Ask a Question

Introduction to Trie

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

We'll cover the following...

About the pattern

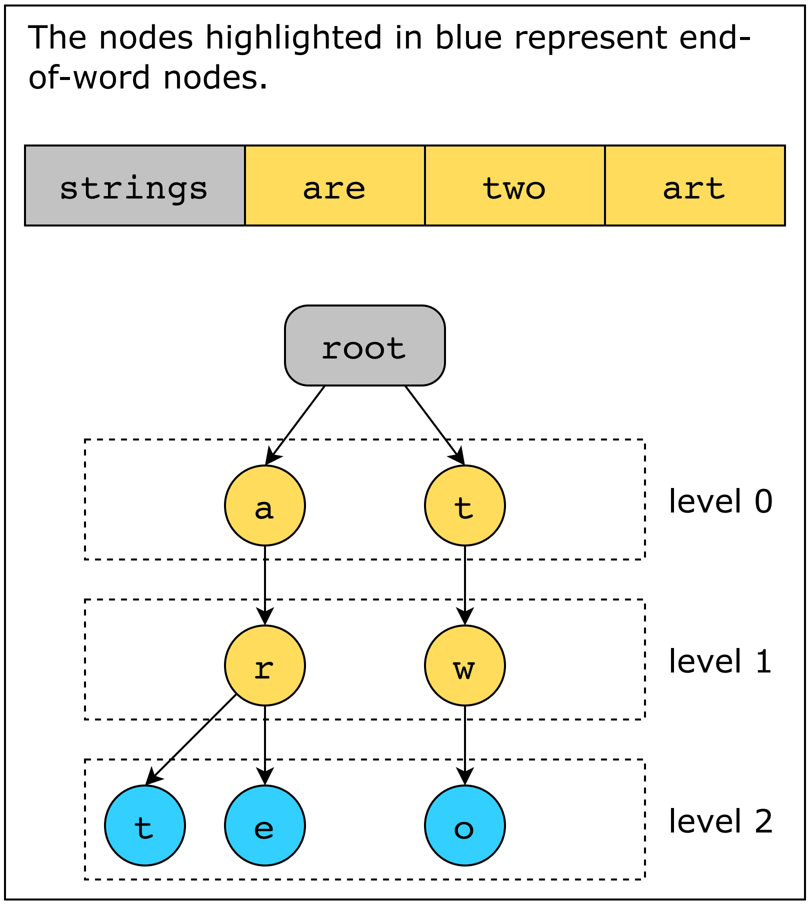
A **trie** is a tree data structure used for storing and locating keys from a set. The keys are usually strings that are stored character by character—each node of a trie corresponds to a single character rather than the entire key.

Below are the key characteristics of a trie:

- The order of characters in a string is represented by edges between the adjacent nodes. For example, in the string “are”, there will be an edge from node `a` to node `r` to node `e`. That is, node `a` will be the parent of node `r`, and node `r` will be the parent of node `e`.
- The level of nodes signifies the position of characters within a word. Each level corresponds to a specific index in the word being represented. In other words, at any given level, each node corresponds to a distinct character in the words stored in the trie. A we traverse down the trie from the root to a leaf node, the character encountered along the path collectively form the word associated with that path.

- It contains end-of-word nodes that mark the conclusion of a word within the trie structure. Each end-of-word node signifies the termination point of a word stored in the trie. This characteristic is crucial for efficient word retrieval and validation operations, as it allows the trie to distinguish between prefixes and complete words during searches or insertions.

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



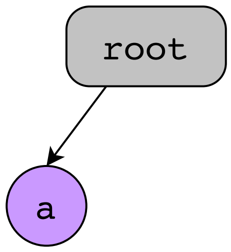
This way, additional space is not required for storing strings with common prefixes. We can keep moving down the tree until a new character that's not present in the node's children is encountered and add it as a new node. Similarly, searches can also be performed using depth-first search by following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

Below is an explanation of the staple methods of a trie:

- **Insert(word):** We start at the root and traverse down the trie, creating new nodes if required for each character in the string. The time complexity of this method is $O(m)$, where m is the length of the word. This is because we have to traverse the trie through the length of the word.
- **Search(word):** We start at the root and traverse down the trie, following the path that corresponds to the characters of the target word. If we encounter a null pointer or reach the end of a word before reaching a leaf node, the word is not present in the trie. The time complexity of this method is $O(m)$ because we have to traverse the trie through the length of the word.
- **Delete(word):** We start at the root and traverse down the trie, following the path that corresponds to the characters of the target word. If found, we remove the nodes corresponding to the characters of the string. This operation may also include cleanup to remove any unnecessary nodes to maintain the trie's efficiency. The time complexity of this method is $O(m)$ because we have to traverse the trie through the length of the word.

