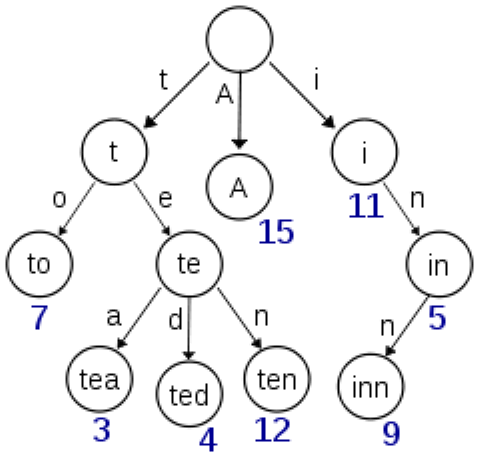
TRIE

DEFINITION: In computer science, a **trie**, also called **digital tree** and sometimes **radix tree** or **prefix tree** (as they can be searched by prefixes), is a kind of search tree—an ordered tree data structure that is used to store a dynamic set or associative array where the keys are usually strings. Unlike a binary search tree, no node in the tree stores the key associated with that node; instead, its position in the tree defines the key with which it is associated. All the descendants of a node have a common prefix of the string associated with that node, and the root is associated with the empty string. Values are not necessarily associated with every node. Rather, values tend only to be associated with leaves, and with some inner nodes that correspond to keys of interest.



SEARCHING (BST vs TRIE): In a balanced search tree containing N keys, time taken for searching key is O(M*log(n)) where M is maximum length of key in tree. With tree we can search key in time O(M). However penalty is storage requirement.

STORAGE REQUIREMENT: Suppose we inserted N word in trie, and MAX_CHAR is the number of distinct character possible and M is maximum length of word. Then maximum number of node possible is N*M. now each node is having MAX_CHAR children pointer, hence total memory requirement is: O(MAX_CHAR*N*M)

STRUCTURE:

```
class TrieNode{
   public:
   TrieNode *children[26];
   bool isEndOfWord;
   TrieNode(){
       cout<<"Constructor Called\n";
       for(int i = 0; i<26; i++){
```

```
                children[i] = nullptr;
            }
            isEndOfWord = false;
        }
        ~TrieNode(){
            cout<<"Destructor Called\n";
        }
    };
```

INSERTION: For insertion, traverse through each character of string and see if this is present in Trie, if present go to next character and corresponding next node; if not present create a new node.

```
    void insert(TrieNode &root, string &str){
        TrieNode *tempPtr = &root;
        for(int i = 0; i<str.size(); i++){
            int index = str[i] - 'a';
            if(tempPtr->children[index] == nullptr){
                tempPtr->children[index] = new TrieNode;
            }
            tempPtr = tempPtr->children[index];
        }
        tempPtr->isEndOfWord = true;
    }
```

TIME COMPLEXITY: O(L) where L is length of string to be inserted.

SEARCHING: For SEARCHING traverse through each character of string and traverse corresponding node in trie, if at any iteration you don't find a node corresponding to current character of string return false. Else if you have traversed through all character just check if the last node is marked as "end of word".

```
    bool find(TrieNode &root, string &str){
        TrieNode *tempPtr = &root;
        for(int i = 0; i<str.size(); i++){
            int index = str[i] - 'a';
            if(tempPtr->children[index] == nullptr) return false;
            tempPtr = tempPtr->children[index];
        }
        return(tempPtr->isEndOfWord);
```

```
            }
TIME COMPLEXITY: O(L) WHERE L IS LENGTH OF STRING.
```

DELETION: Deletion of a node from trie is done recursively. There can be four cases for deletion.

1. Key may not be there in trie. Delete operation should not modify trie.
2. Key present as unique key (no part of key contains another key (prefix), nor the key itself is prefix of another key in trie). Delete all the nodes.
3. Key is prefix key of another long key in trie. Unmark the leaf node.
4. Key present in trie, having at least one other key as prefix key. Delete nodes from end of key until first leaf node of longest prefix key.

```
bool delete_node(TrieNode *node, string &str, int ind){
   int n = str.size();
   if(ind == n){
       if(node->isEndOfWord == true){
           node->isEndOfWord = false;    // mark end of string as false
           if(node->numChild == 0){      // if string is complete string.
               delete node;
               return true;
           }
           return false;
       }
       return false;
   }

     if(node->children[str[ind] - 'a'] == nullptr) return false;   //String doesn't
exist(Case 1).

   else{
       bool check = delete_node(node->children[str[ind] - 'a'], str, ind+1);
       if(check == true){       //String found
           if(ind == 0){
               node->children[str[ind] - 'a'] = nullptr;
               return true;
           }
           if(node->numChild == 1 && node->isEndOfWord == false){
```

```
                    delete node;
                    return true;
                }
                else{
                    node->children[str[ind] - 'a'] = nullptr;
                    return false;
                }
            }
            return false;
        }
    }
```

