**Trie: Introduction**

Let’s go over the Trie pattern, its real-world applications, and some problems we can solve with it.

**We'll cover the following**

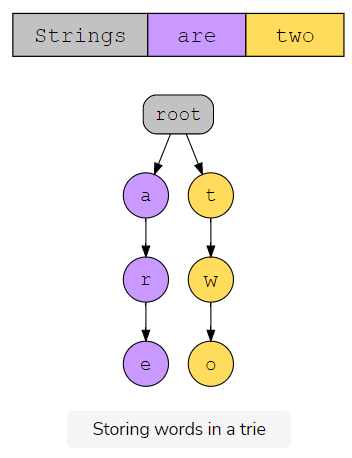
* [Overview](https://www.educative.io/courses/grokking-coding-interview-patterns-java/q2BNGAmMwYD#Overview)
* [Examples](https://www.educative.io/courses/grokking-coding-interview-patterns-java/q2BNGAmMwYD#Examples)
* [Does my problem match this pattern?](https://www.educative.io/courses/grokking-coding-interview-patterns-java/q2BNGAmMwYD#Does-my-problem-match-this-pattern?)
* [Real-world problems](https://www.educative.io/courses/grokking-coding-interview-patterns-java/q2BNGAmMwYD#Real-world-problems)
* [Strategy time!](https://www.educative.io/courses/grokking-coding-interview-patterns-java/q2BNGAmMwYD#Strategy-time!)

**Overview**

**Trie** is a tree data structure used for storing and locating keys from a set. The keys are usually strings that are stored character by character—each node of a trie corresponds to a single character rather than the entire key.

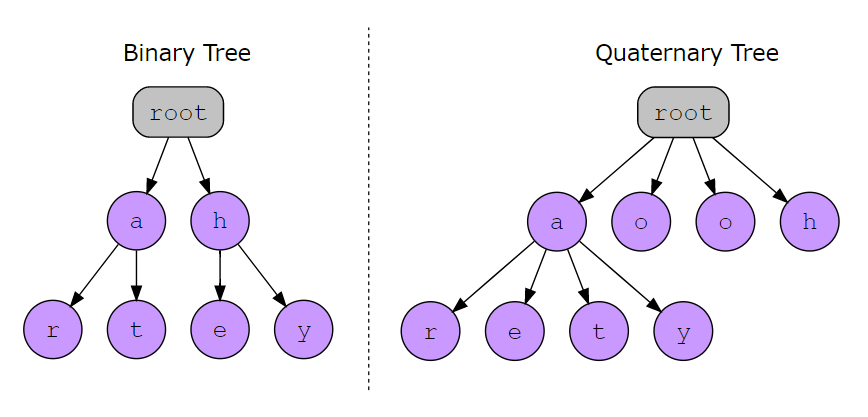
The order of characters in a string is represented by edges between the adjacent nodes. For example, in the string “are”, there will be an edge from node a*a* to node r*r* to node e*e*. That is, node a*a* will be the parent of node r*r*, and node r*r* will be the parent of node e*e*.

The following illustration can help you understand how the strings are stored:



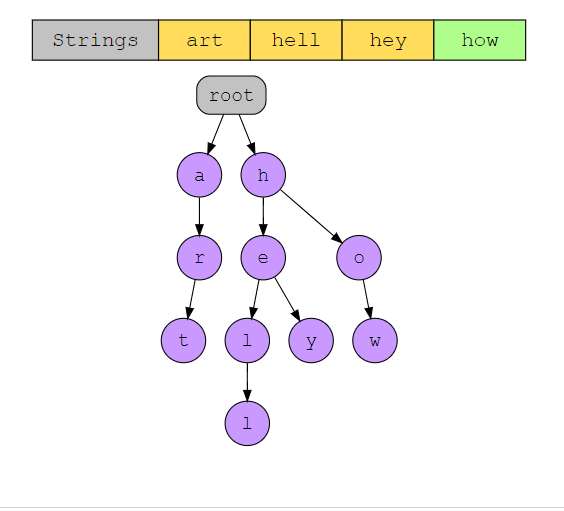
This way, additional space is not required for storing strings with common prefixes. We can keep moving down the tree until a new character, that’s not present in the node’s children, is encountered and add it as a new node. Similarly, searches can also be performed using depth first search by following the edges between the nodes. Essentially, in a trie, words with the same prefix or stem share the memory area that corresponds to the prefix.