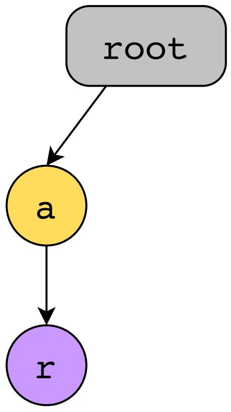
The following illustration shows an example of these methods being used in a trie:

insert(“are”): Because an existing node for ″a″ isn′t available, we create a new node.



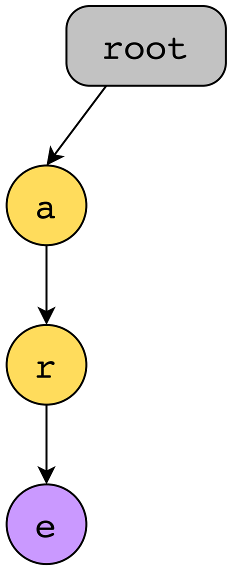
1 of 16

insert(“are”): Because an existing node for ″r″ isn′t available, we create a new node.



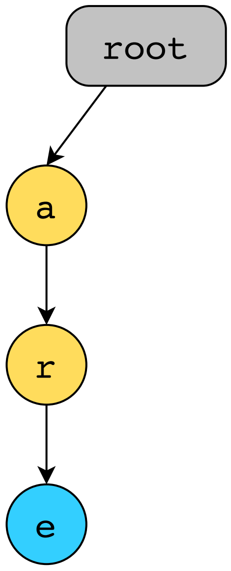
2 of 16

insert(“are”): Because an existing node for "e" isn't available, we create a new node. We have now inserted the word "are".



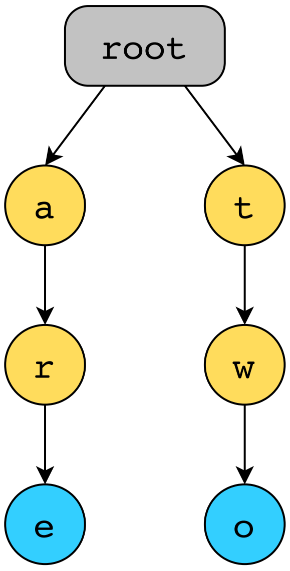
3 of 16

insert("are"): We mark the current node as an end-of-word node.



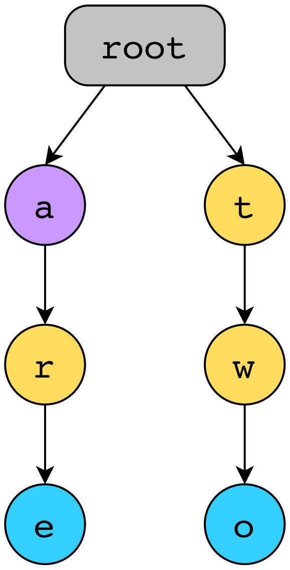
4 of 16

insert(“two“): We will insert this word in the same way in which we inserted the previous word.



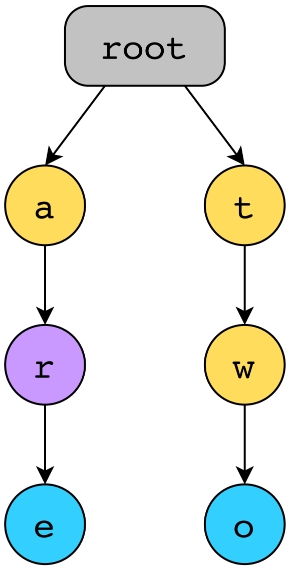
5 of 16

insert(“art“): Because a node for ″a″ is already available, we will not create a new node and simply traverse this node.