TIME COMPLEXITY: O(L) where L is length of string.

SORTING: An array of string can be sorted by making trie of words in the array and then doing the preorder traversal.

Time Complexity for creating Trie is: O(L*n) where is L is length of string and n is number of strings in the array.

Time Complexity for PreOrder Traversal is: **O(L*n*MAX_CHAR(usually 26 english letter)**.

To see the time complexity of PreOrder, let us first calculate the maximum number of node in a trie = n*L where n is number of strings and L is max length of string. We are traversing through each node and checking its MAX_CHAR children hence total Time Complexity = O(L*n*MAx_CHAR);

```
 //index = -1 means it’s not end of word.
void sort_print(const TrieNode *root, string s[]){ //Preorder
     for(int i = 0; i<26; i++){
         if(root->children[i] == nullptr) continue;
          if(root->children[i]->index != -1){
             cout<<s[root->children[i]->index]<<endl;
         }
         sort_print(root->children[i], s);
     }
 }
```

MINIMUM WORD BREAK: Given a string s, break s such that every substring of the partition can be found in the dictionary.Return the minimum break needed. Solution is to calculate min break recursively.

For reducing time complexity save the min break for a particular index once calculated. First make a trie using dictionary word then call the function count_break and pass the string for which minimum break is to be calculated.

```
vector<int> dp(s.size() + 1, -1);    //memoization
int min_word_break(TrieNode *rootPtr, string &s, int ind){
   if(ind >= s.size()) return 0;
   int res = INT_MAX - 1;
   if(dp[ind]!=-1) return dp[ind];
   TrieNode *nodePtr = rootPtr;
   for(int i = ind; i<s.size(); i++){
       int index = s[i] - 'a';
       if(nodePtr->children[index] == nullptr){
               break;
       }
       else if(nodePtr->children[index]->isEndOfWord == true) res = min(res, 1 +
min_word_break(rootPtr, s, i + 1));
       nodePtr = nodePtr->children[index];
   }
   dp[ind] = res;
   return res;
}

void count_break(TrieNode *rootPtr, string &s){
   int res = min_word_break(rootPtr, s, 0);
   cout<<res<<endl;
   if(res>=INT_MAX - 1) cout<<"no break possible"<<endl;
   else cout<<"true\n"<<endl;
}
```

TIME COMPLEXITY: Depend on number of break;

WORD BREAK 1: Given an input string and a dictionary of words, find out if the input string can be segmented into a space-separated sequence of dictionary words.

Method is almost like the previous one. We call a recursive function which returns boolean value(here short). Remember base case is when size of substring which is processed to be is 0 and then we return true.

```
short min_word_break(string &s, int ind, vector<short> &v){
        if(ind >= s.size()){
            return 1;
        }

        if(v[ind]!=-1) return v[ind];    //Memoization
```

```
                    TrieNode *nodePtr = &root;
                    for(int i = ind; i<s.size(); i++){
                        int index = s[i] - 'a';
                        if(nodePtr->children[index] == nullptr){
                            v[ind] = 0;   //Not Found
                            return 0;
                        }
                        else if(nodePtr->children[index]->isEndOfWord == true){
                            if(min_word_break(s, i+1, v) == 1){
                                v[ind] = 1;  //Found
                                return 1;
                            }
                        }
                        nodePtr = nodePtr->children[index];
                    }

                    v[ind] = 0;
                   return 0;
        }
```

TIME COMPLEXITY: Depends on number of break

WORD BREAK 2(LeetCode): Given a **non-empty** string *s* and a dictionary *wordDict* containing a list of **non-empty** words, add spaces in *s* to construct a sentence where each word is a valid dictionary word. Return all such possible sentences.

