**Trie**

**Introduction**

Trie, also called prefix tree, is a special form of a Nary tree.

In this card, we will go deep into the implementation of Trie and talk about how to use this data structure to solve problems.

After completing this card, you should be able to:

1. Understand the concept of Trie;
2. Do insertion and search operations in a Trie;
3. Understand how Trie helps in practical application;
4. Solve practical problems using Trie.

Introduction to Trie

 What is Trie?

 How to represent a Trie?

Basic Operations

 Insertion in Trie

 Search in Trie

 Implement Trie (Prefix Tree)

 Implement Trie - Solution

Practical Application I

 Map Sum Pairs

 Replace Words

 Design Search Autocomplete System

 Add and Search Word - Data structure design

Practical Application II

 Maximum XOR of Two Numbers in an Array

 Word Search II

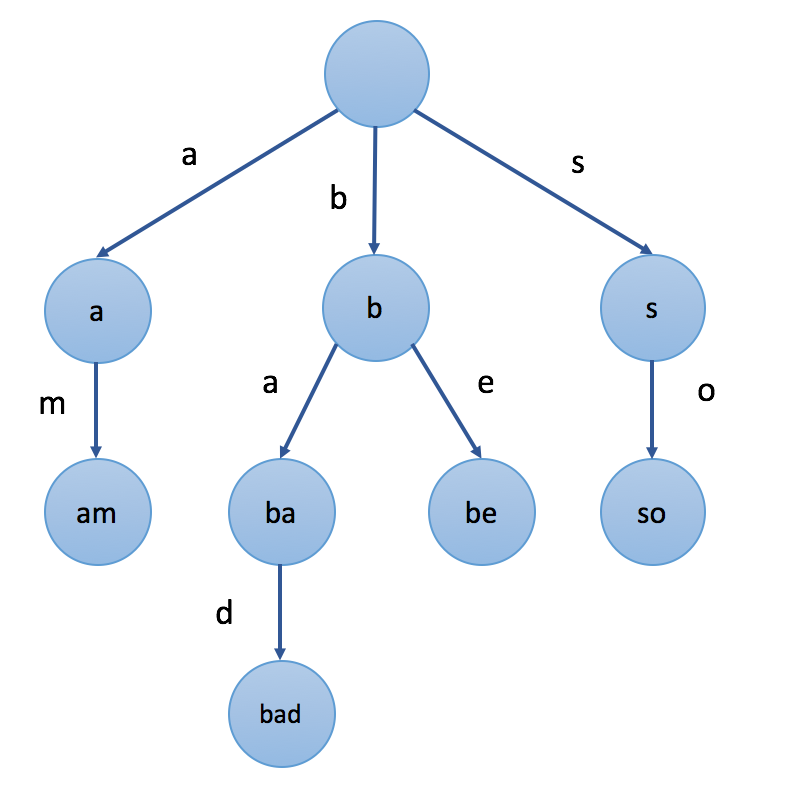
 Word Squares

 Palindrome Pairs

**What is Trie?**

A Trie is a special form of a Nary tree. Typically, a trie is used to store strings. Each Trie node represents a string (a prefix). Each node might have several children nodes while the paths to different children nodes represent different characters. And the strings the child nodes represent will be the origin string represented by the node itself plus the character on the path.

Here is an example of a trie:



In the example, the value we mark in each node is the string the node represents. For instance, we start from the root node and choose the second path 'b', then choose the first child 'a', and choose child 'd', finally we arrived at the node "bad". The value of the node is exactly formed by the letters in the path from the root to the node sequentially.

It is worth noting that the root node is associated with the empty string.

One important property of Trie is that all the descendants of a node have a common prefix of the string associated with that node. That's why Trie is also called prefix tree.

Let's look at the example again. For example, the strings represented by nodes in the subtree rooted at node "b" have a common prefix "b". And vice versa. The strings which have the common prefix "b" are all in the subtree rooted at node "b" while the strings with different prefixes will come to different branches.

Trie is widely used in various applications, such as autocomplete, spell checker, etc. We will introduce the practical applications in later chapters.

**How to represent a Trie?**

In the previous article, we introduce the concept of Trie. In this article, we will talk about how to represent this data structure in coding languages.

Briefly review the node structure of a Nary tree before reading the following contents.

What's special about Trie is the corresponding relationship between characters and children nodes. There are a lot of different representations of a trie node. Here we provide two of them.

### ***First Solution - Array***

The first solution is to use an array to store children nodes.

For instance, if we store strings which only contains letter a to z, we can declare an array whose size is 26 in each node to store its children nodes. And for a specific character c, we can use c - 'a' as the index to find the corresponding child node in the array.

|  |
| --- |
| class TrieNode {  // change this value to adapt to different cases  public static final N = 26;  public TrieNode[] children = new TrieNode[N];    // you might need some extra values according to different cases  };  /\*\* Usage:  \* Initialization: TrieNode root = new TrieNode();  \* Return a specific child node with char c: root.children[c - 'a']  \*/ |

It is really fast to visit a child node. It is comparatively easy to visit a specific child since we can easily transfer a character to an index in most cases. But not all children nodes are needed. So there might be some waste of space.

### ***Second Solution - Map***

The second solution is to use a hashmap to store children nodes.

We can declare a hashmap in each node. The key of the hashmap are characters and the value is the corresponding child node.

|  |
| --- |
| class TrieNode {  public Map<Character, TrieNode> children = new HashMap<>();    // you might need some extra values according to different cases  };  /\*\* Usage:  \* Initialization: TrieNode root = new TrieNode();  \* Return a specific child node with char c: root.children.get(c)  \*/ |

It is even easier to visit a specific child directly by the corresponding character. But it might be a little slower than using an array. However, it saves some space since we only store the children nodes we need. It is also more flexible because we are not limited by a fixed length and fixed range.

### ***More***

We mentioned how to represent the children nodes in Trie node. Besides, we might need some other values.

For example, as we know, each Trie node represents a string but not all the strings represented by Trie nodes are meaningful. If we only want to store words in a Trie, we might declare a boolean in each node as a flag to indicate if the string represented by this node is a word or not.

## Basic Operations

By reading the articles in the previous chapter, you should have a basic understanding of Trie and how to represent this data structure in your code.

We will talk about basic operations in Trie in this chapter. After this chapter, you should be able to implement a Trie by your own.

**Insertion in Trie**

We have talked about insertion in a BST in another card ([Introduction to Data Structure - Binary Search Tree](https://leetcode.com/explore/learn/card/introduction-to-data-structure-binary-search-tree/)).

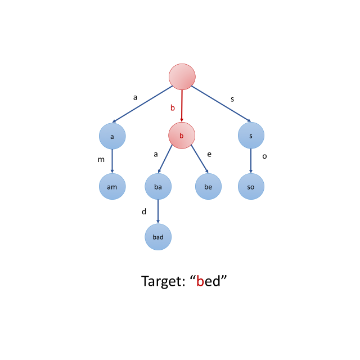
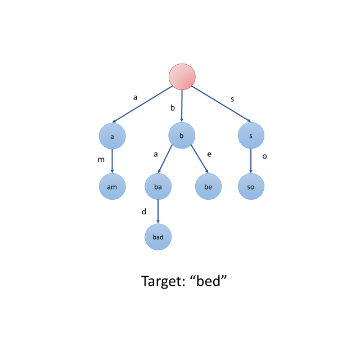
Question:

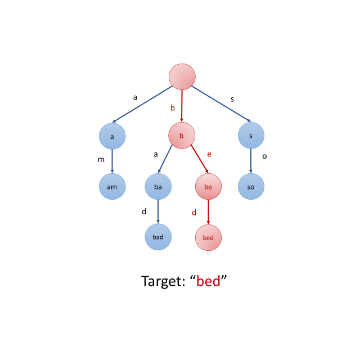
Do you remember how to insert a new node in a binary search tree?

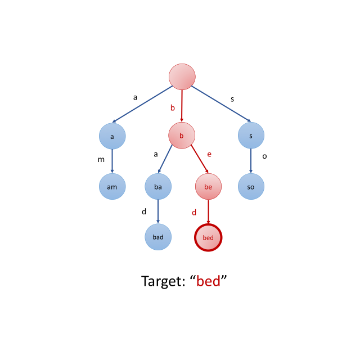
When we insert a target value into a BST, in each node, we need to decide which child node to go according to the relationship between the value of the node and the target value. Similarly, when we insert a target value into a Trie, we will also decide which path to go depending on the target value we insert.

To be more specific, if we insert a string S into Trie, we start with the root node. We will choose a child or add a new child node depending on S[0], the first character in S. Then we go down to the second node and we will make a choice according to S[1]. Then we go down to the third node, so on and so for. Finally, we traverse all characters in S sequentially and reach the end. The end node will be the node which represents the string S.

Here is an example:







Let's summarize the strategy using pseudo-code:

1. Initialize: cur = root

2. for each char c in target string S:

3. if cur does not have a child c:

4. cur.children[c] = new Trie node

5. cur = cur.children[c]

6. cur is the node which represents the string S

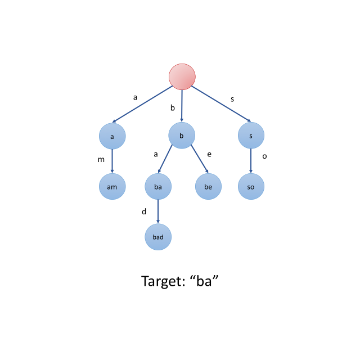
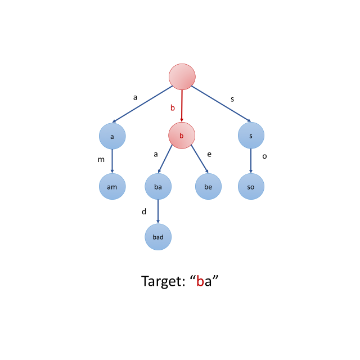
Usually, you will need to build the trie by yourself. Building a trie is actually to call the insertion function several times. But remember to initialize a root node before you insert the strings.

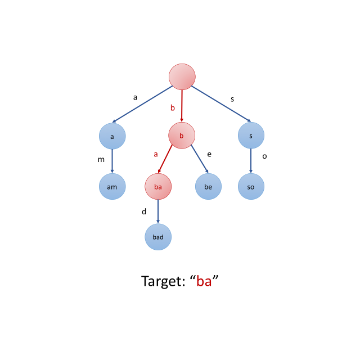
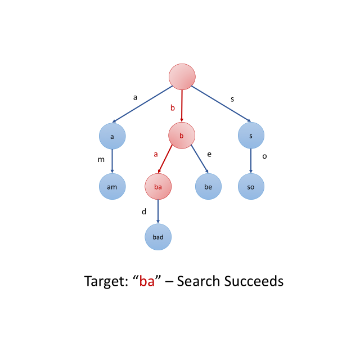
**Search in Trie**

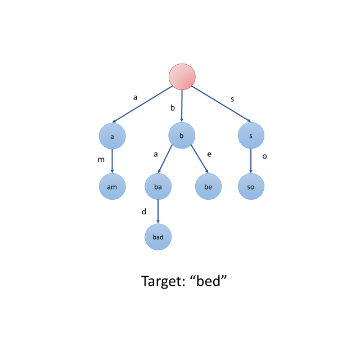
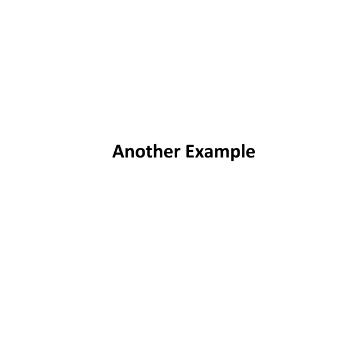
### **Search Prefix**

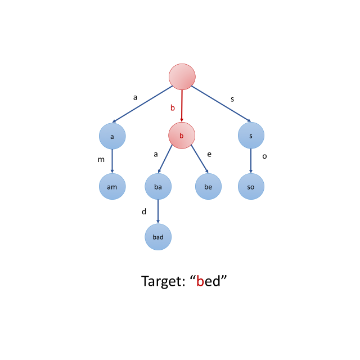
As we mentioned in the introduction to Trie, all the descendants of a node have a common prefix of the string associated with that node. Therefore, it should be easy to search if there are any words in the trie that starts with the given prefix.

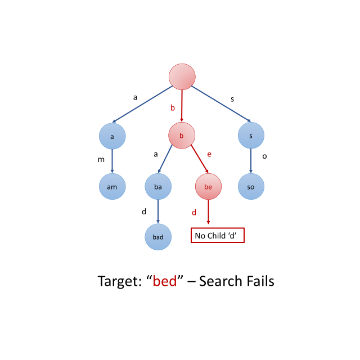
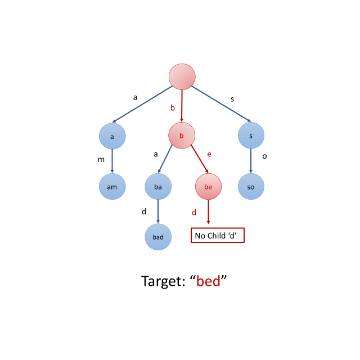
Similarly, we can go down the tree depending on the given prefix. Once we can not find the child node we want, search fails. Otherwise, search succeeds. To be more specific, we provide several examples:

****

****







Let's summarize the strategy using pseudo-code:

1. Initialize: cur = root

2. for each char c in target string S:

3. if cur does not have a child c:

4. search fails

5. cur = cur.children[c]

6. search successes

### **Search Word**

You might also want to know how to search for a specific word rather than a prefix. We can treat this word as a prefix and search in the same way we mentioned above.

1. If search fails which means that no words start with the target word, the target word is definitely not in the Trie.
2. If search succeeds, we need to check if the target word is only a prefix of words in Trie or it is exactly a word. To solve this problem, you might want to modify the node structure a little bit.

Hint: A boolean flag in each node might work.

**Implement Trie (Prefix Tree)**

Implement a trie with insert, search, and startsWith methods.

**Example:**

Trie trie = new Trie();

trie.insert("apple");

trie.search("apple"); // returns true

trie.search("app"); // returns false

trie.startsWith("app"); // returns true

trie.insert("app");

trie.search("app"); // returns true

**Note:**

* You may assume that all inputs are consist of lowercase letters a-z.
* All inputs are guaranteed to be non-empty strings.

## Summary

This article is for intermediate level users. It introduces the following ideas: The data structure Trie (Prefix tree) and most common operations with it.

## Solution

#### **Applications**

Trie (we pronounce "try") or prefix tree is a tree data structure, which is used for retrieval of a key in a dataset of strings. There are various applications of this very efficient data structure such as :

##### **1.**[**Autocomplete**](https://en.wikipedia.org/wiki/Autocomplete)

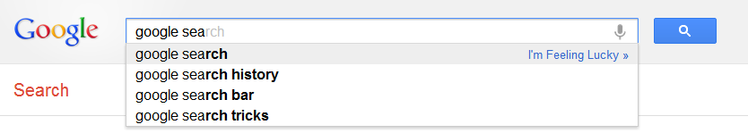


Figure 1. Google Suggest in action.

##### **2.**[**Spell checker**](https://en.wikipedia.org/wiki/Spell_checker)

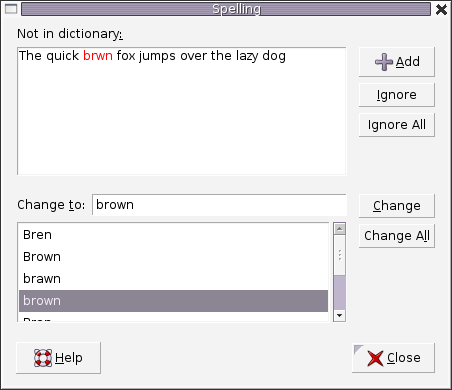


Figure 2. A spell checker used in word processor.

##### **3.**[**IP routing (Longest prefix matching)**](https://en.wikipedia.org/wiki/Longest_prefix_match)

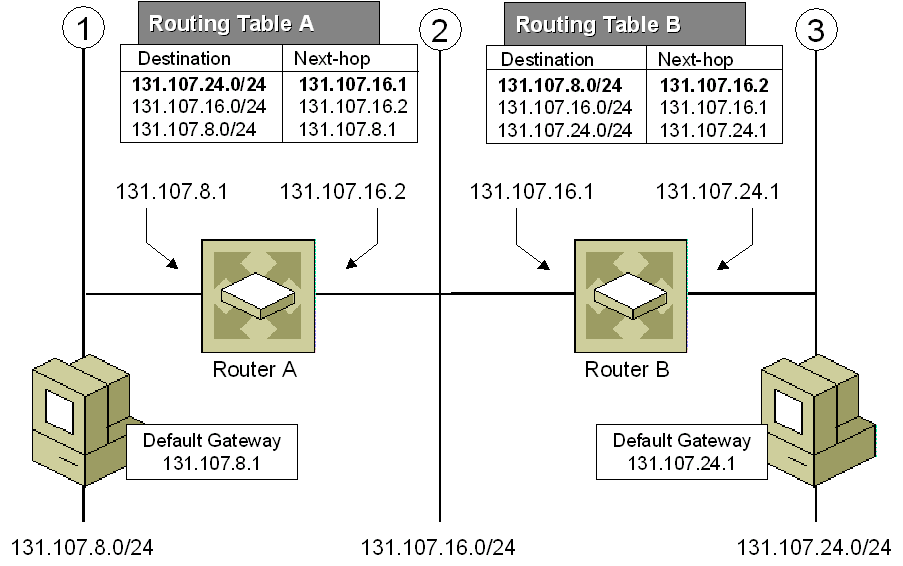


Figure 3. Longest prefix matching algorithm uses Tries in Internet Protocol (IP) routing to select an entry from a forwarding table.

##### **4.**[**T9 predictive text**](https://en.wikipedia.org/wiki/T9_(predictive_text))

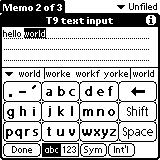


Figure 4. T9 which stands for Text on 9 keys, was used on phones to input texts during the late 1990s.

##### **5.**[**Solving word games**](https://en.wikipedia.org/wiki/Boggle)

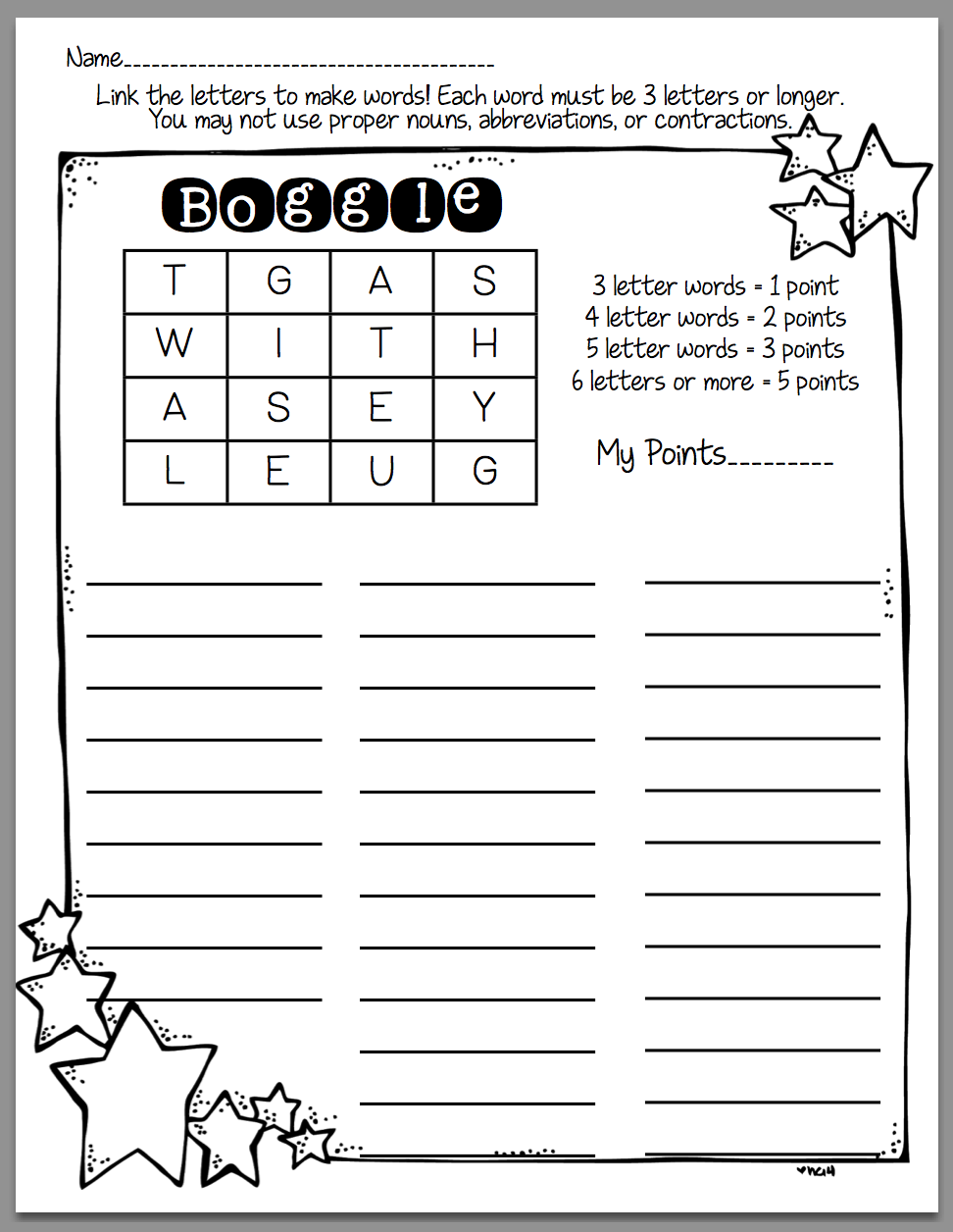


Figure 5. Tries is used to solve Boggle efficiently by pruning the search space.

There are several other data structures, like balanced trees and hash tables, which give us the possibility to search for a word in a dataset of strings. Then why do we need trie? Although hash table has *O*(1) time complexity for looking for a key, it is not efficient in the following operations :

* Finding all keys with a common prefix.
* Enumerating a dataset of strings in lexicographical order.

Another reason why trie outperforms hash table, is that as hash table increases in size, there are lots of hash collisions and the search time complexity could deteriorate to *O*(*n*), where *n* is the number of keys inserted. Trie could use less space compared to Hash Table when storing many keys with the same prefix. In this case using trie has only *O*(*m*) time complexity, where m*m* is the key length. Searching for a key in a balanced tree costs *O*(*m*log*n*) time complexity.

#### **Trie node structure**

Trie is a rooted tree. Its nodes have the following fields:

* Maximum of *R* links to its children, where each link corresponds to one of *R* character values from dataset alphabet. In this article we assume that *R* is 26, the number of lowercase latin letters.
* Boolean field which specifies whether the node corresponds to the end of the key, or is just a key prefix.

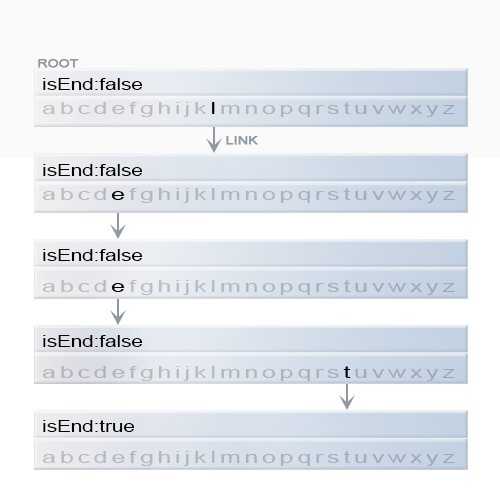


Figure 6. Representation of a key "leet" in trie.

|  |
| --- |
| class TrieNode {  // R links to node children  private TrieNode[] links;  private final int R = 26;  private boolean isEnd;  public TrieNode() {  links = new TrieNode[R];  }  public boolean containsKey(char ch) {  return links[ch -'a'] != null;  }  public TrieNode get(char ch) {  return links[ch -'a'];  }  public void put(char ch, TrieNode node) {  links[ch -'a'] = node;  }  public void setEnd() {  isEnd = true;  }  public boolean isEnd() {  return isEnd;  }  } |

Two of the most common operations in a trie are insertion of a key and search for a key.

#### **Insertion of a key to a trie**

We insert a key by searching into the trie. We start from the root and search a link, which corresponds to the first key character. There are two cases :

* A link exists. Then we move down the tree following the link to the next child level. The algorithm continues with searching for the next key character.
* A link does not exist. Then we create a new node and link it with the parent's link matching the current key character. We repeat this step until we encounter the last character of the key, then we mark the current node as an end node and the algorithm finishes.

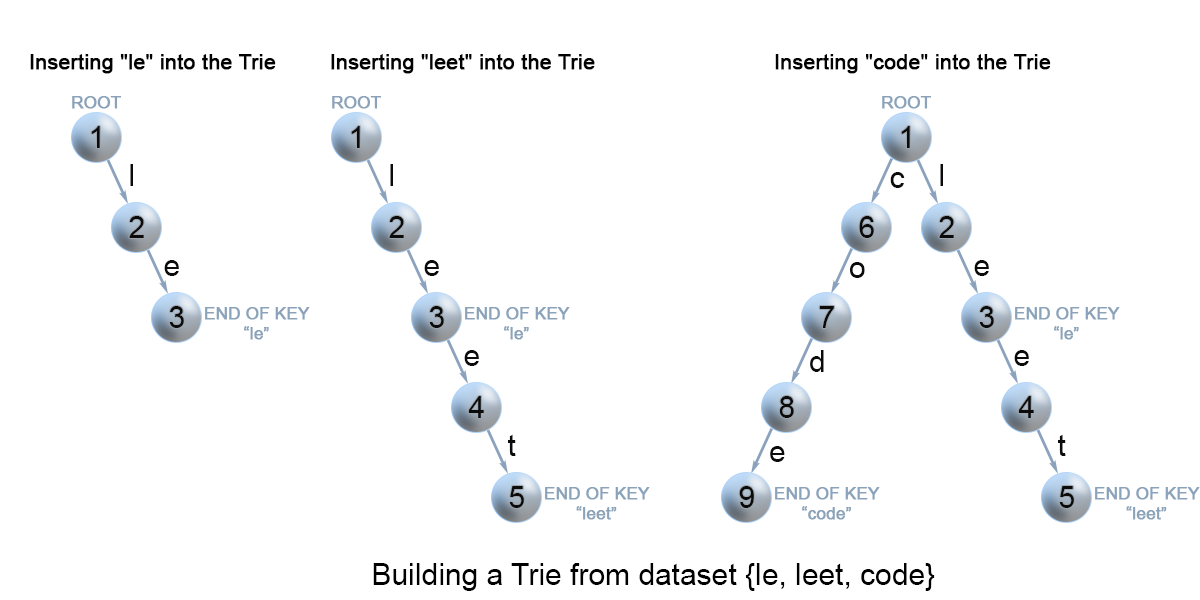


Figure 7. Insertion of keys into a trie.

|  |
| --- |
| class Trie {  private TrieNode root;  public Trie() {  root = new TrieNode();  }  // Inserts a word into the trie.  public void insert(String word) {  TrieNode node = root;  for (int i = 0; i < word.length(); i++) {  char currentChar = word.charAt(i);  if (!node.containsKey(currentChar)) {  node.put(currentChar, new TrieNode());  }  node = node.get(currentChar);  }  node.setEnd();  }  } |

**Complexity Analysis**

* Time complexity : *O*(*m*), where m is the key length.

In each iteration of the algorithm, we either examine or create a node in the trie till we reach the end of the key. This takes only m*m* operations.

* Space complexity : *O*(*m*).

In the worst case newly inserted key doesn't share a prefix with the the keys already inserted in the trie. We have to add m*m* new nodes, which takes us *O*(*m*) space.

#### **Search for a key in a trie**

Each key is represented in the trie as a path from the root to the internal node or leaf. We start from the root with the first key character. We examine the current node for a link corresponding to the key character. There are two cases :

* A link exist. We move to the next node in the path following this link, and proceed searching for the next key character.
* A link does not exist. If there are no available key characters and current node is marked as isEnd we return true. Otherwise there are possible two cases in each of them we return false :
  + There are key characters left, but it is impossible to follow the key path in the trie, and the key is missing.
  + No key characters left, but current node is not marked as isEnd. Therefore the search key is only a prefix of another key in the trie.



Figure 8. Search for a key in a trie.

|  |
| --- |
| class Trie {  ...  // search a prefix or whole key in trie and  // returns the node where search ends  private TrieNode searchPrefix(String word) {  TrieNode node = root;  for (int i = 0; i < word.length(); i++) {  char curLetter = word.charAt(i);  if (node.containsKey(curLetter)) {  node = node.get(curLetter);  } else {  return null;  }  }  return node;  }  // Returns if the word is in the trie.  public boolean search(String word) {  TrieNode node = searchPrefix(word);  return node != null && node.isEnd();  }  } |

**Complexity Analysis**

* Time complexity : O(m)*O*(*m*) In each step of the algorithm we search for the next key character. In the worst case the algorithm performs m*m* operations.
* Space complexity : O(1)*O*(1)

#### **Search for a key prefix in a trie**

The approach is very similar to the one we used for searching a key in a trie. We traverse the trie from the root, till there are no characters left in key prefix or it is impossible to continue the path in the trie with the current key character. The only difference with the mentioned above search for a key algorithm is that when we come to an end of the key prefix, we always return true. We don't need to consider the isEnd mark of the current trie node, because we are searching for a prefix of a key, not for a whole key.

