Advanced Data Structure

15. Trie

Handwritten by pankaj kumar

Trie

Date
Page No. 212,

Date
Page No. 212.

① Introduction:

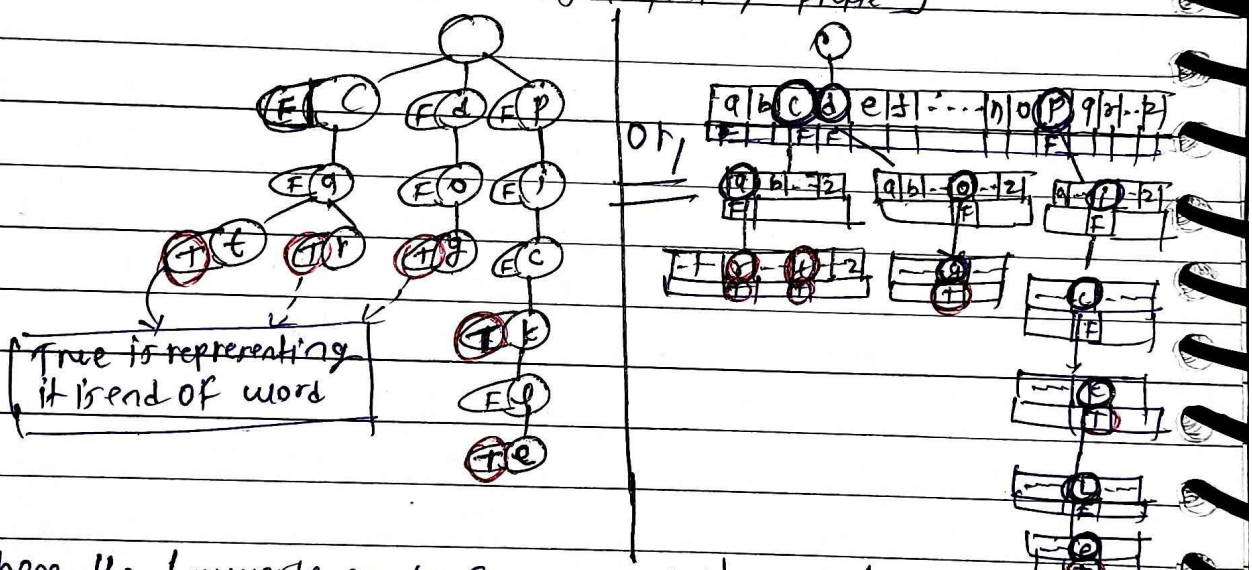
- A Trie is an advanced data structure that is sometimes also known as prefix tree or digital tree. it is a tree that store the data in an ordered and efficient way. we generally use trie's to store strings. Each node of a trie can have as many as 26 references (pointers).
- Each node of a try consists of two things:

① A character ② A boolean value (represent whether this character is end of word)

- Tries in general used to store English characters, hence each character can have 26 references. Node in a trie do not store entires keys, instead they store a part of key (usually next character of string)

ex:- build a trie by inserting 5 words in it

words = ["cat", "car", "dog", "pick", "pickle"]



- here the keywords can be form as we traverse down from the root node to the leaf nodes. The True boolean value is denoted that it is endofword (particular word)
- Tries are not balanced in nature, unlike AVL trees.

* Why use Trie Data structure?

When we talk about the fastest ways to retrieve values from a data structure, hash table generally come to our mind. Though, very

Trie is

efficient in nature but still very less talked about as when compared to hash table, trie's are much more efficient than hash table and also they possess several advantages over the same. mainly:- (Trie is slower than has table but).

- There won't be any collisions hence making the worst performance better than a hash table i.e not implemented properly.
- No need for hash functions.
- Lookup time for a string in trie is O(k), where k = length of the word.
- it can take even less than O(k) time when the word is not there in a trie.

