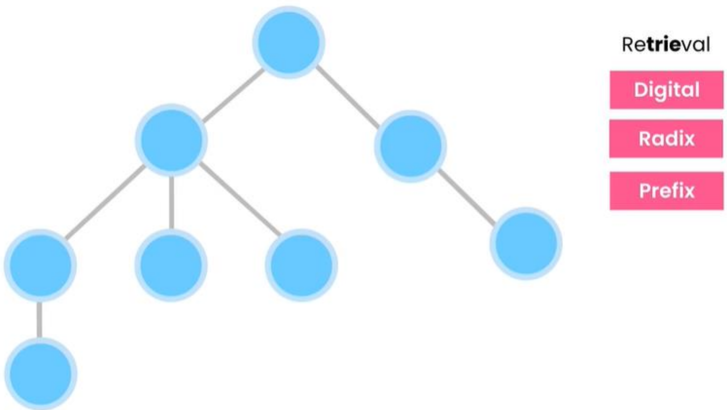
Tries

V-(1+2)-Intro:-

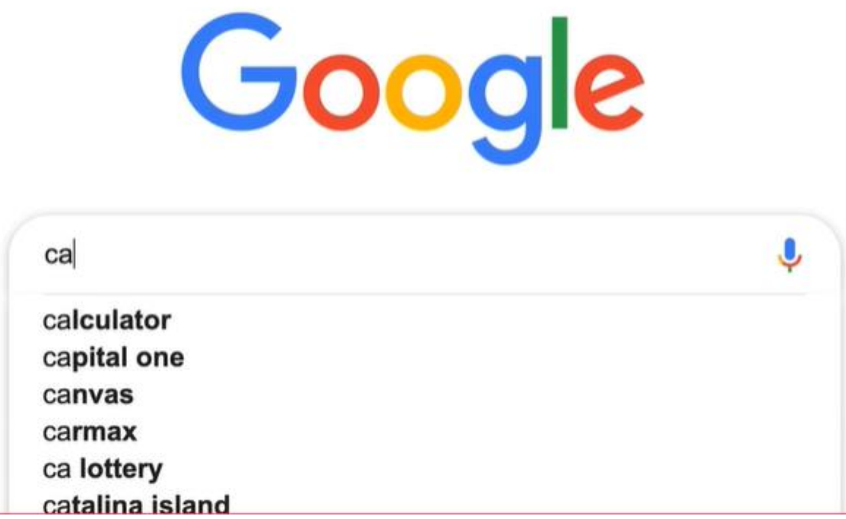
Tries are one of the data structures that most of the universities around the world doesn't include this data structure in their courses. But it is often asked by the interviewer in the technical interview. It's easy to implement.

What are Tries?

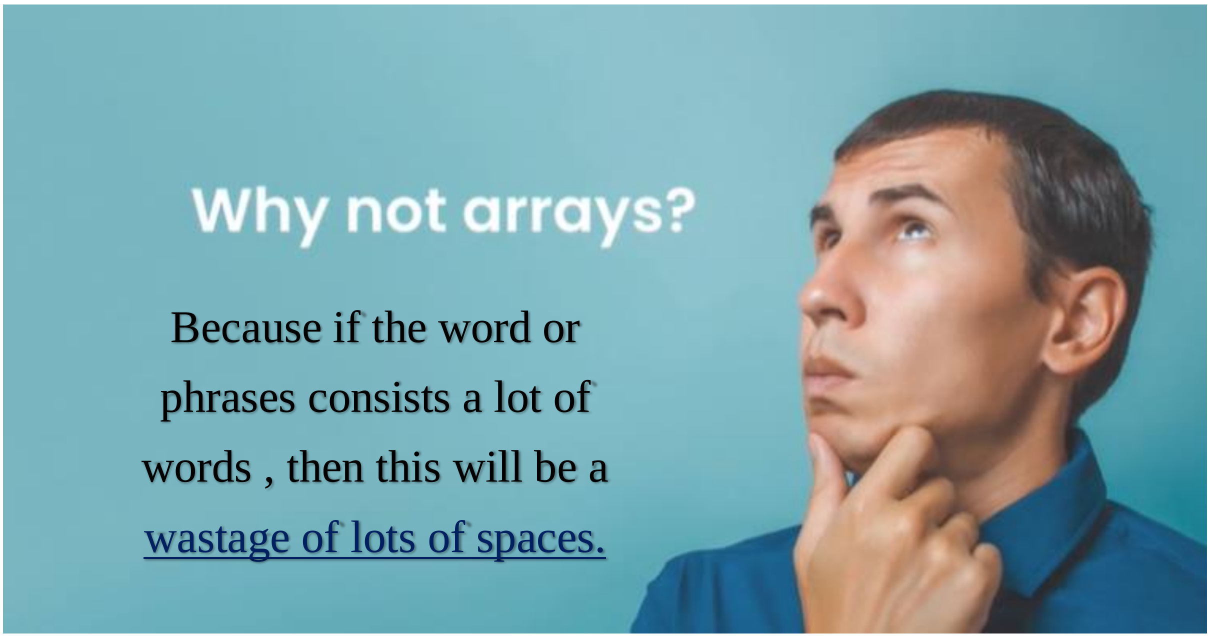
=> Tries are one kind of tree where each child has several nodes that means they are not binary tree. This name came from the word retrieval. Some call it tree only. It has also some other name such as Digital, Radix, Prefix.