Input:

```
s = "catsanddog"
wordDict = ["cat", "cats", "and", "sand", "dog"]
```

Output:(Print)

```
[
       "cats and dog",
       "cat sand dog"
]
```

Idea is to first make a trie using words in dictionary. Then we call a recursive function with s and root of trie as input. Initially we pass ind = 0 to the function and then we iterate from i = ind to i = s.size() and if we find substring(ind to i) of s in trie then again we call function with s, root of trie and ind = i+1. If function returns **true** we construct a graph in which now there will be a direct edge from ind to i+1.

To print all strings(word breaks) we traverse through graph in dfs manner but we don't mark vertex as visited because we need to traverse a vertex more than once.

```
class vertex{
   public:
   vector<int> adj;
   bool visited = false;
   bool string_exist = false;
};

bool word_break(TrieNode *rootPtr, string &s, int ind, vector<vertex> &v){
   if(ind >= s.size()){
       return true;
   }
   if(v[ind].visited == true){
       return v[ind].string_exist;
   }

   v[ind].visited = true;
   TrieNode *nodePtr = rootPtr;
   bool res = false;
   for(int i = ind; i < s.size(); i++){
       int index = s[i] - 'a';
       if(nodePtr->children[index] == nullptr){
           break;
       }

       if(nodePtr->children[index]->isEndOfWord == true){
           // cout<<"found\n";
           if(word_break(rootPtr, s, i+1, v)){
               res = true;
               v[ind].adj.push_back(i+1);
           }
       }
       nodePtr = nodePtr->children[index];
   }

   v[ind].string_exist = res;
   return res;
}

void dfs(vector<vertex> &v, int ind, string s, int n, string &orig){
   if(ind == n){
       cout<<s<<endl;
       return;
   }
   for(auto elem:v[ind].adj){
       dfs(v, elem, s +orig.substr(ind, elem - ind) + " ", n, orig);
   }
}
```

MAXIMUM XOR: Two arrays A and B consisting of N elements are given. The task is to compute the maximum possible XOR of every element in array A with array B.
We maintain a Trie for the binary representation of all elements in array B.Now, for every element of A, we find the maximum xor in trie.
Let's say our number A[i] is {b1, b2…bn}, where b1, b2…bn are binary bits. We start from b1.
Now for the xor to be maximum, we'll try to make most significant bit 1 after performing the xor. We will start from the left most digit and try to make value after xor 1. So if current bit in A[i] is 1 then we will first search if current TrieNode has children[0] if it has then the xor value of A[i] and B[i] is 1 else 0. Similarly if current bit is 1 we will search if current TrieNode has children[1] if it has then the xor value of A[i] and B[i] is 1 else 0. Note that it's a greedy solution and gives optimal solution.

```
void insert(TrieNode &root, int key){    //Insrt each node of Array B in trie
   TrieNode *nodePtr = &root;

   for(int i = 31; i>=0; i--){
       bool current_bit = (key & (1 << i));
       int index = current_bit - 0;
       // cout<<index<<endl;
       if(nodePtr->children[index] == nullptr){
           nodePtr->children[index] = new TrieNode;
       }
       nodePtr = nodePtr->children[index];
   }

   nodePtr->isEndOfWord = true;
}

int max_xor(int a, TrieNode &root){   //call for each element array A as a.
   TrieNode *nodePtr = &root;
   int res = 0;
   for(int i= 31; i>=0; i--){
       bool current_bit = (a &(1<<i));
       // cout<<current_bit<<' ';
       if(current_bit == true){
           if(nodePtr->children[0] != nullptr){    //current bit of result will be 1
               nodePtr = nodePtr->children[0];
               res=res|(1<<i);
           }
           else{                                  //current_bit in result will be 0
```

```
                    nodePtr = nodePtr->children[1];
                }
            }
            else{
                if(nodePtr->children[1]!=nullptr){     //current bit of result will be 1
                    nodePtr = nodePtr->children[1];
                    res = res|(1<<i);
                }
                else{
                    nodePtr = nodePtr->children[0];
                }
            }
        }
        return res;
    }
    TrieNode root;
        for(int i = 0; i<n; i++){
            insert(root, B[i]);
        }

        for(int i = 0; i<4; i++){
            cout<<max_xor(A[i], root)<<endl;
        }
```

TIME COMPLEXITY: for each query to find max xor of A[i] with all elements in B = O(bits in integer format). Time for creating trie = O(N*bits in integer format).

