TRIE

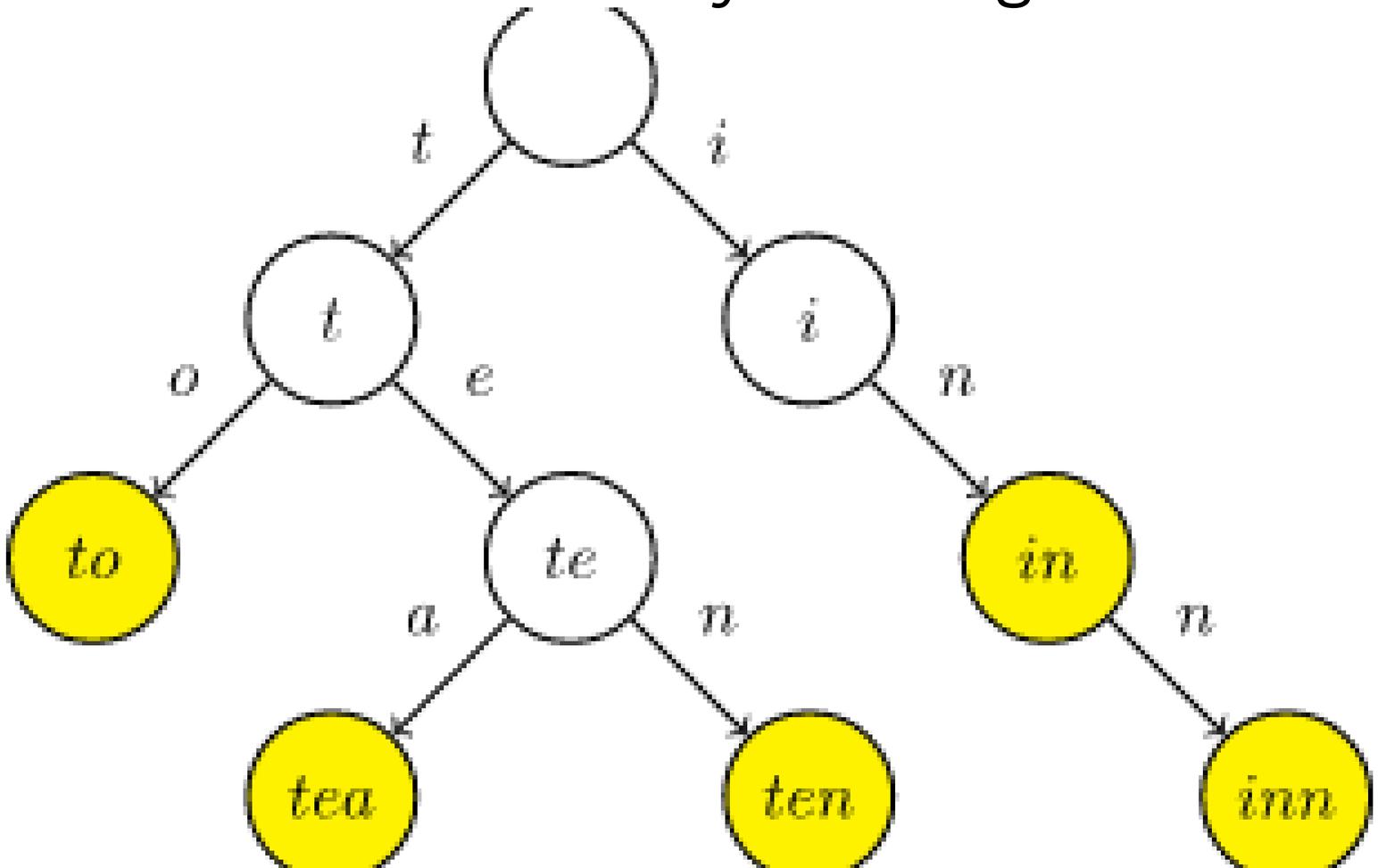
PROBLEM :Given a dictionary that contains a list of strings, and a string , we want to check whether or not is in the dictionary.

1. A hash table (also called a hash map) is a data structure that is used to map keys to values in an unsorted way. In our problem, we can treat each string in the dictionary as a key to the hash table.

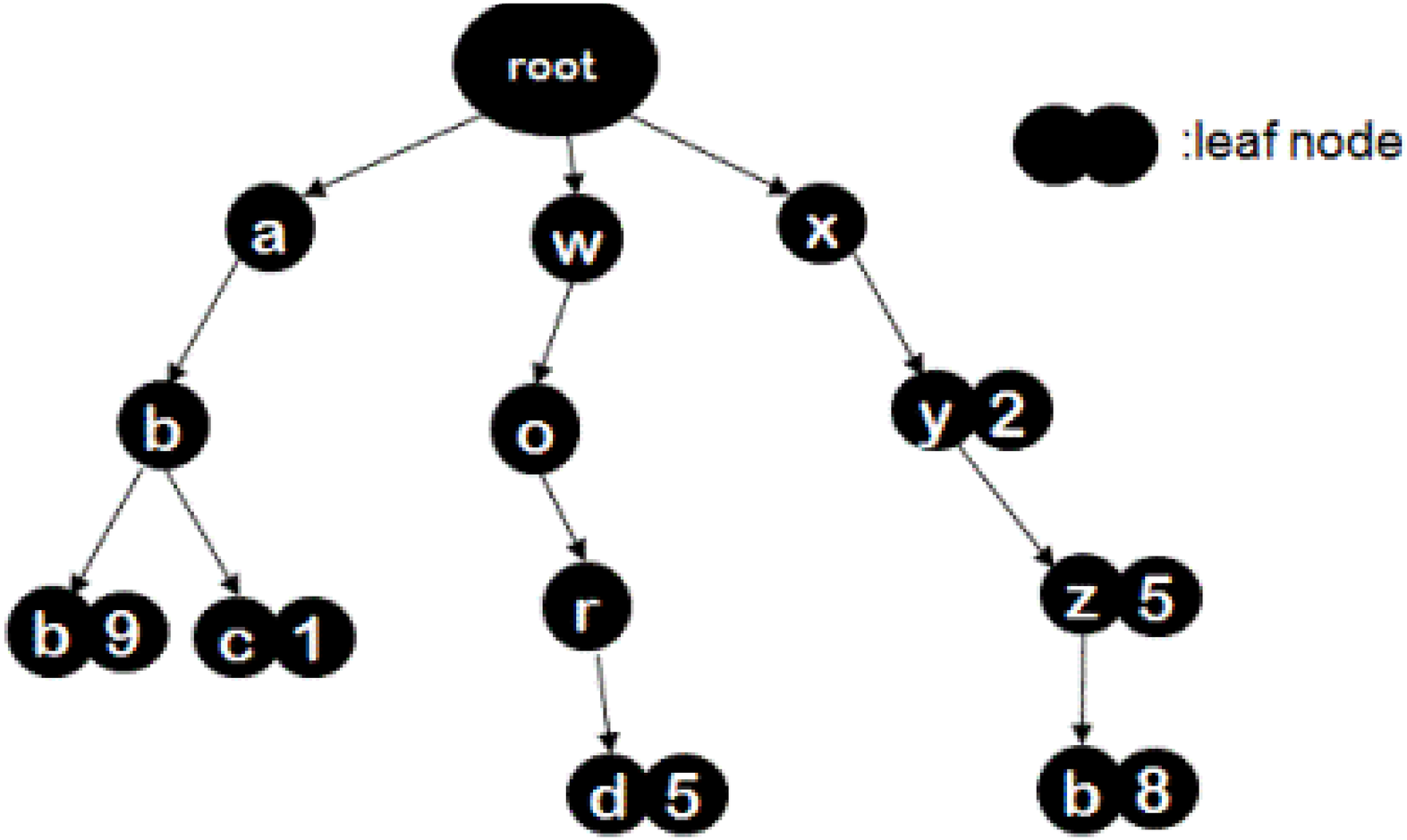
Time complexity : Since a hash function needs to consider all characters of the input string, it is o(n) , where n is the length of the input string.