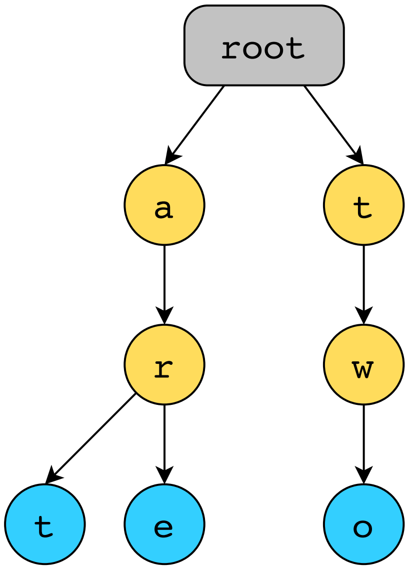
6 of 16

insert(“art“): Because a node for ″r″ is already available, we will not create a new node and simply traverse this node.



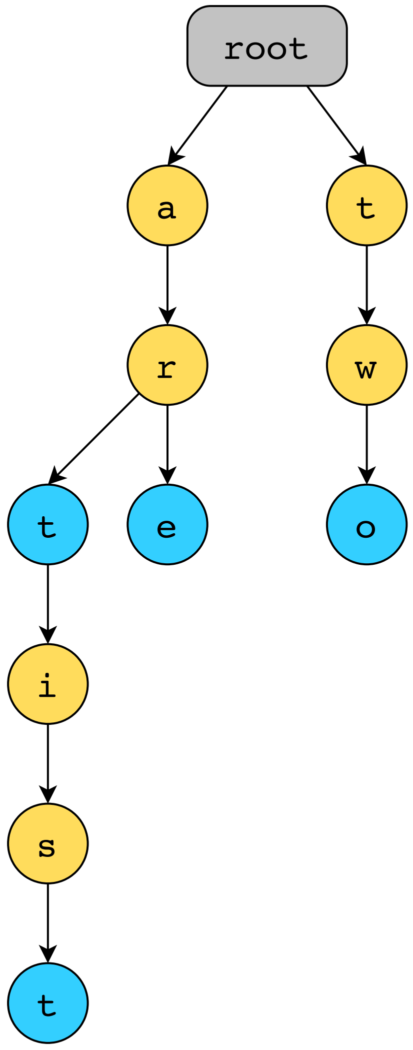
7 of 16

insert(“art“): Because an existing node for "t" isn't available, we create a new node. We have now inserted the word "art" and mark node "t" as an end-of-word node.



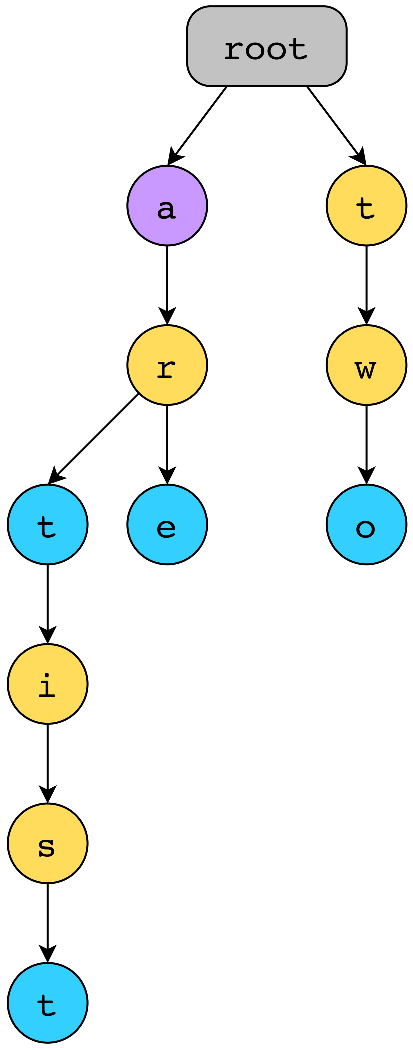
8 of 16

insert(“artist“): Because existing nodes for nodes "a" and "r", and "t" are available, we don't recreate them, and insert the remaining nodes.