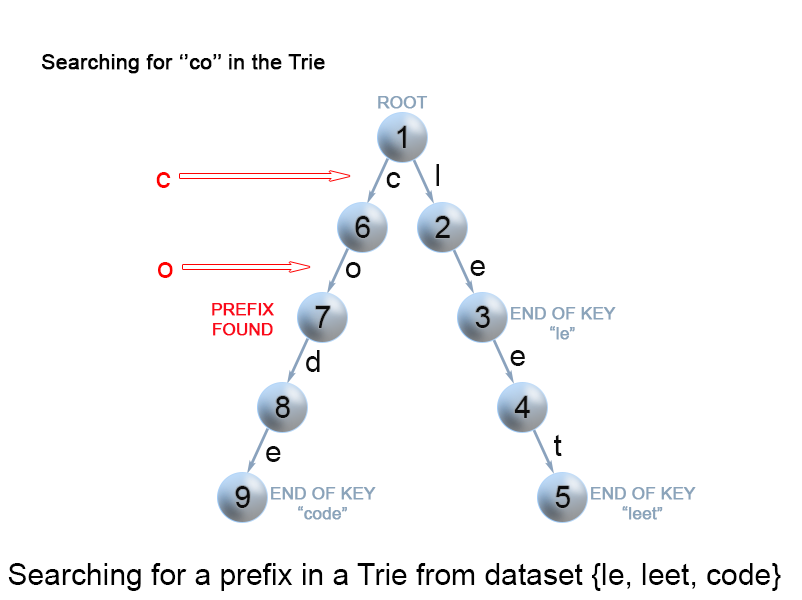


Figure 9. Search for a key prefix in a trie.

|  |
| --- |
| class Trie {  ...  // Returns if there is any word in the trie  // that starts with the given prefix.  public boolean startsWith(String prefix) {  TrieNode node = searchPrefix(prefix);  return node != null;  }  } |

**Complexity Analysis**

* Time complexity : *O*(*m*)
* Space complexity : *O*(1)

## Practice Problems

Here are some wonderful problems for you to practice which uses the Trie data structure.

1. [Add and Search Word - Data structure design](https://leetcode.com/problems/add-and-search-word-data-structure-design/) - Pretty much a direct application of Trie.
2. [Word Search II](https://leetcode.com/problems/word-search-ii/) - Similar to Boggle.

**Implement Trie - Solution**

Problem: Implement a trie with insert, search, and startsWith methods.

### 

### ***Solution***

The key to this problem is to design the Trie node structure. In order to know if the string represented by the node is a word or not, we need an extra boolean flag.

When we insert a new word, we will set the flag in the end node of the word to be true. When we implement the startsWith method, we return true if we successfully find the path. However, when we implement the search method, we return true only if we successfully find the path and the flag of the end node is true.

We provide both C++ and Java codes for your reference. We implement Trie using the map.

|  |
| --- |
| class Trie {  class TrieNode {  public boolean isWord;  public Map<Character, TrieNode> childrenMap = new HashMap<>();  }    private TrieNode root;  /\*\* Initialize your data structure here. \*/  public Trie() {  root = new TrieNode();  }    /\*\* Inserts a word into the trie. \*/  public void insert(String word) {  TrieNode cur = root;  for(int i = 0; i < word.length(); i++){  char c = word.charAt(i);  if(cur.childrenMap.get(c) == null){  // insert a new node if the path does not exist  cur.childrenMap.put(c, new TrieNode());  }  cur = cur.childrenMap.get(c);  }  cur.isWord = true;  }    /\*\* Returns if the word is in the trie. \*/  public boolean search(String word) {  TrieNode cur = root;  for(int i = 0; i < word.length(); i++) {  char c = word.charAt(i);  if(cur.childrenMap.get(c) == null) {  return false;  }  cur = cur.childrenMap.get(c);  }  return cur.isWord;  }    /\*\* Returns if there is any word in the trie that starts with the given prefix. \*/  public boolean startsWith(String prefix) {  TrieNode cur = root;  for(int i = 0;i < prefix.length(); i++){  char c = prefix.charAt(i);  if(cur.childrenMap.get(c) == null) {  return false;  }  cur = cur.childrenMap.get(c);  }  return true;  }  } |

### ***Complexity Analysis***

Let's discuss the complexity of this algorithm.

If the longest length of the word is N, the height of Trie will be N + 1. Therefore, the time complexity of all insert, search and startsWith methods will be O(N).

How about space complexity?

If we have M words to insert in total and the length of words is at most N, there will be at most M\*N nodes in the worst case (any two words don't have a common prefix).

Let's assume that there are maximum K different characters (K is equal to 26 in this problem, but might differs in different cases). So each node will maintain a map whose size is at most K.

Therefore, the space complexity will be O(M\*N\*K).

It seems that Trie is really space consuming, however, the real space complexity of Trie is much smaller than our estimation, especially when the distribution of words is dense.

You can also implement it by the array which will achieve a slightly better time performance but a slightly lower space performance.

### 

### ***Comparison with Hash Table***

You might wonder why not use a hash table to store strings. Let's do a brief comparison between these two data structures. We assume there are N keys and the maximum length of a key is M.

1. Time Complexity

The time complexity to search in hash table is typically O(1), but will be O(logN) in the worst time if there are too many collisions and we solve collisions using height-balanced BST.

The time complexity to search in Trie is O(M).

The hash table wins in most cases.

1. Space Complexity

The space complexity of hash table is O(M \* N). If you want hash table to have the same function with Trie, you might need to store several copies of the key. For instance, you might want to store "a", "ap", "app", "appl" and also "apple" for a keyword "apple" in order to search by prefix. The space complexity can be even much larger in that case.

The space complexity of Trie is O(M \* N) as we estimated above. But actually far smaller than the estimation since there will be a lot of words have the similar prefix in real cases.

Trie wins in most cases.

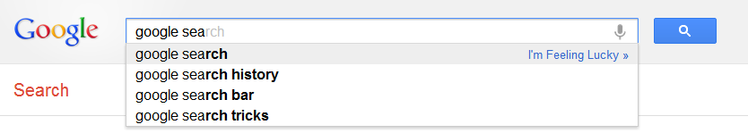
## Practical Application I

Trie is widely used to store strings and retrieve keywords, especially prefix related keywords.

The practical application scenarios of Trie can be very straightforward. Typically, you will need to provide insert method and some kind of search operation related to prefix of words. We provide exercises for you to practice in this chapter.

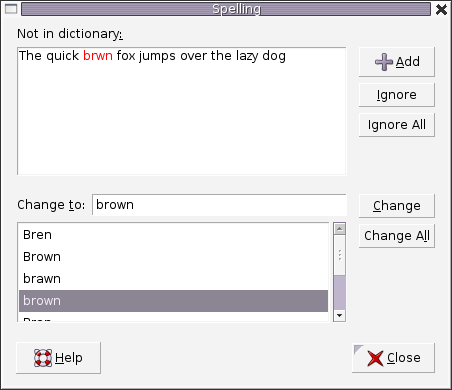
We also provide some more complicated practical problems:

1. [Autocomplete](https://en.wikipedia.org/wiki/Autocomplete)([Design Search Autocomplete System](https://leetcode.com/explore/learn/card/trie/148/practical-application-i/1054/))



One simple way to implement autocomplete is to store ngrams in Trie and do recommendation based on frequency. Consider thoroughly that what will be an ideal node structure to solve this problem.

1. [Spell Checker](https://en.wikipedia.org/wiki/Spell_checker)([Add and Search Word](https://leetcode.com/explore/learn/card/trie/148/practical-application-i/1052/))



It is easy to find words with same prefix in Trie. But how about finding similar words instead? You might want to use some search algorithm to solve this problem.

**Map Sum Pairs**

Implement the MapSum class:

* MapSum() Initializes the MapSum object.
* void insert(String key, int val) Inserts the key-val pair into the map. If the key already existed, the original key-value pair will be overridden to the new one.
* int sum(string prefix) Returns the sum of all the pairs' value whose key starts with the prefix.

**Example 1:**

**Input**

["MapSum", "insert", "sum", "insert", "sum"]

[[], ["apple", 3], ["ap"], ["app", 2], ["ap"]]

**Output**

[null, null, 3, null, 5]

**Explanation**

MapSum mapSum = new MapSum();

mapSum.insert("apple", 3);

mapSum.sum("ap"); // return 3 (apple = 3)

mapSum.insert("app", 2);

mapSum.sum("ap"); // return 5 (apple + app = 3 + 2 = 5)

**Constraints:**

* 1 <= key.length, prefix.length <= 50
* key and prefix consist of only lowercase English letters.
* 1 <= val <= 1000
* At most 50 calls will be made to insert and sum.

#### **Approach #1: Brute Force [Accepted]**

**Intuition and Algorithm**

For each key in the map, if that key starts with the given prefix, then add it to the answer.

|  |
| --- |
| class MapSum {  HashMap<String, Integer> map;  public MapSum() {  map = new HashMap<>();  }  public void insert(String key, int val) {  map.put(key, val);  }  public int sum(String prefix) {  int ans = 0;  for (String key: map.keySet()) {  if (key.startsWith(prefix)) {  ans += map.get(key);  }  }  return ans;  }  } |

\*\*Complexity Analysis\*\*

* Time Complexity: Every insert operation is *O*(1). Every sum operation is *O*(*N*∗*P*) where *N* is the number of items in the map, and *P* is the length of the input prefix.
* Space Complexity: The space used by map is linear in the size of all input key and val values combined.

#### **Approach #2: Prefix Hashmap [Accepted]**

**Intuition and Algorithm**

We can remember the answer for all possible prefixes in a HashMap score. When we get a new (key, val) pair, we update every prefix of key appropriately: each prefix will be changed by delta = val - map[key], where map is the previous associated value of key (zero if undefined.)

|  |
| --- |
| class MapSum {  Map<String, Integer> map;  Map<String, Integer> score;  public MapSum() {  map = new HashMap();  score = new HashMap();  }  public void insert(String key, int val) {  int delta = val - map.getOrDefault(key, 0);  map.put(key, val);  String prefix = "";  for (char c: key.toCharArray()) {  prefix += c;  score.put(prefix, score.getOrDefault(prefix, 0) + delta);  }  }  public int sum(String prefix) {  return score.getOrDefault(prefix, 0);  }  } |

**Complexity Analysis**

* Time Complexity: Every insert operation is O(K^2), where *K* is the length of the key, as *K* strings are made of an average length of *K*. Every sum operation is *O*(1).
* Space Complexity: The space used by map and score is linear in the size of all input key and val values combined.

#### **Approach #3: Trie [Accepted]**

**Intuition and Algorithm**

Since we are dealing with prefixes, a Trie (prefix tree) is a natural data structure to approach this problem. For every node of the trie corresponding to some prefix, we will remember the desired answer (score) and store it at this node. As in Approach #2, this involves modifying each node by delta = val - map[key].

|  |
| --- |
| class MapSum {  HashMap<String, Integer> map;  TrieNode root;  public MapSum() {  map = new HashMap();  root = new TrieNode();  }  public void insert(String key, int val) {  int delta = val - map.getOrDefault(key, 0);  map.put(key, val);  TrieNode cur = root;  cur.score += delta;  for (char c: key.toCharArray()) {  cur.children.putIfAbsent(c, new TrieNode());  cur = cur.children.get(c);  cur.score += delta;  }  }  public int sum(String prefix) {  TrieNode cur = root;  for (char c: prefix.toCharArray()) {  cur = cur.children.get(c);  if (cur == null) return 0;  }  return cur.score;  }  }  class TrieNode {  Map<Character, TrieNode> children = new HashMap();  int score;  } |

**Complexity Analysis**

* Time Complexity: Every insert operation is *O*(*K*), where *K* is the length of the key. Every sum operation is *O*(*K*).
* Space Complexity: The space used is linear in the size of the total input.

**Replace Words**

In English, we have a concept called **root**, which can be followed by some other word to form another longer word - let's call this word **successor**. For example, when the **root** "an" is followed by the **successor** word "other", we can form a new word "another".

Given a dictionary consisting of many **roots** and a sentence consisting of words separated by spaces, replace all the **successors** in the sentence with the **root** forming it. If a **successor** can be replaced by more than one **root**, replace it with the **root** that has **the shortest length**.

Return *the sentence* after the replacement.

**Example 1:**

**Input:** dictionary = ["cat","bat","rat"], sentence = "the cattle was rattled by the battery"

**Output:** "the cat was rat by the bat"

**Example 2:**

**Input:** dictionary = ["a","b","c"], sentence = "aadsfasf absbs bbab cadsfafs"

**Output:** "a a b c"

**Example 3:**

**Input:** dictionary = ["a", "aa", "aaa", "aaaa"], sentence = "a aa a aaaa aaa aaa aaa aaaaaa bbb baba ababa"

**Output:** "a a a a a a a a bbb baba a"

**Example 4:**

**Input:** dictionary = ["catt","cat","bat","rat"], sentence = "the cattle was rattled by the battery"

**Output:** "the cat was rat by the bat"

**Example 5:**

**Input:** dictionary = ["ac","ab"], sentence = "it is abnormal that this solution is accepted"

**Output:** "it is ab that this solution is ac"

**Constraints:**

* 1 <= dictionary.length <= 1000
* 1 <= dictionary[i].length <= 100
* dictionary[i] consists of only lower-case letters.
* 1 <= sentence.length <= 10^6
* sentence consists of only lower-case letters and spaces.
* The number of words in sentence is in the range [1, 1000]
* The length of each word in sentence is in the range [1, 1000]
* Each two consecutive words in sentence will be separated by exactly one space.
* sentence does not have leading or trailing spaces.

#### **Approach #1: Prefix Hash [Accepted]**

**Intuition**

For each word in the sentence, we'll look at successive prefixes and see if we saw them before.

**Algorithm**

Store all the roots in a Set structure. Then for each word, look at successive prefixes of that word. If you find a prefix that is a root, replace the word with that prefix. Otherwise, the prefix will just be the word itself, and we should add that to the final sentence answer.

|  |
| --- |
| class Solution {  public String replaceWords(List<String> roots, String sentence) {  Set<String> rootset = new HashSet();  for (String root: roots) rootset.add(root);  StringBuilder ans = new StringBuilder();  for (String word: sentence.split("\\s+")) {  String prefix = "";  for (int i = 1; i <= word.length(); ++i) {  prefix = word.substring(0, i);  if (rootset.contains(prefix)) break;  }  if (ans.length() > 0) ans.append(" ");  ans.append(prefix);  }  return ans.toString();  }  } |

**Complexity Analysis**

* Time Complexity:  where *wi*​ is the length of the *i*-th word. We might check every prefix, the *i*-th of which is  work.
* Space Complexity: *O*(*N*) where *N* is the length of our sentence; the space used by rootset.

#### **Approach #2: Trie [Accepted]**

**Intuition and Algorithm**

Put all the roots in a trie (prefix tree). Then for any query word, we can find the smallest root that was a prefix in linear time.

|  |
| --- |
| class Solution {  public String replaceWords(List<String> roots, String sentence) {  TrieNode trie = new TrieNode();  for (String root: roots) {  TrieNode cur = trie;  for (char letter: root.toCharArray()) {  if (cur.children[letter - 'a'] == null)  cur.children[letter - 'a'] = new TrieNode();  cur = cur.children[letter - 'a'];  }  cur.word = root;  }  StringBuilder ans = new StringBuilder();  for (String word: sentence.split("\\s+")) {  if (ans.length() > 0)  ans.append(" ");  TrieNode cur = trie;  for (char letter: word.toCharArray()) {  if (cur.children[letter - 'a'] == null || cur.word != null)  break;  cur = cur.children[letter - 'a'];  }  ans.append(cur.word != null ? cur.word : word);  }  return ans.toString();  }  }  class TrieNode {  TrieNode[] children;  String word;  TrieNode() {  children = new TrieNode[26];  }  } |

**Complexity Analysis**

* Time Complexity: *O*(*N*) where *N* is the length of the sentence. Every query of a word is in linear time.
* Space Complexity: *O*(*N*), the size of our trie.

**Design Search Autocomplete System**

Design a search autocomplete system for a search engine. Users may input a sentence (at least one word and end with a special character '#'). For **each character** they type **except '#'**, you need to return the **top 3** historical hot sentences that have prefix the same as the part of sentence already typed. Here are the specific rules:

1. The hot degree for a sentence is defined as the number of times a user typed the exactly same sentence before.
2. The returned top 3 hot sentences should be sorted by hot degree (The first is the hottest one). If several sentences have the same degree of hot, you need to use ASCII-code order (smaller one appears first).
3. If less than 3 hot sentences exist, then just return as many as you can.
4. When the input is a special character, it means the sentence ends, and in this case, you need to return an empty list.

Your job is to implement the following functions:

The constructor function:

AutocompleteSystem(String[] sentences, int[] times): This is the constructor. The input is **historical data**. Sentences is a string array consists of previously typed sentences. Times is the corresponding times a sentence has been typed. Your system should record these historical data.

Now, the user wants to input a new sentence. The following function will provide the next character the user types:

List<String> input(char c): The input c is the next character typed by the user. The character will only be lower-case letters ('a' to 'z'), blank space (' ') or a special character ('#'). Also, the previously typed sentence should be recorded in your system. The output will be the **top 3** historical hot sentences that have prefix the same as the part of sentence already typed.

**Example:**  
**Operation:** AutocompleteSystem(["i love you", "island","ironman", "i love leetcode"], [5,3,2,2])  
The system have already tracked down the following sentences and their corresponding times:  
"i love you" : 5 times  
"island" : 3 times  
"ironman" : 2 times  
"i love leetcode" : 2 times  
Now, the user begins another search:  
  
**Operation:** input('i')  
**Output:** ["i love you", "island","i love leetcode"]  
**Explanation:**  
There are four sentences that have prefix "i". Among them, "ironman" and "i love leetcode" have same hot degree. Since ' ' has ASCII code 32 and 'r' has ASCII code 114, "i love leetcode" should be in front of "ironman". Also we only need to output top 3 hot sentences, so "ironman" will be ignored.  
  
**Operation:** input(' ')  
**Output:** ["i love you","i love leetcode"]  
**Explanation:**  
There are only two sentences that have prefix "i ".  
  
**Operation:** input('a')  
**Output:** []  
**Explanation:**  
There are no sentences that have prefix "i a".  
  
**Operation:** input('#')  
**Output:** []  
**Explanation:**  
The user finished the input, the sentence "i a" should be saved as a historical sentence in system. And the following input will be counted as a new search.

**Note:**

1. The input sentence will always start with a letter and end with '#', and only one blank space will exist between two words.
2. The number of **complete sentences** that to be searched won't exceed 100. The length of each sentence including those in the historical data won't exceed 100.
3. Please use double-quote instead of single-quote when you write test cases even for a character input.
4. Please remember to **RESET** your class variables declared in class AutocompleteSystem, as static/class variables are **persisted across multiple test cases**. Please see [here](https://leetcode.com/faq/#different-output) for more details.

## Solution

#### **Approach 1: Brute Force**

In this solution, we make use of a HashMap map*map* which stores entries in the form (sentence\_i, times\_i)(*sentencei*​,*timesi*​). Here, times\_i*timesi*​ refers to the number of times the sentence\_i*sentencei*​ has been typed earlier.

AutocompleteSystem: We pick up each sentence from sentences*sentences* and their corresponding times from the times*times*, and make their entries in the map*map* appropriately.

input(c): We make use of a current sentence tracker variable, \text{cur\\_sent}cur\_sent, which is used to store the sentence entered till now as the input. For c*c* as the current input, firstly, we append this c*c* to \text{cur\\_sent}cur\_sent and then iterate over all the keys of map*map* to check if a key exists whose initial characters match with \text{cur\\_sent}cur\_sent. We add all such keys to a list*list*. Then, we sort this list*list* as per our requirements, and obtain the first three values from this list*list*.

|  |
| --- |
| class Node {  String sentence;  int times;  Node(String st, int t) {  sentence = st;  times = t;  }  }  class AutocompleteSystem {  private HashMap<String, Integer> map = new HashMap<>();  private String cur\_sent = "";  public AutocompleteSystem(String[] sentences, int[] times) {  for (int i = 0; i < sentences.length; i++) map.put(sentences[i], times[i]);  }  public List<String> input(char c) {  List<String> res = new ArrayList<>();  if (c == '#') {  map.put(cur\_sent, map.getOrDefault(cur\_sent, 0) + 1);  cur\_sent = "";  } else {  List<Node> list = new ArrayList<>();  cur\_sent += c;  for (String key : map.keySet())  if (key.indexOf(cur\_sent) == 0) {  list.add(new Node(key, map.get(key)));  }  Collections.sort(  list,  (a, b) -> a.times == b.times ? a.sentence.compareTo(b.sentence) : b.times - a.times);  for (int i = 0; i < Math.min(3, list.size()); i++) res.add(list.get(i).sentence);  }  return res;  }  }  /\*\*  \* Your AutocompleteSystem object will be instantiated and called as such:  \* AutocompleteSystem obj = new AutocompleteSystem(sentences, times);  \* List<String> param\_1 = obj.input(c);  \*/ |

**Performance Analysis**

* AutocompleteSystem() takes *O*(*k*∗*l*) time. This is because, putting an entry in a hashMap takes *O*(1) time. But, to create a hash value for a sentence of average length *k*, it will be scanned atleast once. We need to put *l* such entries in the *map*.
* input() takes  *O*(*n*+*m*log*m*) time. We need to iterate over the list of sentences, in *map*, entered till now(say with a count *n*), taking *O*(*n*) time, to populate the *list* used for finding the hot sentences. Then, we need to sort the *list* of length m*m*, taking *O*(*m*log*m*) time.

#### **Approach 2: Using One level Indexing**

This method is almost the same as that of the last approach except that instead of making use of simply a HashMap to store the sentences along with their number of occurences, we make use of a Two level HashMap.

Thus, we make use of an array *arr* of HashMapsEach element of this array, *arr*, is used to refer to one of the alphabets possible. Each element is a HashMap itself, which stores the sentences and their number of occurences similar to the last approach. e.g. *arr*[0] is used to refer to a HashMap which stores the sentences starting with an 'a'.

The process of adding the data in AutocompleteSystem and retrieving the data remains the same as in the last approach, except the one level indexing using *arr* which needs to be done prior to accessing the required HashMap.

|  |
| --- |
| class Node {  String sentence;  int times;  Node(String st, int t) {  sentence = st;  times = t;  }  }  class AutocompleteSystem {  private HashMap<String, Integer>[] arr;  private String cur\_sent = "";  public AutocompleteSystem(String[] sentences, int[] times) {  arr = new HashMap[26];  for (int i = 0; i < 26; i++) arr[i] = new HashMap<String, Integer>();  for (int i = 0; i < sentences.length; i++)  arr[sentences[i].charAt(0) - 'a'].put(sentences[i], times[i]);  }  public List<String> input(char c) {  List<String> res = new ArrayList<>();  if (c == '#') {  arr[cur\_sent.charAt(0) - 'a'].put(  cur\_sent, arr[cur\_sent.charAt(0) - 'a'].getOrDefault(cur\_sent, 0) + 1);  cur\_sent = "";  } else {  List<Node> list = new ArrayList<>();  cur\_sent += c;  for (String key : arr[cur\_sent.charAt(0) - 'a'].keySet()) {  if (key.indexOf(cur\_sent) == 0) {  list.add(new Node(key, arr[cur\_sent.charAt(0) - 'a'].get(key)));  }  }  Collections.sort(  list,  (a, b) -> a.times == b.times ? a.sentence.compareTo(b.sentence) : b.times - a.times);  for (int i = 0; i < Math.min(3, list.size()); i++) res.add(list.get(i).sentence);  }  return res;  }  } |

**Performance Analysis**

* AutocompleteSystem() takes *O*(*k*∗*l*+26) time. Putting an entry in a hashMap takes *O*(1) time. But, to create a hash value for a sentence of average length *k*, it will be scanned atleast once. We need to put *l* such entries in the *map*.
* input() takes *O*(*s*+*m*log*m*) time. We need to iterate only over one hashmap corresponding to the sentences starting with the first character of the current sentence, to populate the *list* for finding the hot sentences. Here, s*s* refers to the size of this corresponding hashmap. Then, we need to sort the *list* of length m*m*, taking *O*(*m*log*m*) time.

#### **Approach 3: Using Trie**

A Trie is a special data structure used to store strings that can be visualized like a tree. It consists of nodes and edges. Each node consists of at max 26 children and edges connect each parent node to its children. These 26 pointers are nothing but pointers for each of the 26 letters of the English alphabet A separate edge is maintained for every edge.

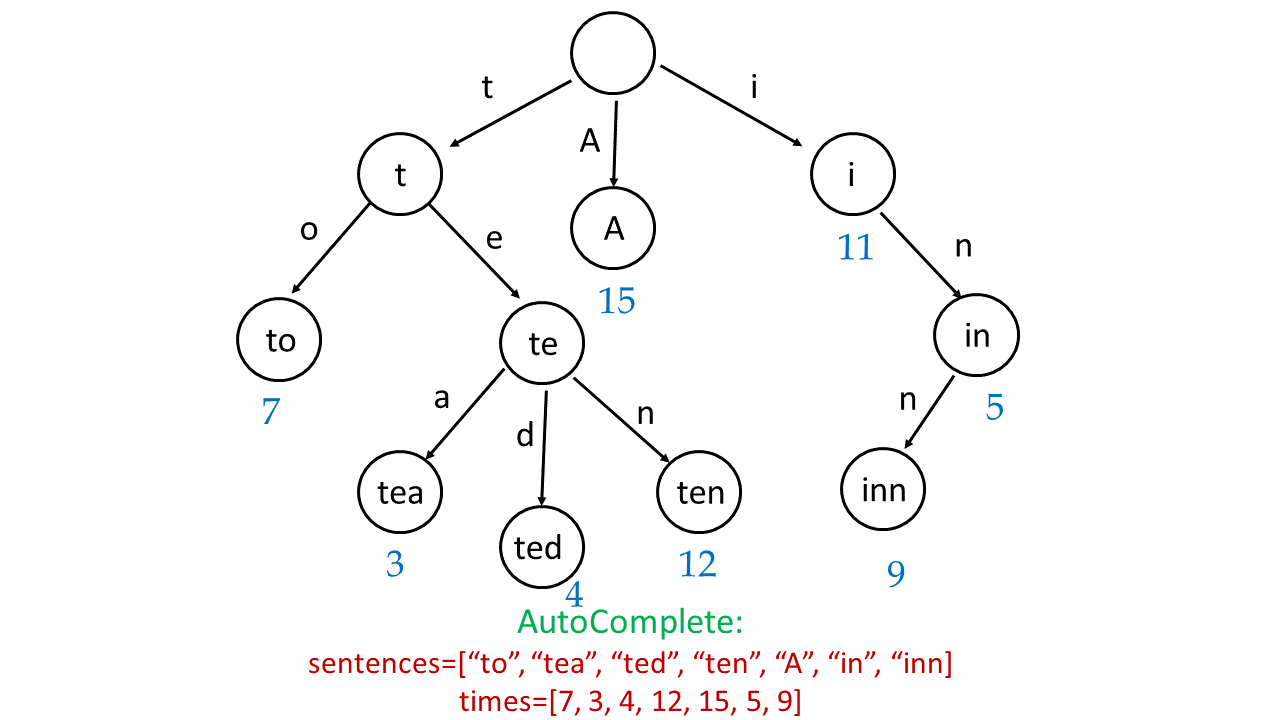
Strings are stored in a top to bottom manner on the basis of their prefix in a trie. All prefixes of length 1 are stored at until level 1, all prefixes of length 2 are sorted at until level 2 and so on.

A Trie data structure is very commonly used for representing the words stored in a dictionary. Each level represents one character of the word being formed. A word available in the dictionary can be read off from the Trie by starting from the root and going till the leaf.

By doing a small modification to this structure, we can also include an entry, times*times*, for the number of times the current word has been previously typed. This entry can be stored in the leaf node corresponding to the particular word.

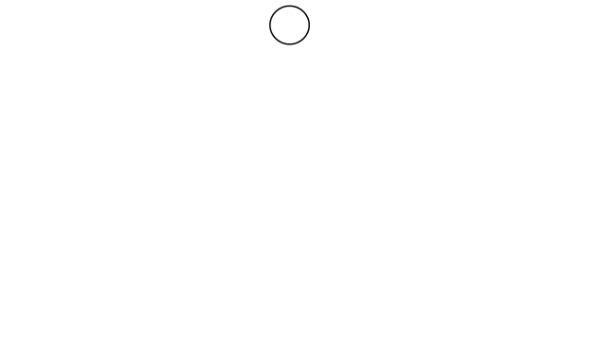
Now, for implementing the AutoComplete function, we need to consider each character of the every word given in sentences*sentences* array, and add an entry corresponding to each such character at one level of the trie. At the leaf node of every word, we can update the times*times* section of the node with the corresponding number of times this word has been typed.

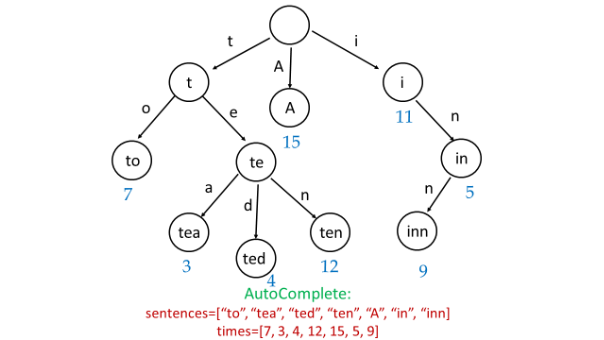
The following figure shows a trie structure for the words "A","to", "tea", "ted", "ten", "i", "in", and "inn", occuring 15, 7, 3, 4, 12, 11, 5 and 9 times respectively.

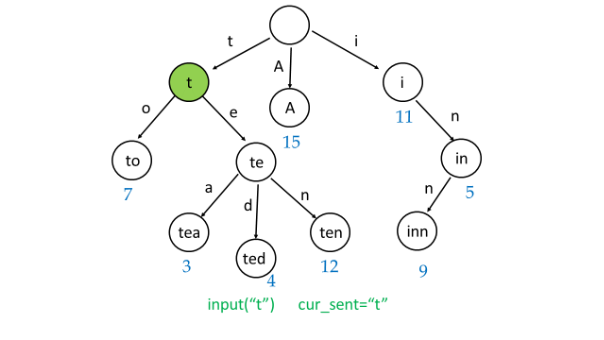


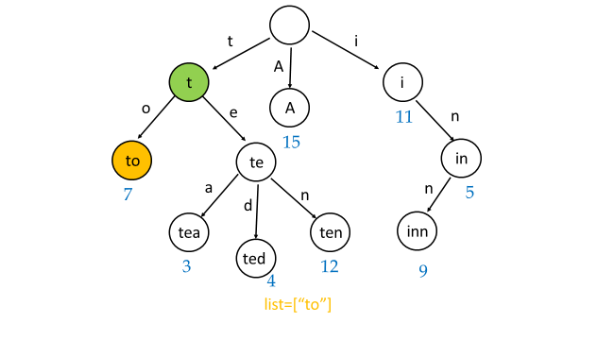
Similarly, to implement the input(c) function, for every input character c*c*, we need to add this character to the word being formed currently, i.e. to \text{cur\\_sent}cur\_sent. Then, we need to traverse in the current trie till all the characters in the current word, \text{cur\\_sent}cur\_sent, have been exhausted.