This data structure is worked for auto completion. If we search something on google , it will suggest some auto completion words just like the picture below:-



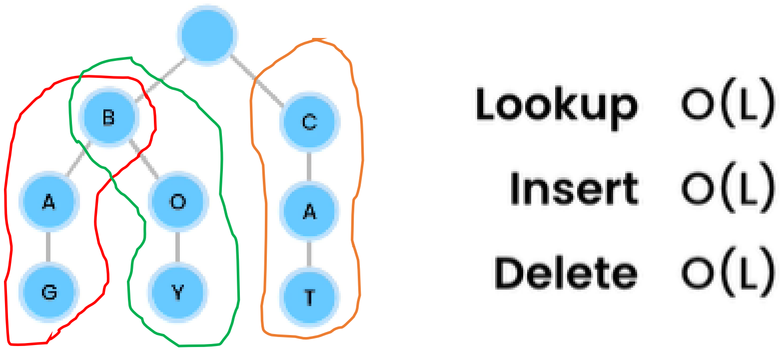
Then there is a question arise like below:-



Take a look at the picture below to understand more:-

[**pic**k,
picky,
pickle,
picture,
picnic]

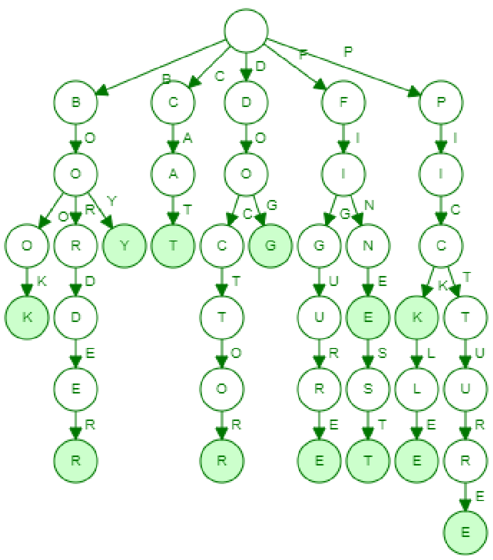
All the words started with “pic” prefix. Now, if we want the word picnic we will have to iterate the whole array then we can find “picnic” which is **so slow.** *We can optimize this but it will not be so good as we* ***optimized with Tries*****.** Tries help us to manage millions of space wisely and give us a chance for **super-fast lookup**.



Now wisely look at the picture above. Can you find **three words?** If you don't find then it's okay. Look at the red marked area the word is **Bag**. Afterwards, the green marked area which is **Boy**. Finally, the brown marked area **Cat**. Now if we wisely think about the words **Bag** & **Boy,** we see both the words **share a** same prefix. Now, suppose we want to **generate a word** from that tree & that is **baggage**. For that, we just have to *extend red marked area* and add **g, a, g, e**. This is how this data structure saves a lot of spaces because we don't have to duplicate the word for same prefix. Now we work with English language that's why we can have at most **26 children for each node (depends)** because English Language **has 26 alphabets.** If we works with **Bangla Language,** then each node can **have 50 children**. One important thing that root node always have **empty character or null.** We **can't have 26 roots** in our tree. We can **only have one root.** We can indicate the words with prefix (beginning can be different or same). Time complexity for searching a word(look up) will be O(L) where L is the length of the word. Insert operation will be also O(L). Removing a word also has a complexity of O(L).