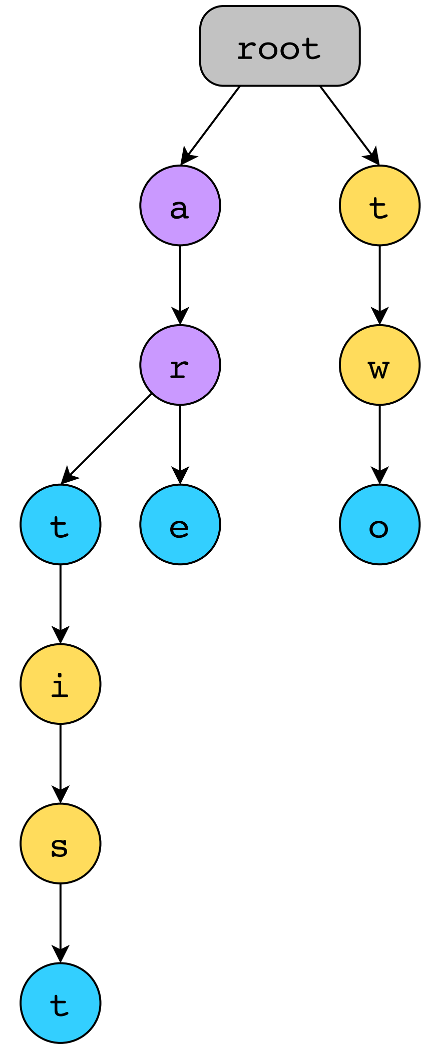
9 of 16

search("art"): We traverse the node "a".



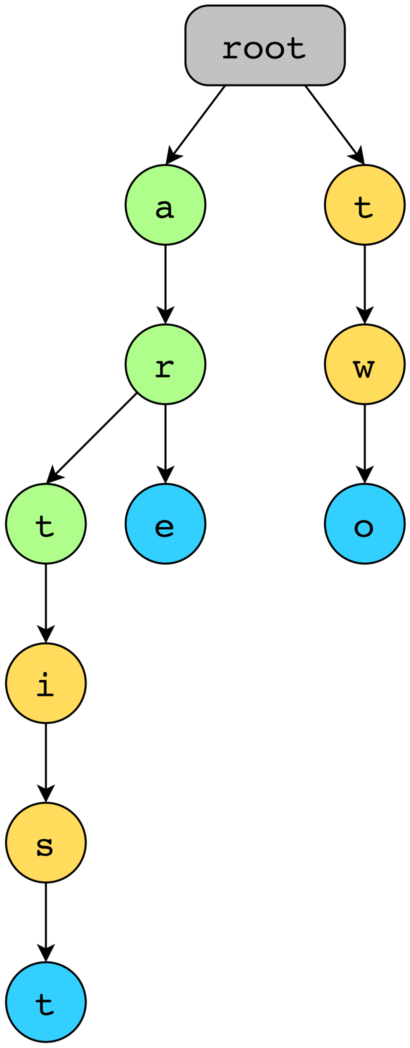
10 of 16

search(“art“): We traverse the node ″r″.



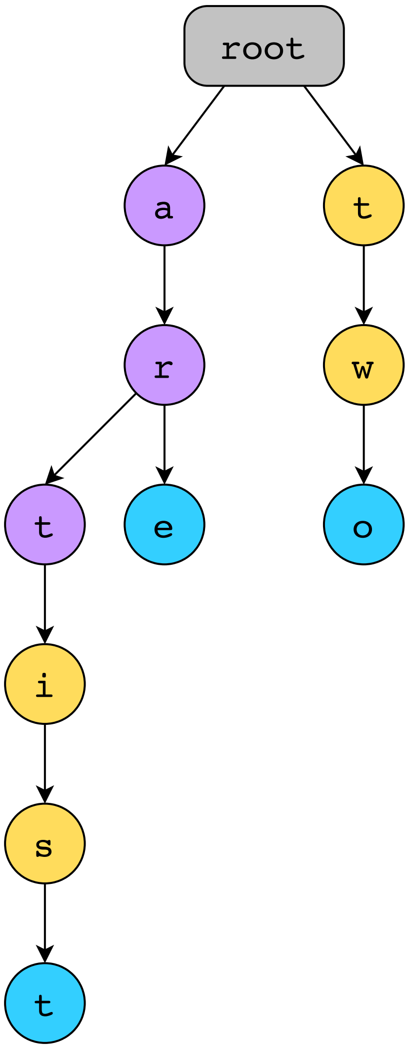
11 of 16

search("art"): We traverse the node "t". Because all the characters of the word have been traversed in order, and the last node is also an end-of-word node, we have found the word "art".