To understand how tries are more efficient for storing and searching strings, consider a binary tree. The time complexity of a binary tree is O(\log n)*O*(log*n*), where we talk in terms of \loglog base 22. Instead, think of a quaternary tree, where every node has a fan-out of four, so each node can have four children. The time complexity of this tree is still O(\log n)*O*(log*n*). However, now we’re talking in terms of \loglog with base 44. That’s an improvement in the performance even if it’s by a constant factor. As our trees become wider and shorter, the operations become more efficient. This is because we don’t have to traverse as deep.



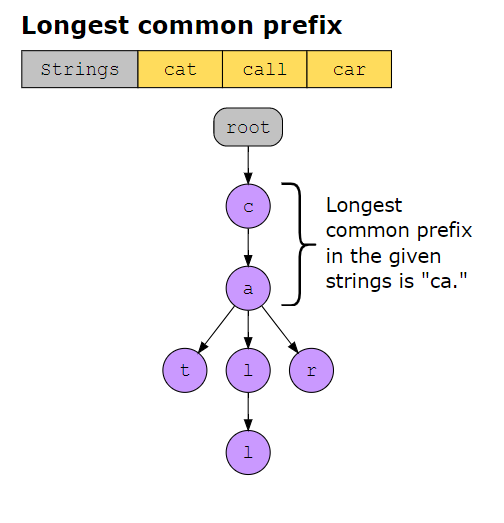
This is exactly the motivation behind a trie. What if we had an n*n*-ary tree with the fan-out equal to the number of unique values in the given dataset? For example, if we’re considering strings in English, the fan-out would be 2626, corresponding to the number of letters in the English language. This makes the tree wider and shorter! The maximum depth of the trie would be the maximum length of a word or string.

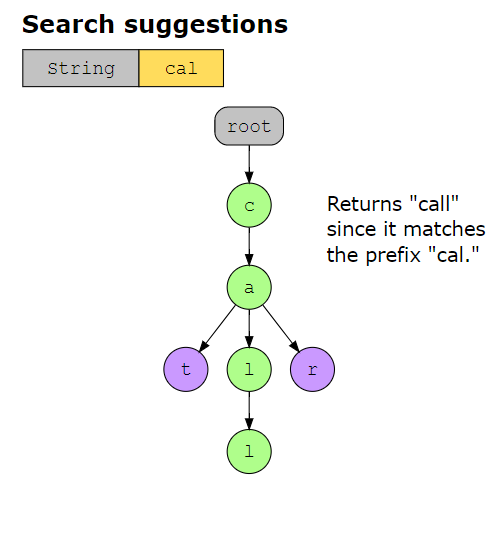
The following illustration shows how strings are stored in a trie:



## Examples

The following examples illustrate some problems that can be solved with this approach:





**Does my problem match this pattern?**

* Yes, if either of these conditions is fulfilled:
  + We need to compare two strings to detect partial matches, based on the initial characters of one or both strings.
  + We wish to optimize the space used to store a dictionary of words. Storing shared prefixes once allows for significant savings.
* No, if either of these conditions is fulfilled:
  + The problem statement restricts us from breaking down the strings into individual characters.
  + Partial matches between pairs of strings are not significant to solving the problem.