V-(3)-Populating a Tire:-

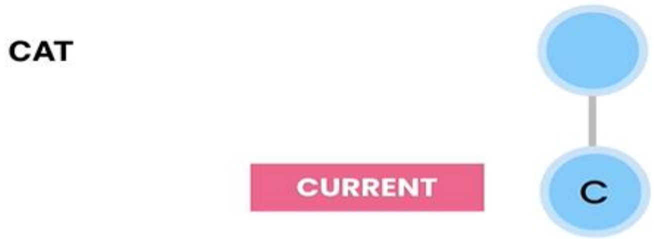
boy, book, border, cat, dog, doctor, fine, finest, figure, pick, pickle, picture

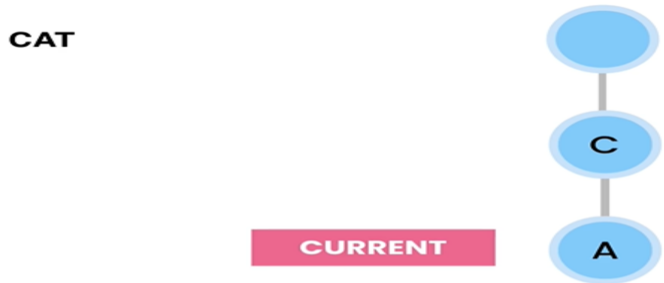


https://www.cs.usfca.edu/~galles/visualization/Trie.html

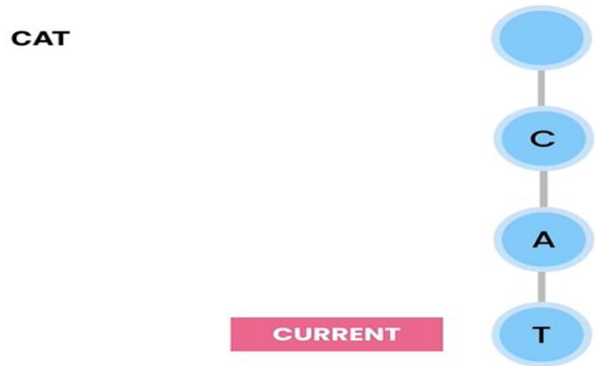
V-(4+5)-Building a Tire:-

First check if there is any child exist for C if not then create a child for C and set current to C.





Then ask if there is any child exist for A. if not then create a child for A and set current to A.



Then ask if there is any child exist for T. if not then create a child for T and set current to T.

If further we have to create tree for CAN. Then we just need to append N after A. Because C and A is already created.

```
Trie
    Node (Private Class)
        value: char
        children: Node[26]
        isEndOfWord: boolean
insert(word: String)
index = ch - 'a'

100 - 97 = 3 (d)
```

Node [26] is an array of characters [a-z] and insert method help us to calculate which index should we insert which character. We can use both hashmap and array for implementing this problem.

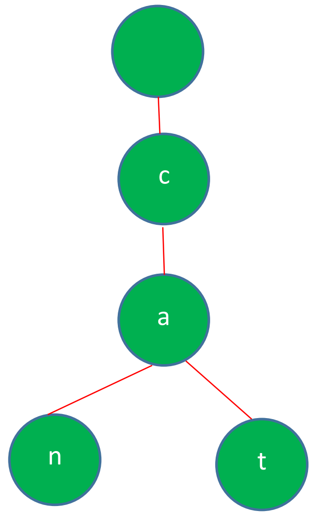
```java
public class Trie {
    public static int Alphabet_Size=26;

    private class Node{
        private char value;
        private Node[] children =  new Node[Alphabet_Size];
        private boolean isEndofWord;

        public Node(char value) {
            this.value = value;
        }

        @Override
        public String toString() {
            return "value=" + value;
        }
    }
    private Node root = new Node(' ');
    public void insert(String word){
        var current = root;
        for(var ch:word.toCharArray()){
            var index = ch-'a';
            if(current.children[index]==null)
                current.children[index]= new Node(ch);
            current = current.children[index];
        }
        current.isEndofWord=true;
    }
}
```

https://medium.com/javarevisited/do-you-know-nested-and-inner-classes-in-java-cc5647f46e07