(2.) Trie Application:-

- **Auto complete Feature:-** it is used in search engine, After we type something in the search bar, the tree of the potential words that we might enter is greatly reduced, which in turn allows the program to enumerate what kinds of string are possible for the words we have typed in.
- it also help in the case where we want to store additional information of a word, say the popularity of the word, which make it powerful. you might have seen that when you type "foot" on the search bar, you get "football" before anything say "footpath". it is bcz "football" is a much popular word.
- Trie also has helped in checking the correct spelling of a word, as a path is similar for a slightly mis pelled word. **Spell checker**
- **String matching**
- **Browser history**

(3) Some key points:-

- creating a trie is take. $O(m*n)$, m = no. of words, n = avg. length of each word.
- Inserting a node in a trie take $O(k)$ time
- Inserting a node in a trie has space complexity $O(n)$
- Time complexity to search a key. $O(n)$
- Time complexity to search for a prefix of a key $O(k)$

④ Function Implementation of trie. (Insertion / search / delete)

* Insertion :- Every character inserted as an individual trie node the children are a array of pointer (or references) to next level trie nodes. the character key acts as an index into the array of children. if the input key is new or an extension of the existing key. we need to construct non-existing node of the key and mark end of the word for last node.
- if the i/p 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.

* Searching :- we compare character and move down. the search can terminate due to end of a string or lack of key in trie.
- if the isEndofword field of the last node is true then the key exist in the trie.
- if key is not present then search terminate without examining all the character of the key.

* deletion :- During delete operation, we delete the key in a bottom-up manner using recursion. the following are cases while deleting.

case 1:- key may not present, then delete operation should not modify trie.

case 2:- part of the key should not contain prefix (i.e other input), nor the key itself as a prefix of another key. in this delete only that key which is not part of another key.

case 3:- if the key is prefix key of another long key in the trie. in this case, simply unmark the leaf node.

code :-

```
#include<bits/stdc++.h>
using namespace std;
const int ALPHABET_SIZE=26;
struct TrieNode{
    struct TrieNode* children[ALPHABET_SIZE];
    bool isEndOfWord;
};
```

```
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;
}
```

①
```
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];
    }
    pCrawl->isEndOfWord = true;
}
```

②
```
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);
}
```

```
bool isEmpty(TrieNode* root){
    for(int i=0; i<ALHABET_SIZE;i++)
        if (root->children[i]) return false;
    return true;
}
```

③ TrieNode* remove (TrieNode* root, string key, int depth=0)

```
if(!root) return NULL;          // if empty tree
if(depth == key.size()){        // if last character being processed
    if(root->isEndOfWord)       // This node is no more
        root->isEndOfWord=false;// end of word.
    if(isEmpty(root)){          // if this word is not prefix of any other word
        delete(root);
        root=NULL;
    }
    return root;
}
```

if not last character recur for the child

```
int index = key[depth]-'a';
root->children[index]= remove(root->children[index], key, depth+1);
```

if root does not have any child and it is not end of another word

```
if(isEmpty(root) && root->isEndOfWord==false){
    delete(root);
    root=NULL;
}
return root;
}
```

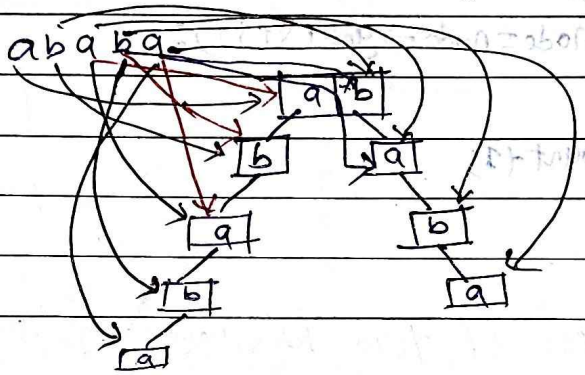
```
int main(){
    string keys[]={"the","a","there","answer","any","by","bye","their","hero","heroplane"};
    int n=sizeof(keys)/sizeof(keys[0]);
    struct TrieNode* root=getNode();
    for(int i=0; i<n; i++) insert(root, keys[i]);
    search(root,"the")? cout<<"Yes\n":cout<<"No\n";      // Yes
    search(root,"there")? cout<<"Yes\n":cout<<"No\n";    // No
    remove(root,"heroplane");
    search(root,"hero")?cout<<"Yes\n":cout<<"No\n";      // Yes
    search(root,"heroplane")? cout<<"Yes":cout<<"No\n";  // NO
    return 0;
}
```

(5) Number of Distinct Substrings in a String Using Trie.