**Real-world problems**

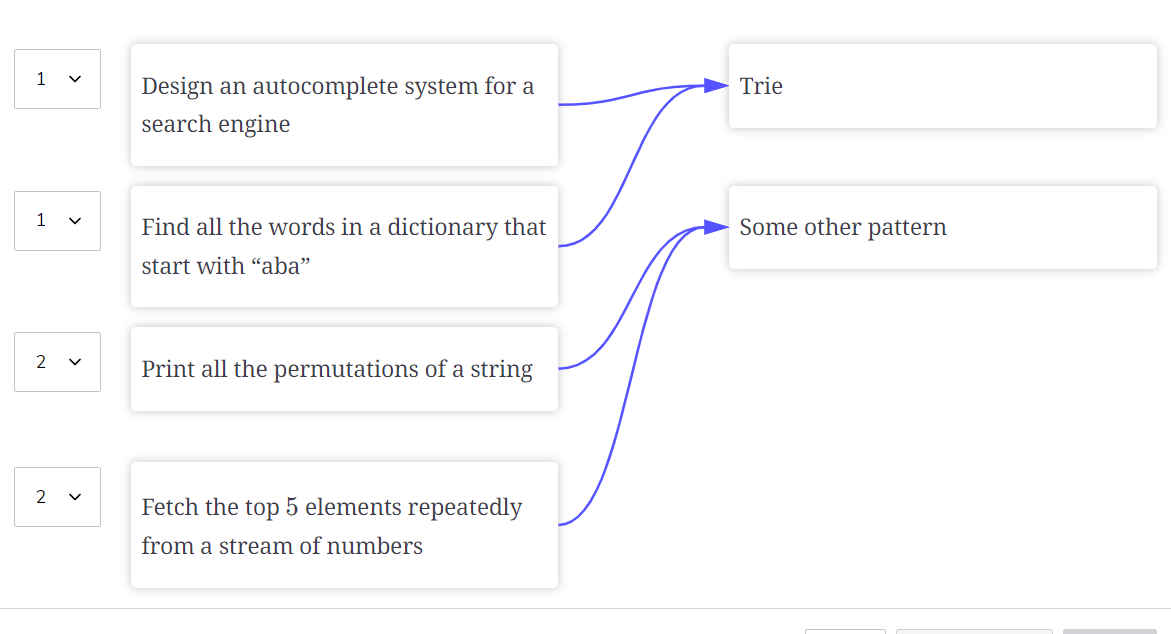
Many problems in the real world use the trie pattern. Let’s look at some examples.

* **Autocomplete system:** One of the most common applications of trie is the autocomplete system in search engines, such as Google. This is the feature that prompts the search engine to give us some suggestions to complete our query when we start typing something in the search bar. These suggestions are given based on common queries that users have searched already that match the prefix we have typed.
* **Orthographic corrector:** Ever seen pop-up suggestions or red lines under a word while you’re typing a message? That’s an orthographic corrector making suggestions and pointing out spelling mistakes by searching through a dictionary. It uses a trie data structure for efficient searches and retrievals from the available database.

**Strategy time!**

Match the problems that can be solved using the trie pattern.

**Note:** In the following exercise, option 11 corresponds to “Trie” and option 22 corresponds to “Some other pattern”.



**Implement Trie**

Try to solve the Implement Trie problem.

**We'll cover the following**

* [Statement](https://www.educative.io/courses/grokking-coding-interview-patterns-java/39jVLW5RkRr#Statement)
* [Examples](https://www.educative.io/courses/grokking-coding-interview-patterns-java/39jVLW5RkRr#Examples)
* [Understand the problem](https://www.educative.io/courses/grokking-coding-interview-patterns-java/39jVLW5RkRr#Understand-the-problem)
* [Figure it out!](https://www.educative.io/courses/grokking-coding-interview-patterns-java/39jVLW5RkRr#Figure-it-out!)
* [Try it yourself](https://www.educative.io/courses/grokking-coding-interview-patterns-java/39jVLW5RkRr#Try-it-yourself)