2. A trie or a prefix tree is a particular kind of search tree, where nodes are usually keyed by strings.

In a trie, a link between two nodes represents a character in the keyed string.



how a trie data structure looks like for key, value pairs ("abc",1),("xy",2),("xyz",5), ("abb",9)("xyzb",8), ("word",5).



Comparisons

Lookup Speed

- When we look up a string for a hash table, we first calculate the hash value of the string, which takes o(n) time. Then, it will take o(1) time to locate the hash value in the memory, assuming we have a good hash function. Therefore, the overall lookup time complexity is o(n)
- When we look up a string for a trie, we go through each character of the string and locate its corresponding node in the trie. The overall lookup time complexity is also o(n)
- However, the trie has some more overhead to retrieve the whole string. We need to access the memory multiple times to locate the trie nodes along the character path. For the hash table, we only need to compute the hash value for the input string once. Therefore, it is relatively faster when we look up a whole string in the hash table.

Memory Requirement

- When we first construct a hash table, we normally pre-allocate a big chunk of memory to avoid collisions by hashing uniformly on the size of the memory. In th future when we insert a string into the hash table, we only need to store the strin content.
- For a trie data structure, we need to store extra data such as character link pointers and complete node flags. Therefore, the trie requires more memory to store the string data.
- However, if there are many common prefixes, the memory requirement becomes smaller as we can share the prefix nodes.

Overall, the memory requirement between a hash table and a trie is based on the size of pre-allocated hash table memory and input dictionary strings.

Applications

Trie : can quickly look up prefixes of keys, enumerate all entries with a given prefix, etc.

Trie advantages :

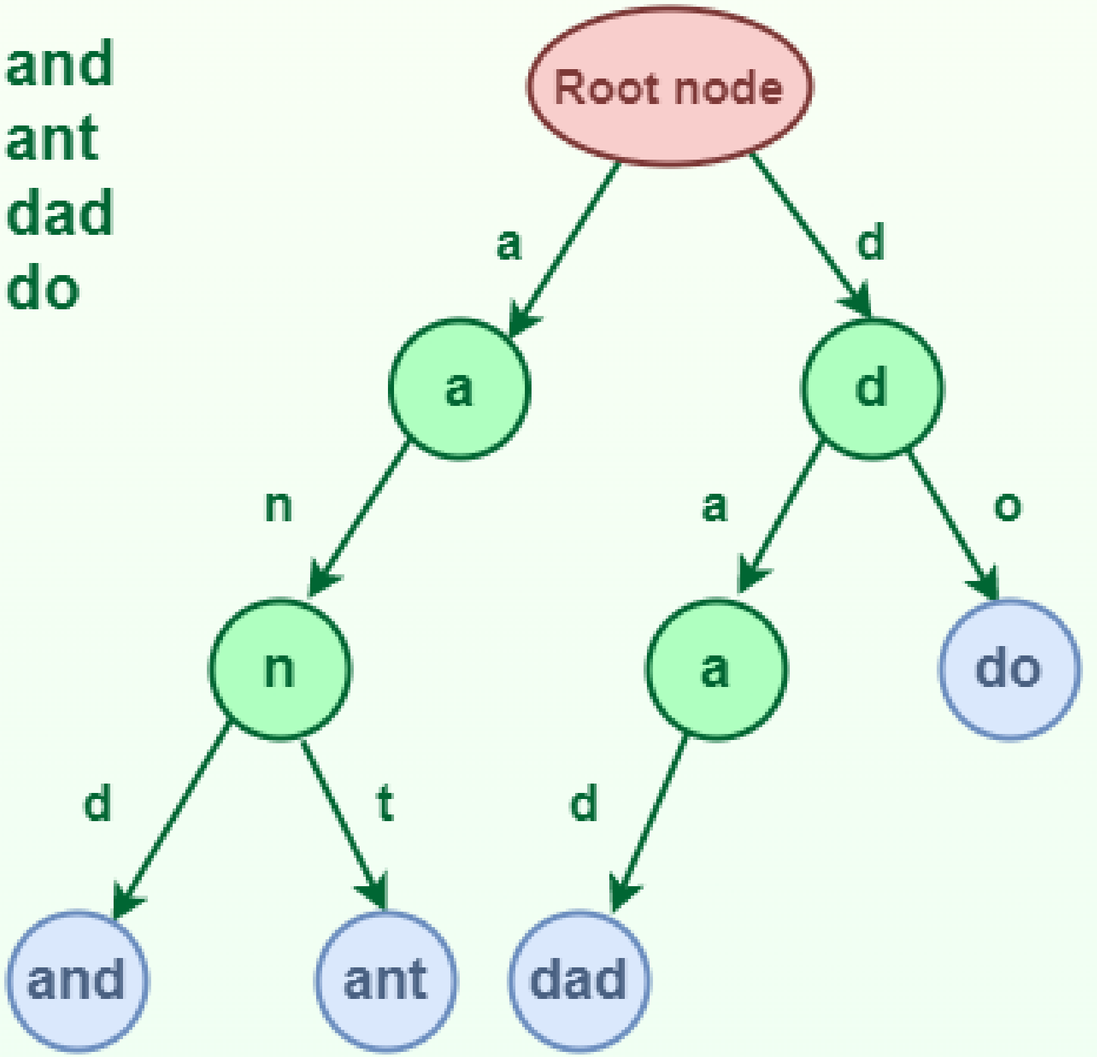
- Predictable O(k) lookup time where k is the size of the key
- Lookup can take less than k time if it's not there
- Supports ordered traversal
- No need for a hash function
- Deletion is straightforward
- it **all depends on what problem you're trying to solve. If all you need to do is insertions and lookups, go with a hash table. If you need to solve more complex problems such as prefix-related queries, then a trie might be the better solution.**

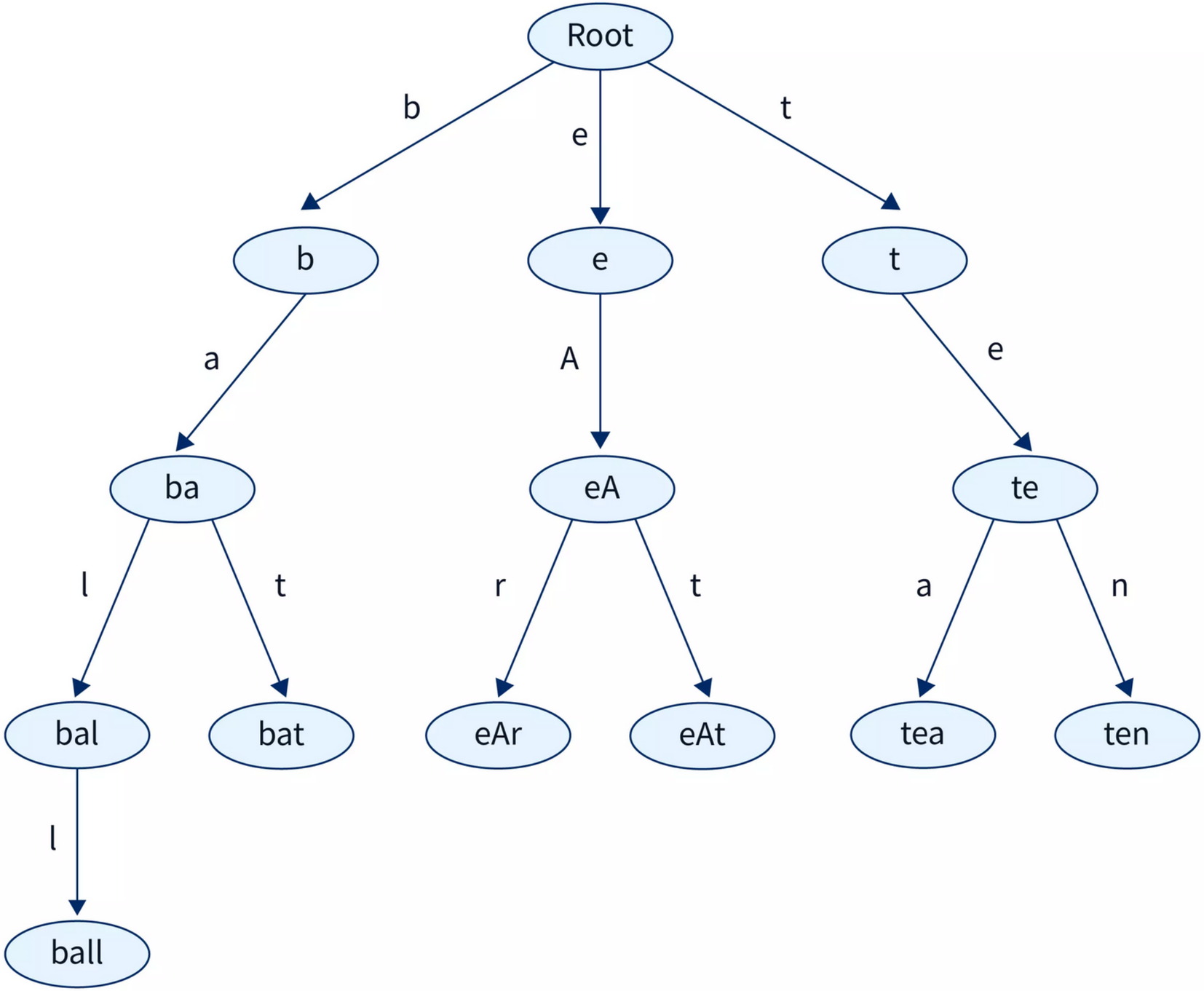
Preprocessing pattern improves the performance of pattern matching algorithm. But if a text is very large then it is better to preprocess text instead of pattern for efficient search.