Given a string of alphabetic characters. Return the count of distinct substrings of the string (including the empty string) using the Trie data structure.

ex:- S = "ababa"
O/P = 10

All the substrings of the strings are a, ab, aba, abab, ababa, b, ba, bab, baba, a, ab, aba, b, ba, a. many of the substring are duplicated. The distinct substrings are a, ab, aba, abab, ababa, b, ba, bab, baba, Total count of distinct substring are (9+1) (1 for empty) = 10.



the total strings inserted in trie are.
[a, ab, aba, abab, ababa, b, ba, bab, baba, a, ab, aba, b, ba, a,]

Intution:- The basic intution is to generate all the substrings & store them in the trie along with its creation. if a substring already exists in the trie then we can ignore, else we can make an entry and increase the count

Code:-

```
struct Node{
    Node * links[26];
    bool containsKey(char ch){
        return (links[ch-'a'] != NULL);
    }
    Node * get(char ch){
        return links[ch-'a'];
    }
    void put(char ch, Node *node){
        links[ch-'a'] = node;
    }
};
```

```
int countDistinctSubstrings(string s){
    Node *root = new Node();
    int count = 0;
    int n = s.size();
    for(int i=0; i<n; i++){
        Node *node = root;
        for(int j=i; j<n; j++){
            if(!node->contains(s[j])){
                node->put(s[j], new Node());
                count++;
            }
            node = node->get(s[j]);
        }
    }
    return count+1;
}
```

Time complexity: $O(n^2)$

SC: $O(n^2)$

(6) Maximum XOR of Two Numbers in two array.

You are given two arrays of non-negative integers say 'arr1' and 'arr2' of size N and M respectively. Find the maxm value of ('A' xor 'B') where 'A' and 'B' are any elements from 'arr1' and 'arr2' respectively and 'xor' represents the bitwise xor operation.

ex:- N=2, m=3

arr1 = [6,8] , arr2 = [7, 8, 2] | o/p : 15

Explanation:- Possible pair are (6,7), (6,8), (6,2), (8,7), (8,8), (8,2) the maxm pair xor are, 8 xor 7 will given 15.

Approach:-

Step-I :- Insert all the elements of arr1 into TRIE.

Step-II :- Take X and Find the maxm number from the arr1 where X ^ arr[i] is maxm

Note:- here, treate every element of arr2 as X and find the MaxXOR and consider the maximum of all.

Example :—

① Insertion into TRIE:

while inserting the numbers into the trie consider the binary format (integer - 32bit) of the arr[i] and treat it as a string and insert the value.

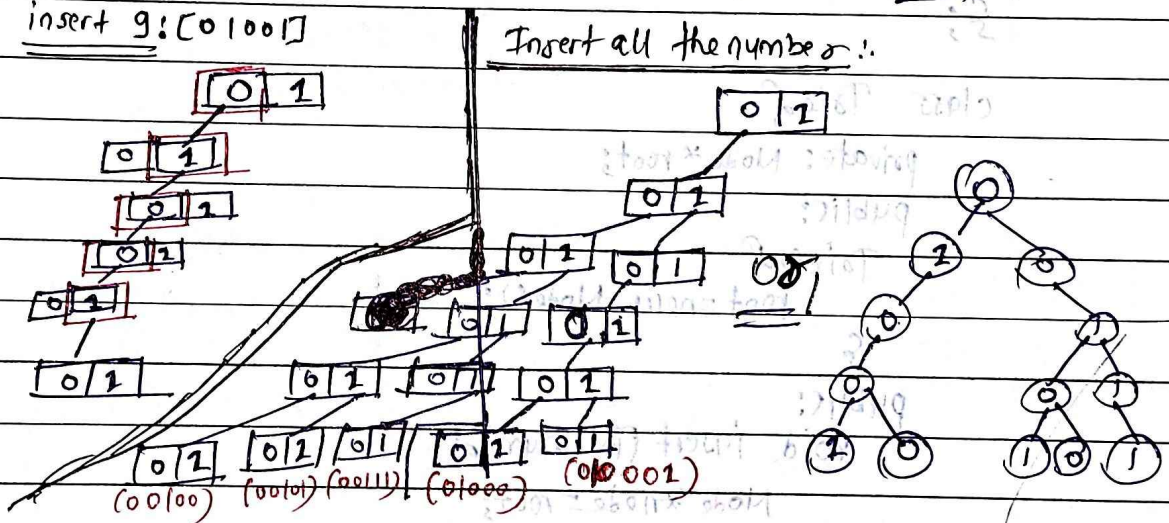
** Note:- here for understanding we consider only 5bit but while coding we have to code it for 32 bit.

arr[] = [9, 8, 7, 5, 4] . and X = 8

~~[here we are taking one element X=8 and will find maxm xor X ^ arr[i]]~~

arr[] = [9, 8, 7, 5, 4]

= ["01001", "01000", "00111", "00101", "00100"]



② maximizing XOR:-

in order to have maxm value, we should have set bits from left - Right.

