TRIE:

1. Implement a Phone Directory

|  |
| --- |
| // Java Program to Implement a Phone  // Directory Using Trie Data Structure  import java.util.\*;    class TrieNode  {  // Each Trie Node contains a Map 'child'  // where each alphabet points to a Trie  // Node.  HashMap<Character,TrieNode> child;    // 'isLast' is true if the node represents  // end of a contact  boolean isLast;    // Default Constructor  public TrieNode()  {  child = new HashMap<Character,TrieNode>();    // Initialize all the Trie nodes with NULL  for (char i = 'a'; i <= 'z'; i++)  child.put(i,null);    isLast = false;  }  }    class Trie  {  TrieNode root;    // Insert all the Contacts into the Trie  public void insertIntoTrie(String contacts[])  {  root = new TrieNode();  int n = contacts.length;  for (int i = 0; i < n; i++)  {  insert(contacts[i]);  }  }    // Insert a Contact into the Trie  public void insert(String s)  {  int len = s.length();    // 'itr' is used to iterate the Trie Nodes  TrieNode itr = root;  for (int i = 0; i < len; i++)  {  // Check if the s[i] is already present in  // Trie  TrieNode nextNode = itr.child.get(s.charAt(i));  if (nextNode == null)  {  // If not found then create a new TrieNode  nextNode = new TrieNode();    // Insert into the HashMap  itr.child.put(s.charAt(i),nextNode);  }    // Move the iterator('itr') ,to point to next  // Trie Node  itr = nextNode;    // If its the last character of the string 's'  // then mark 'isLast' as true  if (i == len - 1)  itr.isLast = true;  }  }    // This function simply displays all dictionary words  // going through current node. String 'prefix'  // represents string corresponding to the path from  // root to curNode.  public void displayContactsUtil(TrieNode curNode,  String prefix)  {    // Check if the string 'prefix' ends at this Node  // If yes then display the string found so far  if (curNode.isLast)  System.out.println(prefix);    // Find all the adjacent Nodes to the current  // Node and then call the function recursively  // This is similar to performing DFS on a graph  for (char i = 'a'; i <= 'z'; i++)  {  TrieNode nextNode = curNode.child.get(i);  if (nextNode != null)  {  displayContactsUtil(nextNode, prefix + i);  }  }  }    // Display suggestions after every character enter by  // the user for a given string 'str'  void displayContacts(String str)  {  TrieNode prevNode = root;    // 'flag' denotes whether the string entered  // so far is present in the Contact List    String prefix = "";  int len = str.length();    // Display the contact List for string formed  // after entering every character  int i;  for (i = 0; i < len; i++)  {  // 'str' stores the string entered so far  prefix += str.charAt(i);    // Get the last character entered  char lastChar = prefix.charAt(i);    // Find the Node corresponding to the last  // character of 'str' which is pointed by  // prevNode of the Trie  TrieNode curNode = prevNode.child.get(lastChar);    // If nothing found, then break the loop as  // no more prefixes are going to be present.  if (curNode == null)  {  System.out.println("\nNo Results Found for \""  + prefix + "\"");  i++;  break;  }    // If present in trie then display all  // the contacts with given prefix.  System.out.println("\nSuggestions based on \""  + prefix + "\" are");  displayContactsUtil(curNode, prefix);    // Change prevNode for next prefix  prevNode = curNode;  }    for ( ; i < len; i++)  {  prefix += str.charAt(i);  System.out.println("\nNo Results Found for \""  + prefix + "\"");  }  }  }    // Driver code  class Main  {  public static void main(String args[])  {  Trie trie = new Trie();    String contacts [] = {"gforgeeks", "geeksquiz"};    trie.insertIntoTrie(contacts);    String query = "gekk";    // Note that the user will enter 'g' then 'e' so  // first display all the strings with prefix as 'g'  // and then all the strings with prefix as 'ge'  trie.displayContacts(query);  }  }  Run on IDE  Output:  Suggestions based on "g" are  geeksquiz  gforgeeks  Suggestions based on "ge" are  geeksquiz  No Results Found for "gek"  No Results Found for "gekk" |

1. Auto-complete feature using 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”}.
2. Word formation using concatenation of two dictionary words:

|  |
| --- |
| Given a dictionary find out if given word can be made by two words in the dictionary.  Note: Words in the dictionary must be unique and the word to be formed should not be a repetition of same words that are present in the Trie.  Examples:  Input : dictionary[] = {"news", "abcd", "tree",  "geeks", "paper"}  word = "newspaper"  Output : Yes  We can form "newspaper" using "news" and "paper"  Input : dictionary[] = {"geeks", "code", "xyz",  "forgeeks", "paper"}  word = "geeksforgeeks"  Output : Yes  Input : dictionary[] = {"geek", "code", "xyz",  "forgeeks", "paper"}  word = "geeksforgeeks"  Output : No |