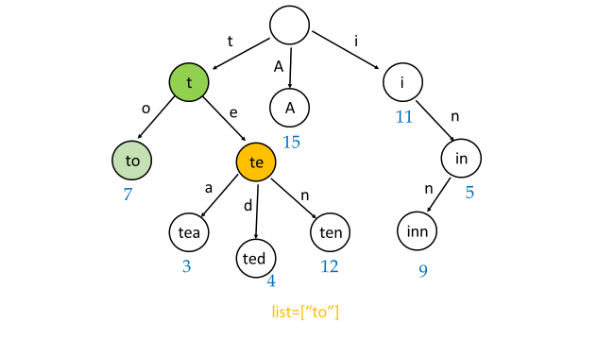
From this point onwards, we traverse all the branches possible in the Trie, put the sentences/words formed by these branches to a list*list* along with their corresponding number of occurences, and find the best 3 out of them similar to the last approach. The following animation shows a typical illustration.

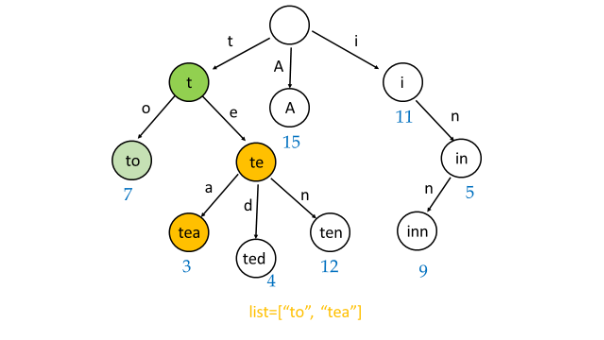


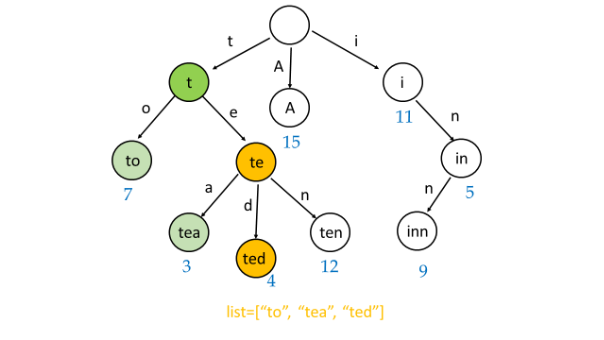


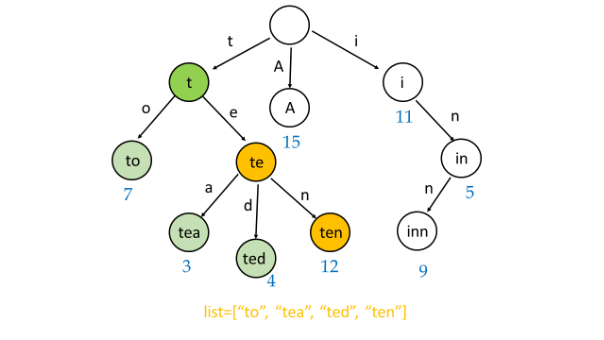


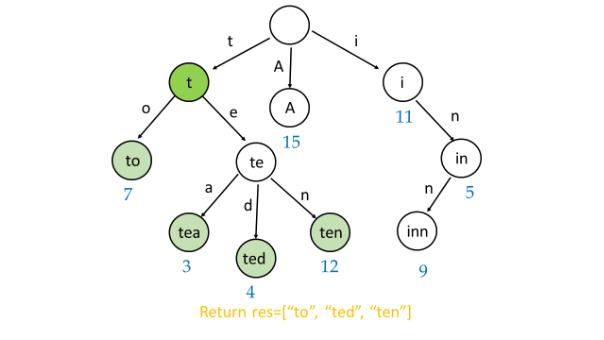


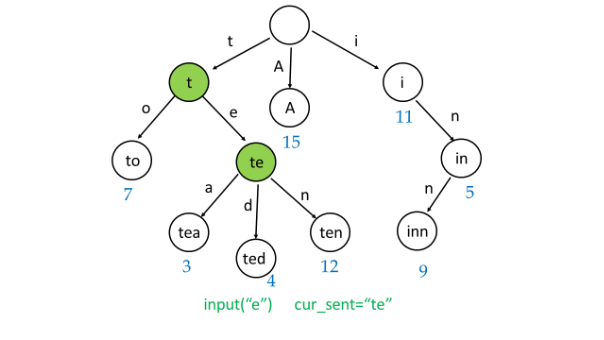


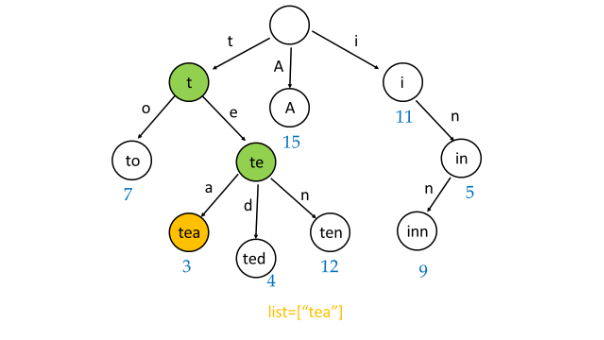


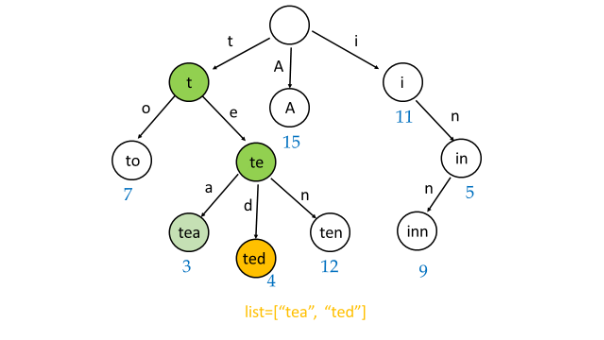


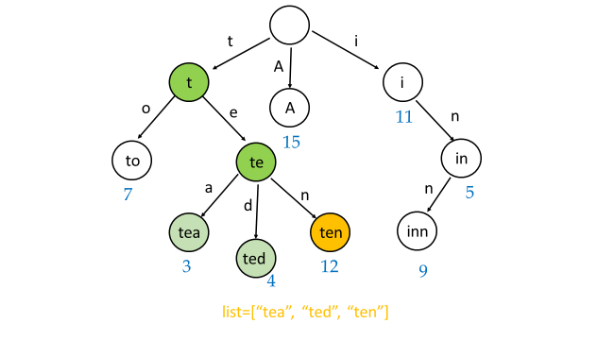


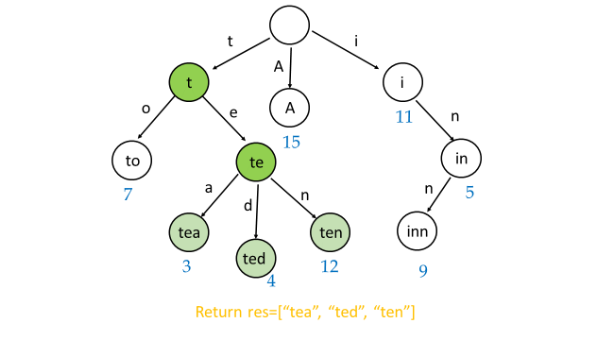












|  |
| --- |
| class Node {  String sentence;  int times;  Node(String st, int t) {  sentence = st;  times = t;  }  }  class Trie {  int times;  Trie[] branches = new Trie[27];  }  class AutocompleteSystem {  private Trie root;  private String cur\_sent = "";  public AutocompleteSystem(String[] sentences, int[] times) {  root = new Trie();  for (int i = 0; i < sentences.length; i++) {  insert(root, sentences[i], times[i]);  }  }  private int toInt(char c) {  return c == ' ' ? 26 : c - 'a';  }  private void insert(Trie t, String s, int times) {  for (int i = 0; i < s.length(); i++) {  if (t.branches[toInt(s.charAt(i))] == null) {  t.branches[toInt(s.charAt(i))] = new Trie();  }  t = t.branches[toInt(s.charAt(i))];  }  t.times += times;  }  private List<Node> lookup(Trie t, String s) {  List<Node> list = new ArrayList<>();  for (int i = 0; i < s.length(); i++) {  if (t.branches[toInt(s.charAt(i))] == null) {  return new ArrayList<>();  }  t = t.branches[toInt(s.charAt(i))];  }  traverse(s, t, list);  return list;  }  private void traverse(String s, Trie t, List<Node> list) {  if (t.times > 0) list.add(new Node(s, t.times));  for (char i = 'a'; i <= 'z'; i++) {  if (t.branches[i - 'a'] != null) {  traverse(s + i, t.branches[i - 'a'], list);  }  }  if (t.branches[26] != null) {  traverse(s + ' ', t.branches[26], list);  }  }  public List<String> input(char c) {  List<String> res = new ArrayList<>();  if (c == '#') {  insert(root, cur\_sent, 1);  cur\_sent = "";  } else {  cur\_sent += c;  List<Node> list = lookup(root, cur\_sent);  Collections.sort(  list,  (a, b) -> a.times == b.times ? a.sentence.compareTo(b.sentence) : b.times - a.times);  for (int i = 0; i < Math.min(3, list.size()); i++) res.add(list.get(i).sentence);  }  return res;  }  } |

**Performance Analysis**

* AutocompleteSystem() takes *O*(*k*∗*l*) time. We need to iterate over *l* sentences each of average length *k*, to create the trie for the given set of *sentences*.
* input() takes *O* (*p*+*q*+*m*log*m*) time. Here, *p* refers to the length of the sentence formed till now,  cur\_sent. *q* refers to the number of nodes in the trie considering the sentence formed till now as the root node. Again, we need to sort the *list* of length *m* indicating the options available for the hot sentences, which takes  *O*(*m*log*m*) time.

**Add and Search Word - Data structure design**

Design a data structure that supports adding new words and finding if a string matches any previously added string.

Implement the WordDictionary class:

* WordDictionary() Initializes the object.
* void addWord(word) Adds word to the data structure, it can be matched later.
* bool search(word) Returns true if there is any string in the data structure that matches word or false otherwise. word may contain dots '.' where dots can be matched with any letter.

**Example:**

**Input**

["WordDictionary","addWord","addWord","addWord","search","search","search","search"]

[[],["bad"],["dad"],["mad"],["pad"],["bad"],[".ad"],["b.."]]

**Output**

[null,null,null,null,false,true,true,true]

**Explanation**

WordDictionary wordDictionary = new WordDictionary();

wordDictionary.addWord("bad");

wordDictionary.addWord("dad");

wordDictionary.addWord("mad");

wordDictionary.search("pad"); // return False

wordDictionary.search("bad"); // return True

wordDictionary.search(".ad"); // return True

wordDictionary.search("b.."); // return True

**Constraints:**

* 1 <= word.length <= 500
* word in addWord consists lower-case English letters.
* word in search consist of  '.' or lower-case English letters.
* At most 50000 calls will be made to addWord and search.

You should be familiar with how a Trie works. If not, please work on this problem: [Implement Trie (Prefix Tree)](https://leetcode.com/problems/implement-trie-prefix-tree/) first.

## Solution

#### **Data Structure Trie**

This article introduces the data structure [trie](https://en.wikipedia.org/wiki/Trie). It could be pronounced in two different ways: as "tree" or "try". Trie which is also called a digital tree or a prefix tree is a kind of search ordered tree data structure mostly used for the efficient dynamic add/search operations with the strings.

Trie is widely used in real life: autocomplete search, spell checker, T9 predictive text, [IP routing (longest prefix matching)](https://www.researchgate.net/figure/An-example-routing-table-and-the-corresponding-binary-trie-built-from-it_fig3_4236637), [some GCC containers](https://gcc.gnu.org/onlinedocs/libstdc++/ext/pb_ds/trie_based_containers.html).

Here is how it looks like

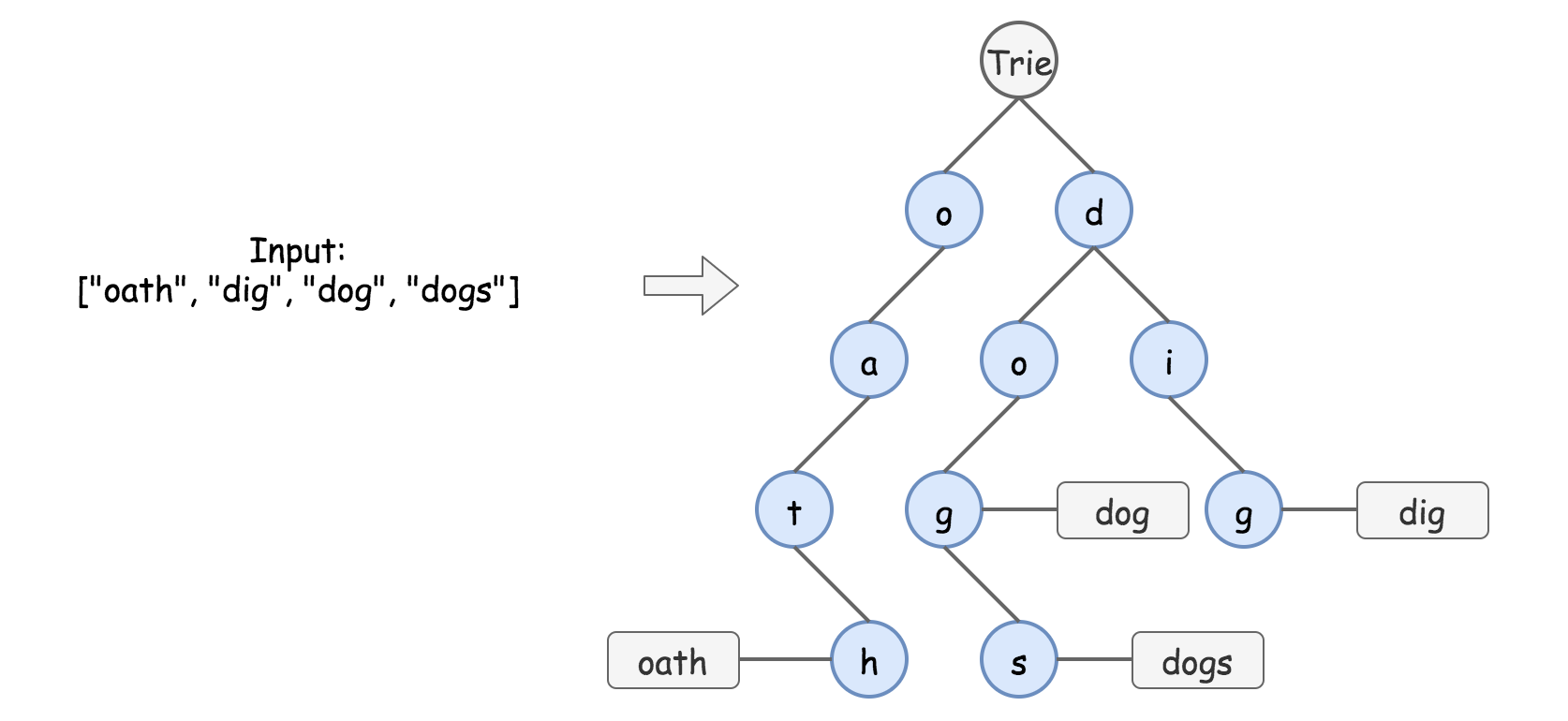


Figure 1. Data structure trie.

There are two main types of trie interview questions:

* [Standard Trie](https://en.wikipedia.org/wiki/Trie). Design a structure to dynamically add and search strings, for example
  + [Add and Search Word](https://leetcode.com/problems/design-add-and-search-words-data-structure/solution/).
  + [Word Search II](https://leetcode.com/articles/word-search-ii).
  + [Design Search Autocomplete System](https://leetcode.com/articles/design-search-autocomplete-system/).
* [Bitwise Trie](https://en.wikipedia.org/wiki/Trie#Bitwise_tries). Design a structure to dynamically add binary strings and compute maximum/minimum XOR/AND/etc, for example
  + [Maximum XOR of Two Number in an Array](https://leetcode.com/articles/maximum-xor-of-two-numbers-in-an-array/).

#### **Why Trie and not HashMap**

It's quite easy to write the solution using such data structures as hashmap or balanced tree.

|  |
| --- |
| class WordDictionary {  Map<Integer, Set<String>> d;  /\*\* Initialize your data structure here. \*/  public WordDictionary() {  d = new HashMap();  }  /\*\* Adds a word into the data structure. \*/  public void addWord(String word) {  int m = word.length();  if (!d.containsKey(m)) {  d.put(m, new HashSet());  }  d.get(m).add(word);  }  /\*\* Returns if the word is in the data structure. A word could contain the dot character '.' to represent any one letter. \*/  public boolean search(String word) {  int m = word.length();  if (d.containsKey(m)) {  for (String w : d.get(m)) {  int i = 0;  while ((i < m) && (w.charAt(i) == word.charAt(i) || word.charAt(i) == '.')) {  i++;  }  if (i == m) return true;  }  }  return false;  }  } |

This solution passes all leetcode test cases, and formally has O(*M*⋅*N*) time complexity for the search, where *M* is a length of the word to find, and *N* is the number of words. Although this solution is not efficient for the most important practical use cases:

* Finding all keys with a common prefix.
* Enumerating a dataset of strings in lexicographical order.
* Scaling for the large datasets. Once the hash table increases in size, there are a lot of hash collisions and the search time complexity could degrade to O(*N2*⋅*M*), where *N* is the number of the inserted keys.

Trie could use less space compared to hashmap when storing many keys with the same prefix. In this case, using trie has only  O(*M*⋅*N*) time complexity, where *M* is the key length, and *N* is the number of keys.

#### **Approach 1: Trie**

**How to Implement Trie: addWord function**

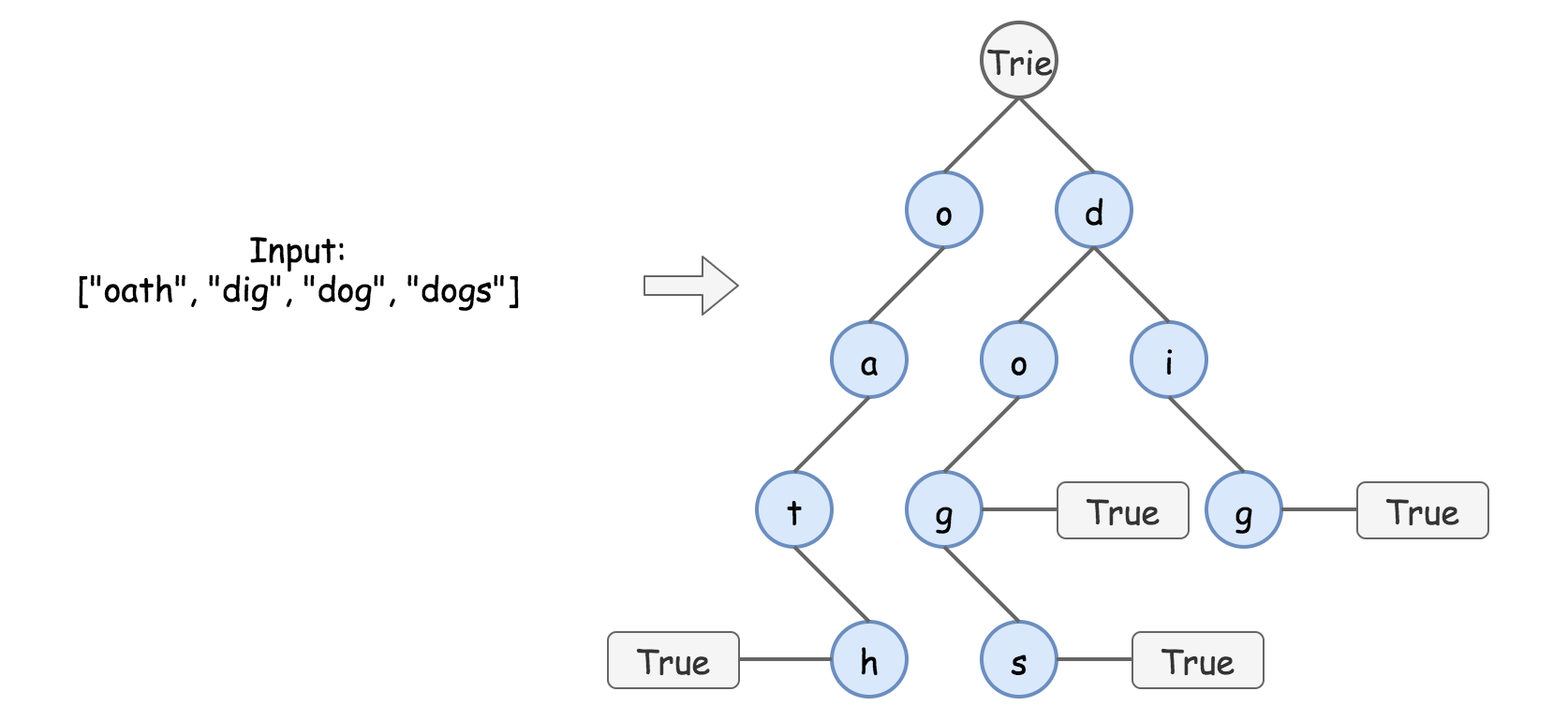
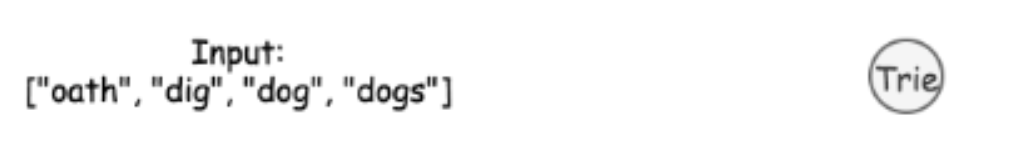
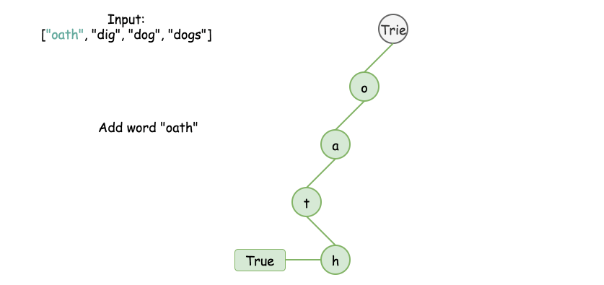


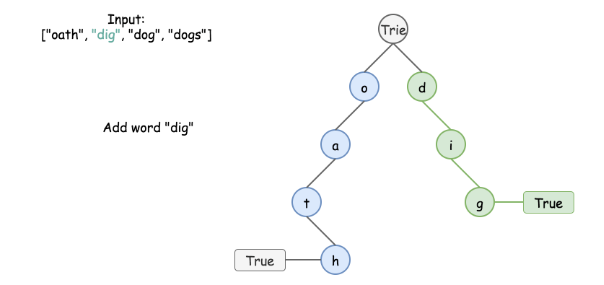
Figure 2. Trie implementation.

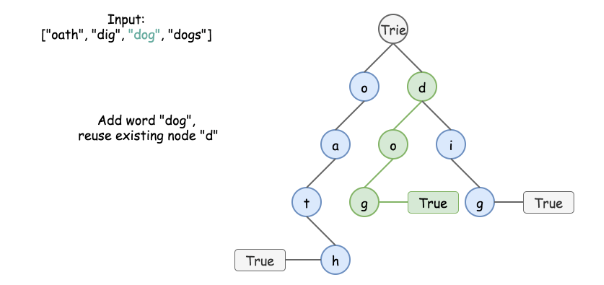
In trie, each path from the root to the "word" node represents one of the input words, for example, o -> a -> t -> h is "oath".

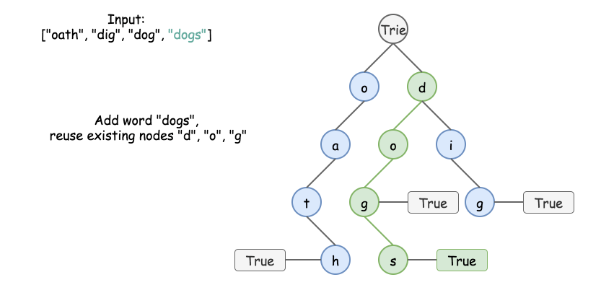
Trie implementation is pretty straightforward, it's basically nested hashmaps. At each step, one has to verify, if the child node to add is already present. If yes, just go one step down. If not, add it into the trie and then go one step down.













|  |
| --- |
| class TrieNode {  Map<Character, TrieNode> children = new HashMap();  boolean word = false;  public TrieNode() {}  }  class WordDictionary {  TrieNode trie;  /\*\* Initialize your data structure here. \*/  public WordDictionary() {  trie = new TrieNode();  }  /\*\* Adds a word into the data structure. \*/  public void addWord(String word) {  TrieNode node = trie;  for (char ch : word.toCharArray()) {  if (!node.children.containsKey(ch)) {  node.children.put(ch, new TrieNode());  }  node = node.children.get(ch);  }  node.word = true;  }  } |

**Complexity Analysis**

* Time complexity: O(*M*), where *M* is the key length. At each step, we either examine or create a node in the trie. That takes only *M* operations.
* Space complexity:  O(*M*). In the worst-case newly inserted key doesn't share a prefix with the keys already inserted in the trie. We have to add *M* new nodes, which takes  O(*M*) space.

**Search in Trie**

In the absence of '.' characters, the search would be as simple as addWord. Each key is represented in the trie as a path from the root to the internal node or leaf. We start from the root and go down in trie, checking character by character.

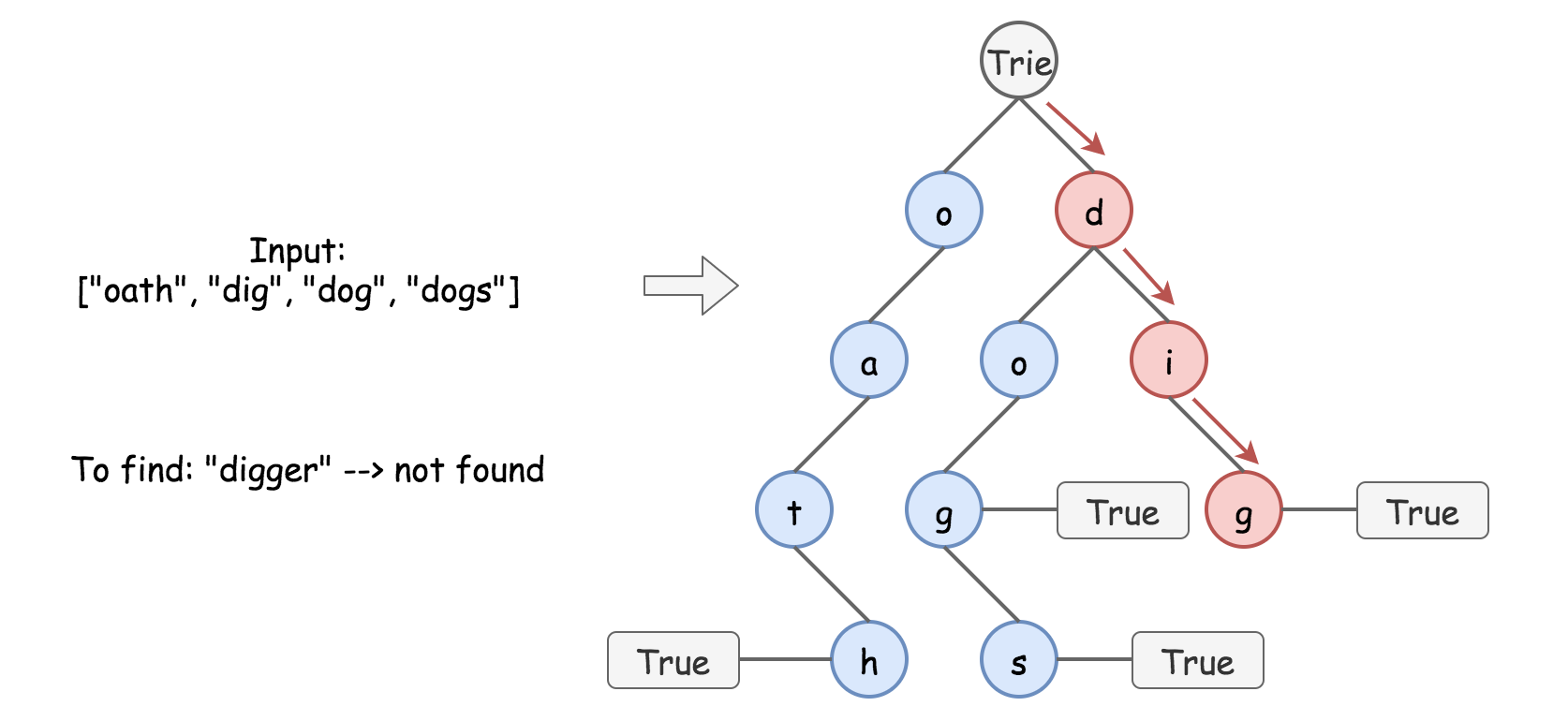


Figure 3. Search in trie.

The presence of '.' characters forces us to explore all possible paths at each . level.