**Statement**

Trie is a tree-like data structure used to store strings. The tries are also called **prefix trees** because they provide very efficient prefix matching operations. Implement a [trie](https://www.educative.io/answers/what-is-a-prefix-tree) data structure with three functions that perform the following tasks:

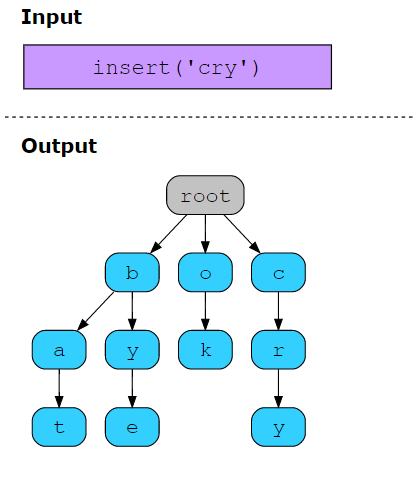
* Insert a string.
* Search a string.
* Search for a given prefix in a string.

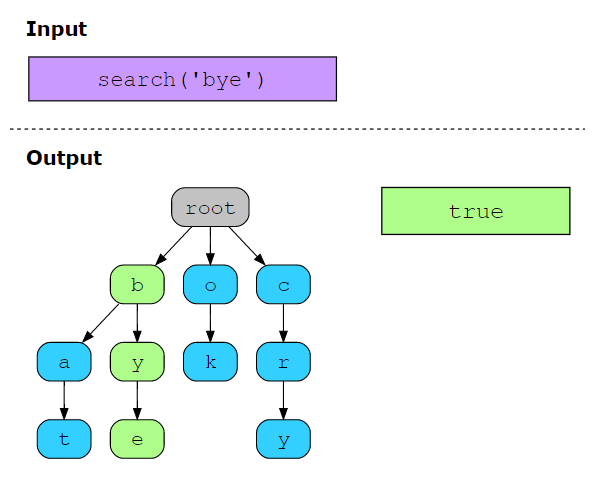
**Constraints:**

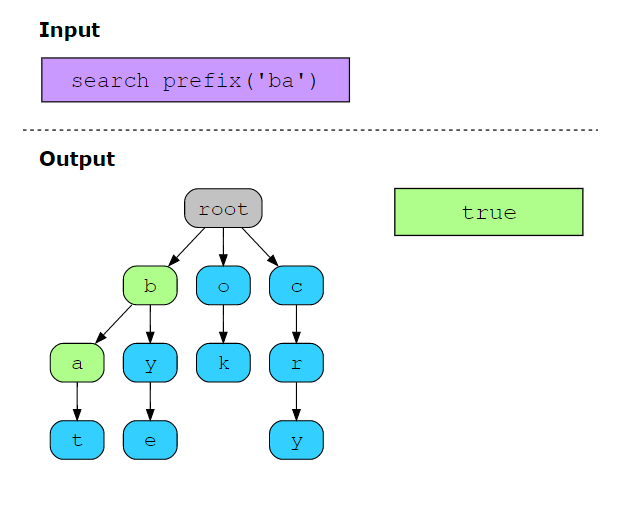
* 1 \leq1≤ word.length, prefix.length \leq 2000≤2000
* The strings consist only of lowercase English letters.
* At most, 3 \times 10^33×103 calls in total will be made to the functions.

**Examples**

The insert*insert* function does not return anything. The search*search* and prefix*prefix* search*search* functions will return TRUE if the input was found in the trie. Otherwise, they will return FALSE.