A trie is a data structure that supports pattern matching queries in time proportional to the pattern size.

Trie Data Structure

- and
- ant
- dad
- do



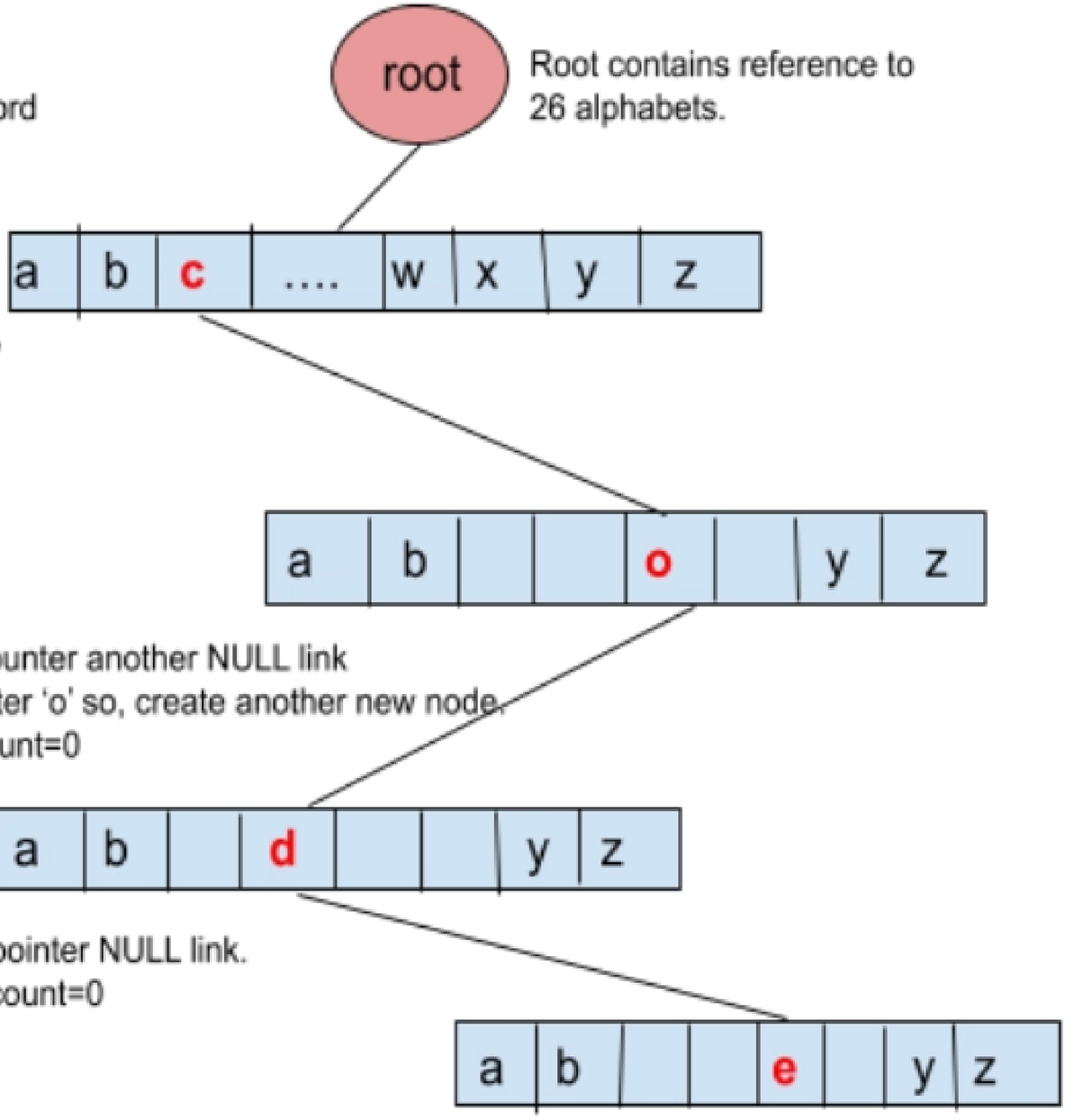

Root
b
e
t
b
e
t
a
A
e
ba
eA
te
l
t
r
t
a
n
bal
bat
eAr
eAt
tea
ten
l
ball

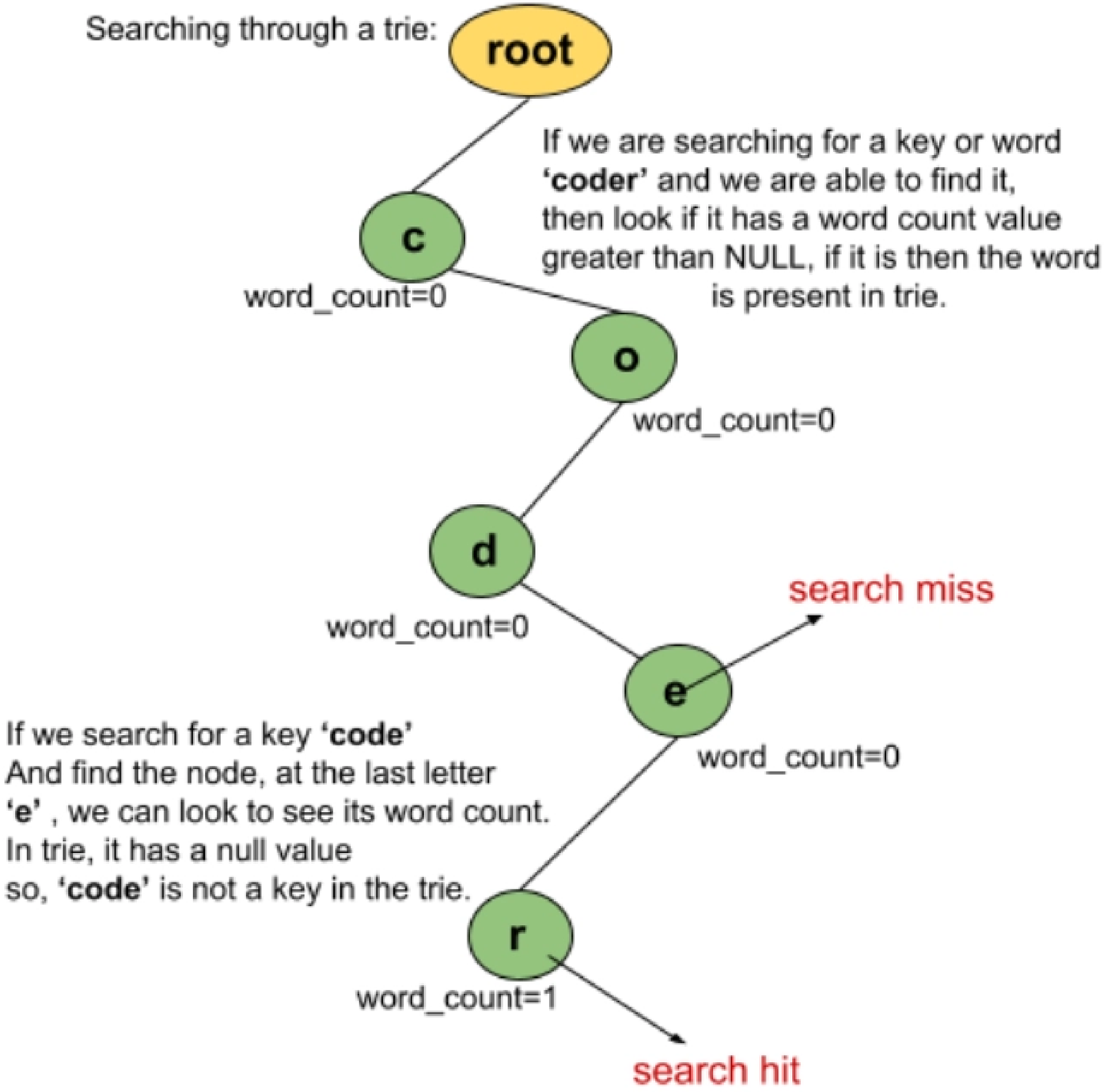
A Trie node field ‘wordEndCnt’ is used to distinguish the node as the end of the word node.