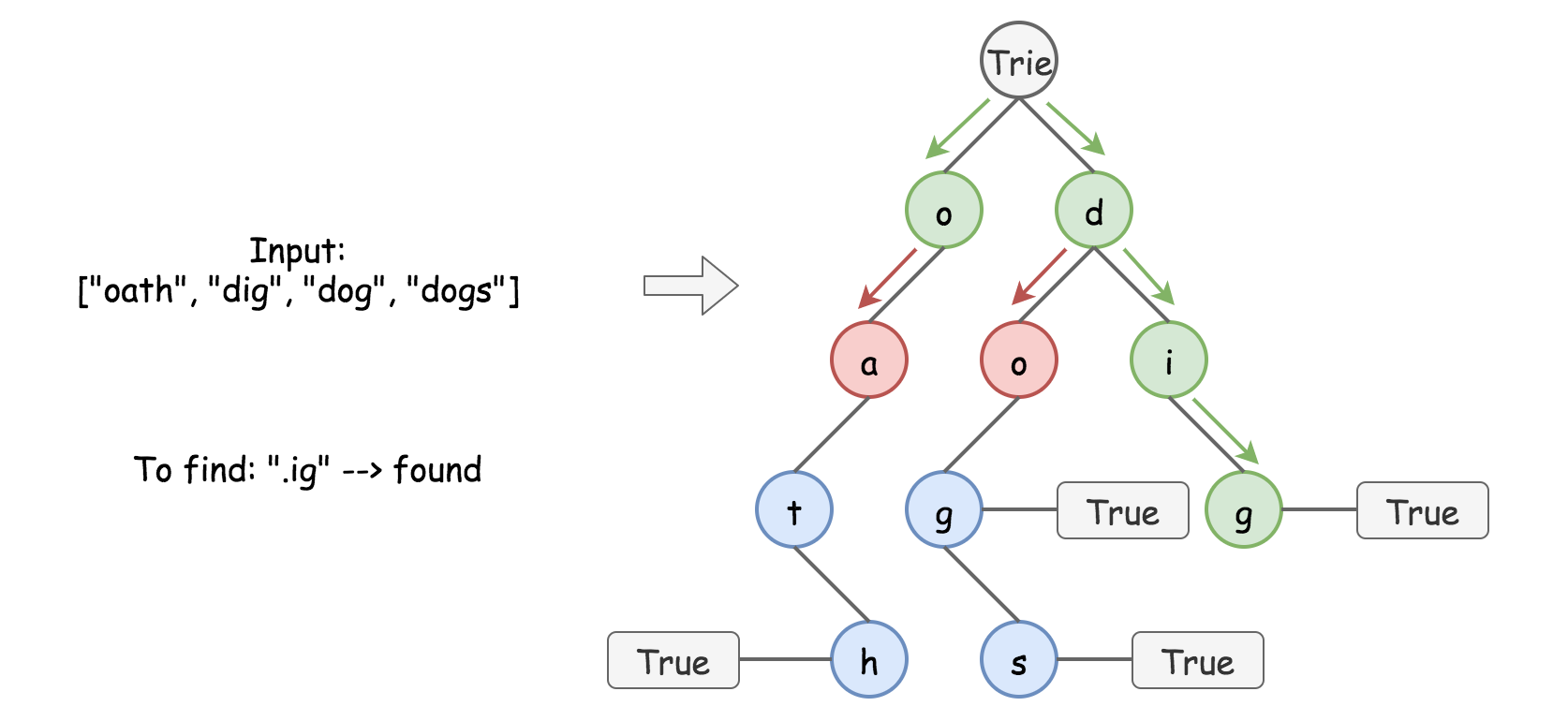


Figure 4. Search in trie.

|  |
| --- |
| /\*\* Returns if the word is in the node. \*/  public boolean searchInNode(String word, TrieNode node) {  for (int i = 0; i < word.length(); ++i) {  char ch = word.charAt(i);  if (!node.children.containsKey(ch)) {  // if the current character is '.'  // check all possible nodes at this level  if (ch == '.') {  for (char x : node.children.keySet()) {  TrieNode child = node.children.get(x);  if (searchInNode(word.substring(i + 1), child)) {  return true;  }  }  }  // if no nodes lead to answer  // or the current character != '.'  return false;  } else {  // if the character is found  // go down to the next level in trie  node = node.children.get(ch);  }  }  return node.word;  }  /\*\* Returns if the word is in the data structure. A word could contain the dot character '.' to represent any one letter. \*/  public boolean search(String word) {  return searchInNode(word, trie);  } |

**Complexity Analysis**

* Time complexity: O(*M*) for the "well-defined" words without dots, where *M* is the key length, and *N* is a number of keys, and  O(*N*⋅26M) for the "undefined" words. That corresponds to the worst-case situation of searching an undefined word  which is one character longer than all inserted keys.
* Space complexity:  O(1) for the search of "well-defined" words without dots, and up to O(*M*) for the "undefined" words, to keep the recursion stack.

**Implementation**

|  |
| --- |
| class TrieNode {  Map<Character, TrieNode> children = new HashMap();  boolean word = false;  public TrieNode() {}  }  class WordDictionary {  TrieNode trie;  /\*\* Initialize your data structure here. \*/  public WordDictionary() {  trie = new TrieNode();  }  /\*\* Adds a word into the data structure. \*/  public void addWord(String word) {  TrieNode node = trie;  for (char ch : word.toCharArray()) {  if (!node.children.containsKey(ch)) {  node.children.put(ch, new TrieNode());  }  node = node.children.get(ch);  }  node.word = true;  }  /\*\* Returns if the word is in the node. \*/  public boolean searchInNode(String word, TrieNode node) {  for (int i = 0; i < word.length(); ++i) {  char ch = word.charAt(i);  if (!node.children.containsKey(ch)) {  // if the current character is '.'  // check all possible nodes at this level  if (ch == '.') {  for (char x : node.children.keySet()) {  TrieNode child = node.children.get(x);  if (searchInNode(word.substring(i + 1), child)) {  return true;  }  }  }  // if no nodes lead to answer  // or the current character != '.'  return false;  } else {  // if the character is found  // go down to the next level in trie  node = node.children.get(ch);  }  }  return node.word;  }  /\*\* Returns if the word is in the data structure. A word could contain the dot character '.' to represent any one letter. \*/  public boolean search(String word) {  return searchInNode(word, trie);  }  } |

## Practical Application II

In the previous chapter, we practice with some typical Trie problems. However, the practical applications of Trie are not always so straightforward.

In this chapter, we provide some interesting practical applications for you to get to know more possibilities about Trie:

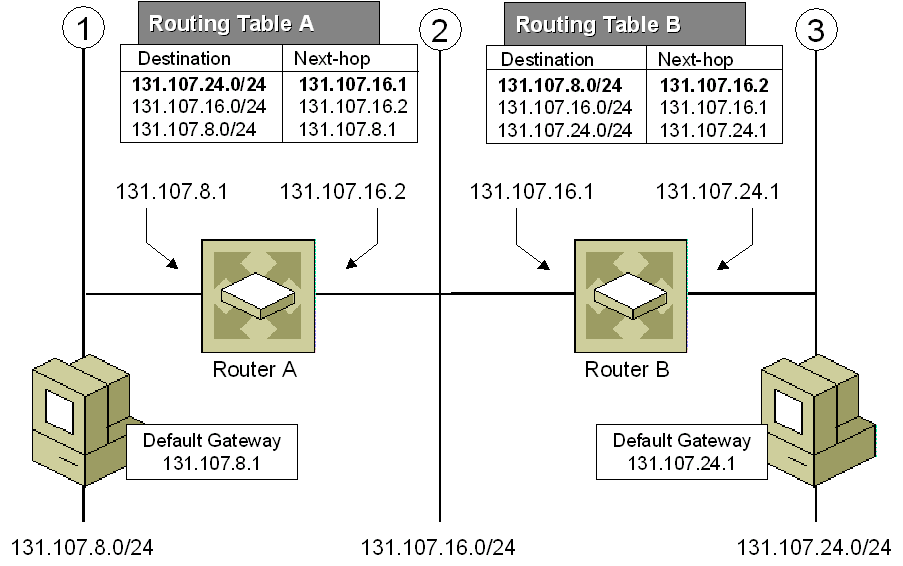
1. Accelerate DFS

Sometimes, we will use Trie to accelerate DFS especially when we do DFS for word games. We provide two word games ([Word Squares](https://leetcode.com/explore/learn/card/trie/149/practical-application-ii/1055/), [Word Search II](https://leetcode.com/explore/learn/card/trie/149/practical-application-ii/1056/)) for you in this chapter. Try to solve the problem by DFS first and then use Trie to improve the performance.

1. Store other Data Type

We use Trie to store strings in most cases but not always. [Maximum XOR of Two Numbers in an Array](https://leetcode.com/explore/learn/card/trie/149/practical-application-ii/1057/) is an interesting example.

There are also some other use cases, such as [IP routing (Longest prefix matching)](https://en.wikipedia.org/wiki/Longest_prefix_match).



**Maximum XOR of Two Numbers in an Array**

Given an integer array nums, return *the maximum result of nums[i] XOR nums[j]*, where 0 ≤ i ≤ j < n.

**Follow up:** Could you do this in O(n) runtime?

**Example 1:**

**Input:** nums = [3,10,5,25,2,8]

**Output:** 28

**Explanation:** The maximum result is 5 XOR 25 = 28.

**Example 2:**

**Input:** nums = [0]

**Output:** 0

**Example 3:**

**Input:** nums = [2,4]

**Output:** 6

**Example 4:**

**Input:** nums = [8,10,2]

**Output:** 10

**Example 5:**

**Input:** nums = [14,70,53,83,49,91,36,80,92,51,66,70]

**Output:** 127

**Constraints:**

* 1 <= nums.length <= 2 \* 104
* 0 <= nums[i] <= 231 - 1

## Solution

#### **Overview**

Requirements are to have O(*N*) time complexity, and we'll discuss here two standard approaches to achieve that complexity.

1. Bitwise Prefixes in HashSet.
2. Bitwise Prefixes in Trie.

The idea behind both solutions is the same: to convert all numbers into the binary form, and to construct the maximum XOR bit by bit, starting from the leftmost one. The difference is in the data structure used to store unique bitwise prefixes, i.e. the first ith bits.

The first approach works faster on the given testcase set, but the second one is standard, more simple, and easily generalised for more complex problems like Find maximum subarray XOR in a given array.

**Prerequisites**

XOR of zero and a bit results in that bit

0⊕*x*=*x*

XOR of two equal bits (even if they are zeros) results in a zero

*x*⊕*x*=0

#### **Approach 1: Bitwise Prefixes in HashSet**

Let's start from rewriting all numbers [3, 10, 5, 25, 2, 8] in binary from

23=(00011)2​

10=(01010)2​

5=(00101)2​

25=(11001)2​

2=(00010)2​

8=(01000)2​

To simplify the work with prefixes, better to use the same number of bits *L* for all the numbers. It's enough to take *L* equal to the length of the max number in the binary representation.

Now let's construct the max XOR starting from the leftmost bit. The absolute maximum one could have with *L*=5 bits here is (11111)2​. So let's check bit by bit:

* Could we have the leftmost bit for XOR to be equal to 1-bit, i.e. max XOR to be equal to (1∗∗∗∗)2​?

Yes, for that it's enough to pair 25=(11001)2​ with another number starting with the zero leftmost bit. So the max XOR is 2(1∗∗∗∗)2​.

* Next step. Could we have max XOR to be equal to (11∗∗∗)2​?

For that, let's consider all prefixes of length 2 and check if there is a pair of them, *p*1​ and *p*2​, such that its XOR is equal to 11: *p*1​⊕*p*2​==11

3=(00∗∗∗)2​

10=(01∗∗∗)2​

5=(00∗∗∗)2​

25=(11∗∗∗)2​

2=(00∗∗∗)2​

8=(01∗∗∗)2​

Yes, it's the case, for example, pair 5=(00∗∗∗)2​ and 25=(11∗∗∗)2​, or 2=(00∗∗∗)2​ and 25=(11∗∗∗)2​, or 3=(00∗∗∗)2​ and 25=(11∗∗∗)2​.

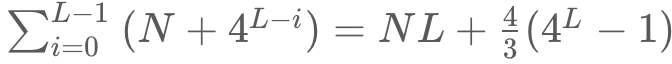
And so on, and so forth. The complexity remains linear. One has to perform *N* operations to compute prefixes, though the number of prefixes containing *L*−*i* bits could not be greater than 2*L*−*i*. Hence the check if XOR could have the ith bit to be equal to 1-bit takes 2*L*−*i*×2*L*−*i* operations.

**Algorithm**

* Compute the number of bits *L* to be used. It's a length of max number in binary representation.
* Initiate max\_xor = 0.
* Loop from *i*=*L*−1 down to *i*=0 (from the leftmost bit *L*−1 to the rightmost bit 0):
  + Left shift the max\_xor to free the next bit.
  + Initiate variable curr\_xor = max\_xor | 1 by setting 1 in the rightmost bit of max\_xor. Now let's check if curr\_xor could be done using available prefixes.
  + Compute all possible prefixes of length *L*−*i* by iterating over nums.
    - Put in the hashset prefixes the prefix of the current number of the length *L*−*i*: num >> i.
  + Iterate over all prefixes and check if curr\_xor could be done using two of them: p1^p2 == curr\_xor. Using self-inverse property of XOR p1^p2^p2 = p1, one could rewrite it as p1 == curr\_xor^p2 and simply check for each p if curr\_xor^p is in prefixes. If so, set max\_xor to be equal to curr\_xor, i.e. set 1-bit in the rightmost bit. Otherwise, let max\_xor keep 0-bit in the rightmost bit.
* Return max\_xor.

|  |
| --- |
| class Solution {  public int findMaximumXOR(int[] nums) {  int maxNum = nums[0];  for(int num : nums) maxNum = Math.max(maxNum, num);  // length of max number in a binary representation  int L = (Integer.toBinaryString(maxNum)).length();  int maxXor = 0, currXor;  Set<Integer> prefixes = new HashSet<>();  for(int i = L - 1; i > -1; --i) {  // go to the next bit by the left shift  maxXor <<= 1;  // set 1 in the smallest bit  currXor = maxXor | 1;  prefixes.clear();  // compute all possible prefixes  // of length (L - i) in binary representation  for(int num: nums) prefixes.add(num >> i);  // Update maxXor, if two of these prefixes could result in currXor.  // Check if p1^p2 == currXor, i.e. p1 == currXor^p2.  for(int p: prefixes) {  if (prefixes.contains(currXor^p)) {  maxXor = currXor;  break;  }  }  }  return maxXor;  }  } |

**Complexity Analysis**

* Time complexity: O(*N*). One has to perform *N* operations to compute prefixes, though the number of prefixes containing *L*−*i* bits is 2*L*−*i*. Check if XOR could have the i*th* bit to be equal to 1-bit takes  2*L*−*i*×2*L*−*i* operations. Altogether that results in  operations, that means O(*N*) time complexity.
* Space complexity: O(1). One has to keep not more than *L* prefixes, and  *L*=1+[log2​*M*], where M is maximum number in nums.

#### **Approach 2: Bitwise Trie**

**Why HashSet is not a Good Structure to Store Prefixes**

Hashset structure, used to store the prefixes in Approach 1, doesn't provide the functionality to cut off some paths which don't lead to the solution.

For example, after two steps of max XOR computation  (11∗∗∗)2​ it's quite obvious that 25 should be paired with 00 prefix, i.e. with 2, 3, or 5.

3 =(00011)2​

10 =(01010)2​

5 = (00101)2​

25 = (11001)2​

2 = (00010)2​

8 = (01000)2​

Although for the third step we'll again compute all possible prefixes, including the ones for 10 and 8, even if it's quite obvious that they will not lead to the solution.

3 = (000∗∗)2​

10 = (010∗∗)2​

5 = (001∗∗)2​

25 = (110∗∗)2​

2 = (000∗∗)2​

8 = (010∗∗)2​

To cut these branches off, would be great to use some sort of tree structure.

**Bitwise Trie: What is it and How to Construct**

The standard way is to use [Bitwise Trie](https://en.wikipedia.org/wiki/Trie#Bitwise_tries). It's a special type of [Trie](https://leetcode.com/articles/word-search-ii/), which is used to store binary prefixes in an efficient way. There are plenty of real-life examples of bitwise trie usage, for example, [in GCC](https://gcc.gnu.org/onlinedocs/libstdc++/ext/pb_ds/trie_based_containers.html).

Let's start with Bitwise Trie for the array [3, 10, 5, 25, 2]

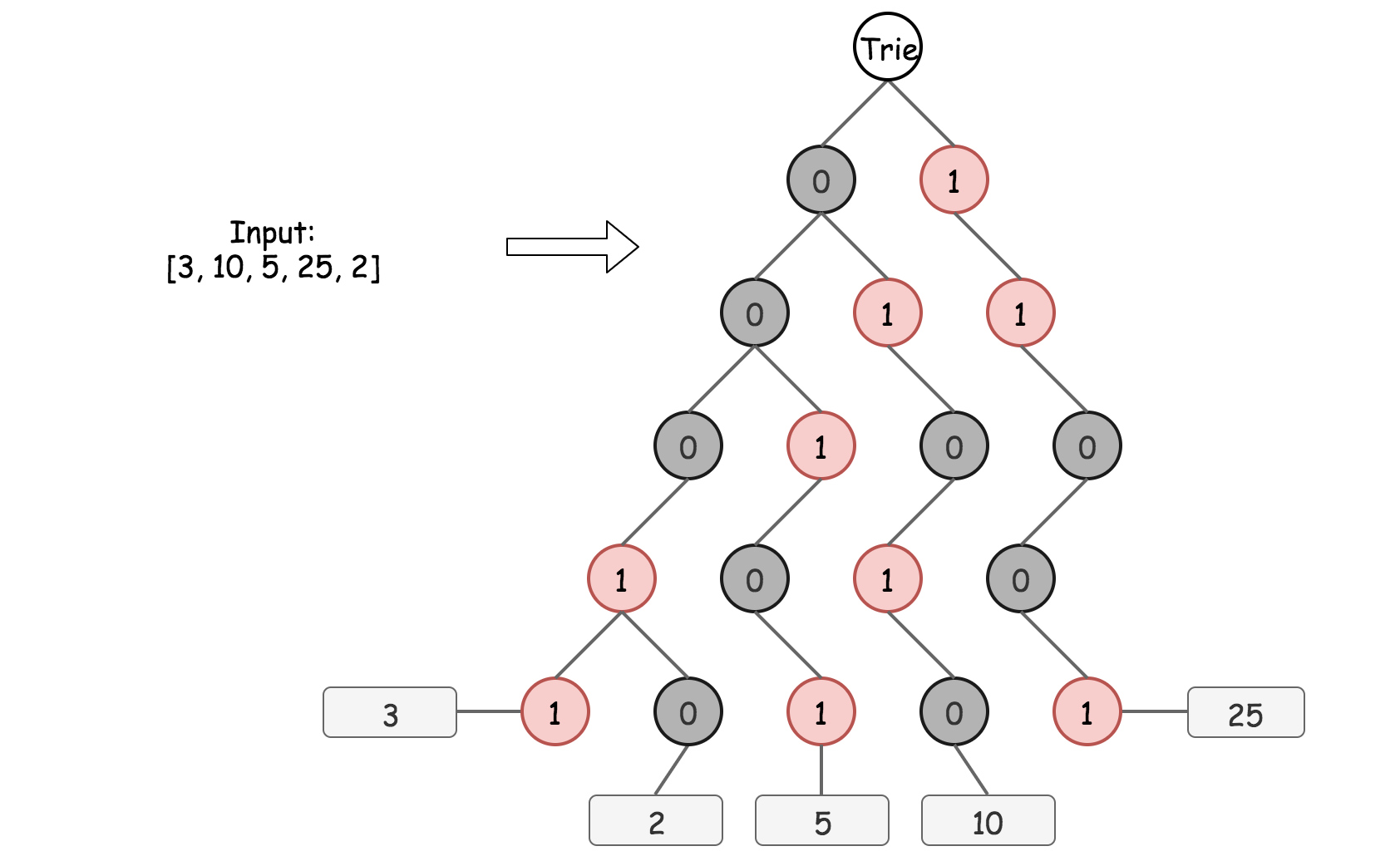
3 = (00011)2​

10 = (01010)2​

5 = (00101)2​

25 = (11001)2​

2 = (00010)2​



Each root -> leaf path in Bitwise Trie represents a binary form of a number in nums, for example, 0 -> 0 -> 0 -> 1 -> 1 is 3. As before, the same number of bits *L* is used for all numbers, and  *L*=1+[log2​*M*], where M is a maximum number in nums. The depth of Bitwise Trie is equal to *L* as well, and all leafs are on the same level.

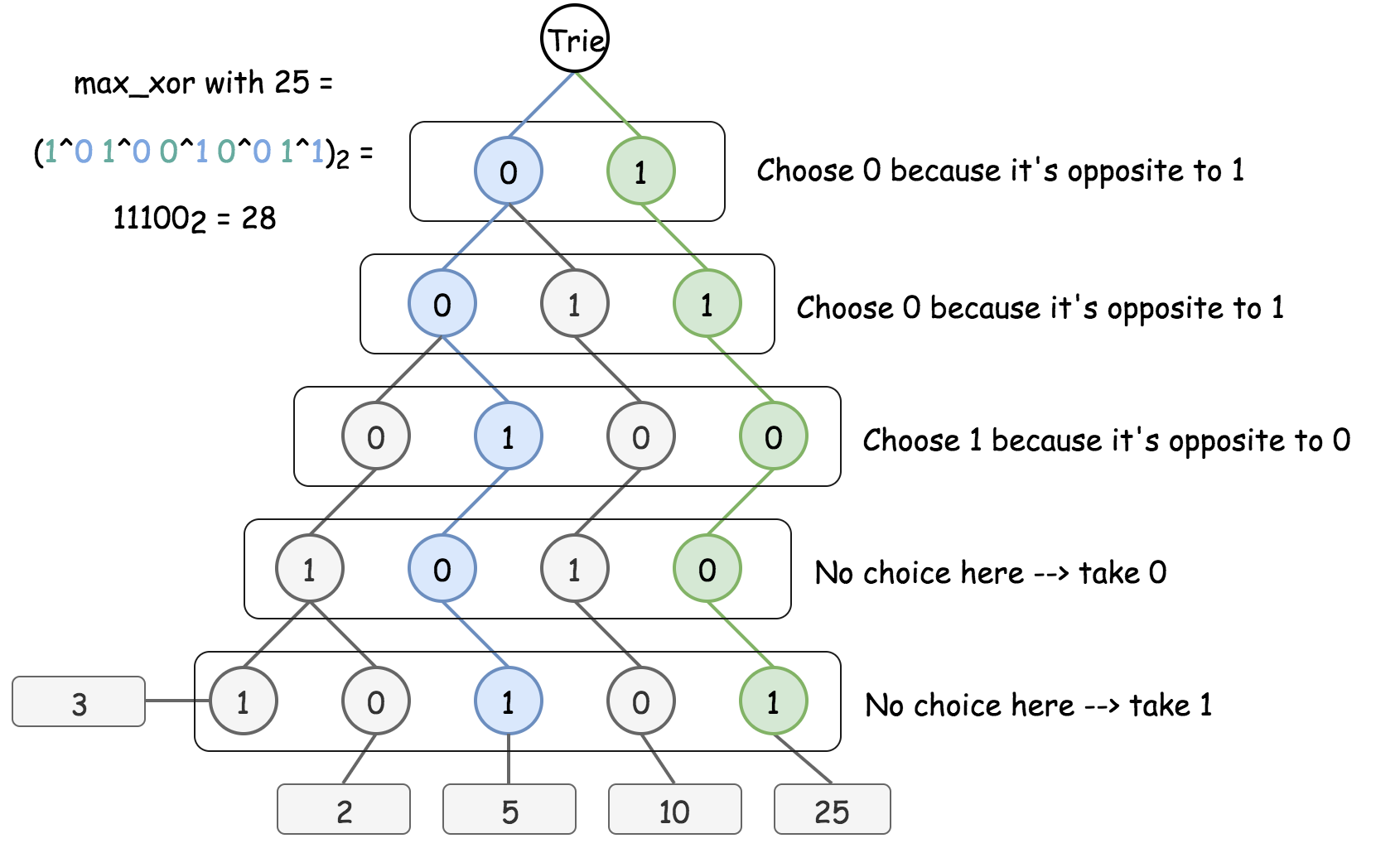
Bitwise Trie is a perfect way to see how different the binary forms of numbers are, for example, 3 and 2 share 4 bits of 5. The construction of Bitwise Trie is pretty straightforward, it's basically nested hashmaps. At each step one has to verify, if the child node to add (0 or 1) is already present. If yes, just go one step down. If not, add it into the Trie and then go one step down.

|  |
| --- |
| TrieNode trie = new TrieNode();  for (String num : strNums) {  TrieNode node = trie;  for (Character bit : num.toCharArray()) {  if (node.children.containsKey(bit)) {  node = node.children.get(bit);  } else {  TrieNode newNode = new TrieNode();  node.children.put(bit, newNode);  node = newNode;  }  }  } |

**Maximum XOR of a Given Number with All Numbers in Trie**

Now the Trie is constructed, so let's find the maximum XOR of a given number with all numbers that have been already inserted into Bitwise Trie.

To maximize XOR, the strategy is to choose the opposite bit at each step whenever it's possible. Step by step for 25 as a given number:



The implementation is also pretty simple:

* Try to go down to the opposite bit at each step if it's possible. Add 1-bit at the end of current XOR.
* If not, just go down to the same bit. Add 0-bit at the end of current XOR.

|  |
| --- |
| TrieNode trie = new TrieNode();  for (String num : strNums) {  TrieNode xorNode = trie;  int currXor = 0;  for (Character bit : num.toCharArray()) {  Character toggledBit = bit == '1' ? '0' : '1';  if (xorNode.children.containsKey(toggledBit)) {  currXor = (currXor << 1) | 1;  xorNode = xorNode.children.get(toggledBit);  } else {  currXor = currXor << 1;  xorNode = xorNode.children.get(bit);  }  }  } |

**Algorithm**

To summarise, now one could

* Insert a number into Bitwise Trie.
* Find maximum XOR of a given number with all numbers that have been inserted so far.

That's all one needs to solve the initial problem:

* Convert all numbers to the binary form.
* Add the numbers into Trie one by one and compute the maximum XOR of a number to add with all previously inserted. Update maximum XOR at each step.
* Return max\_xor.

|  |
| --- |
| class TrieNode {  HashMap<Character, TrieNode> children = new HashMap<Character, TrieNode>();  public TrieNode() {}  }  class Solution {  public int findMaximumXOR(int[] nums) {  // Compute length L of max number in a binary representation  int maxNum = nums[0];  for(int num : nums) maxNum = Math.max(maxNum, num);  int L = (Integer.toBinaryString(maxNum)).length();  // zero left-padding to ensure L bits for each number  int n = nums.length, bitmask = 1 << L;  String [] strNums = new String[n];  for(int i = 0; i < n; ++i) {  strNums[i] = Integer.toBinaryString(bitmask | nums[i]).substring(1);  }  TrieNode trie = new TrieNode();  int maxXor = 0;  for (String num : strNums) {  TrieNode node = trie, xorNode = trie;  int currXor = 0;  for (Character bit : num.toCharArray()) {  // insert new number in trie  if (node.children.containsKey(bit)) {  node = node.children.get(bit);  } else {  TrieNode newNode = new TrieNode();  node.children.put(bit, newNode);  node = newNode;  }  // compute max xor of that new number  // with all previously inserted  Character toggledBit = bit == '1' ? '0' : '1';  if (xorNode.children.containsKey(toggledBit)) {  currXor = (currXor << 1) | 1;  xorNode = xorNode.children.get(toggledBit);  } else {  currXor = currXor << 1;  xorNode = xorNode.children.get(bit);  }  }  maxXor = Math.max(maxXor, currXor);  }  return maxXor;  }  } |

**Complexity Analysis**

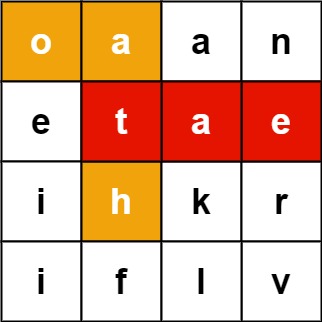
* Time complexity : O(*N*). It takes O(*L*) to insert a number in Trie, and O(*L*) to find the max XOR of the given number with all already inserted ones. *L*=1+[log2​*M*] is defined by the maximum number in the array and could be considered as a constant here. Hence the overall time complexity is  O(*N*).
* Space complexity :  O(1), since one needs at maximum O(2*L*)=O(*M*) space to keep Trie, and L and M could be considered as constants here because of input limitations.

**Word Search II**

Given an m x n board of characters and a list of strings words, return *all words on the board*.

Each word must be constructed from letters of sequentially adjacent cells, where **adjacent cells** are horizontally or vertically neighboring. The same letter cell may not be used more than once in a word.

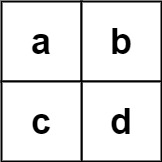
**Example 1:**



**Input:** board = [["o","a","a","n"],["e","t","a","e"],["i","h","k","r"],["i","f","l","v"]], words = ["oath","pea","eat","rain"]

**Output:** ["eat","oath"]

**Example 2:**



**Input:** board = [["a","b"],["c","d"]], words = ["abcb"]

**Output:** []

**Constraints:**

* m == board.length
* n == board[i].length
* 1 <= m, n <= 12
* board[i][j] is a lowercase English letter.
* 1 <= words.length <= 3 \* 104
* 1 <= words[i].length <= 10
* words[i] consists of lowercase English letters.
* All the strings of words are unique.

   Hide Hint #1

You would need to optimize your backtracking to pass the larger test. Could you stop backtracking earlier?

   Hide Hint #2

If the current candidate does not exist in all words' prefix, you could stop backtracking immediately. What kind of data structure could answer such query efficiently? Does a hash table work? Why or why not? How about a Trie? If you would like to learn how to implement a basic trie, please work on this problem: [Implement Trie (Prefix Tree)](https://leetcode.com/problems/implement-trie-prefix-tree/) first.

## Solution

#### **Approach 1: Backtracking with Trie**

**Intuition**

The problem is actually a simplified crossword puzzle game, where the word solutions have been given on the board embedded with some noise letters. All we need to to do is to cross them out.

Intuitively, in order to cross out all potential words, the overall strategy would be to iterate the cell one by one, and from each cell we walk along its neighbors in four potential directions to find matched words. While wandering around the board, we would stop the exploration when we know it would not lead to the discovery of new words.