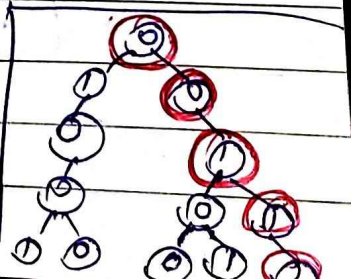
XOR operation:- XOR of same bits = 0 | XOR of different bits = 1

so, for every bit of X=8 (here) Find its opposite bit in Trie if not found then take that bit.

ex:- X=8 ["01000"], 9 → 01001, 8 → 01000, 7 → 00111, 5 → 00101, 4 → 00100

8

0 For, 0 there is no opposite so take 0.
1 For, 1 there is (0) present so choose that path
0 For, 0 there is (1) present at 3rd place so take that path
0 For, 0, there is (1) present take that path
0 For, 0, there is (1) present take that path



here the path is (00111) i.e 7. hence 8^7 = 15 is maxm.

```
  01000
  00111
  01111
```

Code :-

```
struct Node{
    Node *links[2];
    bool containsKey(int ind){
        return (links[ind]!=NULL);
    }
    Node *get(int ind){
        return links[ind];
    }
    void put(int ind, Node *node){
        links[ind]=node;
    }
};

class Trie{
    private: Node *root;
    public:
      Trie(){
          root=new Node();
      }
    public:
      void insert(int num){
          Node *node=root;
          for(int i=31; i>=0; i--){
              int bit = (num>>i)&1;
              if(!node->containsKey(bit)){
                  node->put(bit, new Node());
              }
              node=node->get(bit);
          }
      }
    public:
      int FindMax(int num){
          Node* node = root;
          int maxNum=0;
```

```
for(int i=31; i>=0; i--){
    int bit = (num>>i) & 1;
    if (node->containsKey(!bit)){
        maxNum = maxNum | (1<<i);
        node = node->get(!bit);
    }else{
        node = node->get(bit);
    }
}
return maxNum;
};

int maxXOR(int n, int m, vector<int> &arr1, vector<int> &arr2){
    Trie trie;
    for (int i=0; i<n; i++){
        trie.insert(arr1[i]);
    }
    int maxi=0;
    for (int i=0; i<m; i++){
        maxi = max(maxi, trie.findMax(arr2[i]));
    }
    return maxi;
}
```

TC: O(N*32) + O(M*32)
SC: O(N*32)

(7) Complete String: /Longest words with all prefixes :-

- You have given a array of string, 'A' of size 'N'. Each element of this array is a string.
- A string is called a complete string if every prefix of this string is also present in the array 'A'. You have to find the longest complete string in the array 'A'. If there are multiple string with the same length, return the lexicographically smallest one and if no string exists, return "None".

ex:- N=4
A = ["ab", "abr", "a", "bp"]

- For "ab", prefixes of "ab" i.e "a" and "ab" are present in array. so it is one of the possible strings.
- For "abc", prefixes :- "a", "ab", "abc" all are present in A
- for "a", prefixes:- "a", are present in A.
- for "bp", prefixes:- "b" and "bp" both are not present in A so it is not a complete string.

here the total possible complete strings are, "ab", "abc", "a" in which longest one is "abc"

∴ O/P:- "abc"