(This link above has a clear descriptive idea about inner and outer class.)

```
Variables
  p args = {String[0]@793} []
v ≡ t1 = {Trie@794}
  v f root = {Trie$Node@795} "value= "
      f value = ' ' 32
    v f children = {Trie$Node[26]@823}
        Not showing null elements
      v ≡ 2 = {Trie$Node@825} "value=c"
          f value = 'c' 99
        v f children = {Trie$Node[26]@827}
            Not showing null elements
          v ≡ 0 = {Trie$Node@828} "value=a"
              f value = 'a' 97
            v f children = {Trie$Node[26]@830}
                Not showing null elements
              > ≡ 13 = {Trie$Node@834} "value=n"
              > ≡ 19 = {Trie$Node@835} "value=t"
              f isEndofWord = false
            > f this$0 = {Trie@794}
          f isEndofWord = false
        v f this$0 = {Trie@794}
          v f root = {Trie$Node@795} "value= "
              f value = ' ' 32
            > f children = {Trie$Node[26]@823}
              f isEndofWord = false
            v f this$0 = {Trie@794}
              v f root = {Trie$Node@795} "value= "
```

<u>Here "c" has one child which is "a". After that, "a" has two childs and they are "n" and "t".</u>



V-(6)-An implementation with HashTable:-

```
import java.util.HashMap;

public class Trie {
    public static int Alphabet_Size=26;

    private class Node{
        private char value;
        private HashMap<Character,Node> children =  new HashMap<>();
        private boolean isEndofWord;

        public Node(char value) {
            this.value = value;
        }

        @Override
        public String toString() {
            return "value=" + value;
        }
    }
    private Node root = new Node(' ');
    public void insert(String word){
        var current = root;
        for(var ch:word.toCharArray()){
           // var index = ch-'a';
            //If we use hashmap we don't need index cause we use character for
key.
            if(current.children.get(ch)==null)
                current.children.put(ch,new Node(ch));
            current = current.children.get(ch);
        }
        current.isEndofWord=true;
    }
}
```

V-(7)-A Better Abstraction:-

Abstraction principle says we shouldn't expose our inner working class to outer class. Now, let's see where we broke abstraction principle.

```
public void insert(String word){
    var current = root;
    for(var ch:word.toCharArray()){
       // var index = ch-'a';
        //If we use hashmap we don't need index cause we use character
for key.
        if(current.children.get(ch)==null)//in this line we break the
abstraction principle cause
            //we work directly with children which is in Node
class(inner class) but in Trie(outer class) has directly
            //manipulated Node classes working here. So, we have to set
methods inside inner classes for the operation
            current.children.put(ch,new Node(ch));
        current = current.children.get(ch);
    }
    current.isEndofWord=true;
}
```

Now, if we use abstraction principle properly, then the code will look like below:-

```java
import java.util.HashMap;

public class Trie {
    public static int Alphabet_Size=26;

    private class Node{
        private char value;
        private HashMap<Character,Node> children =  new HashMap<>();
        private boolean isEndofWord;

        public Node(char value) {
            this.value = value;
        }

        @Override
        public String toString() {
            return "value=" + value;
        }
        //for better abstractions we need
        public boolean hasChild(char ch){
            return children.containsKey(ch);
        }
        public void addChild(char ch){
            children.put(ch,new Node(ch));
        }
        public Node getChild(char ch){
            return children.get(ch);
        }
    }
    private Node root = new Node(' ');
    public void insert(String word){
        var current = root;//use of inner class
        for(var ch:word.toCharArray()){
 // var index = ch-'a';
 //If we use hashmap we don't need index cause we use character for key.
            if(!current.hasChild(ch))
//its like a story now. Do you have any children?
                current.addChild(ch);
//If there is no children, then add children
            current = current.getChild(ch);
//also get the children as updated current
        }
        current.isEndofWord=true;//make it true means finish the words.
    }
}
```

V-(8+9)-Looking up a word:-

```java
public class Main {
    public static void main(String[] names) {
        var trie = new Trie();
        trie.insert( word: "canada");
        trie.contains("can")
    }
}
```