1. Pattern Search using all suffix

|  |
| --- |
| import java.util.LinkedList;  import java.util.List;  class SuffixTrieNode {    final static int MAX\_CHAR = 256;    SuffixTrieNode[] children = new SuffixTrieNode[MAX\_CHAR];  List<Integer> indexes;    SuffixTrieNode() // Constructor  {  // Create an empty linked list for indexes of  // suffixes starting from this node  indexes = new LinkedList<Integer>();    // Initialize all child pointers as NULL  for (int i = 0; i < MAX\_CHAR; i++)  children[i] = null;  }    // A recursive function to insert a suffix of  // the text in subtree rooted with this node  void insertSuffix(String s, int index) {    // Store index in linked list  indexes.add(index);    // If string has more characters  if (s.length() > 0) {    // Find the first character  char cIndex = s.charAt(0);    // If there is no edge for this character,  // add a new edge  if (children[cIndex] == null)  children[cIndex] = new SuffixTrieNode();    // Recur for next suffix  children[cIndex].insertSuffix(s.substring(1),  index + 1);  }  }    // A function to search a pattern in subtree rooted  // with this node.The function returns pointer to a  // linked list containing all indexes where pattern  // is present. The returned indexes are indexes of  // last characters of matched text.  List<Integer> search(String s) {    // If all characters of pattern have been  // processed,  if (s.length() == 0)  return indexes;    // if there is an edge from the current node of  // suffix tree, follow the edge.  if (children[s.charAt(0)] != null)  return (children[s.charAt(0)]).search(s.substring(1));    // If there is no edge, pattern doesnt exist in  // text  else  return null;  }  }    // A Trie of all suffixes  class Suffix\_tree{    SuffixTrieNode root = new SuffixTrieNode();    // Constructor (Builds a trie of suffies of the  // given text)  Suffix\_tree(String txt) {    // Consider all suffixes of given string and  // insert them into the Suffix Trie using  // recursive function insertSuffix() in  // SuffixTrieNode class  for (int i = 0; i < txt.length(); i++)  root.insertSuffix(txt.substring(i), i);  }    /\* Prints all occurrences of pat in the Suffix Trie S  (built for text) \*/  void search\_tree(String pat) {    // Let us call recursive search function for  // root of Trie.  // We get a list of all indexes (where pat is  // present in text) in variable 'result'  List<Integer> result = root.search(pat);    // Check if the list of indexes is empty or not  if (result == null)  System.out.println("Pattern not found");  else {    int patLen = pat.length();    for (Integer i : result)  System.out.println("Pattern found at position " +  (i - patLen));  }  }    // driver program to test above functions  public static void main(String args[]) {    // Let us build a suffix trie for text  // "geeksforgeeks.org"  String txt = "geeksforgeeks.org";  Suffix\_tree S = new Suffix\_tree(txt);    System.out.println("Search for 'ee'");  S.search\_tree("ee");    System.out.println("\nSearch for 'geek'");  S.search\_tree("geek");    System.out.println("\nSearch for 'quiz'");  S.search\_tree("quiz");    System.out.println("\nSearch for 'forgeeks'");  S.search\_tree("forgeeks");  }  }  Output:  Search for 'ee'  Pattern found at position 1  Pattern found at position 9  Search for 'geek'  Pattern found at position 0  Pattern found at position 8  Search for 'quiz'  Pattern not found  Search for 'forgeeks'  Pattern found at position 5 |

1. Print unique rows in a given Boolean matrix

|  |
| --- |
| Input:  {0, 1, 0, 0, 1}  {1, 0, 1, 1, 0}  {0, 1, 0, 0, 1}  {1, 1, 1, 0, 0}  Output:  0 1 0 0 1  1 0 1 1 0  1 1 1 0 0  //Given a binary matrix of M X N of integers, you need to return only unique rows of binary array  #include <stdio.h>  #include <stdlib.h>  #include <stdbool.h>    #define ROW 4  #define COL 5    // A Trie node  typedef struct Node  {      bool isEndOfCol;      struct Node \*child[2]; // Only two children needed for 0 and 1  } Node;      // A utility function to allocate memory for a new Trie node  Node\* newNode()  {      Node\* temp = (Node \*)malloc( sizeof( Node ) );      temp->isEndOfCol = 0;      temp->child[0] = temp->child[1] = NULL;      return temp;  }    // Inserts a new matrix row to Trie.  If row is already  // present, then returns 0, otherwise insets the row and  // return 1  bool insert( Node\*\* root, int (\*M)[COL], int row, int col )  {      // base case      if ( \*root == NULL )          \*root = newNode();        // Recur if there are more entries in this row      if ( col < COL )          return insert ( &( (\*root)->child[ M[row][col] ] ), M, row, col+1 );        else // If all entries of this row are processed      {          // unique row found, return 1          if ( !( (\*root)->isEndOfCol ) )              return (\*root)->isEndOfCol = 1;            // duplicate row found, return 0          return 0;      }  }    // A utility function to print a row  void printRow( int (\*M)[COL], int row )  {      int i;      for( i = 0; i < COL; ++i )          printf( "%d ", M[row][i] );      printf("\n");  }    // The main function that prints all unique rows in a  // given matrix.  void findUniqueRows( int (\*M)[COL] )  {      Node\* root = NULL; // create an empty Trie      int i;        // Iterate through all rows      for ( i = 0; i < ROW; ++i )          // insert row to TRIE          if ( insert(&root, M, i, 0) )              // unique row found, print it              printRow( M, i );  }    // Driver program to test above functions  int main()  {      int M[ROW][COL] = {{0, 1, 0, 0, 1},          {1, 0, 1, 1, 0},          {0, 1, 0, 0, 1},          {1, 0, 1, 0, 0}      };        findUniqueRows( M );        return 0;  } |