* Constraints:-

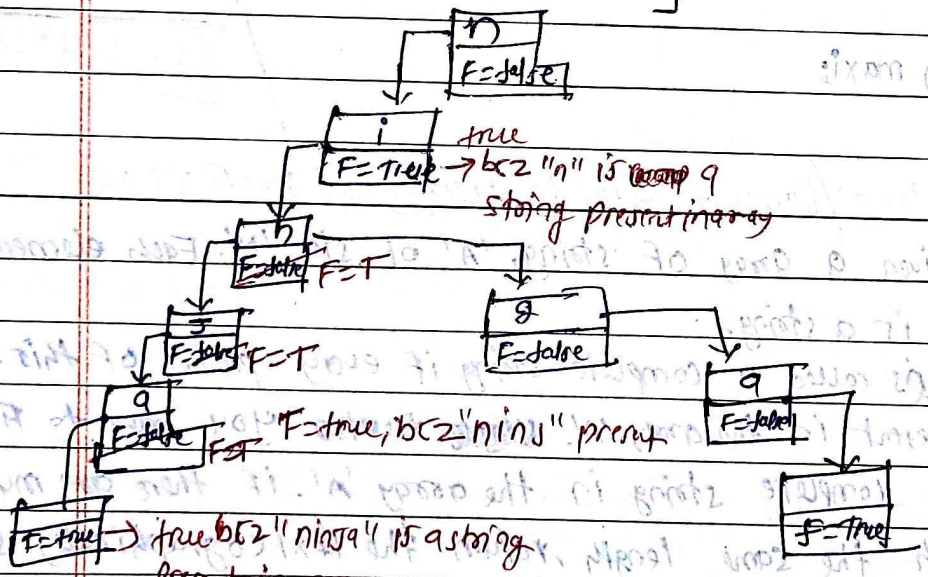
$1 <= N <= 10^5$

~~$1 <= A[i].length <= 10^5$~~

* approach:-

① Insert all in trie ex:- ["n", "ninja", "nin", "ni", "nin", "ninga"]

Trie Node define:- Trie {
arr[26];
bool flag; → Indicate end of string
}



② Find the complete string which is longest

⇒ for every string check if prefixes of every string is present in trie, by checking next flag is True or not.

Insertion T.C ⇒ O(N) × O(len) (len = length of word; N = Average length of all string)

S.C ⇒ We can not find exactly S.C
It can be like 26×26×26×26 ....

* Code :-

```
Struct Node {
    Node *links[26];
    bool flag = false;
    bool containsKey(char ch) {
        return (links[ch-'a'] != NULL);
    }
    Node * get(char ch) {
        return links[ch-'a'];
    }
    void put(char ch, Node * node) {
        links[ch-'a'] = node;
    }
    void setEnd() {
        flag = true;
    }
    bool isEnd() {
        return flag;
    }
};
Class Trie {
    private: Node * root;
  public:
    Trie() {
        root = new Node();
    }

    void Insert(string word) {
        Node *node = root;
        for (int i=0; i<word.size(); i++) {
            if (node -> containsKey(word[i])) {
                node->put(word[i], new Node());
            }
            node = node->get(word[i]);
        }
        node->setEnd();
    }
```

```
bool checkIfAllPrefixExists(string word){
    Node *node = root;
    bool flag = true;
    for(int i=0; i<word.size(); i++){
        if(node->containsKey(word[i])){
            node = node->get(word[i]);
            flag = flag & node->isEnd();
        }
        else{
            return false;
        }
    }
    return flag;
}
};
string completeString(int n, vector<string>& a){
    Trie *obj = new Trie();
    for(auto word: a) obj->insert(word);
    string longest = "";
    for(auto word: a){
        if(obj->checkIfAllPrefixExists(word)){
            if(word.size()>longest.size()){
                longest = word;
            }
            else if(word.size()==longest.size()&&
                word<longest){
                longest = word;
            }
        }
    }
    if(longest=="") return "None";
    return longest;
}
```

TC: O(N * len) — average length of string.

Date

Page No. 225.

(8) Implementing Auto-complete Feature using Trie.

The auto-complete feature is widely useful in showing suggestions when the user types a certain word. In a simpler version, the auto-complete feature list all of the string which have a matching prefix queried by a user.

ex:- if the dictionary stores the following words of "abc", "abcd", "aaa", "abbbaba" & and the user types in "ab" the he must be shown {"abc", "abcd", "abbbaba"} as a result as all of them have the prefix ab.

*Approach :-

(1) Insert all of the given string in a Trie.

(2) Search for the given query using standard Trie search algo.

(3) if query prefix itself is not present, return -1 to indicate the same.

(4) if the query is present and is the end of word in Trie print query. (can be checked by seeing if it is end of word)

(5) if the last matching node of the query has no children return.

(6) Else recursively print all nodes under a subtree of last matching node.