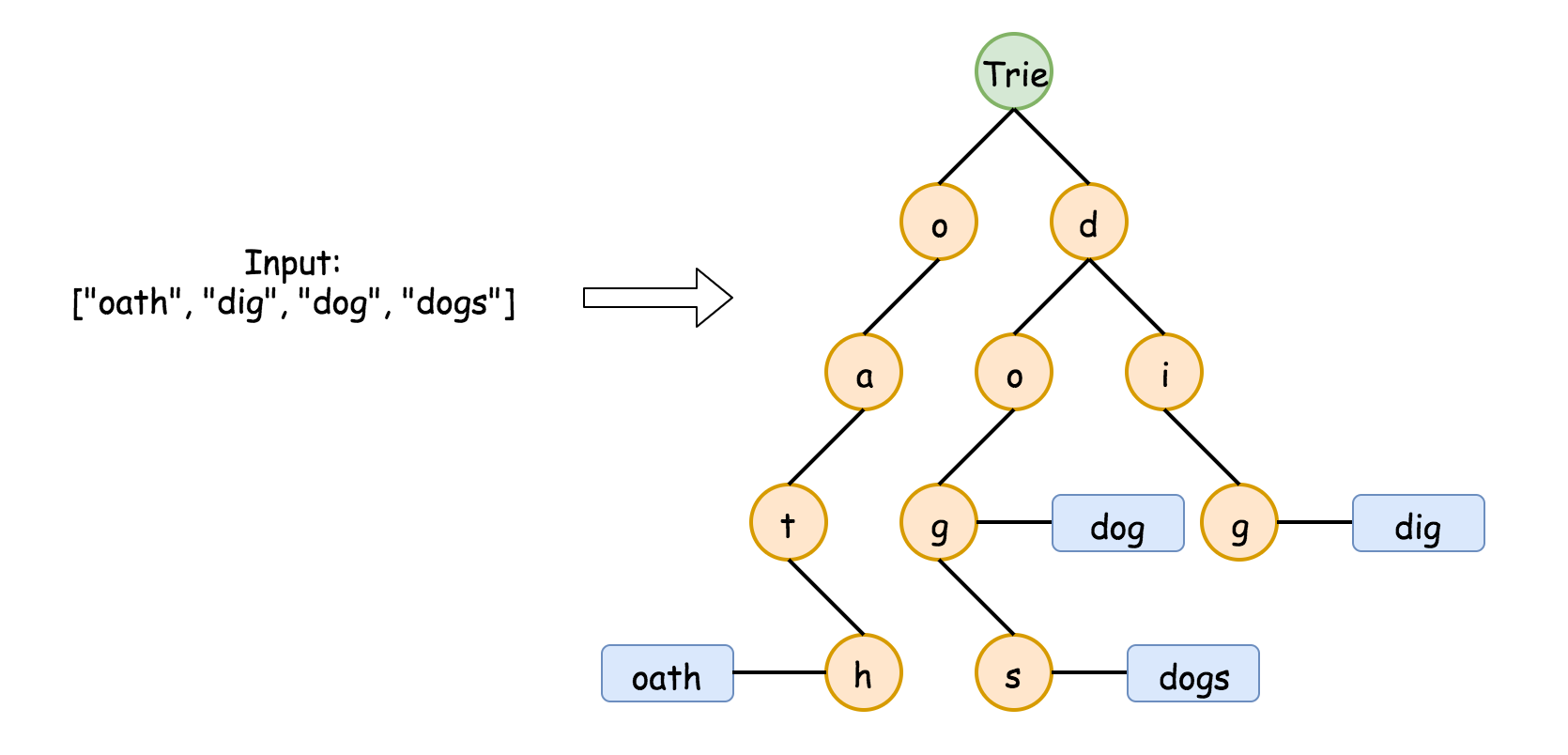
One might have guessed the paradigm that we would use for this problem. Yes, it is **backtracking**, which would be the backbone of the solution. It is fairly simply to construct a solution of backtracking. One could even follow a template given in our [Explore card of Recursion II](https://leetcode.com/explore/learn/card/recursion-ii/472/backtracking/2793/).

The key of the solution lies on how we find the matches of word from the dictionary. Intuitively, one might resort to the hashset data structure (e.g. set() in Python). This could work.

However, during the backtracking process, one would encounter more often the need to tell if there exists any word that contains certain prefix, rather than if a string exists as a word in the dictionary. Because if we know that there does not exist any match of word in the dictionary for a given prefix, then we would not need to further explore certain direction. And this, would greatly reduce the exploration space, therefore improve the performance of the backtracking algorithm.

The capability of finding matching prefix is where the data structure called [Trie](https://leetcode.com/explore/learn/card/trie/) would shine, comparing the hashset data structure. Not only can Trie tell the membership of a word, but also it can instantly find the words that share a given prefix. As it turns out, the choice of data structure (Trie VS. hashset), which could end with a solution that ranks either the top 5\%5% or the bottom 5\%5%.

Here we show an example of Trie that is built from a list of words. As one can see from the following graph, at the node denoted d, we would know there are at least two words with the prefix d from the dictionary.



We have a problem about [implementing a Trie data structure](https://leetcode.com/problems/implement-trie-prefix-tree/). One can start with the Trie problem as warm up, and come back this problem later.

**Algorithm**

The overall workflow of the algorithm is intuitive, which consists of a loop over each cell in the board and a recursive function call starting from the cell. Here is the skeleton of the algorithm.

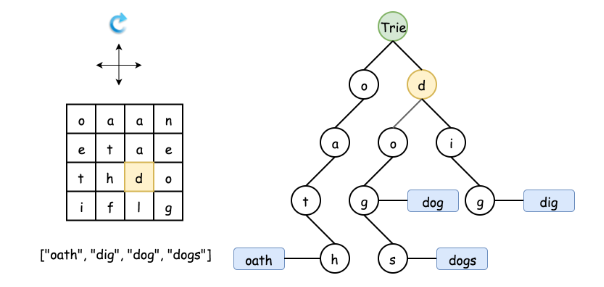
* We build a Trie out of the words in the dictionary, which would be used for the matching process later.
* Starting from each cell, we start the backtracking exploration (i.e. backtracking(cell)), if there exists any word in the dictionary that starts with the letter in the cell.
* During the recursive function call backtracking(cell), we explore the neighbor cells (i.e. neighborCell) around the current cell for the next recursive call backtracking(neighborCell). At each call, we check if the sequence of letters that we traverse so far matches any word in the dictionary, with the help of the Trie data structure that we built at the beginning.

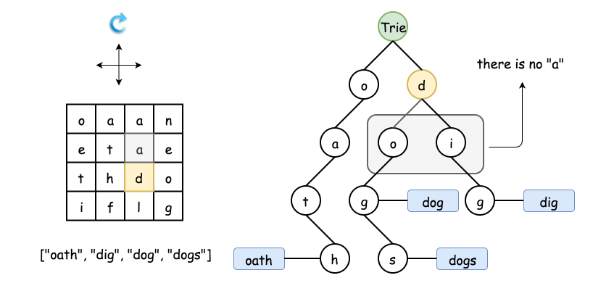
Here is an overall impression how the algorithm works. We give some sample implementation based on the rough idea above. And later, we detail some optimization that one could further apply to the algorithm.

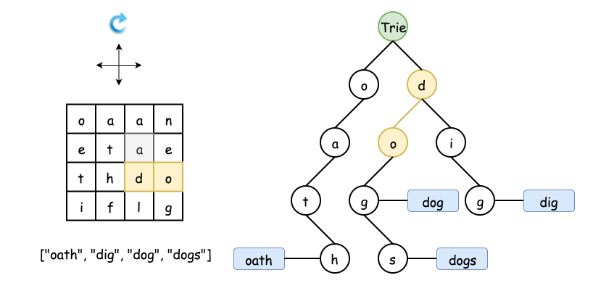


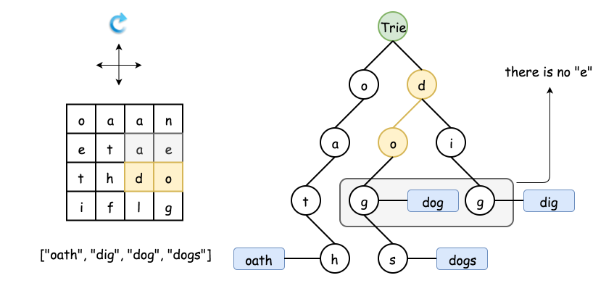
|  |
| --- |
| class TrieNode {  HashMap<Character, TrieNode> children = new HashMap<Character, TrieNode>();  String word = null;  public TrieNode() {}  }  class Solution {  char[][] \_board = null;  ArrayList<String> \_result = new ArrayList<String>();  public List<String> findWords(char[][] board, String[] words) {  // Step 1). Construct the Trie  TrieNode root = new TrieNode();  for (String word : words) {  TrieNode node = root;  for (Character letter : word.toCharArray()) {  if (node.children.containsKey(letter)) {  node = node.children.get(letter);  } else {  TrieNode newNode = new TrieNode();  node.children.put(letter, newNode);  node = newNode;  }  }  node.word = word; // store words in Trie  }  this.\_board = board;  // Step 2). Backtracking starting for each cell in the board  for (int row = 0; row < board.length; ++row) {  for (int col = 0; col < board[row].length; ++col) {  if (root.children.containsKey(board[row][col])) {  backtracking(row, col, root);  }  }  }  return this.\_result;  }    private void backtracking(int row, int col, TrieNode parent) {  Character letter = this.\_board[row][col];  TrieNode currNode = parent.children.get(letter);  // check if there is any match  if (currNode.word != null) {  this.\_result.add(currNode.word);  currNode.word = null;  }  // mark the current letter before the EXPLORATION  this.\_board[row][col] = '#';  // explore neighbor cells in around-clock directions: up, right, down, left  int[] rowOffset = {-1, 0, 1, 0};  int[] colOffset = {0, 1, 0, -1};  for (int i = 0; i < 4; ++i) {  int newRow = row + rowOffset[i];  int newCol = col + colOffset[i];  if (newRow < 0 || newRow >= this.\_board.length || newCol < 0  || newCol >= this.\_board[0].length) {  continue;  }  if (currNode.children.containsKey(this.\_board[newRow][newCol])) {  backtracking(newRow, newCol, currNode);  }  }  // End of EXPLORATION, restore the original letter in the board.  this.\_board[row][col] = letter;  // Optimization: incrementally remove the leaf nodes  if (currNode.children.isEmpty()) {  parent.children.remove(letter);  }  }  } |

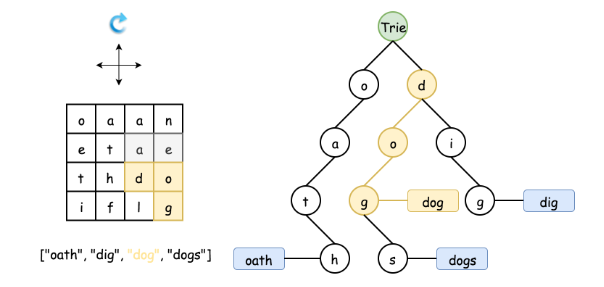
To better understand the backtracking process, we demonstrate how we find the match of dog along the Trie in the following animation.

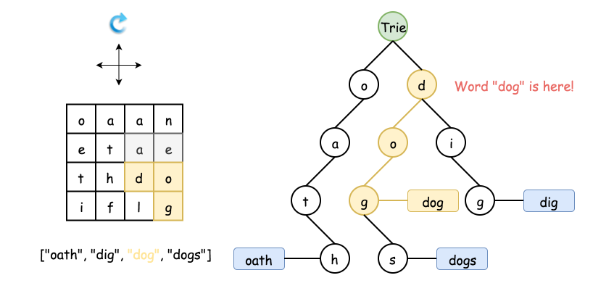












#### **Optimizations**

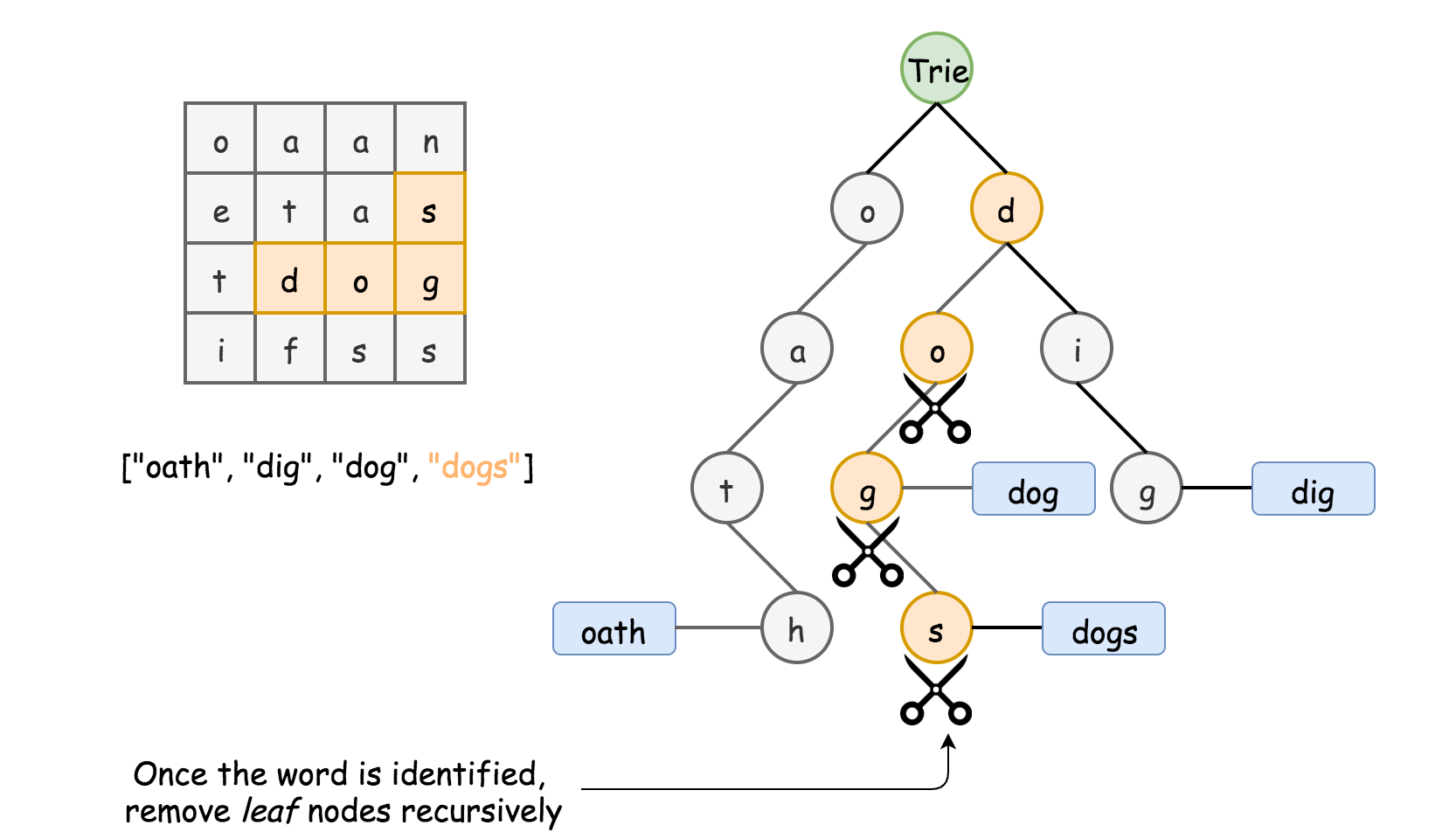
In the above implementation, we applied a few tricks to further speed up the running time, in addition to the application of the Trie data structure. In particular, the Python implementation could run faster than 98% of the submissions. We detail the tricks as follows, ordered by their significance.

Backtrack along the nodes in Trie.

* One could use Trie simply as a dictionary to quickly find the match of words and prefixes, i.e. at each step of backtracking, we start all over from the root of the Trie. This could work.
* However, a more efficient way would be to traverse the Trie together with the progress of backtracking, i.e. at each step of backtracking(TrieNode), the depth of the TrieNode corresponds to the length of the prefix that we've matched so far. This measure could lift your solution out of the bottom 5% of submissions.

Gradually ***prune*** the nodes in Trie during the backtracking.

* The idea is motivated by the fact that the time complexity of the overall algorithm sort of depends on the size of the Trie. For a leaf node in Trie, once we traverse it (i.e. find a matched word), we would no longer need to traverse it again. As a result, we could prune it out from the Trie.
* Gradually, those non-leaf nodes could become leaf nodes later, since we trim their children leaf nodes. In the extreme case, the Trie would become empty, once we find a match for all the words in the dictionary. This pruning measure could reduce up to 50\%50% of the running time for the test cases of the online judge.



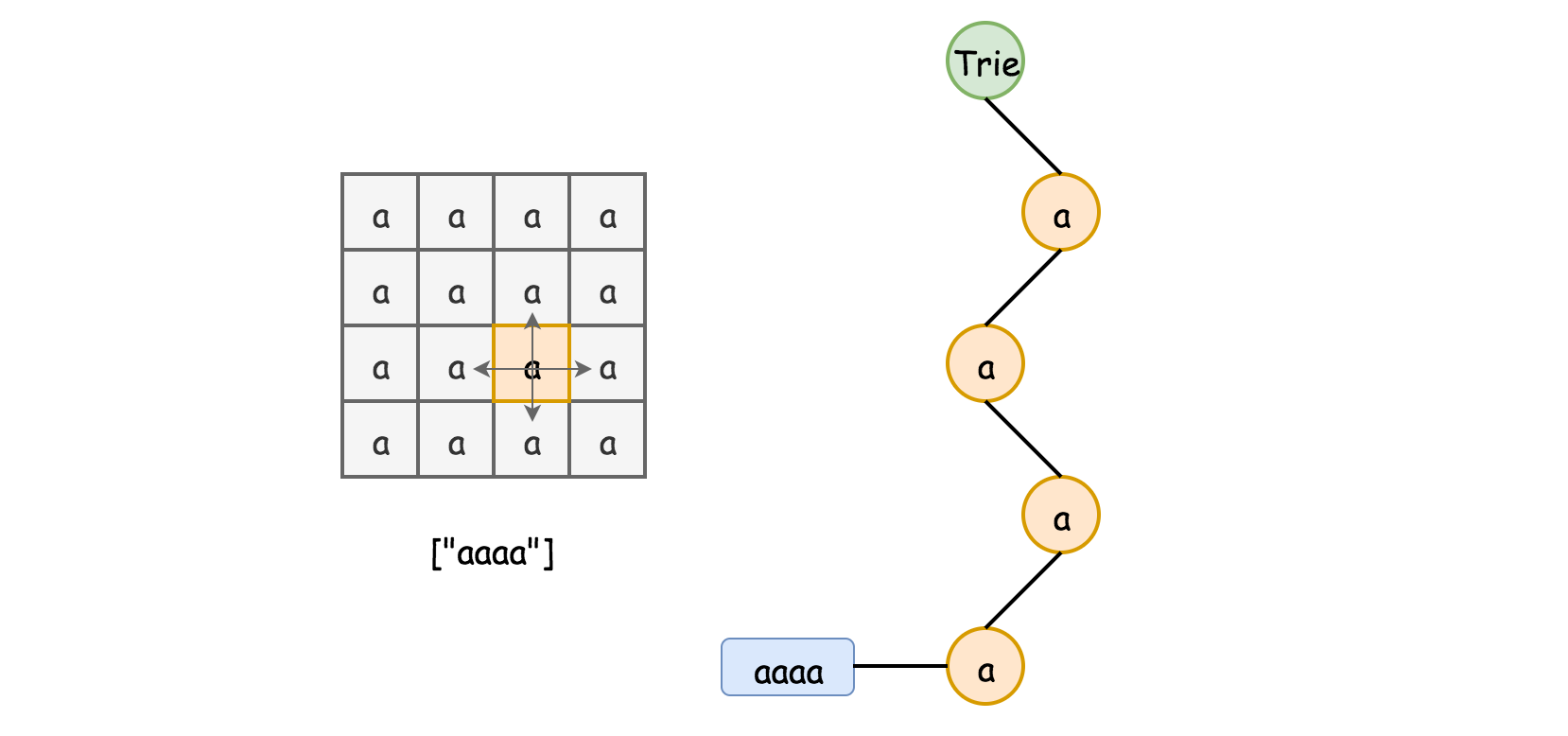
Keep words in the Trie.

* One might use a flag in the Trie node to indicate if the path to the current code match any word in the dictionary. It is not necessary to keep the words in the Trie.
* However, doing so could improve the performance of the algorithm a bit. One benefit is that one would not need to pass the prefix as the parameter in the backtracking() call. And this could speed up a bit the recursive call. Similarly, one does not need to reconstruct the matched word from the prefix, if we keep the words in Trie.

Remove the matched words from the Trie.

* In the problem, we are asked to return all the matched words, rather than the number of potential matches. Therefore, once we reach certain Trie node that contains a match of word, we could simply remove the match from the Trie.
* As a side benefit, we do not need to check if there is any duplicate in the result set. As a result, we could simply use a list instead of set to keep the results, which could speed up the solution a bit.

**Complexity**

* Time complexity: O(*M*(4⋅3*L*−1)), where *M* is the number of cells in the board and *L* is the maximum length of words.
  + It is tricky is calculate the exact number of steps that a backtracking algorithm would perform. We provide a upper bound of steps for the worst scenario for this problem. The algorithm loops over all the cells in the board, therefore we have *M* as a factor in the complexity formula. It then boils down to the maximum number of steps we would need for each starting cell (i.e. 4⋅3*L*−1).
  + Assume the maximum length of word is *L*, starting from a cell, initially we would have at most 4 directions to explore. Assume each direction is valid (i.e. worst case), during the following exploration, we have at most 3 neighbor cells (excluding the cell where we come from) to explore. As a result, we would traverse at most  4⋅3*L*−1 cells during the backtracking exploration.
  + One might wonder what the worst case scenario looks like. Well, here is an example. Imagine, each of the cells in the board contains the letter a, and the word dictionary contains a single word ['aaaa']. Voila. This is one of the worst scenarios that the algorithm would encounter. 
  + Note that, the above time complexity is estimated under the assumption that the Trie data structure would not change once built. If we apply the optimization trick to gradually remove the nodes in Trie, we could greatly improve the time complexity, since the cost of backtracking would reduced to zero once we match all the words in the dictionary, i.e. the Trie becomes empty.

* Space Complexity: O(*N*), where *N* is the total number of letters in the dictionary.
  + The main space consumed by the algorithm is the Trie data structure we build. In the worst case where there is no overlapping of prefixes among the words, the Trie would have as many nodes as the letters of all words. And optionally, one might keep a copy o

**Word Squares**

Given a set of words **(without duplicates)**, find all [word squares](https://en.wikipedia.org/wiki/Word_square) you can build from them.

A sequence of words forms a valid word square if the *k*th row and column read the exact same string, where 0 ≤ *k* < max(numRows, numColumns).

For example, the word sequence ["ball","area","lead","lady"] forms a word square because each word reads the same both horizontally and vertically.

b a l l

a r e a

l e a d

l a d y

**Note:**

1. There are at least 1 and at most 1000 words.
2. All words will have the exact same length.
3. Word length is at least 1 and at most 5.
4. Each word contains only lowercase English alphabet a-z.

**Example 1:**

**Input:**

["area","lead","wall","lady","ball"]

**Output:**

[

[ "wall",

"area",

"lead",

"lady"

],

[ "ball",

"area",

"lead",

"lady"

]

]

**Explanation:**

The output consists of two word squares. The order of output does not matter (just the order of words in each word square matters).

**Example 2:**

**Input:**

["abat","baba","atan","atal"]

**Output:**

[

[ "baba",

"abat",

"baba",

"atan"

],

[ "baba",

"abat",

"baba",

"atal"

]

]

**Explanation:**

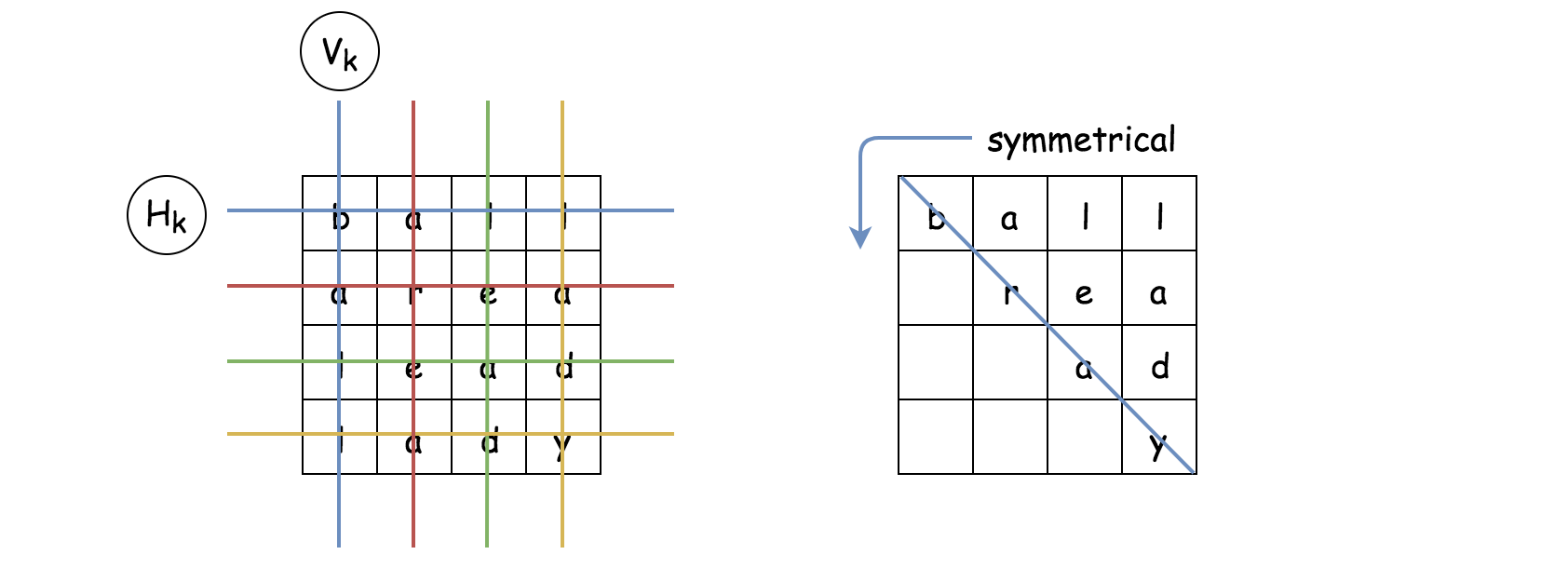
The output consists of two word squares. The order of output does not matter (just the order of words in each word square matters).

## Solution

Before diving into the solutions, it could be helpful to take a step back and clarify the requirements of the problem first.

Given a list of non-duplicate words, we are asked to construct all possible combinations of word squares. And here is the definition of ***word square***.

A sequence of words forms a valid word square, if and only if each string (*Hk*​) that is formed horizontally from the kth row equals to the string (*Vk*​) that is formed vertically from the kth column, i.e. Hk == Vk .



As we can see from the definition, for a word square with equal-sized row and column, the resulting letter matrix should be **symmetrical** across its diagonal.

In other words, if we know the upper-right part of the word square, we could infer its lower-left part, and vice versa. This symmetric property of the word square could also be interpreted as the **constraint** of the problem (as in the constraint programming), which could help us to narrow down the valid combinations.

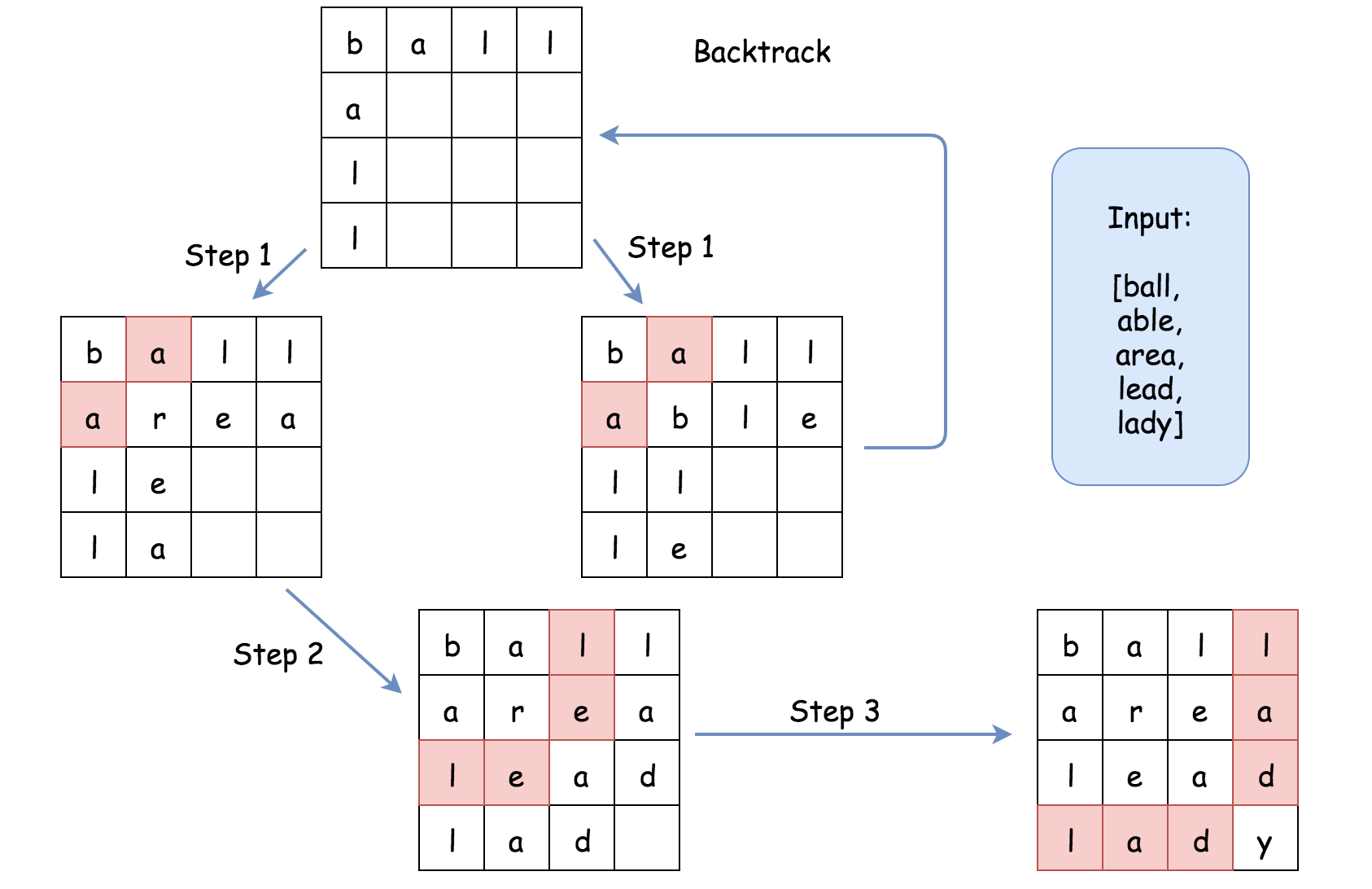
#### **Algorithm: Backtracking**

Given a list of words, we are asked to find a combination of words upon with we could construct a word square. The backbone of the algorithm to solve the above problem could be surprisingly simple.