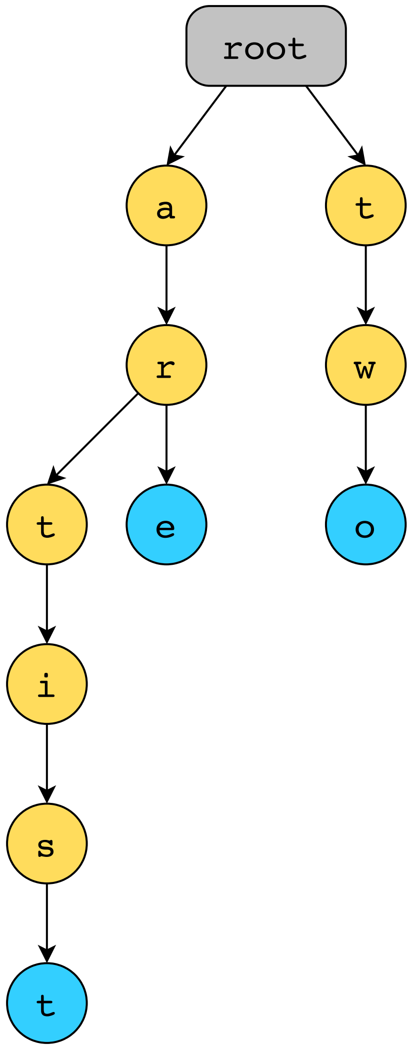
12 of 16

delete(“art“): We will first search for the word as described previously.



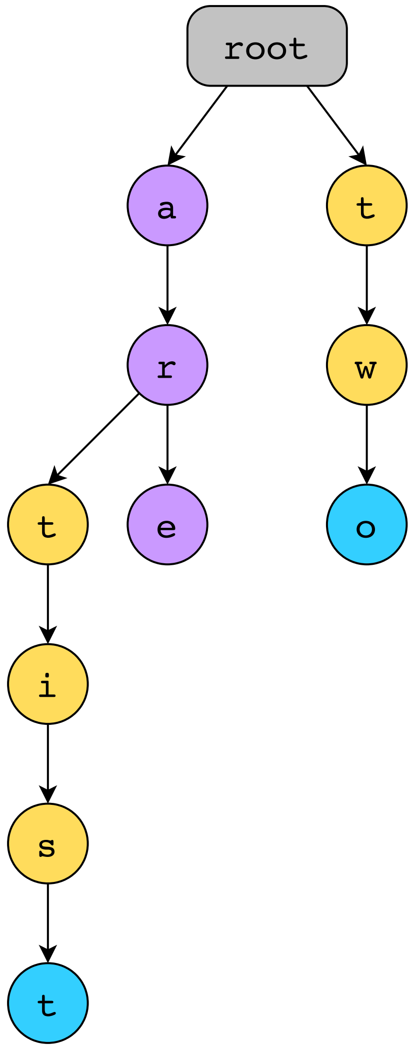
13 of 16

delete("art"): Because the prefix "are" is also present in "artist", we can not remove the "a", "r" and "t" nodes. So we simply remove the end-of-word property from node "t".



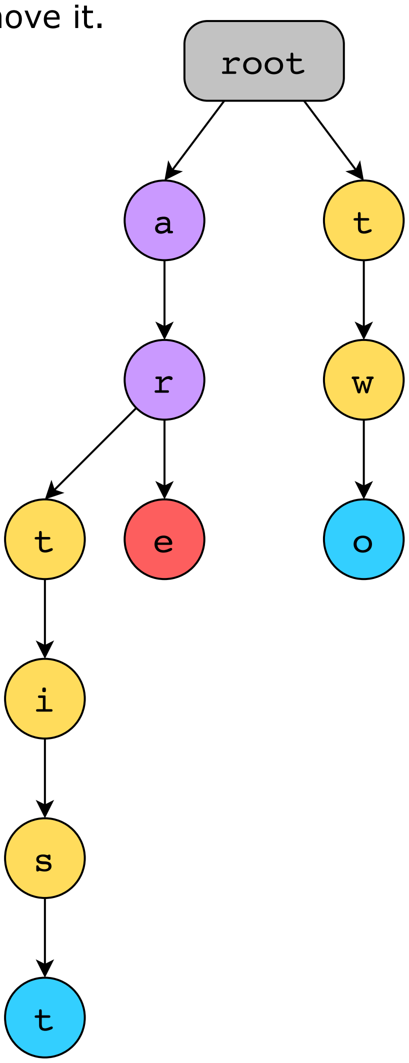
14 of 16

delete(“are“): We will first search for the word as described previously.