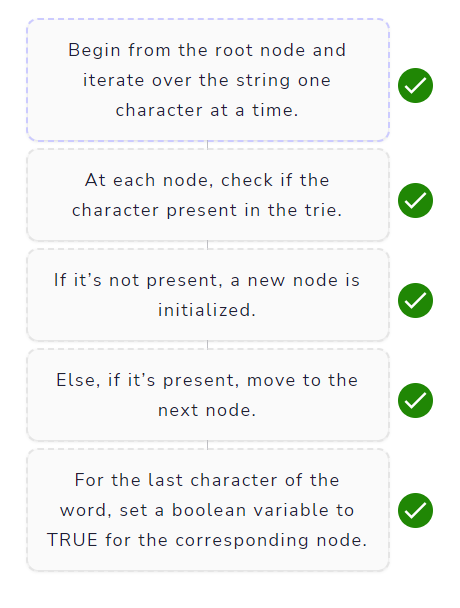






## Figure it out!

We have a game for you to play. Rearrange the logical building blocks to develop a clearer understanding of how to solve this problem.



# Solution: Implement Trie

Let's solve the Implement Trie problem using the Trie pattern.

**We'll cover the following**

* [Statement](https://www.educative.io/courses/grokking-coding-interview-patterns-java/B85RpAJZ0ko#Statement)
* [Pattern: Trie](https://www.educative.io/courses/grokking-coding-interview-patterns-java/B85RpAJZ0ko#Pattern:-Trie)
* [Step-by-step solution construction](https://www.educative.io/courses/grokking-coding-interview-patterns-java/B85RpAJZ0ko#Step-by-step-solution-construction)
  + [Solution summary](https://www.educative.io/courses/grokking-coding-interview-patterns-java/B85RpAJZ0ko#Solution-summary)
  + [Time complexity](https://www.educative.io/courses/grokking-coding-interview-patterns-java/B85RpAJZ0ko#Time-complexity)
  + [Space complexity](https://www.educative.io/courses/grokking-coding-interview-patterns-java/B85RpAJZ0ko#Space-complexity)

## Statement

Trie is a tree-like data structure used to store strings. The tries are also called **prefix trees** because they provide very efficient prefix matching operations. Implement a [trie data structure](https://www.educative.io/answers/what-is-a-prefix-tree) with three functions that perform the following tasks:

* Insert a string.
* Search a string.
* Search for a given prefix in a string.

**Constraints:**

* 1 \leq1≤ word.length, prefix.length \leq 2000≤2000
* The strings consist only of lowercase English letters.
* At most, 3 \times 10^33×103 calls in total will be made to the functions.

## Pattern: Trie

There are multiple data structures that can be used to store strings. However, for a majority of them, searching a string requires a complete traversal of the stored data. A more efficient approach is to use a trie.

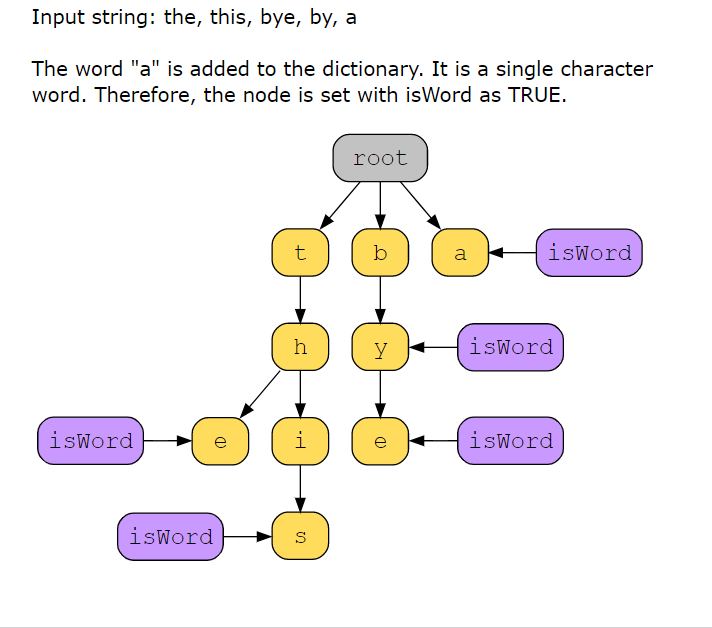
A trie is a tree of characters. The tree takes a word as a parameter and creates a new node for each character of that word. This way, the given string is searched character by character and does not require an exhaustive search. Hence, this problem fits perfectly in the trie pattern.

