**Trie:**

**1.Trie Good Memory Saving Implementation Based On Map:**

struct trie\_node

{

map<char,struct trie\_node \*> nodes;

bool is\_word\_ended;

};

struct trie\_node \*create\_new\_trie\_node()

{

struct trie\_node \*new\_trie\_node=NULL;

try

{

new\_trie\_node=new trie\_node();

new\_trie\_node->is\_word\_ended=false;

}

catch(const bad\_alloc& e)

{

printf("Memory could be allocated for new\_trie\_node\n");

exit(EXIT\_FAILURE);

}

return new\_trie\_node;

}

struct trie\_node \*insert(struct trie\_node \*root,const char \*str)

{

if(str==NULL)

{

return root;

}

int str\_length=strlen(str);

if(str\_length==0)

{

return root;

}

if(root==NULL)

{

root=create\_new\_trie\_node();

}

struct trie\_node \*curr=root;

for(int i=0;i<str\_length;i++)

{

if(curr->nodes.find(str[i])==curr->nodes.end())

{

curr->nodes[str[i]]=create\_new\_trie\_node();

}

curr=curr->nodes[str[i]];

}

curr->is\_word\_ended=true;

return root;

}

bool search(struct trie\_node \*root,const char \*str)

{

if(str==NULL)

{

printf("The pointer str does not point to any memory location\n");

return true;

}

int str\_length=strlen(str);

if(str\_length==0)

{

return true;

}

if(root==NULL)

{

printf("The root of the trie\_node is NULL\n");

return false;

}

struct trie\_node \*curr=root;

for(int i=0;i<str\_length;i++)

{

if(curr->nodes[str[i]]==NULL)

{

return false;

}

else

{

curr=curr->nodes[str[i]];

}

}

if(curr->is\_word\_ended==true)

{

return true;

}

else

{

//otherwise, the search string is a prefix of some other string

return false;

}

}

So, this a better implementation in compare to the fixed 26 pointer to the trie node approach

**2.Longest Prefix Matching**

**3.Find Shortest Unique Prefix For Every Word In A Given List:**

I can do it.

However, still let’s discuss the approach:  
  
An Efficient Solution is to use Trie. The idea is to maintain a count in every node. Below are steps.

1) Construct a Trie 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.

1. 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.

**(Now, since, we have to find a node nearest to root, we need to do BFS rather than DFS)**

Time complexity if this step also is O(N) where N is total number of characters in all words.

**4.Longest Common Prefix In A Trie:**

Given a set of strings, find the longest common prefix.

Insert all the words one by one in the trie. After inserting we perform a walk on the trie.

In this walk, go deeper until we find a node having more than 1 children(branching occurs) or 0 children (one of the string gets exhausted).

This is because the characters (nodes in trie) which are present in the longest common prefix must be the single child of its parent, i.e- there should not be a branching in any of these nodes.

1. **Print all words matching a pattern in CamelCase Notation Dictionary:**

I can do it.

# Implement a Phone Directory

Given a list of contacts which exist in a phone directory. The task is to implement search query for the phone directory. The search query on a string ‘str’ displays all the contacts which prefix as ‘str’. One special property of the search function is that, when a user searches for a contact from the contact list then suggestions (Contacts with prefix as the string entered so for) are shown after user enters each character.

Note : Contacts in the list consist of only lower case alphabets.

I can do it.

**7.Print Unique Rows In a Boolean Matrix:**

I can do it.

**8.Find the k most frequent words from a file**

We can use Trie and Min Heap to get the k most frequent words efficiently. The idea is to use Trie for searching existing words adding new words efficiently. Trie also stores count of occurrences of words. A Min Heap of size k is used to keep track of k most frequent words at any point of time(Use of Min Heap is same as we used it to find k largest elements in this post).

Trie and Min Heap are linked with each other by storing an additional field in Trie ‘indexMinHeap’ and a pointer ‘trNode’ in Min Heap. The value of ‘indexMinHeap’ is maintained as -1 for the words which are currently not in Min Heap (or currently not among the top k frequent words). For the words which are present in Min Heap, ‘indexMinHeap’ contains, index of the word in Min Heap. The pointer ‘trNode’ in Min Heap points to the leaf node corresponding to the word in Trie.

Following is the complete process to print k most frequent words from a file.

Read all words one by one. For every word, insert it into Trie. Increase the counter of the word, if already exists. Now, we need to insert this word in min heap also. For insertion in min heap, 3 cases arise:

1. The word is already present. We just increase the corresponding frequency value in min heap and call minHeapify() for the index obtained by “indexMinHeap” field in Trie. When the min heap nodes are being swapped, we change the corresponding minHeapIndex in the Trie. Remember each node of the min heap is also having pointer to Trie leaf node.