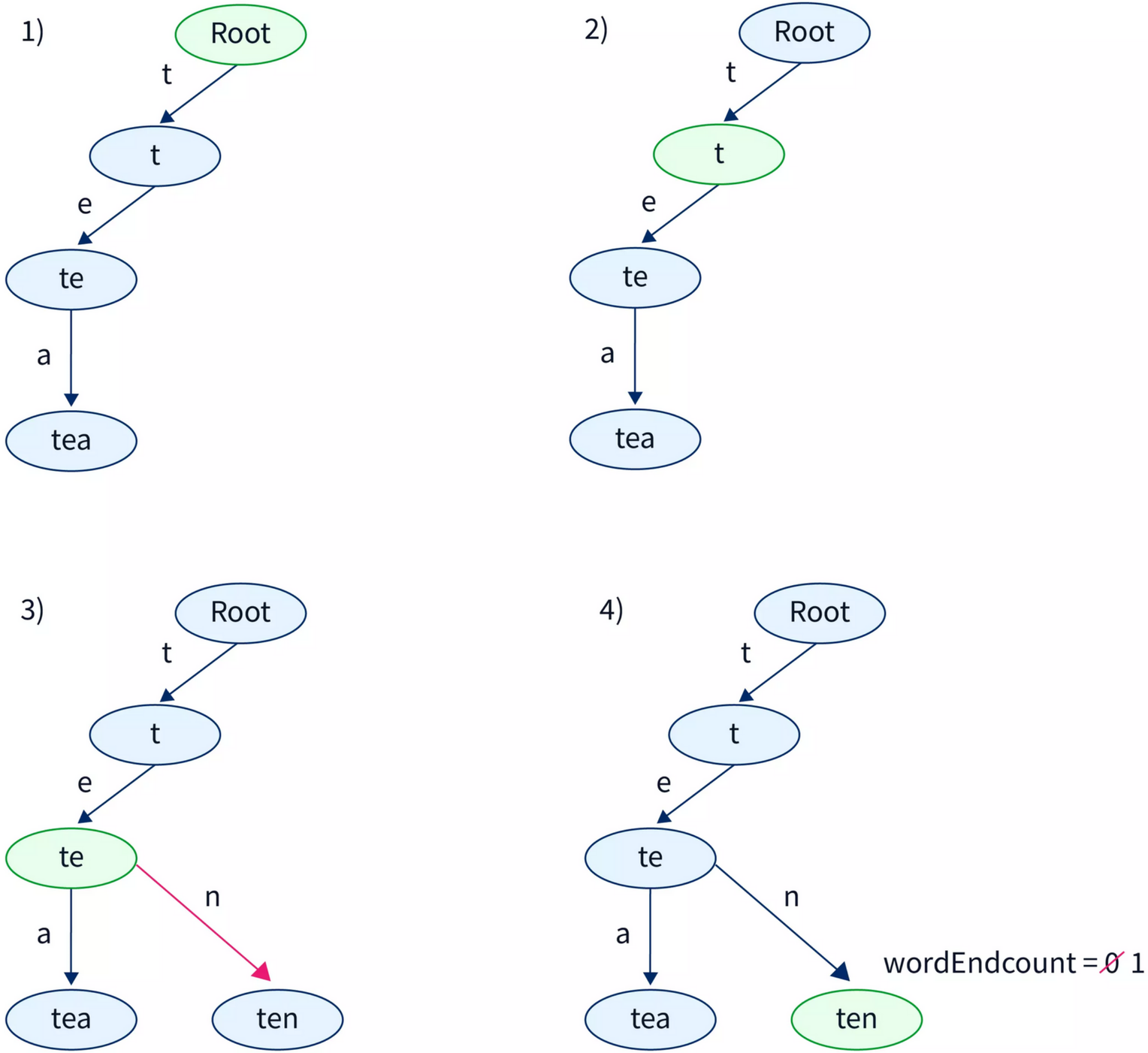
Insert Operation in Trie:

- Every character of the input key is inserted as an individual Trie node. Note that the children is an array of pointers (or references) to next-level trie nodes.
- The key character acts as an index to the array children.
- If the input key is new or an extension of the existing key, construct non-existing nodes of the key, and mark the end of the word for the last node.
- If the input key is a prefix of the existing key in Trie, Simply mark the last node of the key as the end of a word.
- The key length determines Trie depth.

First look for pointer for ‘c’
As we want to insert the word
‘code’.




Searching through a trie:
root
c
word_count=0
If we are searching for a key or word
'coder' and we are able to find it,
then look if it has a word count value
greater than NULL, if it is then the word
is present in trie.
o
word_count=0
d
word_count=0
search miss
e
word_count=0
If we search for a key 'code'
And find the node, at the last letter
'e' , we can look to see its word count.
In trie, it has a null value
so, 'code' is not a key in the trie.
r
word_count=1
search hit


1)
Root
t
t
e
te
a
tea
2)
Root
t
t
e
te
a
tea
3)
Root
t
t
e
te
n
a
tea
ten
4)
Root
t
t
e
te
n
a
wordEndcount = 0 1
tea
ten

Initially, the current node is the root node. The first character of the input string is t. As the t-a th index of the childNode pointer array is not NULL we will not create a new node. The current node is shifted from the root node to the node pointed by t-a th index of the childNode pointer array of the root node

Now we check for the second character of the input string,i.e e. As the e-a th index of the current node is also not null we shift the current node to the node pointed by e-a th index of the current node.

Now we check for the last character of the input string, i.e n. As the n-a th index of the current node is NULL we make a new node and follow the same steps as discussed above. We increment the wordEndCnt variable for this node as this character is the last character of the input string.