## Step-by-step solution construction

To implement a Trie class, we’ll initialize the root node of the tree in the constructor. This node will be of type Node. The Node class contains a dictionary of nodes and a boolean variable, which determines if the character is at the end of a word.

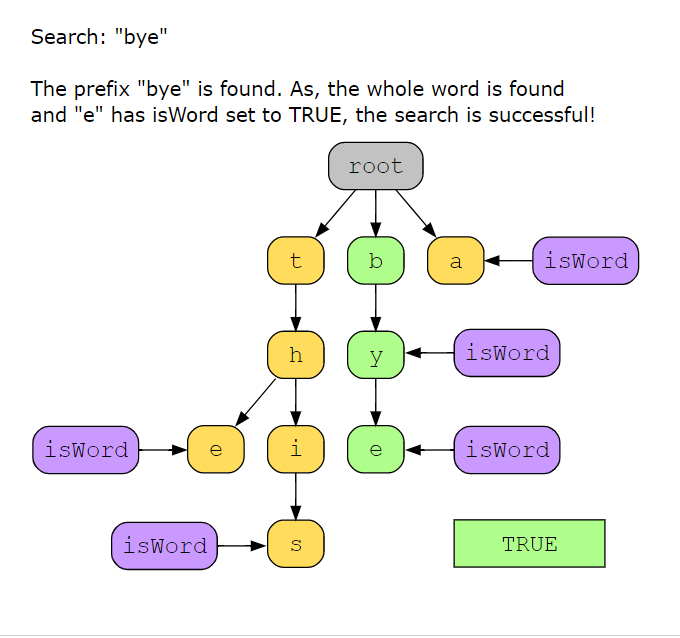
Let’s discuss the following functions now:

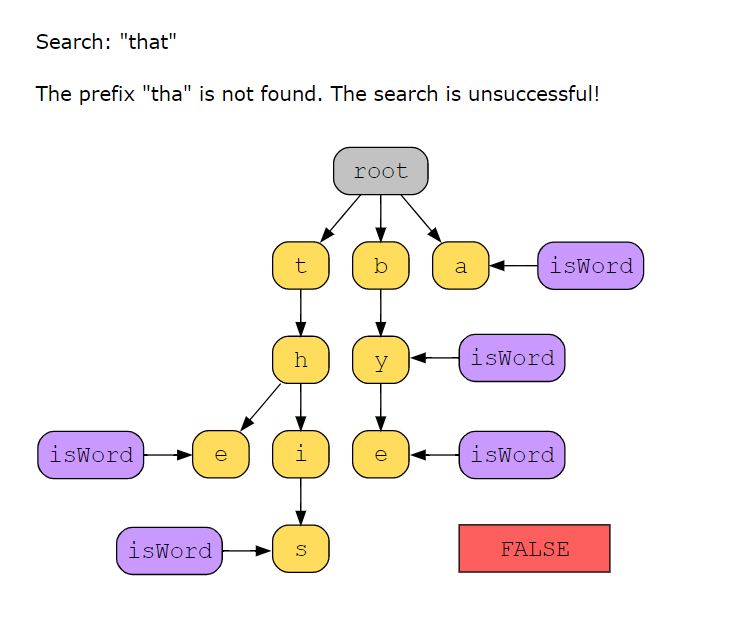
* **Insert():** In this function, we take a word as input. We begin from the root node and iterate over the string one character at a time. At each node, we check if a child node with the character is present or not. If it’s not present, a new node is initialized. For the last character of the word, we also set the boolean variable to TRUE for the corresponding node.