The idea is that we construct the word square **row by row** from top to down. At each row, we simply do trial and error, i.e. we try with one word, if it does not meet the constraint then we try another one.

As one might notice, the above idea of the algorithm is actually known as [backtracking](https://leetcode.com/explore/learn/card/recursion-ii/472/backtracking/2654/), which is often associated with recursion and DFS (Depth-First Search) as well.

Let us illustrate the idea with an example. Given a list of words [ball, able, area, lead, lady], we should pile up 4 words together in order to build a word square.



* Let us start with the word ball as the first word in the word square, i.e. the word that we would put in the first row.
* We then move on to the second row. Given the symmetric property of the word square, we now know the letters that we should fill on the first column of the second row. In other words, we know that the word in the second row should start with the prefix a.
* Among the list of words, there are two words with prefix a (i.e. able, area). Both of them could be the candidates to fill the second row of the square. We then should try both of them in the next step.
* In the next step (1), let us fill the second row with the word able. Then we could move on to the third row. Again, due to the symmetric property, we know that the word in the third row should start with the prefix ll. Unfortunately, we do not find any word start with ll. As a result, we could no longer move forwards. We then abandon this path, and **backtrack** to the previous state (with the first row filled).
* As an alternative next step (1), we could try with the word area in the second row. Once we fill the second row, we would know that in the next row, the word to be filled should start with the prefix le. And this time, we find the candidate (i.e. lead).
* As a result, in the next step (2), we fill the third row with the word lead. So on and so forth.
* At the end, if one repeats the above steps with each word as the starting word, one would exhaust all the possibilities to construct a valid word square.

One could follow the [code template of backtracking](https://leetcode.com/explore/learn/card/recursion-ii/472/backtracking/2793/) from our Explore card to implement the above algorithm. Here is one example.

|  |
| --- |
| class Solution:  def wordSquares(self, words: List[str]) -> List[List[str]]:  self.words = words  self.N = len(words[0])  results = []  word\_squares = []  for word in words:  # try with every word as the starting word  word\_squares = [word]  self.backtracking(1, word\_squares, results)  return results  def backtracking(self, step, word\_squares, results):  if step == self.N:  results.append(word\_squares[:])  return  prefix = ''.join([word[step] for word in word\_squares])  # find out all words that start with the given prefix  for candidate in self.getWordsWithPrefix(prefix):  # iterate row by row  word\_squares.append(candidate)  self.backtracking(step+1, word\_squares, results)  word\_squares.pop()  def getWordsWithPrefix(self, prefix):  for word in self.words:  if word.startswith(prefix):  yield word |

At the first glance of the code, the problem does not seem to be as daunting as it is labeled. Actually if one could come up the skeleton of code in the interview, it would be fair to say that one has scored the interview.

The above implementation is correct and would pass most of the test cases in the online judge. However, it would run into Time Limit Exceeded exception for certain test cases with large input. But, there is no need for dismay, since we've already figured out the hard part of the algorithm. We just need to iron out the last bit of optimization which actually could be a followup question during the interview.

#### **Approach 1: Backtracking with HashTable**

**Intuition**

As one might notice in the above backtracking algorithm, the bottleneck lies in the function getWordsWithPrefix() which is to find all words with the given prefix. At each invocation of the function, we were iterating through the entire input list of words, which is of linear time complexity O(*N*).

One of the ideas to optimize the getWordsWithPrefix() function would be to process the words beforehand and to build a data structure that could speed up the lookup procedure later.

As one might recall, one of the data structures that provide a fast lookup operation is called ***hashtable*** or dictionary. We could simply build a hashtable with all possible prefixes as keys and the words that are associated with the prefix as the values in the table. Later, given the prefix, we should be able to list all the words with the given prefix in constant time O(1).

**Algorithm**

* We build upon the backtracking algorithm that we listed above, and tweak two parts.
* In the first part, we add a new function buildPrefixHashTable(words) to build a hashtable out of the input words.
* Then in the second part, in the function getWordsWithPrefix() we simply query the hashtable to retrieve all the words that possess the given prefix.

Here are some sample implementations. The idea is inspired by the post of [gabbu](https://leetcode.com/problems/word-squares/discuss/91360/3-Python-Solutions-with-very-detailed-explanations) in the discussion forum.

|  |
| --- |
| class Solution {  int N = 0;  String[] words = null;  HashMap<String, List<String>> prefixHashTable = null;  public List<List<String>> wordSquares(String[] words) {  this.words = words;  this.N = words[0].length();  List<List<String>> results = new ArrayList<List<String>>();  this.buildPrefixHashTable(words);  for (String word : words) {  LinkedList<String> wordSquares = new LinkedList<String>();  wordSquares.addLast(word);  this.backtracking(1, wordSquares, results);  }  return results;  }  protected void backtracking(int step, LinkedList<String> wordSquares,  List<List<String>> results) {  if (step == N) {  results.add((List<String>) wordSquares.clone());  return;  }  StringBuilder prefix = new StringBuilder();  for (String word : wordSquares) {  prefix.append(word.charAt(step));  }  for (String candidate : this.getWordsWithPrefix(prefix.toString())) {  wordSquares.addLast(candidate);  this.backtracking(step + 1, wordSquares, results);  wordSquares.removeLast();  }  }  protected void buildPrefixHashTable(String[] words) {  this.prefixHashTable = new HashMap<String, List<String>>();  for (String word : words) {  for (int i = 1; i < this.N; ++i) {  String prefix = word.substring(0, i);  List<String> wordList = this.prefixHashTable.get(prefix);  if (wordList == null) {  wordList = new ArrayList<String>();  wordList.add(word);  this.prefixHashTable.put(prefix, wordList);  } else {  wordList.add(word);  }  }  }  }  protected List<String> getWordsWithPrefix(String prefix) {  List<String> wordList = this.prefixHashTable.get(prefix);  return (wordList != null ? wordList : new ArrayList<String>());  }  } |

**Complexity Analysis**

* Time complexity:  O(*N*⋅26*L*), where *N* is the number of input words and *L* is the length of a single word.
  + It is tricky to calculate the exact number of operations in the backtracking algorithm. We know that the trace of the backtrack would form a n-ary tree. Therefore, the upper bound of the operations would be the total number of nodes in a full-blossom n-ary tree.
  + In our case, at any node of the trace, at maximum it could have 26 branches (*i.e.* 26 alphabet letters). Therefore, the maximum number of nodes in a 26-ary tree would be approximately 26*L*.
  + In the loop around the backtracking function, we enumerate the possibility of having each word as the starting word in the word square. As a result, in total the overall time complexity of the algorithm should be O(*N*⋅26*L*).
  + As large as the time complexity might appear, in reality, the actual trace of the backtracking is much smaller than its upper bound, thanks to the *constraint* checking (symmetric of word square) which greatly prunes the trace of the backtracking.
* Space Complexity:  where *N* is the number of words and L*L* is the length of a single word.
  + The first half of the space complexity (*i.e.* *N*⋅*L*) is the values in the hashtable, where we store *L* times all words in the hashtable.
  + The second half of the space complexity (*i.e.* *N*⋅*L/2*​) is the space took by the keys of the hashtable, which include all prefixes of all words.
  + In total, we could say that the overall space of the algorithm is proportional to the total words times the length of a single word.

Approach 2: Backtracking with Trie

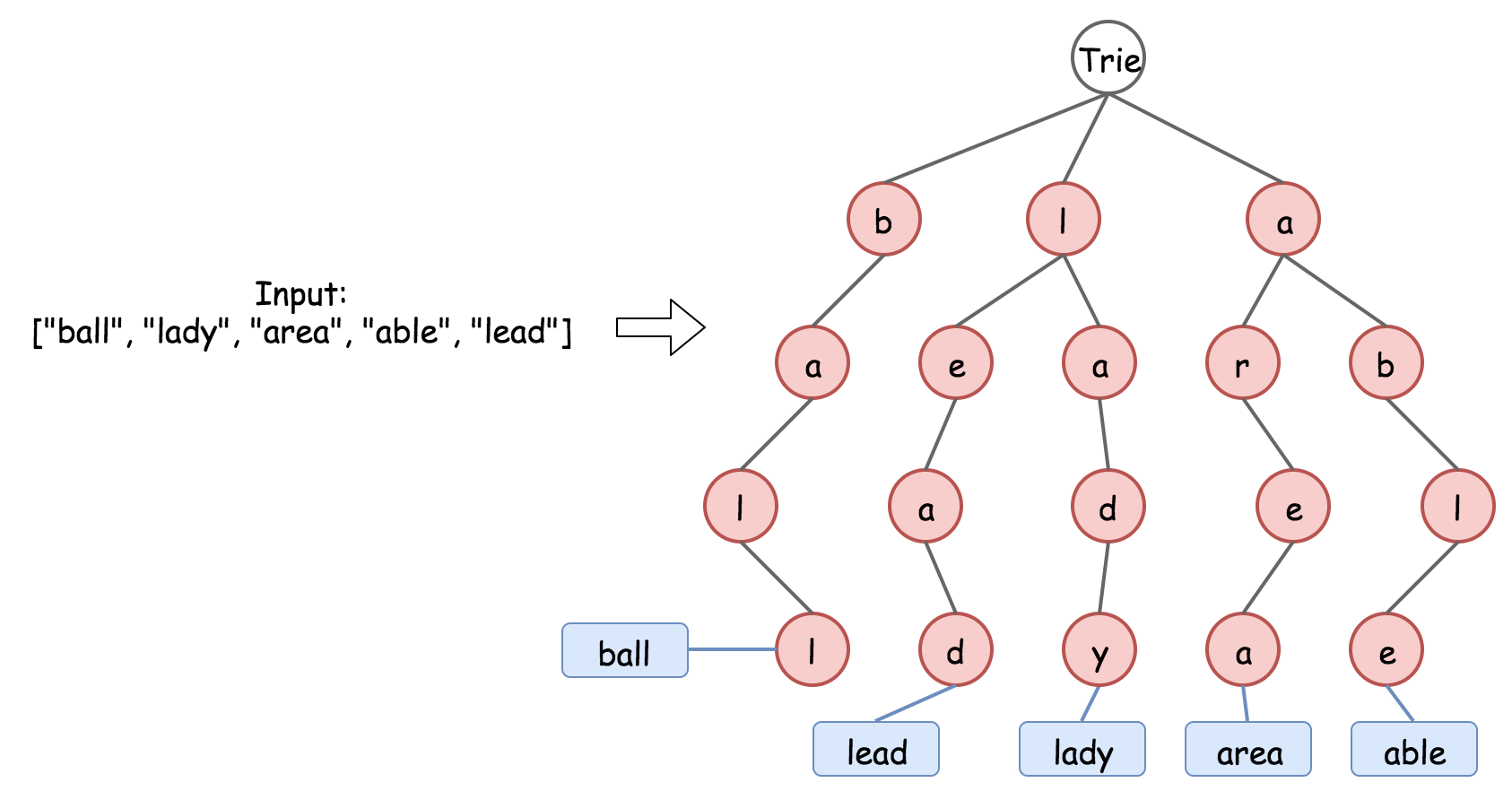
**Intuition**

Speaking about prefix, there is another data structure called ***Trie*** (also known as ***prefix tree***), which could find its use in this problem.

In the above approach, we have reduce the time complexity of retrieving a list of words with a given prefix from the linear O(*N*) to the constant time O(1). In exchange, we have to spend some extra space to store all the prefixes of each words.

The Trie data structure provides a *compact* and yet still *fast* way to retrieve words with a given prefix.

In the following graph, we show an example on how we can build a Trie from a list of words.



As we can see, basically Trie is a n-aray tree, where each node represents a character in a prefix. The Trie data structure is **compact** to store the prefixes, since it deduplicate the redundant prefixes, *e.g.* the prefixes of le and la would share one node. And yet it is still fast to retrieve words from the Trie. It takes O(*L*) to retrieve a word, where *L* is the length of the word, which is much faster than the brute-force enumeration.

**Algorithm**

* We build upon the backtracking algorithm that we listed above, and tweak two parts.
* In the first part, we add a new function buildTrie(words) to build a Trie out of the input words.
* Then in the second part, in the function getWordsWithPrefix(prefix) we simply query the Trie to retrieve all the words that possess the given prefix.

Here are some sample implementations. Note that, we tweak the Trie data structure a bit, in order to further optimize the time and space complexity.

* Instead of labeling the word at the leaf node of the Trie, we label the word at each node so that we don't need to perform a further traversal once we reach the last node in the prefix. This trick could help us with the time complexity.
* Instead of storing the actual words in the Trie, we keep only the index of the word, which could greatly save the space.

|  |
| --- |
| class TrieNode {  HashMap<Character, TrieNode> children = new HashMap<Character, TrieNode>();  List<Integer> wordList = new ArrayList<Integer>();  public TrieNode() {}  }  class Solution {  int N = 0;  String[] words = null;  TrieNode trie = null;  public List<List<String>> wordSquares(String[] words) {  this.words = words;  this.N = words[0].length();  List<List<String>> results = new ArrayList<List<String>>();  this.buildTrie(words);  for (String word : words) {  LinkedList<String> wordSquares = new LinkedList<String>();  wordSquares.addLast(word);  this.backtracking(1, wordSquares, results);  }  return results;  }  protected void backtracking(int step, LinkedList<String> wordSquares,  List<List<String>> results) {  if (step == N) {  results.add((List<String>) wordSquares.clone());  return;  }  StringBuilder prefix = new StringBuilder();  for (String word : wordSquares) {  prefix.append(word.charAt(step));  }  for (Integer wordIndex : this.getWordsWithPrefix(prefix.toString())) {  wordSquares.addLast(this.words[wordIndex]);  this.backtracking(step + 1, wordSquares, results);  wordSquares.removeLast();  }  }  protected void buildTrie(String[] words) {  this.trie = new TrieNode();  for (int wordIndex = 0; wordIndex < words.length; ++wordIndex) {  String word = words[wordIndex];  TrieNode node = this.trie;  for (Character letter : word.toCharArray()) {  if (node.children.containsKey(letter)) {  node = node.children.get(letter);  } else {  TrieNode newNode = new TrieNode();  node.children.put(letter, newNode);  node = newNode;  }  node.wordList.add(wordIndex);  }  }  }  protected List<Integer> getWordsWithPrefix(String prefix) {  TrieNode node = this.trie;  for (Character letter : prefix.toCharArray()) {  if (node.children.containsKey(letter)) {  node = node.children.get(letter);  } else {  // return an empty list.  return new ArrayList<Integer>();  }  }  return node.wordList;  }  } |

**Complexity Analysis**

* Time complexity:  O(*N*⋅26*L*⋅*L*), where *N* is the number of input words and *L* is the length of a single word.
  + Basically, the time complexity is same with the Approach #1 O(*N*⋅26*L*)), except that instead of the constant-time access for the function getWordsWithPrefix(prefix), we now need O(*L*).
* Space Complexity:  where *N* is the number of words and *L* is the length of a single word.
  + The first half of the space complexity (*i.e.*  *N*⋅*L*) is the word indice that we store in the Trie, where we store *L* times the index for each word.
  + The second half of the space complexity (*i.e.* *N*⋅L/2​) is the space took by the prefixes of all words. In the worst case, we have no overlapping among the prefixes.
  + Overall, this approach has the same space complexity as the previous approach. Yet, in running time, it would consume less memory thanks to all the optimization that we have done.

**Palindrome Pairs**

Given a list of **unique** words, return all the pairs of the ***distinct*** indices (i, j) in the given list, so that the concatenation of the two words words[i] + words[j] is a palindrome.

**Example 1:**

**Input:** words = ["abcd","dcba","lls","s","sssll"]

**Output:** [[0,1],[1,0],[3,2],[2,4]]

**Explanation:** The palindromes are ["dcbaabcd","abcddcba","slls","llssssll"]

**Example 2:**

**Input:** words = ["bat","tab","cat"]

**Output:** [[0,1],[1,0]]

**Explanation:** The palindromes are ["battab","tabbat"]

**Example 3:**

**Input:** words = ["a",""]

**Output:** [[0,1],[1,0]]

**Constraints:**

* 1 <= words.length <= 5000
* 0 <= words[i].length <= 300
* words[i] consists of lower-case English letters.

## Solution

Here's a few words of advice before we get started.

This is a very popular interview question. A concern I've seen brought up on the forums is that this question is too big to do in an interview.

Keep in mind though, that you're being compared to other candidates. They too will struggle with this, unless they've seen it before and memorized it. This however will be obvious to an experienced interviewer. It is the candidate who has clearly never seen it before yet makes great progress (probably not writing a complete implementation) who will be considered the most impressive. The secret would be to prioritize your time so that you are focusing on the core of the problem and not implementations of straightforward helper methods.

For this question, great progress would probably be deriving the intuition discussed in approach 2 and then writing code for the core algorithm of Approach 2 or Approach 3.

Remember that you don't necessarily have to "implement" every helper method. For example, some implementations rely on checking if a part of a string is a palindrome. This detail is easy-level by Leetcode standards, and in particular if you're using a whiteboard, it's a waste of time and space to write it unless you have finished the core algorithm. Simply state how you'd do it and leave it as a method signature unless asked to do otherwise. Also (for Approach 3), keep the TrieNode class simple. Don't waste half the whiteboard writing getters and setters for it.

#### **Approach 1: Brute force**

**Intuition**

The brute force solution is a good place to start. For this question, it is straightforward. Iterate over every possible pair of strings and check whether or not they form a palindrome.

You probably won't be writing this code, there simply won't be time. But make sure you know what it would be, and that you could describe the algorithm line-by-line if needed.

**Algorithm**

We can do this using 2 nested loops, each loop going over each index in the array. For each pair we need to check whether or not it forms a palindrome. There are many ways of doing this step, here I recommend the simplest way: creating the combined word and the reversed combined word and checking if they're equal. Doing the check in a more efficient way at this stage is not worth it — we want to focus our efforts on optimizing the main inefficiencies in this algorithm, which are discussed further in the complexity analysis section.

**An important edge case to be careful of** is where i = j. The problem states that i and j must be distinct (in other words, not the same). Identifying this edge case now is important, because we'll also need to be careful of it when we are optimizing our algorithm.

|  |
| --- |
| class Solution {  public List<List<Integer>> palindromePairs(String[] words) {  List<List<Integer>> pairs = new ArrayList<>();  for (int i = 0; i < words.length; i++) {  for (int j = 0; j < words.length; j++) {  if (i == j) continue;  String combined = words[i].concat(words[j]);  String reversed = new StringBuilder(combined).reverse().toString();  if (combined.equals(reversed)) {  pairs.add(Arrays.asList(i, j));  }  }  }  return pairs;  }  } |

**Complexity Analysis**

Let n*n* be the number of words, and k*k* be the length of the longest word.

* Time Complexity : O(n2 \* k).

There are n2 pairs of words. Then appending 2 words requires time 2*k*, as does reversing it and then comparing it for equality. The constants are dropped, leaving *k*. So in total, we get *O*(n2⋅*k*). We can't do better than this with the brute-force approach.

* Auxiliary Space Complexity : O(n2 + k).

Auxiliary space is where we do *not* consider the size of the input.

Let's start by working out the size of the output. In the worst case, there'll be *n*⋅(*n*−1) pairs of integers in the output list, as each of the *n* words could pair with any of the other *n*−1 words. Each pair will add 2 integers to the input list, giving a total of  2*n*⋅(*n*−1)=2⋅ n2−2⋅*n*. Dropping the constant and insignificant terms, we are left with an output size of *O*(n2).

Now, how much space do we use to find all the pairs? Each time around the loop, we are combining 2 words and creating an additional (reversed) copy of the combined words. This is 4⋅*k*, which gives us *O*(*k*). We ***don't*** need to multiply this by n2 because we aren't keeping the combined/ reversed words.

In total, this gives us *O*(n2+*k*). It might initially seem like the *k* should be dropped, as it's less significant than the n2. This isn't *always* the case though. If the words were really long, and the list very short, then it's possible for k*k* to be bigger than n2.

It's possible to optimize this slightly to *O*(n2). By using an in-place algorithm to determine whether or not 2 given words form a palindrome, the *k* would become a 1 and therefore be dropped. Like I said above though, it'd be wasted effort to do so. Especially given that in practice it's likely that k*k* is smaller than n2 anyway.

* Space Complexity : *O*(*n*⋅*k*+ n2).

For this, we also need to take into account the size of the input. There are *n* words, with a length of up to *k* each. This gives us *O*(*n*⋅*k*).

Like above, we can't assume anything about whether *k*>*n* or *k*<*n*. Therefore, we don't know whether *O*(n2+*k*) or  *O*(*n*⋅*k*) is bigger.

Approach 2: Hashing

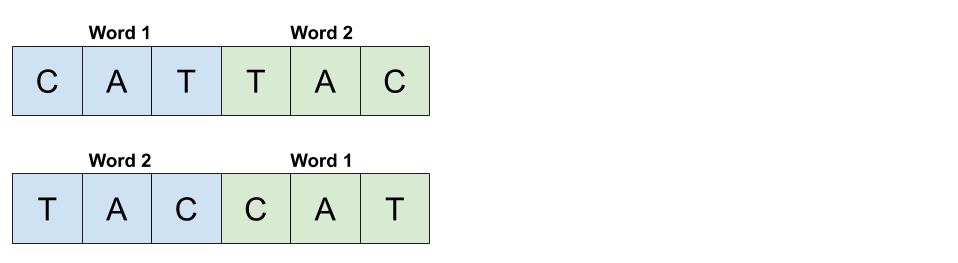
**Intuition**

Testing every pair is too expensive. Is there a way we can avoid checking pairs that will definitely not form a palindrome? Inorder to answer this question, we'll need to explore the properties of pairs that *do* form a palindrome.

This type of exploration and reasoning can be a bit challenging if you're not used to it, so we'll tackle it with some examples and then we'll try and prove our discoveries more formally. After that, we'll take a look at how it could be implemented efficiently in code.

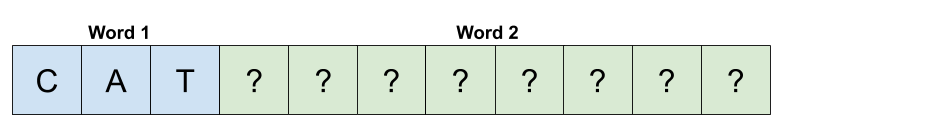
*What are the ways we could form a palindrome with 2 words?*

The simplest way to make a palindrome is to take 2 words that are the reverse of each other and put them together. In this case, we get 2 different palindromes, as we can put either word first.

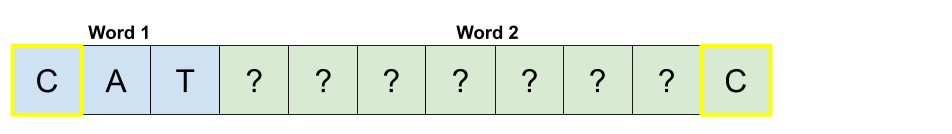


We know that there are always 2 unique palindromes that can be formed by 2 words that are the reverse of each other, because the words *must be different*. The problem statement is clear that there are no duplicates in the word list.

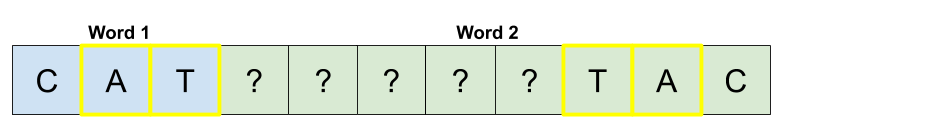
Let's now think about all the words that could pair with a word 1 of "CAT" to make a palindrome. We'll assume that all the possibilities for word 2 we're looking at are 8 letters long. While this assumption might seem too specific, remember that we're just using it as a starting point to identify possible cases. We'll do a more general proof later.



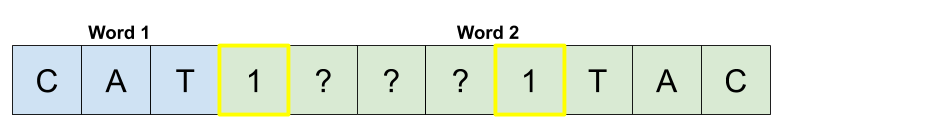
To start with, we know that the last letter of word 2 has to be "C". Otherwise, it would be impossible to form a palindrome.



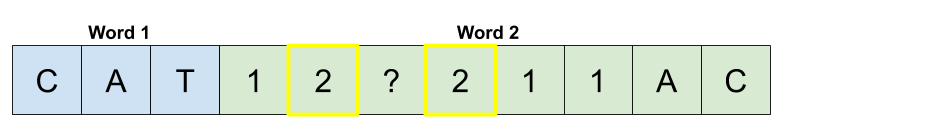
By that same logic, we also know the 2nd to last and 3rd to last characters must be "A" and "T" respectively.



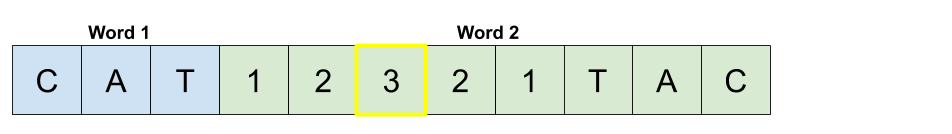
Here's where things start to get a bit interesting. We know that the 2 letters highlighted in the next diagram must be the same for the combined word to be a palindrome. We'll use numbers to show where letters must be the same.



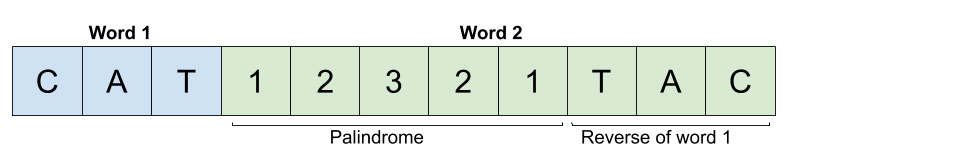
The same argument applies for the next pair of highlighted letters.



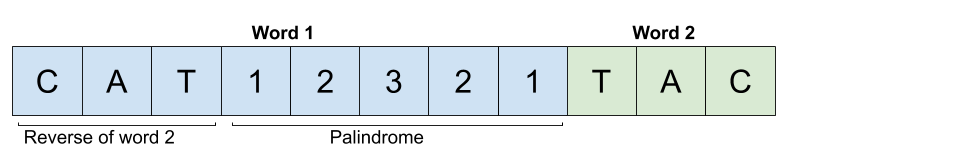
And that last letter in the center can be anything.



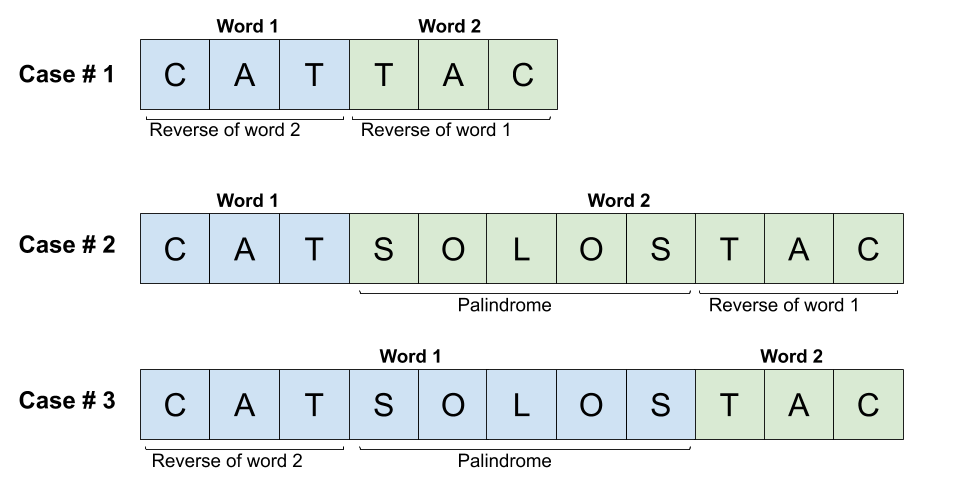
Let's now take a step back and see what we have. Our experimenting has shown us that if word 2 is the concatenation of a 5-letter palindrome and then the reverse of word 1, that the combined pair of word 1 and word 2 is a palindrome.



Another case can also be seen here. If instead word 1 was the concatenation of the reverse of word 2 and then a 5-letter palindrome, the combined pair of word 1 and word 2 would also be a palindrome.



We have now identified 3 cases.



Don't forget that the *empty string* is also a valid word. How could we form a palindrome with it? This is an important edge case we'll now think about.

Appending an empty string with another word will simply give *the non-empty string* word. If this word was a palindrome by itself, we will have a valid palindrome pair. If it wasn't, we won't. So any words that by themselves are a palindrome will form a palindrome pair with the empty string.

Depending on the implementation you use, you might not need to treat this as a special case, as it is really just a sub case of **case 2** and **case 3**. It's just that the bit that is reversed is of length-0. Make sure to test your implementation on this case though!