Trie insertion

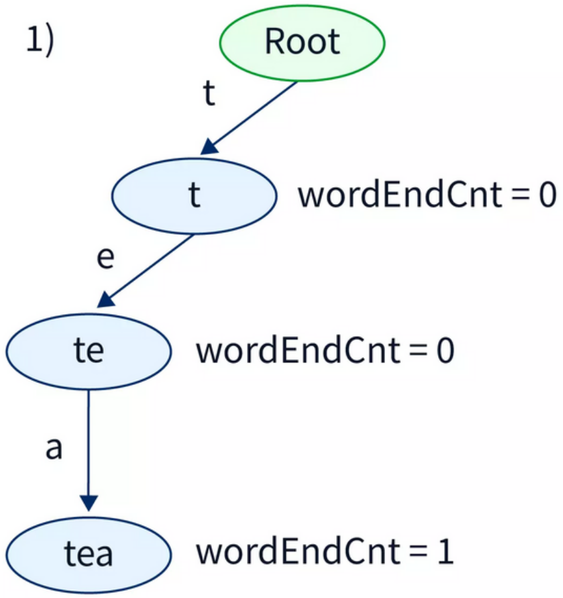
```
struct TrieNode {    //pointer array for child nodes of each node
 TrieNode *childNode[26];
   int wordEndCnt;
   //constructor    TrieNode()
   {
      //initialize the wordEndCnt variable with 0       //initialize every index of
childNode array with NULL
   wordEndCnt = 0;
      for (int i = 0; i < 26; i++)
      {
         childNode[i] = NULL;
      }
   }
};
```

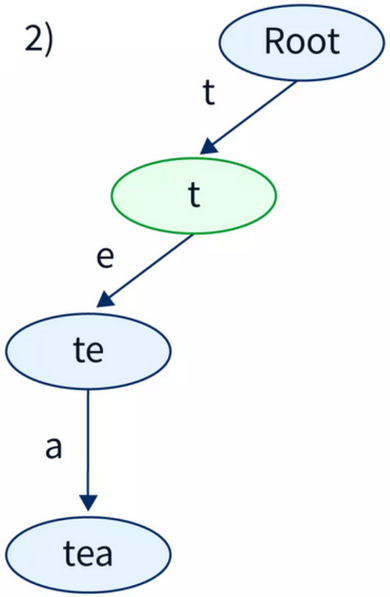
- Each TrieNode will have 26 children from a-z represented by a character pointer array.

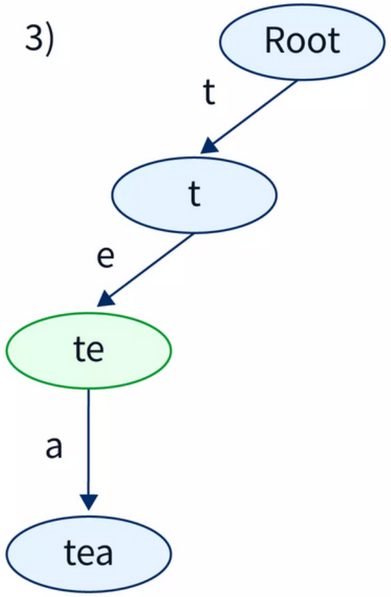
- Each node will have a wordEndCnt integer variable. This variable will store the count of the strings in the Trie which are the same as that of the prefix represented by that node of the Trie.

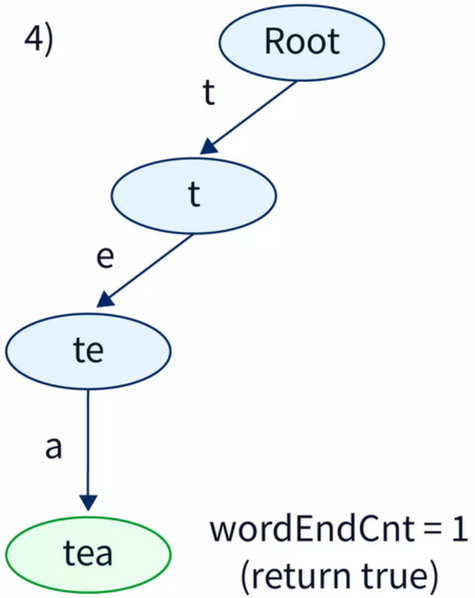
- Inside the structure of a TrieNode we made a constructor TrieNode() which will initialize every index of the childNode pointer array with NULL whenever a new node is created. It will also initialize the wordEndCnt value for every node with 0.

```
TrieNode* insert_key(TrieNode *root, string &key){
 //initialize the currentNode pointer with the root node TrieNode *currentNode = root;

 //Store the length of the key string
 int length = key.size();
 //iterate across the length of the string
for (int i = 0; i < length; i++)
 {
  //Check X-'a' th index is NULL or not
 if (currentNode->childNode[key[i] - 'a'] == NULL)
  {
   //If null make a new node   TrieNode * newNode = new TrieNode();
   //Point the X-'a' th index of current node to the new node
 currentNode->childNode[key[i] - 'a'] = newNode;     }

  //Move the current node pointer to the newly created node.
currentNode = currentNode->childNode[key[i] - 'a'];
}
  currentNode->wordEndCnt++;

 //return the updated root node return root;}
```


1)
Root
t
t
wordEndCnt = 0
e
te
wordEndCnt = 0
a
tea
wordEndCnt = 1


2)
Root
t
t
e
te
a
tea


3)
Root
t
t
e
te
a
tea


4)
Root
t
t
e
te
a
tea
wordEndCnt = 1
(return true)

Initially, the current node is the root node. The first character of the query string is t. We check whether the t-a th index of the childNode pointer array of the current node is NULL or not. As it is not null we move the current node pointer to the node pointed by t-a th index of the current node which is the root node in step 1.

Now we check for the second character of the query string i.e e. As the e-a th index of the current node is also not NULL. We move the current node pointer to the node pointed by e-a th index of the current node.

Then, we process the last character of the query string, i.e a. As the a-a index of the current node is not equal to null we move the current node pointer to the node pointed by the a-a th index of the childNode pointer array of the current node.

Finally, we check if the wordEndCnt for the current node is greater than 0 or not. If the wordEndCnt is greater than 0 which is the case for the query string tea we return true. Otherwise, we return false.

discuss the case in which the query string is not present in the Trie. Lets us try to search the string to in the Trie.

Search Operation : "to"

We start with the root node and check for the first character of the query string, i.e t. As the t-a th index is not NULL. We move the current node pointer to the node pointed by the t-a th index of the current node.
In step 2 we look for the character o in the current node. However, as the o-a th index of the current node is equal to NULL. This means that the query string to is not present in the trie. Thus we return false in this case.

Implementation of the Search Operation in a Trie Data Structure

```
bool search_key(TrieNode *root, string &queryString){
 //Initialize the currentNode pointer with the root node TrieNode *currentNode = root;

 //Store the length of the query string int length = queryString.size();
 for (int i = 0; i < length; i++)
 {
  //Check if the X-'a' th index is NULL or not  if (currentNode->childNode[queryString[i] - 'a'] == NULL)
  {return false;}
   //If null then the query string is not present in the Trie   //return false

  //If not NULL  //Move the currentNode pointer to the node pointed by X-'a' th index of the  //
current node  currentNode = currentNode->childNode[queryString[i] - 'a'];
 }

 //If currentNode pointer is not NULL //and wordEndCnt for the currentNode pointer  //is greater than
0 then return true else //return false
 return true if  currentNode != NULL && currentNode->wordEndCnt > 0; else false
}
```

Deletion

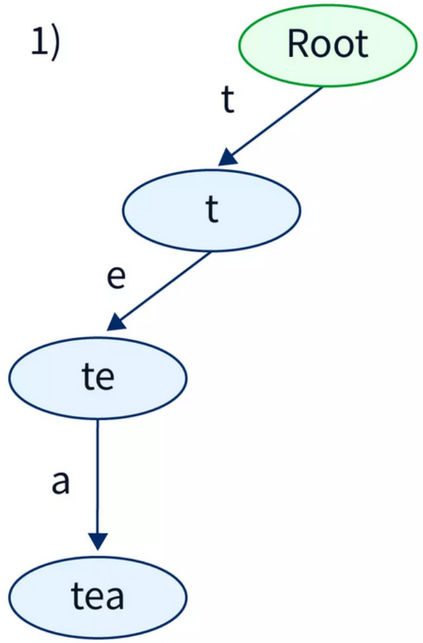
1. If key 'k' is not present in trie, then we should not modify trie in any way.
2. If key 'k' is not a prefix nor a suffix of any other key and nodes of key 'k' are not part of any other key then all the nodes starting from root node(excluding root node) to leaf node of key 'k' should be deleted.
3. If key 'k' is a prefix of some other key, then leaf node corresponding to key 'k' should be marked as 'not a leaf node'. No node should be deleted in this case.
4. If key 'k' is a suffix of some other key 'k1', then all nodes of key 'k' which are not part of key 'k1' should be deleted.