Here’s how it’s implemented:

* **Search():** In this function, we begin checking from the root node to see if the first character exists in its children’s dictionary. If it exists, we move on to that node and check its children for the next character. If at any point the node corresponding to a character is not found, we return FALSE. If the whole string is found but the boolean variable is not set to TRUE for the last node, we return FALSE as well. TRUE is only returned if all the characters match and the word also ends at that point.





* **Search prefix():** This function is mostly the same as the search function. The only exception is that we do not check if the boolean variable is also set to TRUE in the last-found node because we aren’t looking for a complete word but just a prefix. In the above trie, for instance, *search prefix(“by”)* should return TRUE.

====================== Code =======================================

import java.util. \* ;

class TrieNode {

boolean isWord;

HashMap < Character,

TrieNode > children;

public TrieNode() {

this.children = new HashMap < Character,

TrieNode > ();

this.isWord = false;

}

}

import java.util. \* ;

class Trie {

TrieNode root;

public Trie() {

this.root = new TrieNode();

}

public static String repeat(String str, int pValue) {

String out = "";

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

out += str;

}

return out;

}

public static String printWithMarkers(String word, int index) {

String out = "";

for (int i = 0; i < word.length(); i++) {

if (i == index) {

out += "«";

out += String.valueOf(word.charAt(i)) + "» ";

}

else out += String.valueOf(word.charAt(i)) + " ";

}

return out;

}

// inserting string in trie

public void insert(String word) {

TrieNode trieNode = this.root;

// adding string characters in the tree

System.out.println("\n\tAdding string characters to the trie");

int j = 0;

for (int i = 0; i < word.length(); i++) {

char c = word.charAt(i);

System.out.println("\tLoop index " + j);

System.out.println("\t\t" + printWithMarkers(word, j));

j += 1;

System.out.println("\t\tcharacter: " + c);

if (!trieNode.children.containsKey(c)) {

System.out.println("\t\t\t" + c + " is not in the trie, creating a new node");

trieNode.children.put(c, new TrieNode());

}

else {

System.out.println("\t\t\t" + c + " is already in the trie, hence we move to the next character");

}

System.out.println("\t\t\tCurrent state of trie");

Print.printState(this.root, "\n\t\t\t");

System.out.println();

trieNode = trieNode.children.get(c);

}

j = 0;

System.out.println("\n\t\t\tThe word is complete, setting the isWord variable to true");

trieNode.isWord = true;

System.out.println("\t\t\t\tisWord: " + trieNode.isWord);

Print.printState(this.root, "\n\t\t\t");

}

public boolean search(String word) {

System.out.println("\n\tSearching for a string");

TrieNode trieNode = this.root;

int j = 0;

// iterate over the string characters and check in the node's children

for (int i = 0; i < word.length(); i++) {

char c = word.charAt(i);

System.out.println("\tLoop index " + j);

System.out.println("\t\t" + printWithMarkers(word, j));

System.out.println("\t\tcharacter: " + c);

if (j == 0) System.out.println("\t\tRoot's children: " + trieNode.children.keySet());

else System.out.println("\t\tNode " + word.charAt(j - 1) + "'s children: " + trieNode.children.keySet());

if (!trieNode.children.containsKey(c)) {

System.out.println("\t\tCharacter '" + c + "' is not present in the node's children, returning False");

return false;

}

else System.out.println("\t\tCharacter '" + c + "' is present in the node's children, moving to the next character.");

trieNode = trieNode.children.get(c); // update the node since as we're moving to the next character

j += 1;

}

if (trieNode.isWord) {

System.out.println("\tAll characters are present in the trie and isWord: " + trieNode.isWord);

System.out.println("\tWord found!");

}

else {

System.out.println("\tAll characters are found, however, isWord: " + trieNode.isWord);

System.out.println("\tWord not found!");

}

return trieNode.isWord; //if isWord is true, the string exists

}

// searching for a prefix

public boolean searchPrefix(String prefix) {

System.out.println("\n\tSearching for a prefix");

