true;

res.push\_back(temp);

}

}

return res;

}

**Method 1:** This method explains the simple approach towards solving the above problem.

**Approach:** A simple approach would be to check each row with all processed rows. Print the first row. Now, starting from the second row, for each row, compare the row with already processed rows. If the row matches with any of the processed rows, skip it else print it.

**Algorithm:**

1. Traverse the matrix row-wise
2. For each row check if there is any similar row less than the current index.
3. If any two rows are similar then do not print the row.
4. Else print the row.

**Complexity Analysis:**

* **Time complexity:** O( ROW^2 x COL ).   
  So for every row check if there is any other similar row. So the time complexity is O( ROW^2 x COL ).
* **Auxiliary Space:** O(1).   
  As no extra space is required.

**Method 2:** This method uses [Binary Search Tree](https://www.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/) to solve the above operation. The Binary Search Tree is a node-based binary tree data structure which has the following properties:

* The left subtree of a node contains only nodes with keys lesser than the node’s key.
* The right subtree of a node contains only nodes with keys greater than the node’s key.
* The left and right subtree each must also be a binary search tree.
* There must be no duplicate nodes.

The above properties of Binary Search Tree provide ordering among keys so that the operations like search, minimum and maximum can be done fast. If there is no order, then we may have to compare every key to search a given key.

**Approach:**The process must begin from finding the decimal equivalent of each row and inserting them into a BST. As we know, each node of the BST will contain two fields, one field for the decimal value, other for row number. One must not insert a node if it is duplicated. Finally, traverse the BST and print the corresponding rows.

**Algorithm:**

1. Create a BST in which no duplicate elements can be stored. Create a function to convert a row into decimal and to convert the decimal value into binary array.
2. Traverse through the matrix and insert the row into the BST.
3. Traverse the BST (inorder traversal) and convert the decimal into binary array and print it.

**Output:**

1 0 1 0 0

1 0 1 1 0

0 1 0 0 1

**Complexity Analysis:**

* **Time complexity:** O( ROW x COL + ROW x log( ROW ) ).   
  To traverse the matrix time complexity is O( ROW x COL) and to insert them into BST time complexity is O(log ROW) for each row. So overall time complexity is O( ROW x COL + ROW x log( ROW ) )
* **Auxiliary Space:** O( ROW ).   
  To store the BST O(ROW) space is needed.

**Method 3:**This method uses [Trie data structure](https://www.geeksforgeeks.org/trie-insert-and-search/) to solve the above problem. Trie is an efficient information retrieval data structure. Using Trie, search complexities can be brought to an optimal limit (key length). If we store keys in the binary search tree, a well-balanced BST will need time proportional to M \* log N, where M is maximum string length and N is the number of keys in the tree. Using Trie, we can search the key in O(M) time. However, the penalty is on Trie storage requirements.  
**Note:** This method will lead to Integer Overflow if the number of columns is large.

**Approach:**  
Since the matrix is boolean, a variant of Trie data structure can be used where each node will be having two children one for 0 and other for 1. Insert each row in the Trie. If the row is already there, don’t print the row. If the row is not there in Trie, insert it in Trie and print it.

**Algorithm:**

1. Create a Trie where rows can be stored.
2. Traverse through the matrix and insert the row into the Trie.
3. Trie cannot store duplicate entries so the duplicates will be removed
4. Traverse the Trie and print the rows.

**Output:**

0 1 0 0 1

1 0 1 1 0

1 0 1 0 0

**Complexity Analysis:**

* **Time complexity:** O( ROW x COL ).   
  To traverse the matrix and insert in the trie the time complexity is O( ROW x COL). This method has better time complexity. Also, the relative order of rows is maintained while printing but it takes a toll on space.
* **Auxiliary Space:** O( ROW x COL ).   
  To store the Trie O(ROW x COL) space complexity is needed.

**Method 4:** This method uses [HashSet data structure](https://www.geeksforgeeks.org/hashset-in-java/) to solve the above problem. The HashSet class implements the Set interface, backed by a hash table which is actually a HashMap instance. No guarantee is made as to the iteration order of the set which means that the class does not guarantee the constant order of elements over time. This class permits the null element. The class offers constant time performance for the basic operations like add, remove, contains and size assuming the hash function disperses the elements properly among the buckets.

**Approach:** In this method convert the whole row into a single String and then if check it is already present in the HashSet or not. If the row is present then we will leave it otherwise we will print unique row and add it to HashSet.

**Algorithm:**

1. Create a HashSet where rows can be stored as a String.
2. Traverse through the matrix and insert the row as String into the HashSet.
3. HashSet cannot store duplicate entries so the duplicates will be removed
4. Traverse the HashSet and print the rows.

**Complexity Analysis:**

* **Time complexity:** O( ROW x COL ).   
  To traverse the matrix and insert in the HashSet the time complexity is O( ROW x COL)
* **Auxiliary Space:** O( ROW ).   
  To store the HashSet O(ROW x COL) space complexity is needed.