AUTO COMPLETE: We are given a Trie with a set of strings stored in it. Now the user types in a prefix of his search query, we need to give him all recommendations to auto-complete his query based on the strings stored in the Trie.
For example if the Trie store {“abc”, “abcd”, “aa”, “abbbaba”} and the User types in “ab” then he must be shown {“abc”, “abcd”, “abbbaba”}. **Method**: suppose user types prefix “ab” then we search for “ab” in trie and store last trie node(nodePtr) for this prefix(“ab”). Now with **nodePtr** as root print all the strings stored in trie below this node.

```
    void auto_complete_util(TrieNode *nodePtr, string &s, int level){
        if(nodePtr->isEndOfWord == true){
            cout<<s.substr(0, level)<<endl;
        }

        for(int i = 0; i<26; i++){
            if(nodePtr->children[i] == nullptr) continue;
            s[level] = 'a' + i;
            auto_complete_util(nodePtr->children[i], s, level + 1);
        }
    }

    void auto_complete(TrieNode &root, string &s){
        TrieNode *currPtr = search(root, s);
        if(currPtr == nullptr){
            cout<<"prefix not found\n";
            return;
        }
        int level = s.size();
        s.resize(s.size() + 20, '\0');
        auto_complete_util(currPtr, s, level);
    }
```

TIME COMPLEXITY: O(N*L*MAX_CHAR)

DISPLAY CONTENT: Print all strings stored in trie. Method is similar to auto complete, instead of searching for prefix we start from root and print all string below it, recursively. String is printed when we find a node where **isEndOfWord** is true.

TIME COMPLEXITY: O(N*L*MAX_CHAR)

LONGEST COMMON PREFIX: Given a set of strings, find the longest common prefix.

```
Input  : {“geeksforgeeks”, “geeks”, “geek”, “geezer”}
Output : "gee"
Input  : {"apple", "ape", "april"}
Output : "ap"
```

First insert all strings in trie and for each node store the number of children pointer which are not pointing to null(**numChildren**). After inserting traverse the string and stopr when you find a node having **numChildren>1**. Num of nodes(including current node) between current node and root is required answer.

TIME COMPLEXITY: For Creating Trie: O(N*L). For finding longest common prefix after trie is created: O(L). where L is maximum length of string.

WORD COUNT: A Trie is used to store dictionary words so that they can be searched efficiently and prefix search can be done. The task is to write a function to count the number of words.
Method is to traverse through each node of trie recursively and check if isEndOfWord is true. Answer is number of nodes in trie for which idEndOfWord is true.

```
                int word_count(TrieNode *nodePtr){
                    int cnt = 0;
                    if(nodePtr->isEndOfWord == true){
                         cnt = 1;
                    }
                    for(int i = 0; i<26; i++){
                        if(nodePtr->children[i] == nullptr) continue;
                        cnt+=word_count(nodePtr->children[i]);
                    }
                    return cnt;
                }
```
TIME COMPLEXITY: O(L*N*MAX_CHAR)

WORD FREQUENCY: Idea is to store another variable **occurrence** for each node. Whenever we insert a string in trie, then the occurrence value of last node(node which is marked as End of Word) is incremented. Now to find occurence of a string in trie, we just traverse trie for corresponding string and return value of last node's occurrence value. If string is not present in trie return 0.

```
void insert(TrieNode &root, string s){
    TrieNode *nodePtr = &root;
    for(int i = 0; i<s.size(); i++){
        int index = s[i] - 'a';
        if(nodePtr->children[index] == nullptr){
            nodePtr->children[index] = new TrieNode;
        }

        nodePtr = nodePtr->children[index];
    }

    nodePtr->isEndOfWord = true;
    nodePtr->occurrence+=1;
}
int word_freuency(TrieNode &root, string &s){
    TrieNode *nodePtr = &root;

    for(int i = 0; i<s.size(); i++){
        int index = s[i] - 'a';
        if(nodePtr->children[index] == nullptr){
            return 0;
        }
        nodePtr = nodePtr->children[index];
    }

    return nodePtr->occurrence;
}
```
TIME COMPLEXITY: For Creating Trie: O(N*L). For finding frequency of a given string:O(L) where L is length of string.