In the picture we see in insert method that we insert "canada". But, take a look at contains method. It is searching for **"can".** That means as we insert "canada" it will find all the nodes **c,a,n** but we don't insert **"can"** as word before. So "n" is not the end of word for "canada" that means ***trie.contains("can")*** method should return **false**. We should keep in mind all that things when implementing contains method.

```
import java.util.HashMap;

public class Trie {
    public static int Alphabet_Size=26;

    private class Node{
        private char value;
        private HashMap<Character,Node> children =  new HashMap<>();
        private boolean isEndofWord;

        public Node(char value) {
            this.value = value;
        }

        @Override
        public String toString() {
            return "value=" + value;
        }
        //for better abstractions we need
        public boolean hasChild(char ch){
            return children.containsKey(ch);
        }
        public void addChild(char ch){
            children.put(ch,new Node(ch));
        }
        public Node getChild(char ch){
            return children.get(ch);
        }
    }
    private Node root = new Node(' ');
    public void insert(String word){
        var current = root;//use of inner class
        for(var ch:word.toCharArray()){
           // var index = ch-'a';
            //If we use hashmap we don't need index cause we use character for key.
            if(!current.hasChild(ch))//its like a story now. Do you have any children?
                current.addChild(ch);//If there is no children, then add children
            current = current.getChild(ch);//also get the children as updated current
        }
        current.isEndofWord=true;//make it true means finish the words.
    }
    public boolean contains(String word){
        if(word==null)//if null input then return false
            return false;
        var current = root;//call root because we need to iterate the trie
        for(var ch:word.toCharArray()){//string to array converted for iteration
            if(!current.hasChild(ch))//if we don't find any particular character we
immediately return false
                return false;
            current = current.getChild(ch);//otherwise set current to next character and
check if it's exist in trie
        }
        return current.isEndofWord;
    }
}
```

V-(10)-Tree Traversal:-

We can do two types of traversals. One is pre-order traversal and another one is post-order traversal. These traversals are not like binary traversals where we have leftchild or rightchild. These are like visiting all children from root which is known as pre-order traversal and another one is visiting all nodes from leaf to root which is post-order traversal. When we need to look up for a word we can use pre-order concept and when we have to delete a word we use post order because we have to delete the nodes starting from leaf.

```
import java.util.HashMap;

public class Trie {
    public static int Alphabet_Size=26;

    private class Node{
        private char value;
        private HashMap<Character,Node> children =  new HashMap<>();
        private boolean isEndofWord;

        public Node(char value) {
            this.value = value;
        }

        @Override
        public String toString() {
            return "value=" + value;
        }
        //for better abstractions we need
        public boolean hasChild(char ch){
            return children.containsKey(ch);
        }
        public void addChild(char ch){
            children.put(ch,new Node(ch));
        }
        public Node getChild(char ch){
            return children.get(ch);
        }
        public Node[] getChildren(){
            return children.values().toArray(new Node[0]);
            //As children.values().toArray() is returning a collection of nodes
            //but we need array to be returned that's why we create a new node
object
            //That will put the result in Node array
        }
    }
    private Node root = new Node(' ');
    public void insert(String word){
        var current = root;//use of inner class
        for(var ch:word.toCharArray()){
           // var index = ch-'a';
            //If we use hashmap we don't need index cause we use character for
key.
            if(!current.hasChild(ch))//its like a story now. Do you have any
children?
                current.addChild(ch);//If there is no children, then add
children
            current = current.getChild(ch);//also get the children as updated
current
        }
        current.isEndofWord=true;//make it true means finish the words.
    }
```

```
    public boolean contains(String word){
        if(word==null)//if null input then return false
            return false;
        var current = root;//call root because we need to iterate the trie
        for(var ch:word.toCharArray()){//string to array converted for
iteration
            if(!current.hasChild(ch))//if we don't find any particular
character we immediately return false
                return false;
            current = current.getChild(ch);//otherwise set current to next
character and check if it's exist in trie
        }
        return current.isEndofWord;
    }
    public void traverse_Pre_order(){
        traverse_Pre_order(root);
    }
    private void traverse_Pre_order(Node root){
        //In pre-order traversal we have to visit root First
        System.out.println(root.value);
        for(var child:root.getChildren())
            traverse_Pre_order(child);
    }
    public void traverse_Post_order(){
        traverse_Post_order(root);
    }
    private void traverse_Post_order(Node root){
        //In post-order traversal we have to visit the leaf first
        for(var child:root.getChildren())
            traverse_Post_order(child);
        System.out.println(root.value);
    }
}
```