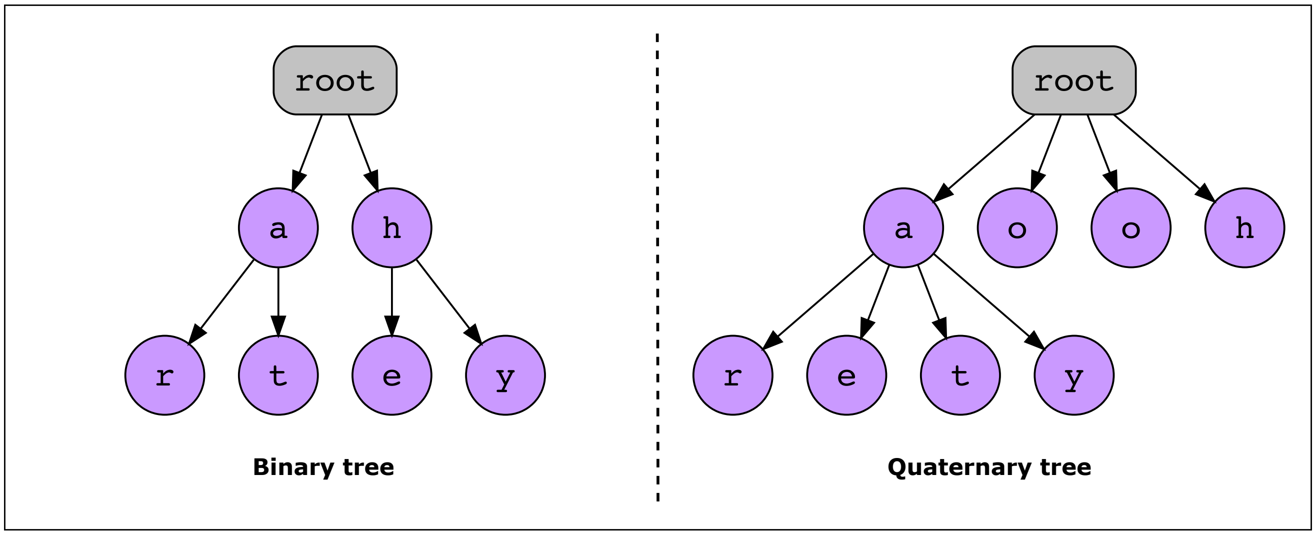
15 of 16

delete("are"): Because the prefix "ar" is also present in "artist", we can not remove nodes "a" and "r".
However, Because node "e" is not connected to any other node, we can safely remove it.



16 of 16

To understand how tries are more efficient for storing and searching strings, consider a binary tree. The time complexity of a binary tree is $O(\log n)$, where we talk in terms of $\log$ base 2. Instead, think of a quaternary tree, where every node has a fan-out of four, so each node c have four children. Although the time complexity of this tree is still $O(\log n)$, we're now talking in terms of $\log$ with base 4. That's an improvement in the performance even if it's by a constant factor. As our trees become wider and shorter, the operations become more efficient. This is because we don't have to traverse as deep.



This is exactly the motivation behind a trie. What if we had an n-ary tree with the fan-out equal to the number of unique values in the given dataset? For example, if we're considering strings in English, the fan-out would be 26, corresponding to the number of letters in the English language. This makes the tree wider and shorter! The maximum depth of the trie would be the maximum length of a word or string.

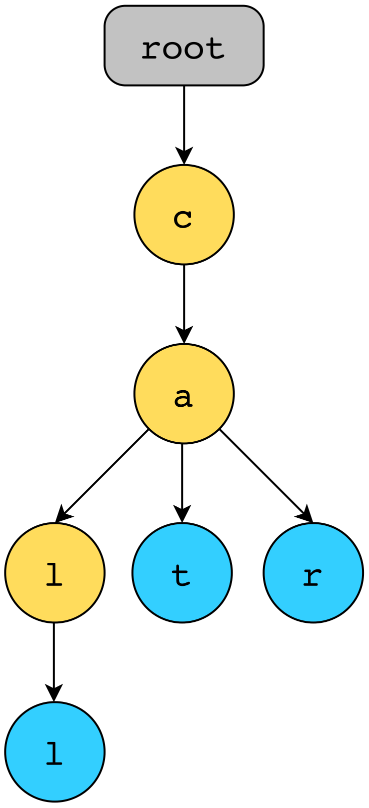
Examples

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

1. **Longest common prefix:** Find the longest common prefix that is shared among a given list of strings.

We will first insert the words in the trie.

strings	cat	call	car



1 of 3

Next, we traverse the trie from the root, keeping track of the common prefix encountered so far until we reach a node with multiple children, or a leaf node.

strings	cat	call	car