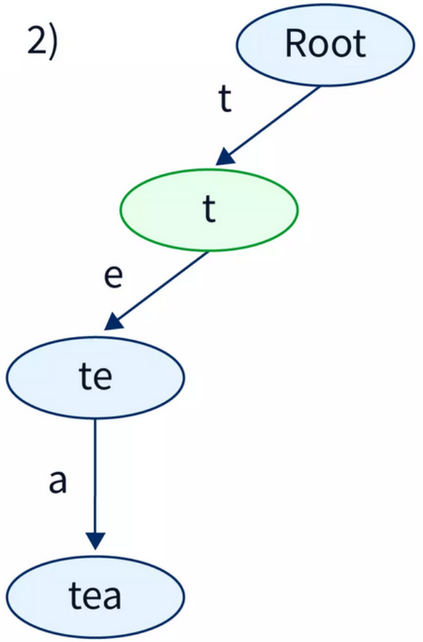
5. If key 'k' is not a prefix nor a suffix of any other key but some nodes of key 'k' are shared with some other key 'k1', then nodes of key 'k' which are not common to any other key should be deleted and shared nodes should be kept intact.

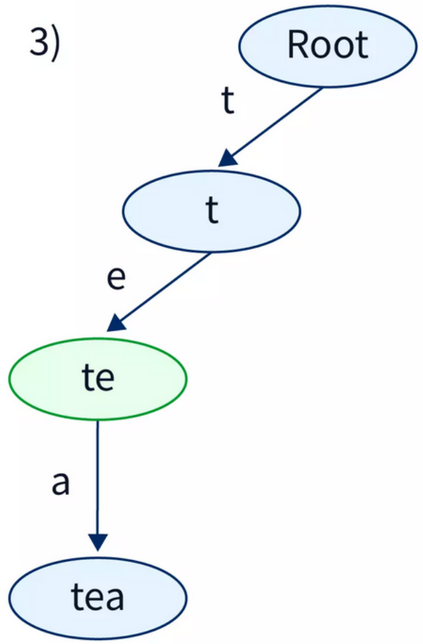
Delete Operation in Trie Data Structure.

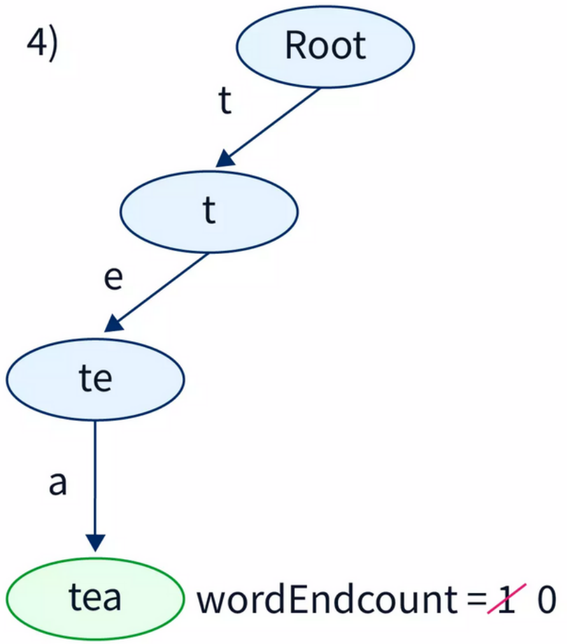
With the help of the delete operation, we can delete the occurrence of a string previously present in the Trie data structure. If a string is not present in the Trie data structure we can't delete that string. If a string is successfully deleted using the delete operation we return boolean True. If the string is not present in Trie then we return boolean false.

To implement the delete operation in a Trie data structure we first search whether the query string is present in the Trie or not. To search the string we apply the same logic discussed in the above section. If the string is not present in the Trie data structure we return false. If the string is present in the Trie data structure we reduce the wordEndCnt for the node pointe by the last current node pointer by 1.


1)
Root
t
t
e
te
a
tea

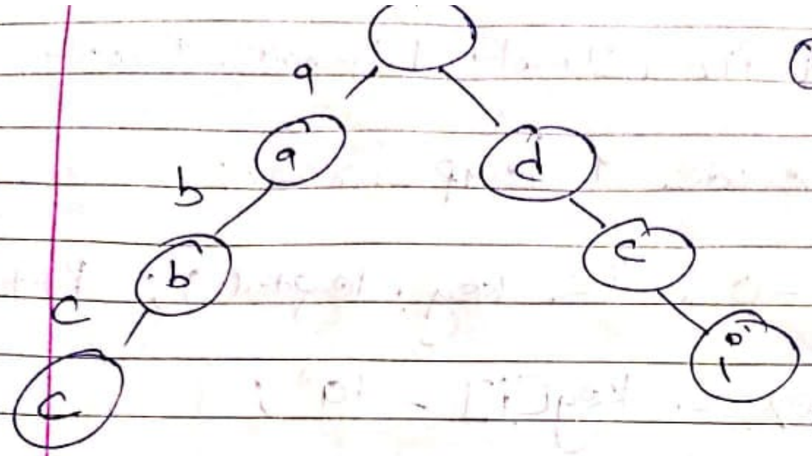

2)
Root
t
t
e
te
a
tea


3)
Root
t
t
e
te
a
tea


4)
Root
t
t
e
te
a
tea
wordEndcount = 1 0

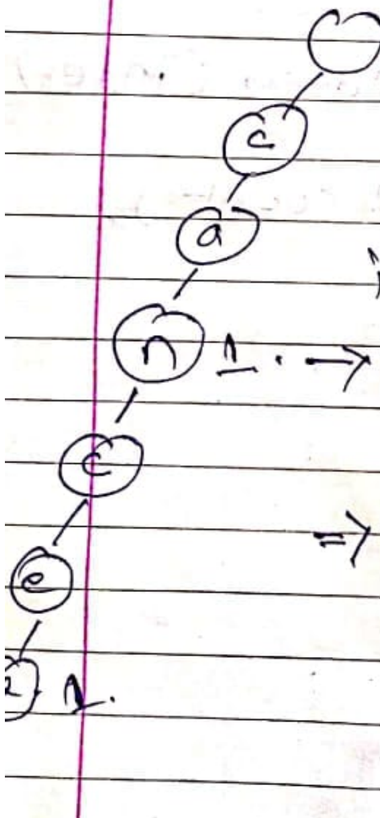
steps involved in deleting the string tea from the given Trie.

At first, we search for the string tea using the same logic discussed in the search operation in a Trie.
As the string tea is present in the Trie the current node pointer will move from the root node to the last node. The last node represents the string tea. Now we will check if the wordEndCnt for this node is greater than 0 or not. As in this case, the wordEndCnt for the node is greater than 0 we reduce the wordEndCnt for this node and return True.