V-(11+12)-Removing a Word:-

In Trie Class:-

```
public void remove(String word){
    if(word==null)//in null string , nothing is needed to remove
        return;
    remove(root,word,0); //remove the string
}
private void remove(Node root,String word,int index){
    if(index==word.length()) {//base case
        root.isEndofWord=false;
        //as we don't always physically remove the word
        //we can only remove the word if it has no child
        //in example cat & cattle
        //for removing "cat" when we reach at the first 't' of cattle then we
set
        //the root.isEndofWord = false cause we are not allow to remove
permanently
        //cause we also insert cattle as a word.
        //if we remove c,a,t from the Trie permanently , then if we search for
cattle
        //it will show us that no word such cattle exist in Trie.
        return;
    }
    var ch = word.charAt(index);//get the character
    var child = root.getChild(ch);//get the child for that character
    if(child==null)//if no child found , simply return
        return;
```

```
    remove(child,word,index+1);

    //if no child found & it is the not the end of another word then only
remove the characters
//permanently from the Trie
if(!child.hasChildren() && !child.isEndofWord)
    root.removeChild(ch);
}
```

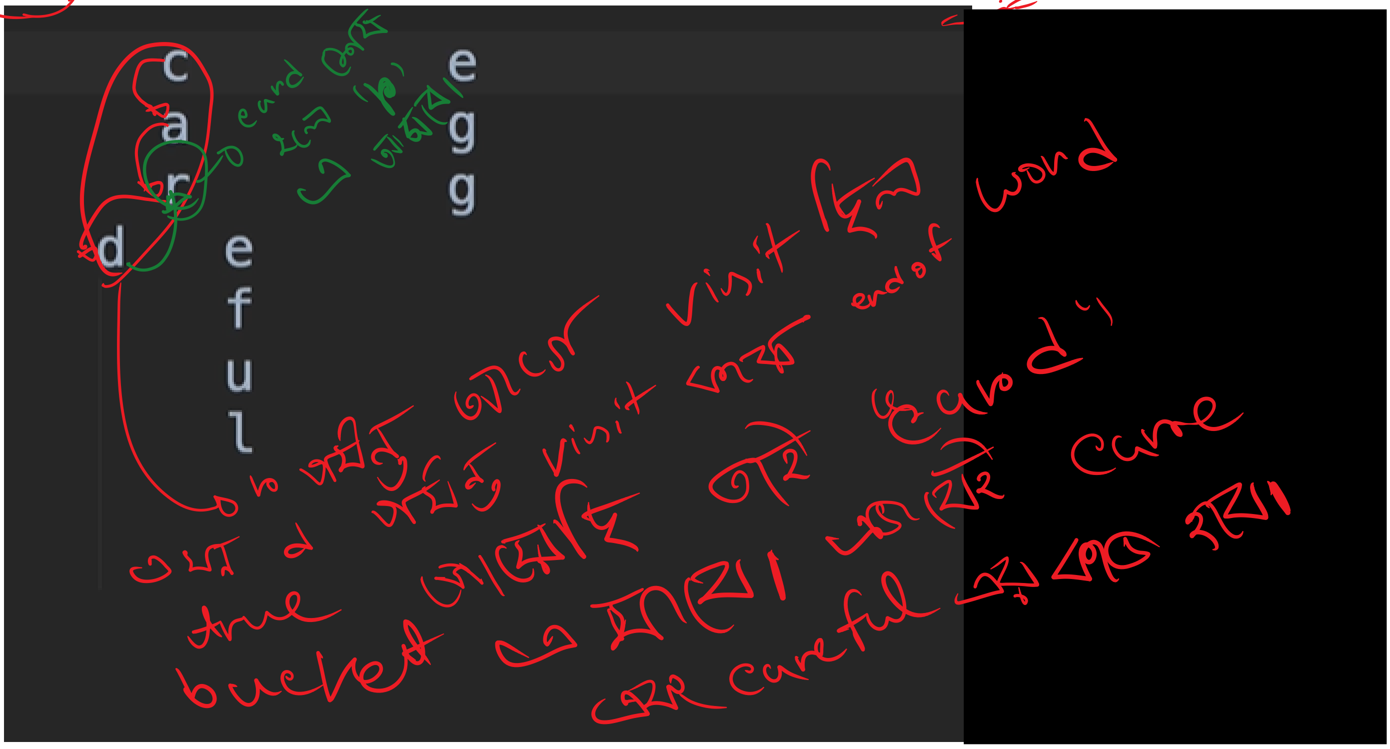
In Node Class:-

```
public boolean hasChild(char ch){
    return children.containsKey(ch);
}
public void addChild(char ch){
    children.put(ch,new Node(ch));
}
public Node getChild(char ch){
    return children.get(ch);
}
public Node[] getChildren(){
    return children.values().toArray(new Node[0]);
    //As children.values().toArray() is returning a collection of nodes
    //but we need array to be returned that's why we create a new node object
    //That will put the result in Node array
}
public boolean hasChildren(){
    return !children.isEmpty();//if children exist , then return false
}
public void removeChild(char ch){
    children.remove(ch);
}
```