TrieNode trieNode = this.root;

int j = 0;

// iterate over the string characters and check in the node's children

for (int i = 0; i < prefix.length(); i++) {

char c = prefix.charAt(i);

System.out.println("\tLoop index " + j);

System.out.println("\t\t" + printWithMarkers(prefix, j));

System.out.println("\t\tcharacter: " + c);

if (j == 0) System.out.println("\t\tRoot's children: " + trieNode.children.keySet());

else System.out.println("\t\tNode " + prefix.charAt(j - 1) + "'s children: " + trieNode.children.keySet());

if (!trieNode.children.containsKey(c)) {

System.out.println("\t\tCharacter '" + c + "' is not present in the node's children, returning False");

return false;

}

else System.out.println("\t\tCharacter '" + c + "' is present in the node's children, moving to the next character.");

trieNode = trieNode.children.get(c); // update the node since as we're moving to the next character

j += 1;

}

System.out.println("\tPrefix found!");

return true;

}

public static void main(String args[]) {

List < String > keys = Arrays.asList("the", "a", "there", "answer");

Trie trieOfKeys = new Trie();

int num = 1;

for (String x: keys) {

System.out.println(num + ".\tInserting key: " + x);

trieOfKeys.insert(x);

num += 1;

System.out.println();

System.out.println(repeat("-", 100));

}

List < String > search = Arrays.asList("a", "answer", "xyz", "an");

for (String y: search) {

System.out.println(num + ".\tSearching key: " + y);

System.out.println("\t" + trieOfKeys.search(y));

num += 1;

System.out.println(repeat("-", 100));

}

List < String > searchPrefix = Arrays.asList("b", "an");

for (String z: search) {

System.out.println(num + ".\tSearching prefix: " + z);

System.out.println("\t" + trieOfKeys.searchPrefix(z));

num += 1;

System.out.println(repeat("-", 100));

}

}

}

import java.util. \* ;

class Print {

public static void printState(TrieNode trieNode, String indent) {

Stack < TrieNode > stack = new Stack < TrieNode > ();

Stack < Character > keys = new Stack < Character > ();

TrieNode temp = trieNode;

for (Character i: temp.children.keySet()) {

stack.push(temp.children.get(i));

keys.push(i);

}

while (!stack.isEmpty()) {

System.out.print(indent);

System.out.print(keys.pop());

indent += " ";

temp = stack.pop();

if (temp.isWord == true) System.out.print("\*");

while (!temp.children.isEmpty()) {

for (Character i: temp.children.keySet()) {

System.out.print(indent);

System.out.print(i);

temp = temp.children.get(i);

if (temp.isWord == true) System.out.print("\*");

indent += " ";

}

}

indent = "\n\t\t\t";

}

}

}

==============================Code complete ============================

### Solution summary

To recap, the solution to this problem can be divided into the following parts:

* **Insert()**: Add words to a trie data structure.
* **Search()**: Search by starting from the root and going down the trie, checking character by character. For a complete word, check if the boolean variable is set to TRUE.
* **Search prefix()**: Search by starting from the root and going down the trie. However, the boolean variable doesn’t necessarily have to be TRUE.

### Time complexity

* **Insert()**: The time complexity is O(l)*O*(*l*), where l*l* is the length of the word being inserted.
* **Search():** The time complexity is O(l)*O*(*l*), where l*l* is the length of the word that we need to search in the trie.
* **Search prefix():** The time complexity is O(l)*O*(*l*), where l*l* is the length of the prefix that we need to search in the trie.

### Space complexity

* **Insert():** The space complexity is O(l)*O*(*l*) because, in the worst case, we will add l*l* nodes in the trie.
* **Search():** The space complexity is O(1)*O*(1) because constant space is used while searching the trie.
* **Search prefix():** The space complexity is O(1)*O*(1) because constant space is used while searching the trie.