① delete . All nodes

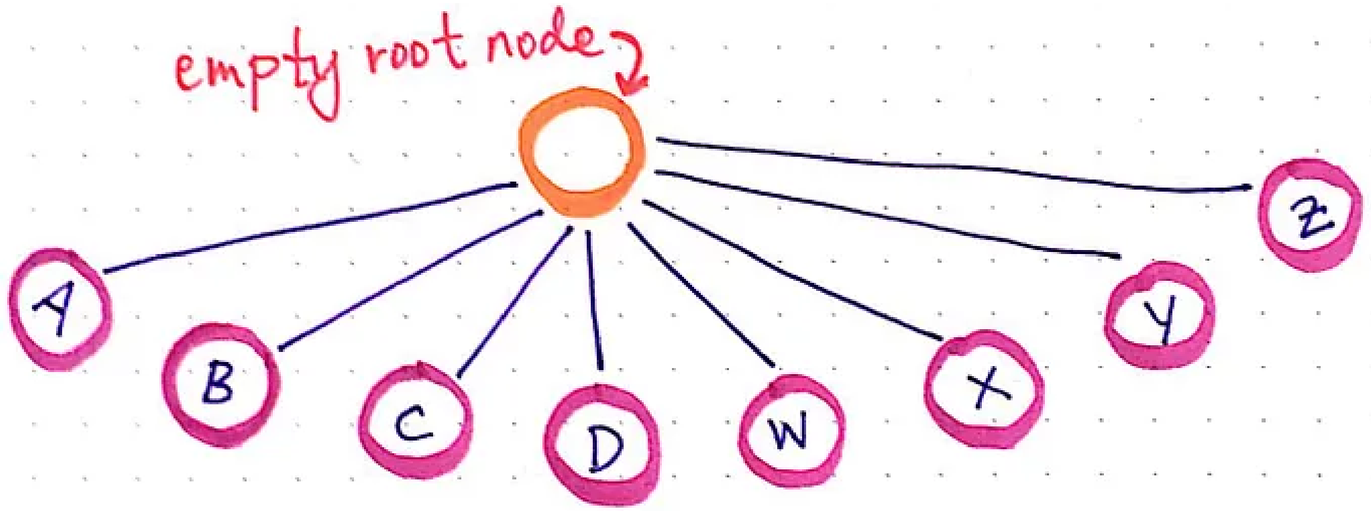
② Prefix of some other key



⇒ if delete "can" then leaf node corresp. to last key → word count = 0 / not a leaf node.

⇒ No node can be deleted here.

Compressed Trie



- The shape and structure of a trie is always a set of linked nodes, all connecting back to an empty root node. Each node contains an array of pointers (child "nodes"), one for each possible alphabetic value.

*Note: the size of a trie is directly connected/correlated to the size of the alphabet that is being represented.

compressed Trie

Standard Trie :The size of a trie is directly correlated to the size of all the possible values that the trie could represent.

Big O Notation of a trie structure:

- The worst case runtime for creating a trie structure is dependent on how many words the trie contains, and how long they might potentially be. This is known as $O(m \cdot n)$ time, where m is the longest word length and n is the total number of words.
- The time of searching, inserting, & deleting from a trie depends on the length of the word a and the total number of words: $O(a \cdot n)$

- The first thing that we'll notice when we look at this trie is that there are two keys for which we have ***redundant nodes*** as well as a ***redundant chain of edges***.
- A redundant node is one that takes up an undue amount of space because it only has one child node of its own. We'll see that for the key "deck", the node for the character "e" is redudant, because it only has a single child node, but we still have to initialize an entire node, with all of its pointers, as a result.
- Similarly, the edges that connect the key "did" are redundant, as they connect redundant nodes that don't *really* all need to be initialized, since they each have only one child node of their own.

The redundancy of a standard trie comes from the fact that we are repeating ourselves by allocating space for nodes or edges that contain only one possible string or word. Another way to to think about is that we repeat ourselves by allocating a lot of space for something that only has one possible branch path.

A compressed trie is a trie that has been compacted down to save space. This is done by following one crucial rule: each internal node (every parent node) has to have 2 or more children.

* In other words, a node must have more than one branch path to leaf nodes in order for it to have children.
* In order to "compact" the trie, each node that contained a single child is merged together — an only child node is merged with its parent.

RULE FOR COMPRESSED TRIE

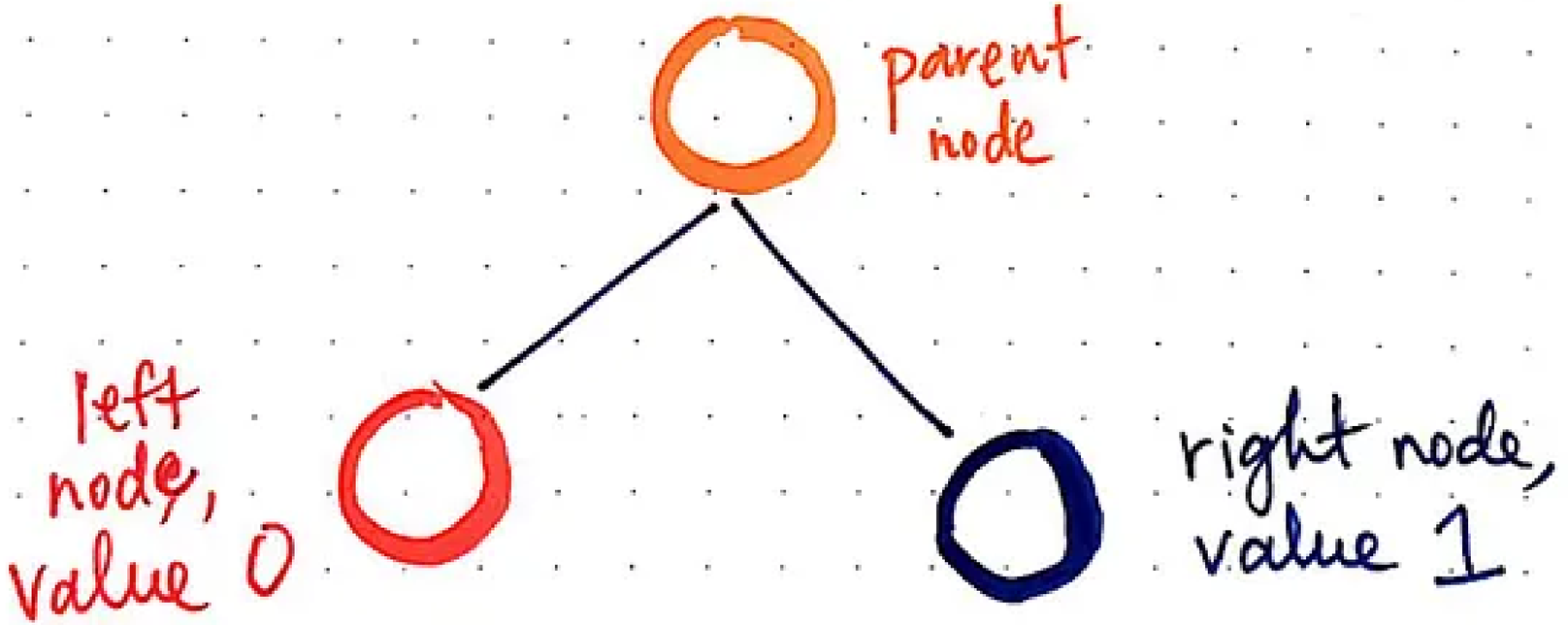
- *each internal node (every parent node) must have two or more child nodes*. If a parent has two child nodes, which is at two branch paths to potential leaf nodes, then it doesn't need to be compressed, since we actually need to allocate space and memory for both of these branch paths.
- However, if a parent node only has one child node — that is to say, if it only has one possible branch path to a leaf node — then it can be compressed. In order to do the work of "compacting" the trie, each node that is the only child of its parent node is merged into its parent. The parent node and the single-child node are fused together, as are the values that they contain.
- Compressed tries are also known as **radix trees, radix tries, or compact prefix trees.**
- **a space-optimized version of a standard trie. Unlike regular tries, the references/edges/pointers of a radix tree can hold a sequence of a string, and not just a single character element.**

PATRICIA tree.

- A trie’s keys could be read and processed a byte at a time, half a byte at a time, or two bits at a time. However, there is one particular type of radix tree that processes keys in a really interesting way, called a **PATRICIA tree**.
- PATRICIA stands for “Practical Algorithm To Retrieve Information Coded In Alphanumeric”.

- The most important thing to remember about a PATRICIA tree is that its radix is 2. Since we know that the way that keys are compared happens r bits at a time, where 2 to the power of r is the radix of the tree, we can use this math to figure out how a PATRICIA tree reads a key.