V-(13+14)-Auto Completion:-

P.T.O

End of word = true মানে সেটা একটা valid word.



1/ Node root → visiting node
2/ String word → current word
3/ List<String> words → collecting the words

```
public List<String> findWords(String prefix) {
    List<String> words = new ArrayList<>();
    var lastNode = findLastNodeOf(prefix);
    findWords(lastNode, prefix, words);
//after completing this method execution words which is an arrayList
// this arrayList contains the words with same prefix
//this task is called autocompletion
    return words;
}

private void findWords(Node root, String prefix, List<String> words) {
    //This method will add the words in the arraylist called "words"
    //if user input (String prefix) is null then just simply return
    //nothing to find
    if (root == null)
        return;
    //if root.isEndofword  true that means the word is valid and we
have
    //to add the word to arrayList
    //Now the question is why we just add the prefix in the list?
    //The answer is very simple ,prefix is updated everytime it visits
a node
    //and string concatenation happens here until it found that the
node.isEndofWord is true
    //we will help to understand this via a example
    //  c
    //  a
    //  r
```

```
    // d t
    // findWords(Node root, String prefix, List<String> words) method
already
    //have the last node of the prefix which is called outside this
method via
    //different method name findLastNodeOf(String prefix)
    //which means if user pass the prefix car
    //the findlastNode method return 'r' which is obvious the last node
    //of the prefix 'car'
    //now for the very first time when the
    // findWords(Node root, String prefix, List<String> words) method
is called
    //it will have 'r' in the root parameter
    // now its checking for the first time is 'r' .isEndofWord ==true
or not
    //if root.isEndofWord that means r is the end of any word then the
prefix
    //car will be added in the arraylist 'words'
    //then the first ever recursion will called by the findWords method
itself
    //it will get the child for example now we will find the child of
'r'
    //  c
    //  a
    //  r
    // d t
    //see r has two childs d and t
    //first it will do "car"+d and check d isEndofWord = true or not if
yes the 'card'
    //will be added in the word which is an arrayList
    //parallel call happen here it is 'car' for the very first time
    //it works for children d and t parallelly
    //so next recursion generate car+t = cart and check if t
.isEndofWord=ture or not
    //if yes then cart will be added in the arraylist 'word'
    if (root.isEndofWord)
        words.add(prefix);

    for (var child : root.getChildren()){
        findWords(child, prefix + child.value, words);
        //System.out.println(child.value);
    }
```

Exercise:1-

1- Implement the contains method recursively. Compare the iterative and recursive solutions.

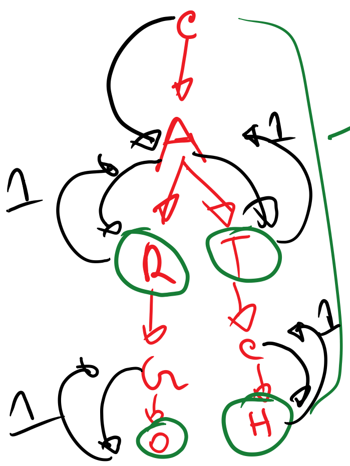
```
public boolean containsRecursive(String word) {
    if (word == null)
        return false;

    return containsRecursive(root, word, 0);
}
```

```
private boolean containsRecursive(Node root, String word, int index) {
    // Base condition
    if (index == word.length())
        return root.isEndofWord;

    if (root == null)
        return false;

    var ch = word.charAt(index);
    var child = root.getChild(ch);
    if (child == null)
        return false;

    return containsRecursive(child, word, index + 1);
}
```