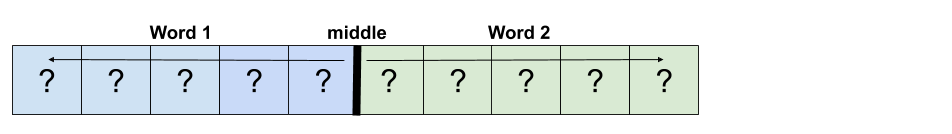
*How can we prove that we have identified all the cases?*

By experimenting, we've discovered a few cases. But for these kinds of questions, it's very important to convince ourselves that we haven't overlooked any cases. One way we can do this is by considering the relative length of each pair of words. There are 2 cases for the relative lengths within each pair.

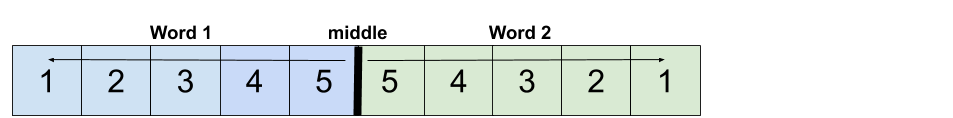
1. The words are both of the same length.
2. The words are of different lengths.

We then need to show how each of these 2 cases fully map onto the palindrome pair cases we've already discovered. We'll do this by considering where the middle of the combined word (word we get by appending the second word to the first word) is.

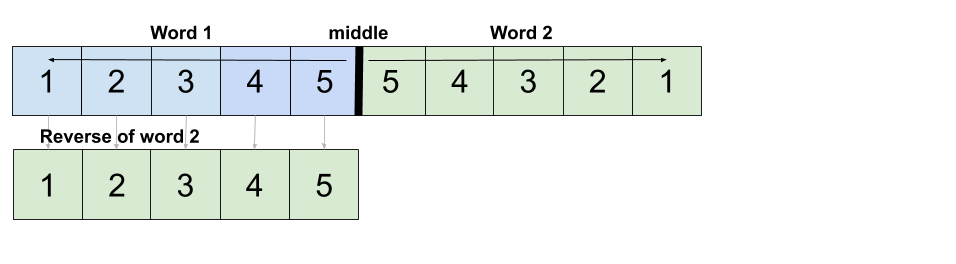
For the first possibility, the center of the combined word is *between the two words*.



For the pair to form a palindrome, the letters before the center must be the *reverse* of the letters after the center. The following diagram uses numbers to show where 2 letters must be the same.



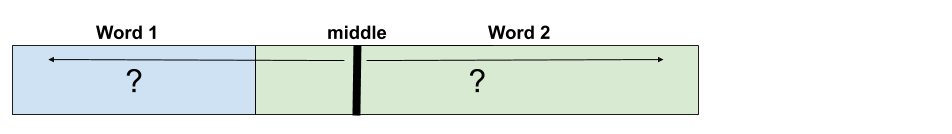
We can also see that this means word 1 must be the reverse of word 2.



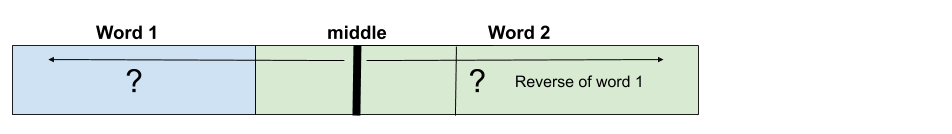
Therefore, when 2 words of the *same length* form a palindrome, it must be because word 1 is the reverse of word 2 (which also means word 2 is the reverse of word 1). This is equivalent to palindrome pair **case 1**.

For the second relative word-length case, we know that *one of the words must be shorter than the other*. We'll assume for now that *word 1 is shorter*. The exact same argument will make will also apply for when word 2 is shorter.

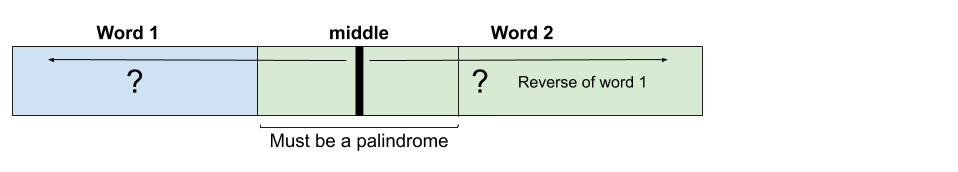
Like before, there must be a middle of the combined word. We know that because word 1 is shorter, word 2 will overlap this center point.



We know that a palindrome must mirror around that center point. Therefore, we know that the end of word 2 must be the reverse of word 1.



We are now left with the region *between* word 1 and the reverse of word 1. We know that this middle region is divided equally in 2 by the middle line because we took the same number of characters off each end of the combined word. Therefore, for the overall combined word to be a palindrome, the piece in the middle must be a palindrome.



Which is equivalent to palindrome-pair **case 2**.

Using this same line of reasoning, you can easily show that when word 2 is shorter, it is equivalent to palindrome pair **case 3**.

Therefore, we have proven that the only possible ways of forming a palindrome pair out of 2 words are covered by the 3 palindrome-pair cases we discovered during our exploration.

*How can we put all this into code?*

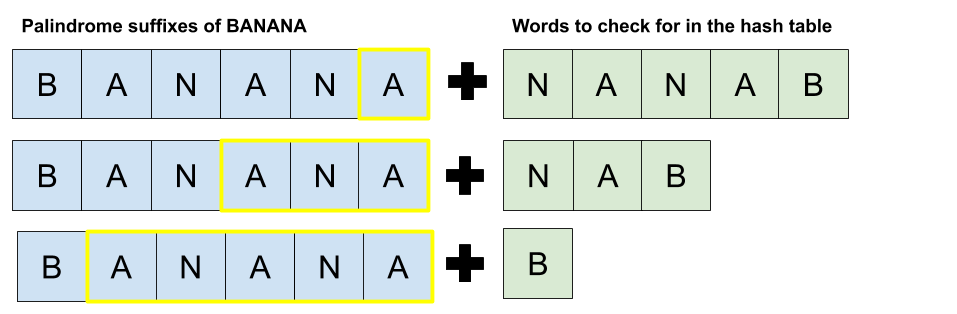
The simplest way to put all of this into code is to iterate over the list of words and do the following for each **word**.

If these initial explanations are confusing, don't panic. There's further examples just below the list.

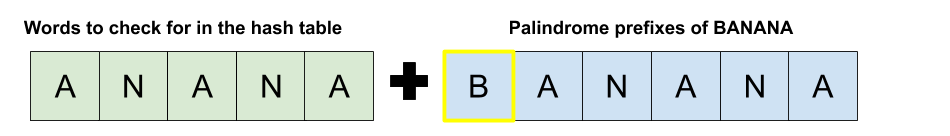
1. Check if the reverse of **word** is present. If it is, then we have a **case 1** pair by appending the reverse onto the end of **word**.
2. For each **suffix** of **word**, check if the **suffix** is a palindrome. **if it is a palindrome**, then reverse the remaining **prefix** and check if it's in the list. If it is, then this is an example of **case 2**.
3. For each **prefix** of **word**, check if the **prefix** is a palindrome. **if it is a palindrome**, then reverse the remaining **suffix** and check if it's in the list. If it is, then this is an example of **case 3**.

For example, imagine we have the word "banana". Start by checking whether or not "ananab" is in the list.

Now identify all palindrome suffixes of "banana". For each one, we take the remaining prefix, reverse it, and check if we have that word in the list.



Do the same for all palindrome prefixes of "banana". There is only one of these.



If we do this for each word, we will get all palindrome pairs exactly once. The most challenging idea here is that we are treating our current word as *word 2* for case 2. The reason we do this is because treating it as *word 1* would mean we had to guess possible prefixes for *word 2*, which would be very, very inefficient.

To ensure the implementation is efficient, we can put all the words into a hash table with the word as the key and the original index as the value (as the output must be the original indexes of the words).

**Algorithm**

We'll call a *suffix* a "valid suffix" of a word if the remainder (prefix) of the word forms a palindrome. The function allValidSuffixes finds all such suffixes. For example, the "valid suffixes" of the word "exempt" are "xempt" (remove "e") and "mpt" (remove 'exe').

We'll call a *prefix* a "valid prefix" of a word if the remainder (suffix) of the word forms a palindrome. The function allValidPrefixes finds all such prefixes in a similar way to how the allValidSuffixes function does. It is possible to combine more of the code for these functions here, but after going back and forth on the issue, I decided against it for this explanation because while it decreases the length of the code and some repetition, the cognitive load to understand it is higher. In your own code, it would be fine to combine it.

Examples of case 1 can be found by reversing the current word and looking it up. One edge case to be careful of is that if a word is a palindrome by itself, then we don't want to add a pair that includes that same word twice. This case only comes up in case 1, because case 1 is the only case that deals with pairs where the words are of equal length.

Examples of case 2 can be found by calling allValidSuffixes and then reversing each of the suffixes found and looking them up.

Examples of case 3 can be found by calling allValidPrefixes and then reversing each of the prefixes found and looking them up.

It would be possible to simplify further (not done here) by recognizing that **case 1** is really just a special case of **case 2** and **case 3**. This is because the empty string is a palindrome prefix/ suffix of any word.

|  |
| --- |
| class Solution {  private List<String> allValidPrefixes(String word) {  List<String> validPrefixes = new ArrayList<>();  for (int i = 0; i < word.length(); i++) {  if (isPalindromeBetween(word, i, word.length() - 1)) {  validPrefixes.add(word.substring(0, i));  }  }  return validPrefixes;  }  private List<String> allValidSuffixes(String word) {  List<String> validSuffixes = new ArrayList<>();  for (int i = 0; i < word.length(); i++) {  if (isPalindromeBetween(word, 0, i)) {  validSuffixes.add(word.substring(i + 1, word.length()));  }  }  return validSuffixes;  }  // Is the prefix ending at i a palindrome?  private boolean isPalindromeBetween(String word, int front, int back) {  while (front < back) {  if (word.charAt(front) != word.charAt(back)) return false;  front++;  back--;  }  return true;  }  public List<List<Integer>> palindromePairs(String[] words) {  // Build a word -> original index mapping for efficient lookup.  Map<String, Integer> wordSet = new HashMap<>();  for (int i = 0; i < words.length; i++) {  wordSet.put(words[i], i);  }  // Make a list to put all the palindrome pairs we find in.  List<List<Integer>> solution = new ArrayList<>();  for (String word : wordSet.keySet()) {  int currentWordIndex = wordSet.get(word);  String reversedWord = new StringBuilder(word).reverse().toString();  // Build solutions of case #1. This word will be word 1.  if (wordSet.containsKey(reversedWord)  && wordSet.get(reversedWord) != currentWordIndex) {  solution.add(Arrays.asList(currentWordIndex, wordSet.get(reversedWord)));  }  // Build solutions of case #2. This word will be word 2.  for (String suffix : allValidSuffixes(word)) {  String reversedSuffix = new StringBuilder(suffix).reverse().toString();  if (wordSet.containsKey(reversedSuffix)) {  solution.add(Arrays.asList(wordSet.get(reversedSuffix), currentWordIndex));  }  }  // Build solutions of case #3. This word will be word 1.  for (String prefix : allValidPrefixes(word)) {  String reversedPrefix = new StringBuilder(prefix).reverse().toString();  if (wordSet.containsKey(reversedPrefix)) {  solution.add(Arrays.asList(currentWordIndex, wordSet.get(reversedPrefix)));  }  }  }  return solution;  }  } |

**Complexity Analysis**

Let n*n* be the number of words, and k*k* be the length of the longest word.

* Time Complexity : O(k2 ⋅ n).

Building the hash table takes *O*(*n*⋅*k*) time. Each word takes *O*(*k*) time to insert and there are *n* words.

Then, for each of the *n* words we are searching for 3 different cases. First is the word's own reverse. This takes *O*(*k*) time. Second is words that are a palindrome followed by the reverse of another word. Third is words that are the reverse of another word followed by a palindrome. These second 2 cases have the same cost, so we'll just focus on the first one. We need to find all the prefixes of the given word, that are palindromes. Finding all palindrome prefixes of a word can be done in O(k2) time, as there are *k* possible prefixes, and checking each one takes *O*(*k*) time. So, for each word we are doing k2 + k2 + k processing, which in big-oh notation is O(k2). Because are doing this with *n* words, we get a final result of O(k2 \* n).

It's worth noting that the previous approach had a cost of *O*(*n*2⋅*k*). Therefore, this approach isn't better in *every* case. It is only better where *n*>*k*. In the test cases your solution is tested on, this is indeed the case.

* Space Complexity : O((k + n)2).

Like before, there are several components we need to consider. This time however, the space complexity is the same regardless of whether or not we include the input in the calculations. This is because the algorithm immediately creates a hash table the same size as the input.

In the input, there are *n* words, with a length of up to *k* each. This gives us *O*(*n*⋅*k*). We are then building a hash table with *n* keys of size *k*. The hash table is the same size as the original input, so it too is *O*(*n*⋅*k*).

For each word, we're making a list of all possible pair words that need to be looked up in the hash table. In the worst case, there'll be *k* words to look up, with lengths of up to *k*. This means that at each cycle of the loop, we're using up to k2 memory for the lookup list. This could be optimized down to *O*(*k*) by only creating one of the words at a time. In practice though, it's unlikely to make much difference due to the way strings are handled under the hood. So, we'll say that we're using an additional O(k2) memory.

Determining the size of the output is the same as the other approaches. In the worst case, there'll be *n*⋅(*n*−1) pairs of integers in the output list, as each of the *n* words could pair with any of the other *n*−1 words. Each pair will add 2 integers to the input list, giving a total of 2⋅*n*⋅(*n*−1)=2⋅*n*2−2⋅*n*. Dropping the constant and insignificant terms, we are left with an output size of O(n^2).

Putting this all together, we get 

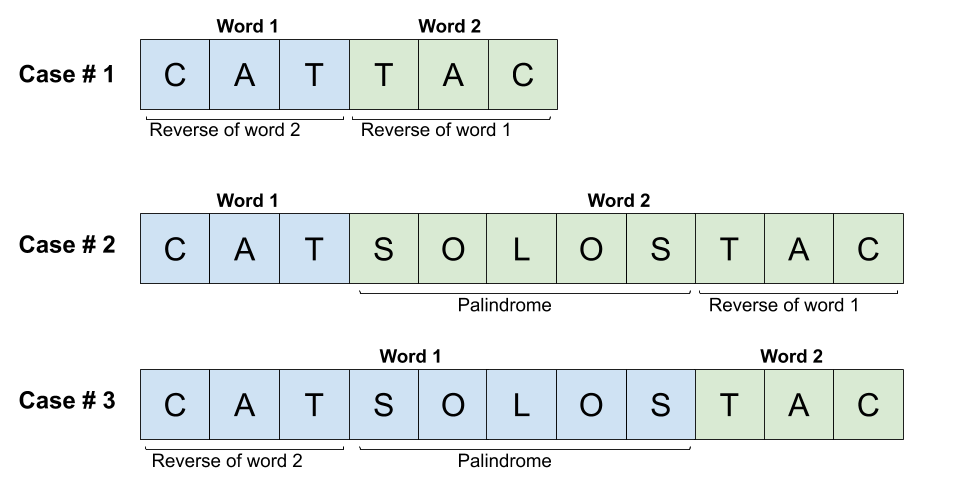
Approach 3: Using a Trie

**Intuition**

*This section assumes you've previously been introduced to the Trie data structure. If you are not familiar with the Trie, work through*[*Leetcode's module on them first*](https://leetcode.com/explore/learn/card/trie/)*You'll also need to have read the previous section's intuition, as this section further builds on those ideas.*

From the previous section, you probably noticed that the prefixes and suffixes of each word were important. If you're familiar with the *Trie* data structure, you may be wondering if there's a way we could use one to solve this problem. It turns out there is, so let's investigate!

We'll start by reminding ourselves of the palindrome pair cases we discovered in the previous section's intuition.



Now, we want to build some kind of Trie with the words. Then, we want to go down the list of words and identify all words from the Trie that our current word from the list would form a palindrome pair with. In words, we are looking for:

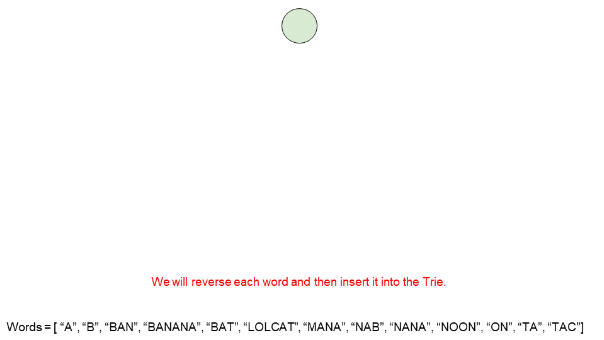
1. Words in the Trie that are the reverse of our current word.
2. Words in the Trie that start with the reverse of our current word and then finish in a palindrome.
3. Words in the Trie that are the reverse of the first part of our current word, and then what's left of our current word forms a palindrome.

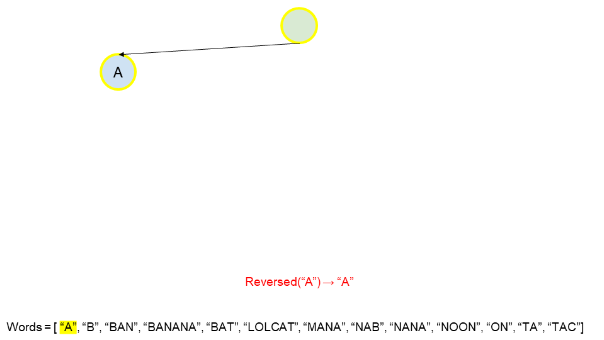
Because we are interested in the reverse of words, it makes sense to put all the words into the Trie in reverse. You could also put the words forward into the Trie, and then reversed each word in the list. Both approaches are equally valid, and have their own pros and cons in terms of clarity.

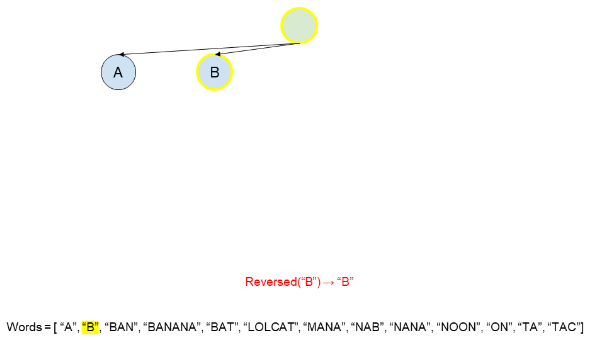
Anyway, let's jump to an example now. Our word list is as follows:

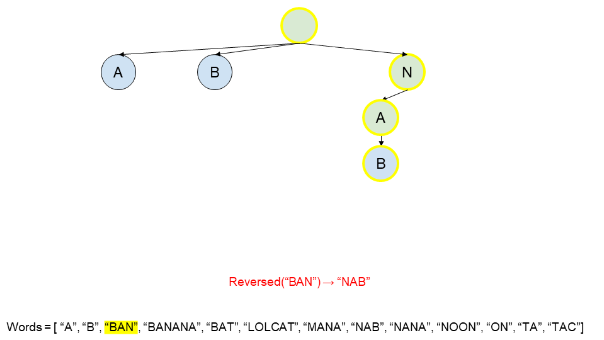
words = [ "A", "B", "BAN", "BANANA", "BAT", "LOLCAT", "MANA", "NAB", "NANA", "NOON", "ON", "TA", "TAC"]

We'll start by inserting the reverse of each word into a Trie, as shown in the following animation.

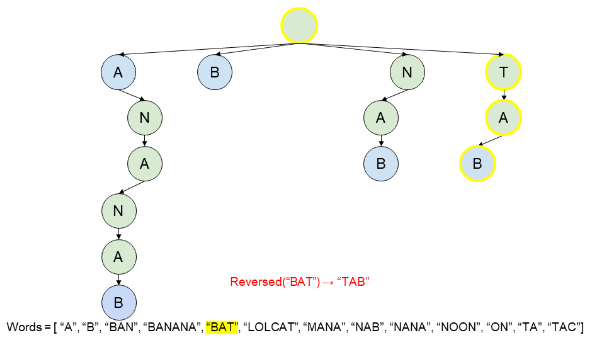


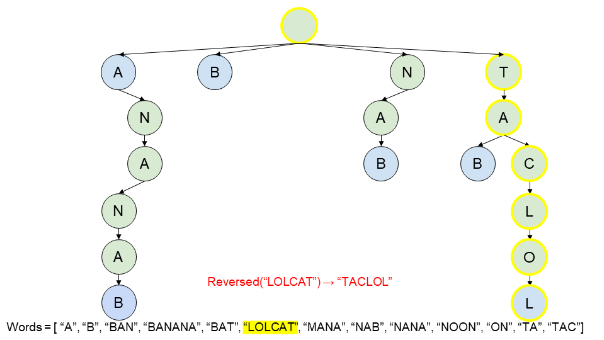


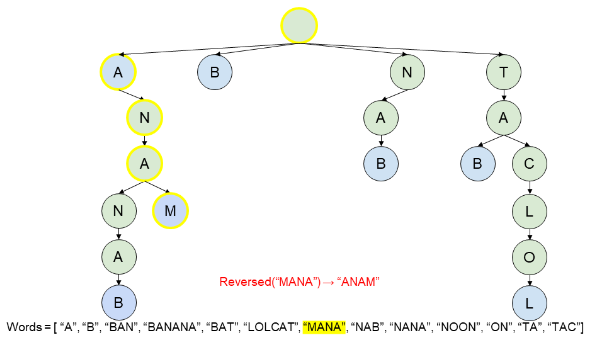


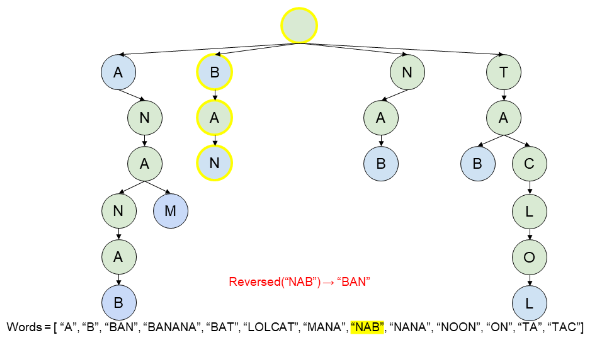
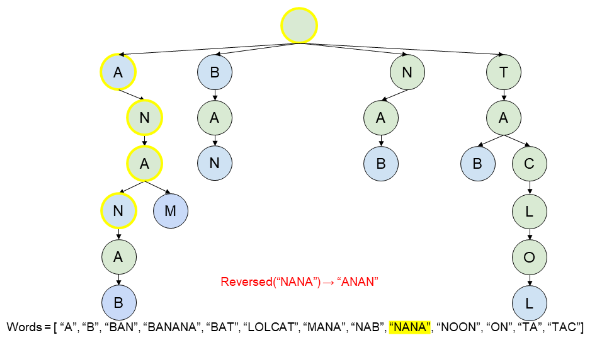


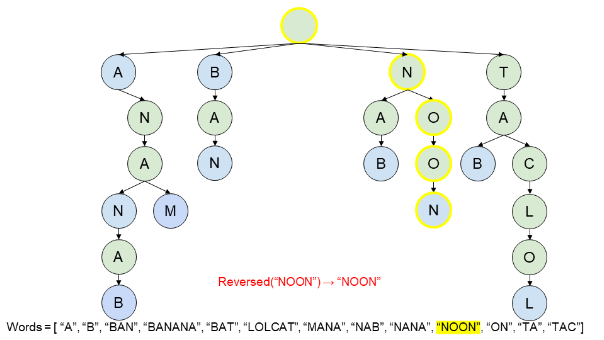


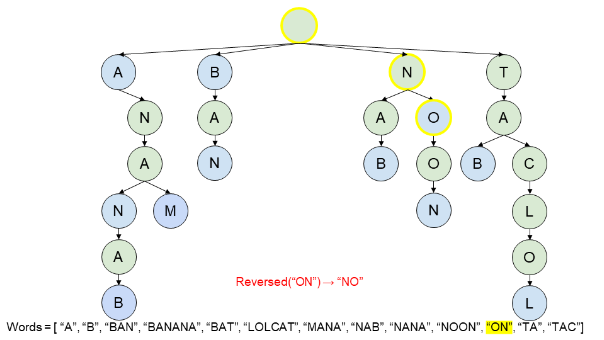


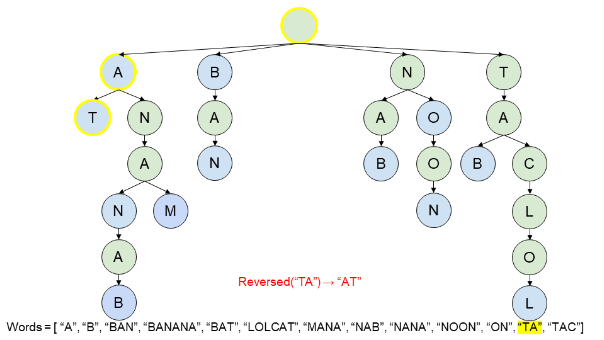


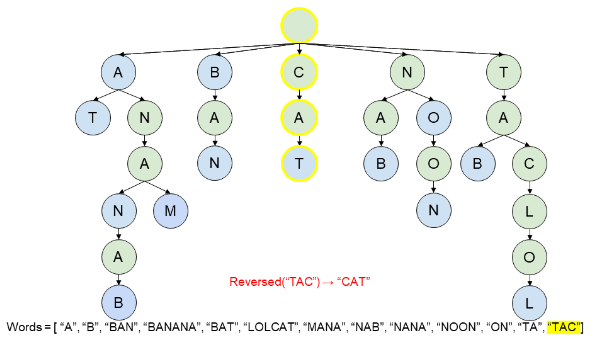


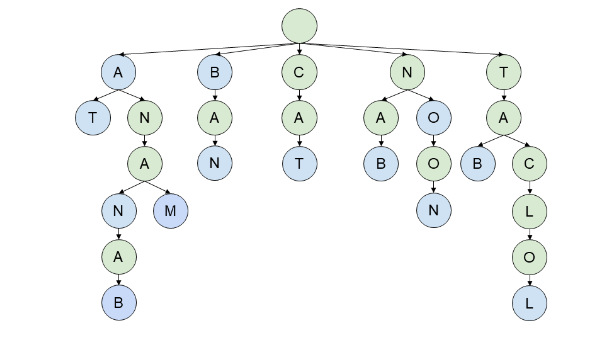
 



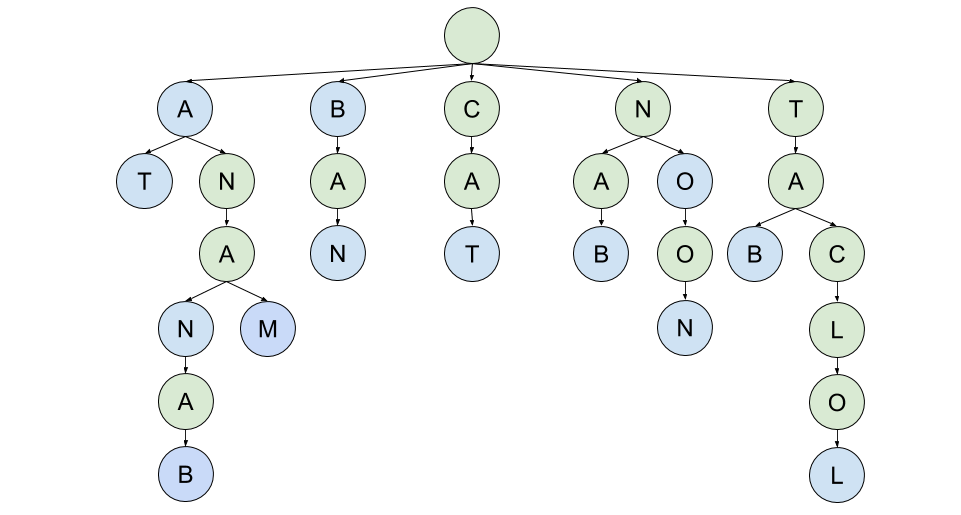








For ease of reference, here's the final Trie we got after inserting all the words.



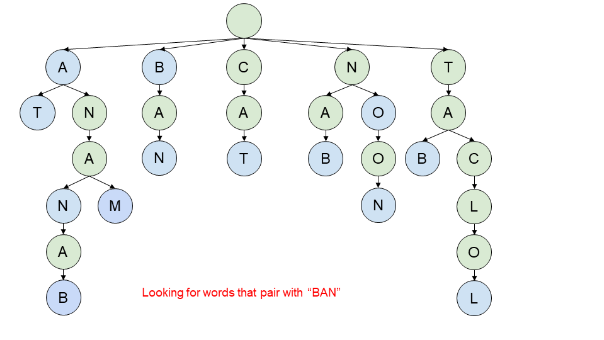
Great! We have a Trie. So, how do we use it? We'll look at each of the 3 cases, one-by-one.

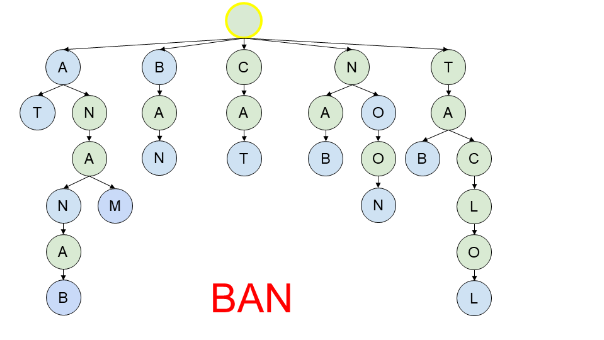
Case 1 with the Trie

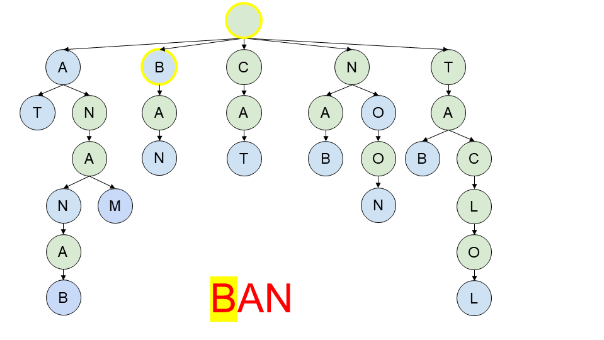
Case 1 is where a palindrome pair is formed by 2 words that are the reverse of each other. We'll use the word "BAN" as our example. The reverse of "BAN" is "NAB". Therefore, we need to use our Trie to see if the word "NAB" exists.