- Since the radix of a PATRICIA tree is 2, we know that r must be equal to 1, since $2^1 = 2$. Thus, a PATRICIA tree processes its keys one bit at a time.



* In a radix and (PATRICIA) tree, keys are read in streams of bits, and are compared r bits at a time, where 2^r is the radix of the tree. Since a PATRICIA tree's radix is 2, we compare 1 bit at a tim

Let’s say that we want to turn our original set of keys, ["dog", "doge", "dogs"] into a PATRICIA tree representation. Since a PATRICIA tree reads keys one bit at a time, we’ll need to convert these strings down into binary so that we can look at them bit by bit.

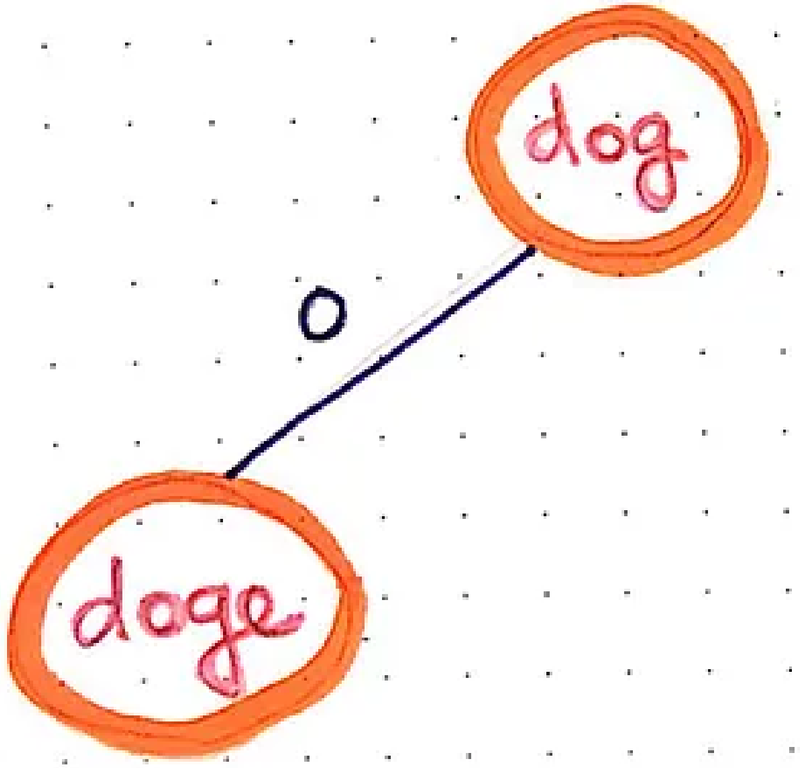
dog: 01100100 01101111 01100111

doge: 01100100 01101111 01100111 **0**1100101

dogs: 01100100 01101111 01100111 011**1**0011

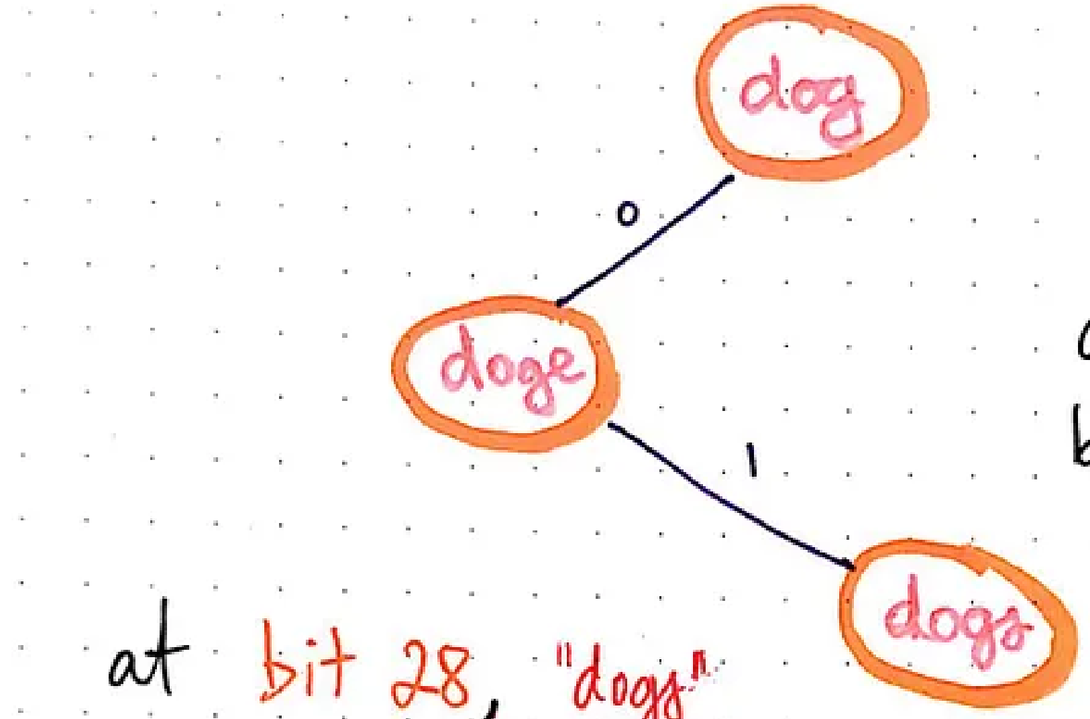
Notice how the keys "doge" and "dogs" are both substrings of "dog". The binary representation of these words is the exact same up until the 25th digit. Interestingly, even "doge" is a substring of "dogs"; the binary representation of both of these two words is the same up until the 28th digit!

so since we know that "dog" is a prefix of "doge", we will compare them bit by bit. The point at which they diverge is at bit 25, where "doge" has a value of 0. Since we know that our binary radix tree can only have 0’s and 1’s, we just need to put "doge" in the correct place. Since it diverges with a value of 0, we’ll add it as the left child node of our root node "dog".



"dog" is a prefix of "doge"; the point at which "doge" differs from "dog" is at bit 25, where "doge" has a value of 0.

Now we'll do the same thing with "dogs". Since "dogs" differs from its binary prefix "doge" at bit 28, we'll compare bit by bit up until that point.



the point at which "dogs" differs from its binary prefix "doge" is at bit 28

at bit 28, "dogs" has a bit value of 1, while "doge" has a bit value of 0.

Suffix trie

- Suffix tree is nothing but an extended version of trie. It's a compressed trie which includes all of a string's suffixes. There are some string-related problems which can be solved using suffix trees. Some of those problems are pattern matching, identifying unique substrings inside a string, and determining the longest palindrome. The suffix tree for the string S of length n is defined as a tree that has the following properties:

1. There are exactly n leaves on the tree, numbered 1 through n.
2. Each edge is identified by a non-empty S substring.
3. Every internal node has at least two child, with the exception of the root.
4. String-labels can't start with the same character on two edges that emerge from the same node.
5. The suffix S[i..n], for I from 1 to n, is formed by combining all the string-labels encountered on the path from the root to the leaf i.

Functionality of Suffix Tree

A suffix tree for a string S of length n can be created in Theta (n) time if the letters come from an alphabet of integers with a polynomial range of -infinity to +infinity (in particular, this is true for fixed-sized alphabets). The majority of the time is spent sorting the letters into an O(n)-sized range for larger alphabets; on average, it takes O(nlog n) time. Imagine that over the string S of length n, a suffix tree has been constructed, then you can:

1. Look for strings:

- In O(m) time, determine whether a string P of length m is a substring.
- In O(m) time, find the very first occurrence of the sequences P1...PQ with a total length of m as substrings.
- In O(m+z) time, find all z occurrences of the patterns P1...PQ of length m in substrings.

Trie applications

- Consider a web browser. Do you know how the web browser can auto complete your text or show you many possibilities of the text that you could be writing? Yes, with the trie you can do it very fast. Do you know how an orthographic corrector can check that every word that you type is in a dictionary? Again a trie. You can also use a trie for suggested corrections of the words that are present in the text but not in the dictionary.
-