Exercise:2-

2- Count the number of words in a trie.

```
public int countWords() {
    return countWords(root);
}

private int countWords(Node root) {
    var total = 0;//initialize a pointer

    if (root.isEndofWord)
        total++;
    //if any node endofword is true that means it is a complete word

    for (var child : root.getChildren())
        total += countWords(child);
    //1 call total = total + countWords(child)

    return total;
}
```



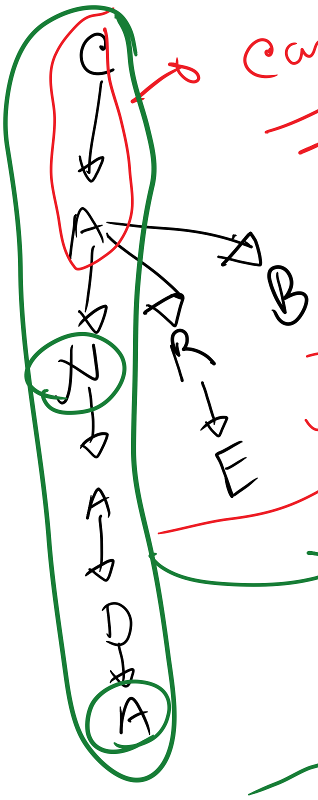
Total 4 words

1+1+1+1=4

parallel calling

Recursion

3- Given an array of strings, find the longest common prefix. Test your algorithm against these test cases.



case-1

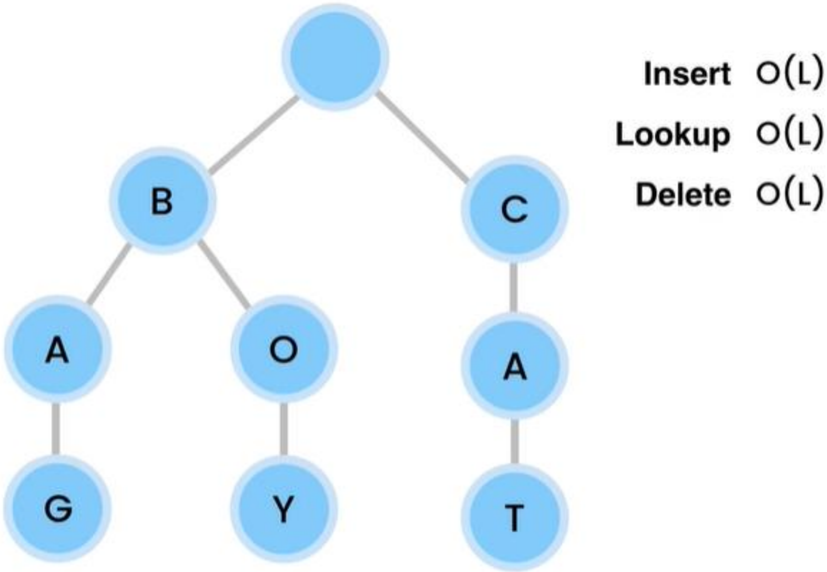
এই Node থেকে child একের বেশি হলে আমরা সেখানে থেমে যাব

case-2 same branch এ একটি word শেষ হলে। এই ক্ষেত্রে একদম শেষ node পর্যন্ত যাবো।

max character ≤ shortest word

```java
public  String longestCommonPrefix(String[] words) {
    if (words == null)
        return "";

    var trie = new Trie();
    for (var word : words)
        trie.insert(word);

    var prefix = new StringBuffer();
    var maxChars = getShortest(words).length();
    var current = trie.root;
    while (prefix.length() < maxChars) {
        var children = current.getChildren();
        if (children.length != 1)
            break;
        current = children[0];
        prefix.append(current.value);
    }

    return prefix.toString();
}

private  String getShortest(String[] words) {
    if (words == null || words.length == 0)
        return "";

    var shortest = words[0];
    for (var i = 1; i < words.length; i++) {
        if (words[i].length() < shortest.length())
            shortest = words[i];
    }

    return shortest;
}
```

Exercise:2-


B
C
A
O
A
G
Y
T
Insert O(L)
Lookup O(L)
Delete O(L)