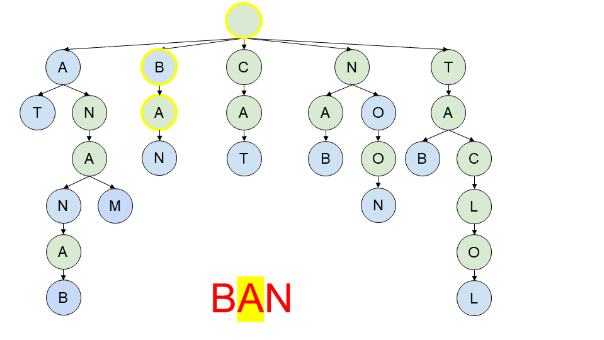
How will "NAB" appear in the Trie? Well, remember how all words were inserted into the Trie ***backwards***? This means that "NAB" will appear as "BAN" in the Trie. Therefore, we are simply searching for the word itself, in this case "BAN". If we can find the word in the Trie, **and** be on a blue (end of word) node when we're done, we know the reverse exists.

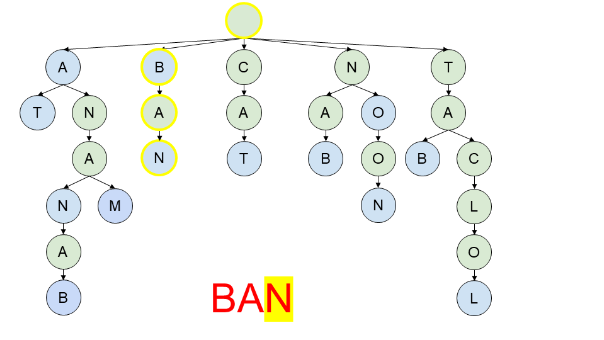
Here's an animation showing how we determine that the reverse of "BAN" is in the Trie.

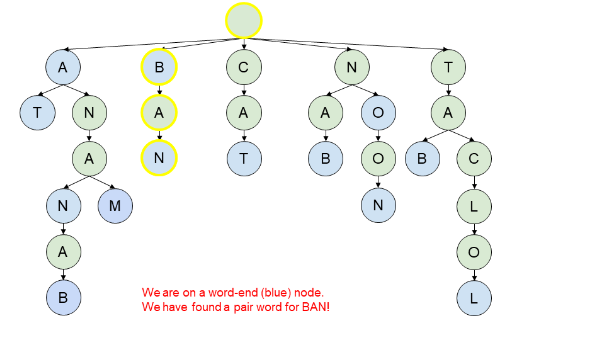


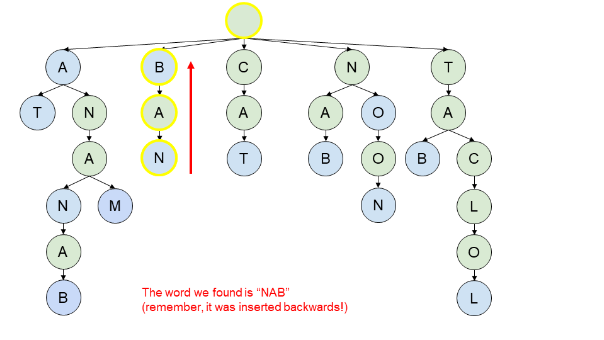






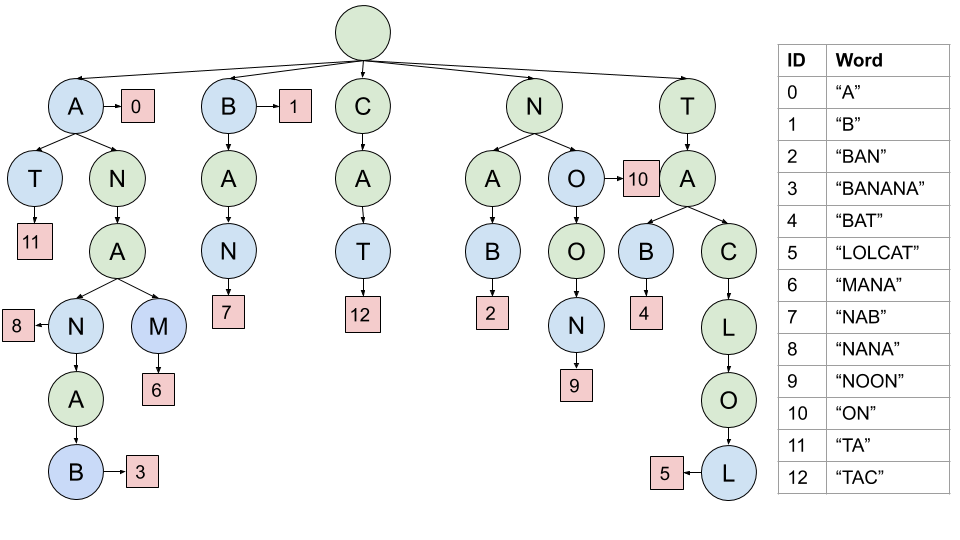








Remember that for the output, we need to give the indexes of each pair. Currently, finding this information would be annoying. To fix it, we'll add an index field onto each end of word node. If we do that, this is our resulting Trie.



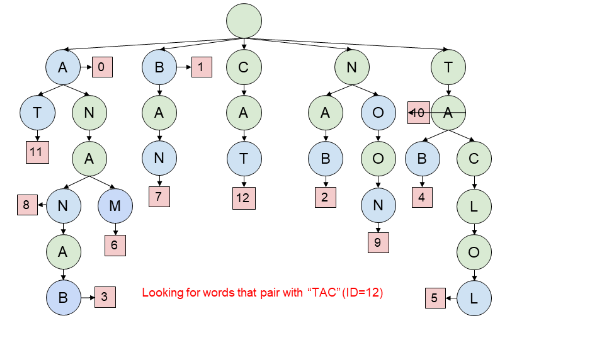
So, we knew that "BAN" had an index of 2. When we get to the end of the word it matches with, we see that it is word 7. The first word of the pair was "BAN", and "NAB" was the second. Therefore, we can add the pair [2, 7] to our output list.

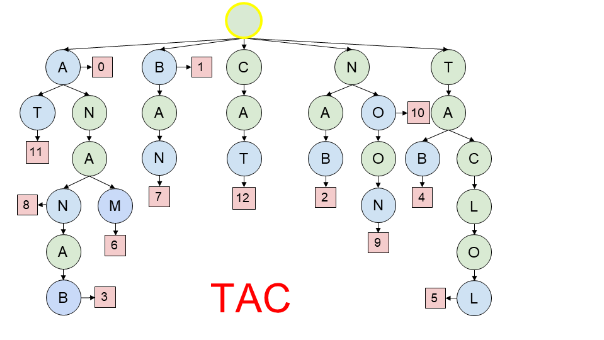
Case 2 with the Trie

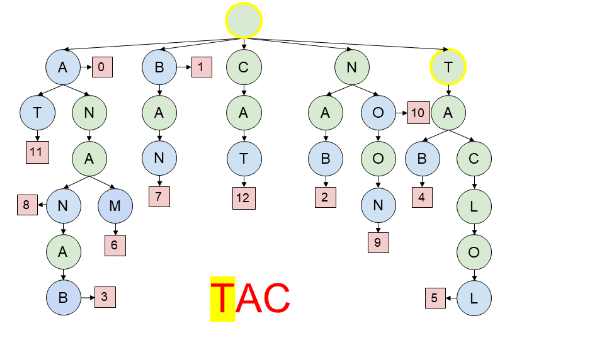
Case 2 is the one where the first word is shorter than the second word. The second word starts with a palindrome, and ends with the reverse of the first word. So, how will this look in our Trie?

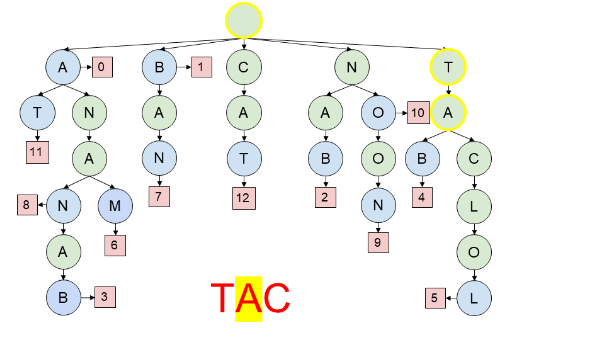
Well, let's just have a look. The example we'll work with this time is "TAC". Like before, we know that the last 3 letters of the second word must start with "CAT". Now, remembering that these would have been inserted in reverse, we will start by looking for "TAC". Once we have found those letters, we would expect to not yet be at the end of a word, but for there to be a word that only has a palindrome left.

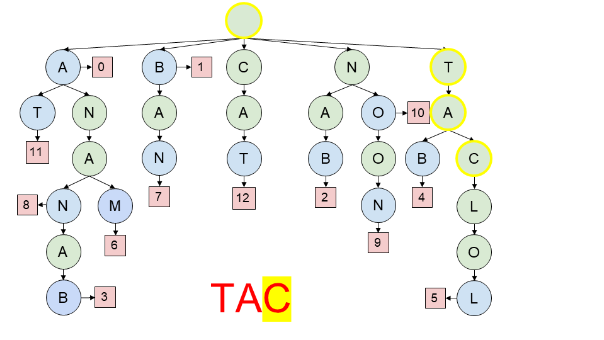
Here's an animation showing this search.

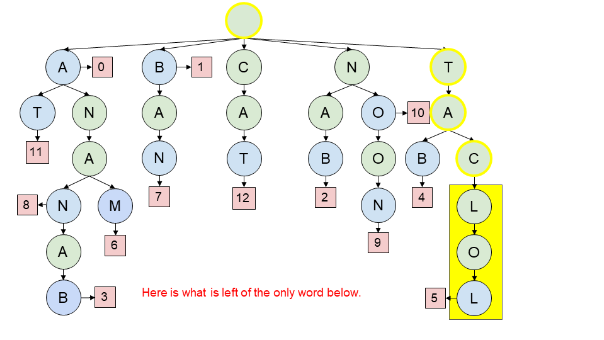


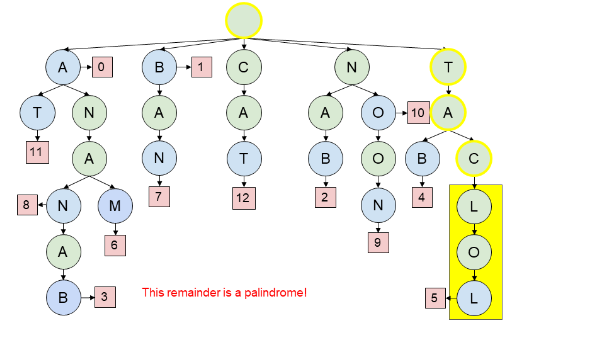


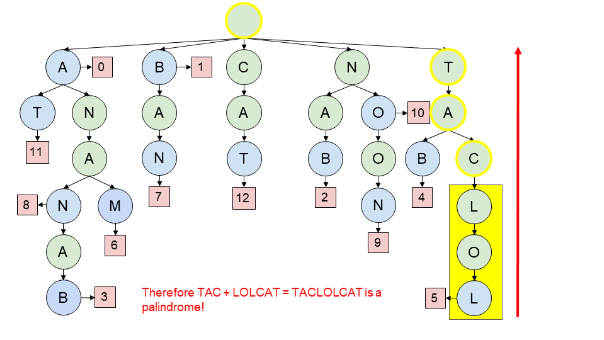










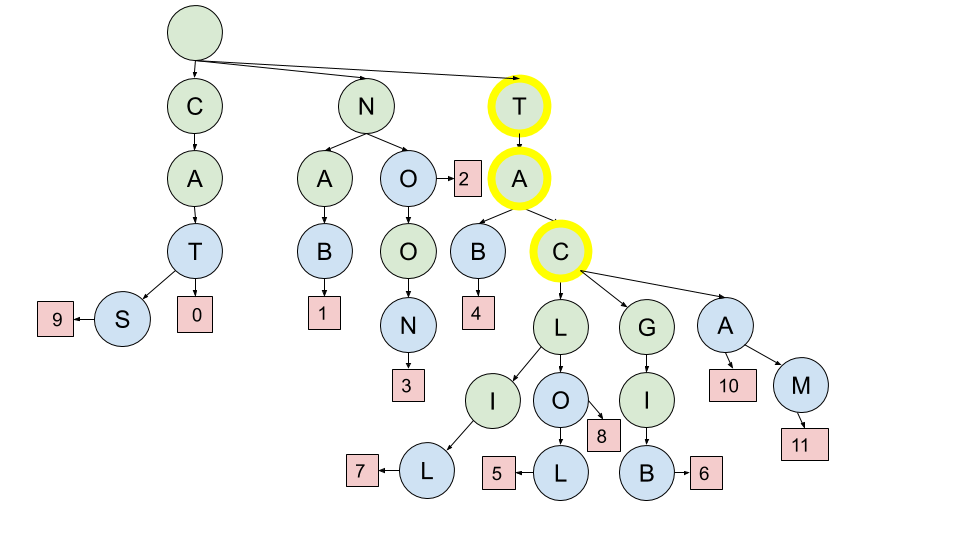


In this particular example, there was only one word left, and it did indeed form a palindrome.

*A quick understanding check*

Before we continue on with case 3, it's time to make sure you're following okay. Here's a different Trie, (closely) related to the one above. Like before, the word we're looking at is the word "TAC". We want to find all second words that it forms a palindrome pair with. Answer the following questions before scrolling down to the explanation.

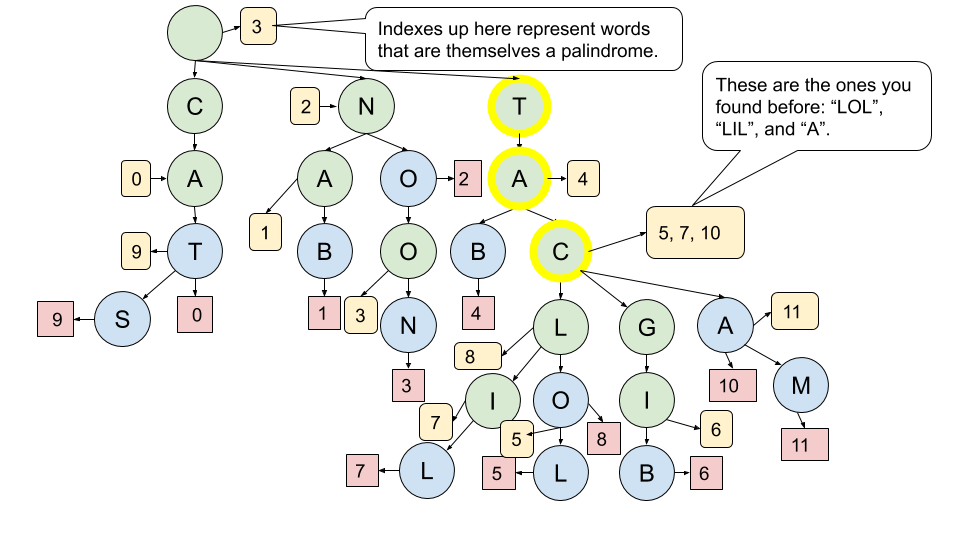
1. What are *ALL* the words that end in "CAT"?
2. What are all the words that form a palindrome pair with "TAC"?
3. How do we know the word "CAT" itself is not in the Trie?



1. We know that any blue circles "below" the letters we have found so far represent the end of a word. Remember that they should be read backwards! The answer is, therefore: "LILCAT". "OLCAT", "LOLCAT", "BIGCAT", "MACAT" and "ACAT".
2. To do this, you need to look for palindromes hanging below the highlighted "C". ALL of the words below the C are: "LIL", "OL", "LOL", "BIG", "MA", and "A". Of these, the palindromes are "LIL", "LOL", and "A", which correspond to the words "LOLCAT", "LILCAT", and "ACAT". Therefore, we know each of these 3 words will form a palindrome pair with "TAC".
3. If "CAT" were in the Trie, we'd expect to see the "C" at the end of the highlighted letters be blue and have an index field. It doesn't. Therefore, we know the word "CAT" is not in the Trie.

It might have been a little annoying having to carefully read each branch below the word "TAC", that ended in a blue circle. Luckily, there's an easy way we can improve the Trie structure to simplify this process.

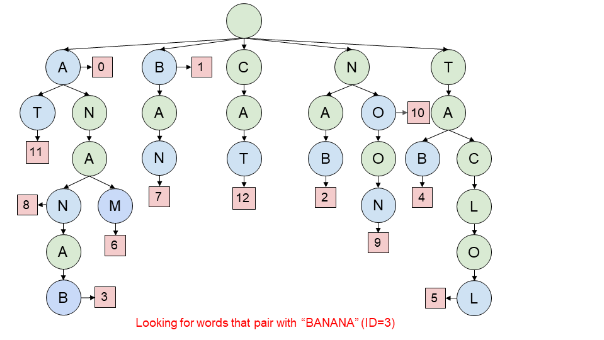
When we insert a word, we can start by determining all of its palindrome *prefixes*. Now, on each node we'll attach a list of all words that have a palindrome remaining on them. For the example you worked through, this is the words you identified in part 2. The new Trie for that example would be as follows. The indexes are shown in yellow.

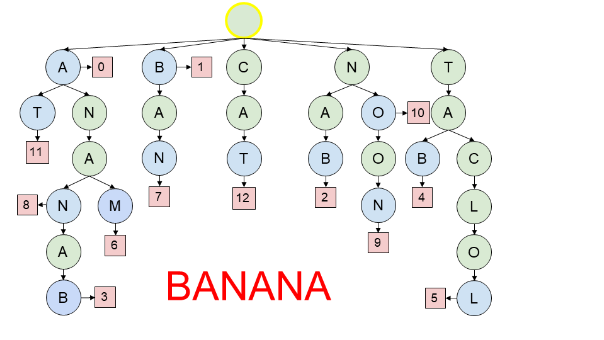


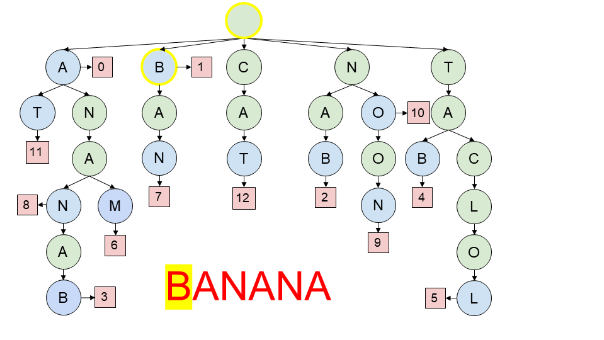
Have a think about our original example. What would these new lists be for it?

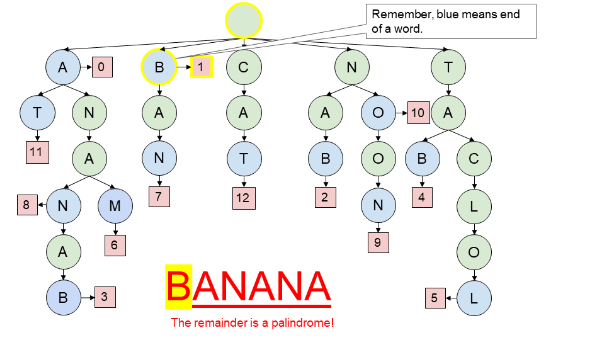
*Case 3 with the Trie*

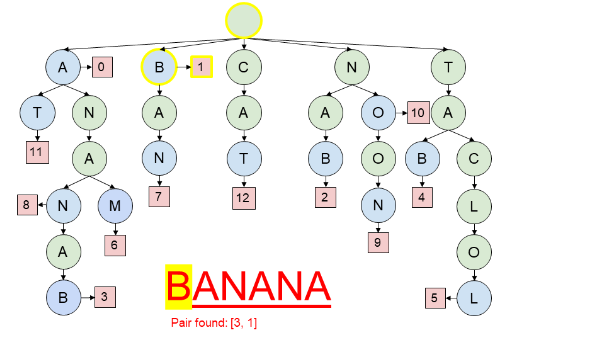
Case 3 is the one where the first word is longer than the second word. In terms of our Trie, it would come up where we get to a blue node and still have some letters left from our current word. If those letters that are left form a palindrome, then we have a case 3 palindrome pair. Again, let's look at an example. This time, we are searching for the word "BANANA". Both times we reach a blue node, there is a palindrome remaining. Therefore, we find 2 pairs in this example.

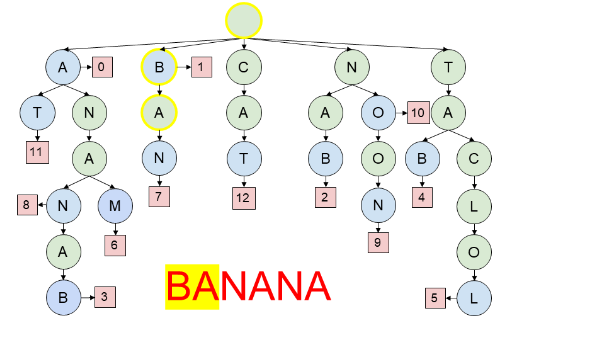


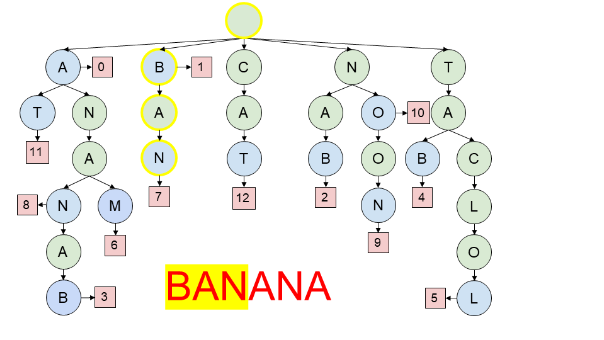


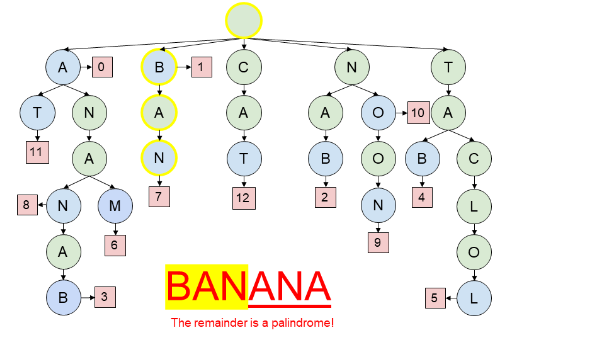














This case is conceptually simpler than case 2. The key thing to remember is that we *only* do this palindrome check if we are *on a blue node*. If there is a palindrome remaining on our word, then a single pair is formed with the word that ended at that blue node (remember, blue nodes can only represent the end of a single word. There were no duplicates in the input list). Also, don't look at the "palindrome remaining" lists that we added for case 2, as this would lead to invalid pairs.

**Algorithm**

We start by building the Trie. For each word, reverse it and identify its palindrome prefixes (suffixes of the reversed word). Insert the word into the Trie, and mark the final letter as an ending node, and include the word's index. Also, while inserting, note any points where the remainder of the word is a palindrome suffix by including the index in an additional list (used for case 2).

Then, we go back through the list of words, looking each up in the Trie. Any of the following situations give us palindrome pairs.

1. We have no letters left on the word and are at a word end node (case 1).
2. We have no letters left on the word and there are indexes in the list attached to the node (case 2).
3. We have a palindrome left on the word and are on a word end node (case 3).

|  |
| --- |
| class TrieNode {  public int wordEnding = -1; // We'll use -1 to mean there's no word ending here.  public Map<Character, TrieNode> next = new HashMap<>();  public List<Integer> palindromePrefixRemaining = new ArrayList<>();  }  class Solution {  // Is the given string a palindrome after index i?  // Tip: Leave this as a method stub in an interview unless you have time  // or the interviewer tells you to write it. The Trie itself should be  // the main focus of your time.  public boolean hasPalindromeRemaining(String s, int i) {  int p1 = i;  int p2 = s.length() - 1;  while (p1 < p2) {  if (s.charAt(p1) != s.charAt(p2)) return false;  p1++; p2--;  }  return true;  }  public List<List<Integer>> palindromePairs(String[] words) {  TrieNode trie = new TrieNode();  // Build the Trie  for (int wordId = 0; wordId < words.length; wordId++) {  String word = words[wordId];  String reversedWord = new StringBuilder(word).reverse().toString();  TrieNode currentTrieLevel = trie;  for (int j = 0; j < word.length(); j++) {  if (hasPalindromeRemaining(reversedWord, j)) {  currentTrieLevel.palindromePrefixRemaining.add(wordId);  }  Character c = reversedWord.charAt(j);  if (!currentTrieLevel.next.containsKey(c)) {  currentTrieLevel.next.put(c, new TrieNode());  }  currentTrieLevel = currentTrieLevel.next.get(c);  }  currentTrieLevel.wordEnding = wordId;  }  // Find pairs  List<List<Integer>> pairs = new ArrayList<>();  for (int wordId = 0; wordId < words.length; wordId++) {  String word = words[wordId];  TrieNode currentTrieLevel = trie;  for (int j = 0; j < word.length(); j++) {  // Check for pairs of case 3.  if (currentTrieLevel.wordEnding != -1  && hasPalindromeRemaining(word, j)) {  pairs.add(Arrays.asList(wordId, currentTrieLevel.wordEnding));  }  // Move down to the next trie level.  Character c = word.charAt(j);  currentTrieLevel = currentTrieLevel.next.get(c);  if (currentTrieLevel == null) break;  }  if (currentTrieLevel == null) continue;  // Check for pairs of case 1. Note the check to prevent non distinct pairs.  if (currentTrieLevel.wordEnding != -1 && currentTrieLevel.wordEnding != wordId) {  pairs.add(Arrays.asList(wordId, currentTrieLevel.wordEnding));  }  // Check for pairs of case 2.  for (int other : currentTrieLevel.palindromePrefixRemaining) {  pairs.add(Arrays.asList(wordId, other));  }  }  return pairs;  }  } |

**Complexity Analysis**

Let n*n* be the number of words, and k*k* be the length of the longest word.

* Time Complexity : *O*(k2⋅*n*).

There were 2 major steps to the algorithm. Firstly, we needed to build the Trie. Secondly, we needed to look up each word in the Trie.

Inserting each word into the Trie takes *O*(*k*) time. As well as inserting the word, we also checked at each letter whether or not the remaining part of the word was a palindrome. These checks had a cost of *O*(*k*), and with *k* of them, gave a total cost of O(k2). With *n* words to insert, the total cost of building the Trie was therefore *O*(k2⋅*n*).

Checking for each word in the Trie had a similar cost. Each time we encountered a node with a word ending index, we needed to check whether or not the current word we were looking up had a palindrome remaining. In the worst case, we'd have to do this *k* times at a cost of *k* for each time. So like before, there is a cost of k2 for looking up a word, and an overall cost of k2⋅*n* for all the checks.

This is the same as for the hash table approach.

* Space Complexity : O((k + n)2).

The Trie is the main space usage. In the worst case, each of the *O*(*n*⋅*k*) letters in the input would be on separate nodes, and each node would have up to *n* indexes in its list. This gives us a worst case of O(n2 \* k), which is strictly larger than the input or the output.

Inserting and looking up words only takes *k* space though, because we're not generating a list of prefixes like we were in approach 2. This is insignificant compared to the size of the Trie itself.

So in total, the size of the Trie has a worst case of *O*(*k*⋅*n*2). In practice however, it'll use a lot less, as we based this on the worst case. Tries are difficult to analyze in the general case, because their performance is so dependent on the type of data going into them. As *n* gets really, really, big, the Trie approach will eventually beat the hash table approach on both time and space. For the values of *n* that we're dealing with in this question though, you'll probably notice that the hash table approach performs better.

#### **Additional Discussion: Online Algorithms**

This section is beyond what is needed for an interview, and is included only for interest.

When developing algorithms for the real world, an often desirable property is that the algorithm works **online**. This does ***not mean on the internet***, instead it means that the algorithm can still work if the input data is provided bit-by-bit. In this case, it'd be that we want to feed the algorithm the words one at a time, and each time, we want to update the list of all pairs without doing too much extra work.

So, let's think through how this would work for approach 2. We'd simply be maintaining a hash table of words to indexes. Each time a new word arrives, we'd need to add it to the hash table and also check which existing words it'd form a palindrome pair with. It's a little bit different to before, because we need to find all pairs with previous words that

For case 1, this is straightforward. We simply check if its reverse is already in the hash table. If it is, then we have 2 new pairs (the new word can be either first or second).

But it breaks for case 2 and case 3. It's straightforward to find pairs where our new word is the longer word of the pair (i.e. second in case 2 and first in case 3), however not where the new word is shorter. The problem is that the additional letters of the longer word could be anything, and therefore we have no way of knowing what to look up in the index. Approach 2 worked as an offline algorithm because pairs were always identified by starting with their longer word, and then looking up their shorter word. Going the other way is intractable.

Approach 3, however, works differently. If we build up a Trie as we go, we can always identify words from the Trie that will form the second half of the pair. It doesn't matter whether it is the current word, or the word from the Trie, that is longer. This solves half the problem—each time we get a new word, we can efficiently find all "second" words for it.

We aren't done yet though—the algorithm wouldn't find pairs where our current word was second. We still need to find a way of identifying all "first" words for the current word. It turns out that if we hadn't reversed words when putting them into the Trie, but instead had reversed the word we are looking up, that we'd be looking up "first" words in the Trie.

Therefore, we can make an online algorithm by maintaining 2 Tries—one with the words forward, and one with the words in reverse. The reverse Trie tells us where the new word will be the first word of a pair, and the forward Trie tells us where the new word will be the second of a pair.