2. The minHeap is not full. we will insert the new word into min heap & update the root node in the min heap node & min heap index in Trie leaf node. Now, call buildMinHeap().

3. The min heap is full. Two sub-cases arise.

….3.1 The frequency of the new word inserted is less than the frequency of the word stored in the head of min heap. Do nothing.

….3.2 The frequency of the new word inserted is greater than the frequency of the word stored in the head of min heap. Replace & update the fields. Make sure to update the corresponding min heap index of the “word to be replaced” in Trie with -1 as the word is no longer in min heap.

4. Finally, Min Heap will have the k most frequent words of all words present in given file. So we just need to print all words present in Min Heap.

**9.Count Distinct Substrings Of A String Using Suffix Trie:**

**(important)**

The idea is create a Trie of all suffixes of given string. Once the Trie is constructed, our answer is total number of nodes in the constructed Trie.

Because,

Each root to node path of a Trie represents a prefix of words present in Trie. Here we words are suffixes. So each node represents a prefix of suffixes.

**Every substring of a string “str” is a prefix of a suffix of “str”.**

**10.Find pair of rows in a binary matrix that has maximum bit difference**

1). Create an empty Trie. Every node of Trie contains two children for 0 and 1 bits.

2). Insert First Row of Binary matrix into Trie

3).Traverse rest of the rows of given Binary Matrix

a). For Each Row First we search maximum bit difference

with rows that we insert before that in Trie and

count bit difference

b). For every search we update maximum bit\_diff count

if needed else not store pair of index that have

maximum bit difference

c). At Last Print Pair

**Weighted Prefix Search:**

I can do it.

The idea is to create and maintain a Trie. Instead of the normal Trie where we store the character, store a number with it, which is maximum value of its prefix. When we encounter the prefix again update the value with maximum of existing and new one.  
Now, search prefix for maximum value, run through the characters starting from the root, if one of character is missing return -1, else return the number stored in the root.

Now, what will happen? There will be a weighted path with each node containing the maximum value. We have to go for that.

1. **Minimum XOR value Pair: (Important)**

1). Create an empty trie. Every node of trie contains two children

for 0 and 1 bits.

2). Initialize min\_xor = INT\_MAX, insert arr[0] into trie

3). Traversal all array element one-by-one starting from second.

a. First find minimum setbet difference value in trie

do xor of current element with minimum setbit diff that value

b. update min\_xor value if required

c. insert current array element in trie

4). return min\_xor

1. **Find The Maximum Subarray XOR: (important)**

1) Create an empty Trie. Every node of Trie is going to

contain two children, for 0 and 1 value of bit.

2) Initialize pre\_xor = 0 and insert into the Trie.

3) Initialize result = minus infinite

4) Traverse the given array and do following for every

array element arr[i].

a) pre\_xor = pre\_xor ^ arr[i]

pre\_xor now contains xor of elements from

arr[0] to arr[i].

b) Query the maximum xor value ending with arr[i]

from Trie.

c) Update result if the value obtained in step

4.b is more than current value of result.

**How does this step work?**

Because, the trie already contains xor values of arr[0],arr[0..1],arr[0..2], to arr[0…i-1]

1. **Boggle Using Trie:**

SearchWord(Trie \*root, i, j, visited[][N])

if root->leaf == true

print word

if we seen this element first time then make it visited.

visited[i][j] = true

do

traverse all child of current root

k goes (0 to 26 ) [there are only 26 Alphabet]

add current char and search for next character

find next character which is adjacent to boggle[i][j]

they are 8 adjacent cells of boggle[i][j] (i+1, j+1),

(i+1, j) (i-1, j) and so on.

make it unvisited visited[i][j] = false

1. **Print all valid words that are possible using Characters of Array**

Given a dictionary and a character array, print all valid words that are possible using characters from the array.

Examples:

Input : Dict - {"go","bat","me","eat","goal",

"boy", "run"}

arr[] = {'e','o','b', 'a','m','g', 'l'}

Output : go, me, goal.

The idea is to use Trie data structure to store dictionary, then search words in Trie using characters of given array.

Create an empty Trie and insert all words of given dictionary into the Trie.

After that, we have pick only those characters in ‘Arr[]’ which are a child of the root of Trie.

To quickly find whether a character is present in array or not, we create a hash of character arrays.

1. **Palindromic Pairs In An Array Of Words:**

I can do that.