MOST FREQUENT WORD: Process is same as word frequency problem. We keep on updating occurrence value while inserting a string in trie and simultaneously check if frequency is greater than current max_frequency.

COUNT DISTINCT SUBSTRING: Count of distinct substrings of a string using Suffix Trie.
Given a string of length n of lowercase alphabet characters, we need to count total number of distinct substrings of this string. Insert all suffix of word in a trie. Now traverse through recursively and print all strings(not only inserted one), here all string means, print path from root to current node(current node need not to be endOfWord).

```
void print_substring(TrieNode *nodePtr, string &s, int level){

    if(level> 0 ){

        cout<<s.substr(0, level)<<endl;

    }

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

        if(nodePtr->children[i] == nullptr) continue;

        s[level] = i + 'a';

        print_substring(nodePtr->children[i], s, level+1);

    }

}
```

TIME COMPLEXITY: Let our input string is str and it has length of L, For Creating Trie: We insert L strings,Hence Time = O(L*L). For Printing substring:O(L*L*MAX_CHAR).

PATTERN SEARCHING: Problem Statement: Given a text txt[0..n-1] and a pattern pat[0..m-1], write a function search(char pat[], char txt[]) that prints all occurrences of pat[] in txt[]. You may assume that n > m. Since we have to print all occurrences of pat[] in txt[], for each trie node we store a vector which stores starting index of suffix being inserted.

```
void insert(TrieNode &root, string s, int suffix_index){  //Insert Suffix
    TrieNode *nodePtr = &root;

    for(int i = 0; i<s.size(); i++){
        int index = s[i] - 'a';
        if(nodePtr->children[index] == nullptr){
            nodePtr->children[index] = new TrieNode;
        }
        nodePtr = nodePtr->children[index];
        nodePtr->index.push_back(suffix_index);
    }

    nodePtr->isEndOfWord = true;
}

TrieNode *pattern_search(TrieNode *nodePtr, string &s){
    for(int i = 0; i<s.size(); i++){
        int index = s[i] - 'a';
        if(nodePtr->children[index] == nullptr) return nullptr;
        nodePtr = nodePtr->children[index];
    }
    return nodePtr;
}

cin>>s;
 TrieNode *tempPtr = pattern_search(&root, s);
```

```
            if(tempPtr == nullptr){
                cout<<"Pattern Not Found\n";
            }
            else{
                for(int i = 0; i<tempPtr->index.size(); i++){
                    cout<<"Pattern found at index "<<tempPtr->index[i]<<endl;
                }
            }
```

TIME COMPLEXITY: O(L + k) where L is length of pattern and k is number of occurrences of pattern in txt.

DUPLICATE ROWS: Given a binary matrix whose elements are only 0 and 1, we need to print the rows which are duplicate of rows which are already present in the matrix. Method is to treat every row as a string and insert them in a trie. Now while inserting a row if that row is already present in trie, we have found a duplicate.

```
        bool insert(TrieNode &root, vector<int> v){  //Here v is a row.
                TrieNode *nodePtr = &root;

                for(int i = 0; i<v.size(); i++){
                    int index = v[i];
                    if(nodePtr->children[index] == nullptr){
                        nodePtr->children[index] = new TrieNode;
                    }

                    nodePtr = nodePtr->children[index];
                 }

                 bool status = nodePtr->isEndOfWord;

                 nodePtr->isEndOfWord = true;

                 return status;     //status:true if duplicate found
        }
```

TIME COMPLEXITY: O(no of rows * no of cols).
LAST UNIQUE URL: Given a very long list of URLs, find out last unique URL. Only one traversal of all URLs is allowed. We store urls in a trie and also in a doubly linked list of strings. We store a pointer from a leaf node(leaf node means node which is marked as end of word for a url) of a url to the node in doubly linked list which(doubly linked list node) also store the url. If we are inserting url first time we insert it in trie and also in doubly linked list then store a pointer from leaf node for current url to node in linked list. If while inserting a url we find that it's already inserted then we delete url from linked list(in O(1)) using pointer from leaf node to linked list and mark the pointer(which in pointing to node in linked list) null. In the last,doubly linked list has urls which are unique and hence print the last inserted url in linked list(list.back()). See the code in **Documents->competitive->Algorithm->Trie->last_unique_url.cpp** in computer folder.
TIME COMPLEXITY: O(N*L),where L in maximum length of url and N is number of url.