1. Palindrome pair in an array of words (or strings)

|  |
| --- |
| Given a list of words, find if any of the two words can be joined to form a palindrome.  Examples:  Input : list[] = {"geekf", "geeks", "or",  "keeg", "abc", "bc"}  Output : Yes  There is a pair "geekf" and "keeg"  Input : list[] = {"abc", "xyxcba", "geekst", "or",  "keeg", "bc"}  Output : Yes  There is a pair "abc" and "xyxcba"  Approach:  1) Create an empty Trie.  2) Do following for every word:-  a) Insert reverse of current word.  b) Also store up to which index it is  a palindrome.  3) Traverse list of words again and do following  for every word.  a) If it is available in Trie then return true  b) If it is partially available  Check the remaining word is palindrome or not  If yes then return true that means a pair  forms a palindrome.  Note: Position upto which the word is palindrome  is stored because of these type of cases.  //Java program to check if there is a pair that  //of above method using Trie  import java.util.ArrayList;  import java.util.Arrays;  import java.util.List;    public class Palin\_pair2 {    // Alphabet size (# of symbols)  static final int ALPHABET\_SIZE = 26;    // Trie node  static class TrieNode {  TrieNode[] children = new TrieNode[ALPHABET\_SIZE];  List<Integer> pos; // To store palindromic  // positions in str  int id;    // isLeaf is true if the node represents  // end of a word  boolean isLeaf;    // constructor  public TrieNode() {  isLeaf = false;  pos = new ArrayList<>();  for (int i = 0; i < ALPHABET\_SIZE; i++)  children[i] = null;  }  }    // Utility function to check if a string is a  // palindrome  static boolean isPalindrome(String str, int i, int len) {  // compare each character from starting  // with its corresponding character from last  while (i < len) {  if (str.charAt(i) != str.charAt(len))  return false;    i++;  len--;  }  return true;  }    // If not present, inserts reverse of key into Trie. If  // the key is prefix of a Trie node, just mark leaf node  static void insert(TrieNode root, String key, int id) {  TrieNode pCrawl = root;    // Start traversing word from the last  for (int level = key.length() - 1; level >= 0; level--) {  // If it is not available in Trie, then  // store it  int index = key.charAt(level) - 'a';  if (pCrawl.children[index] == null)  pCrawl.children[index] = new TrieNode();    // If current word is palindrome till this  // level, store index of current word.  if (isPalindrome(key, 0, level))  (pCrawl.pos).add(id);    pCrawl = pCrawl.children[index];  }  pCrawl.id = id;  pCrawl.pos.add(id);    // mark last node as leaf  pCrawl.isLeaf = true;  }    // list to store result  static List<List<Integer>> result;    // Returns true if key presents in Trie, else false  static void search(TrieNode root, String key, int id) {  TrieNode pCrawl = root;  for (int level = 0; level < key.length(); level++) {  int index = key.charAt(level) - 'a';    // If it is present also check upto which index  // it is palindrome  if (pCrawl.id >= 0 && pCrawl.id != id  && isPalindrome(key, level, key.length() - 1)) {  List<Integer> l = new ArrayList<>();  l.add(id);  l.add(pCrawl.id);  result.add(l);  }    // If not present then return  if (pCrawl.children[index] == null)  return;    pCrawl = pCrawl.children[index];  }    for (int i : pCrawl.pos) {  if (i == id)  continue;  List<Integer> l = new ArrayList<>();  l.add(id);  l.add(i);  result.add(l);  }  }    // Function to check if a palindrome pair exists  static boolean checkPalindromePair(List<String> vect) {    // Construct trie  TrieNode root = new TrieNode();  for (int i = 0; i < vect.size(); i++)  insert(root, vect.get(i), i);    // Search for different keys  result = new ArrayList<>();  for (int i = 0; i < vect.size(); i++) {  search(root, vect.get(i), i);    if (result.size() > 0)  return true;  }    return false;  }    // Driver code  public static void main(String args[]) {  List<String> vect = Arrays.asList("geekf", "geeks",  "or", "keeg", "abc", "bc");    if (checkPalindromePair(vect) == true)  System.out.println("Yes");  else  System.out.println("No");  }  } |