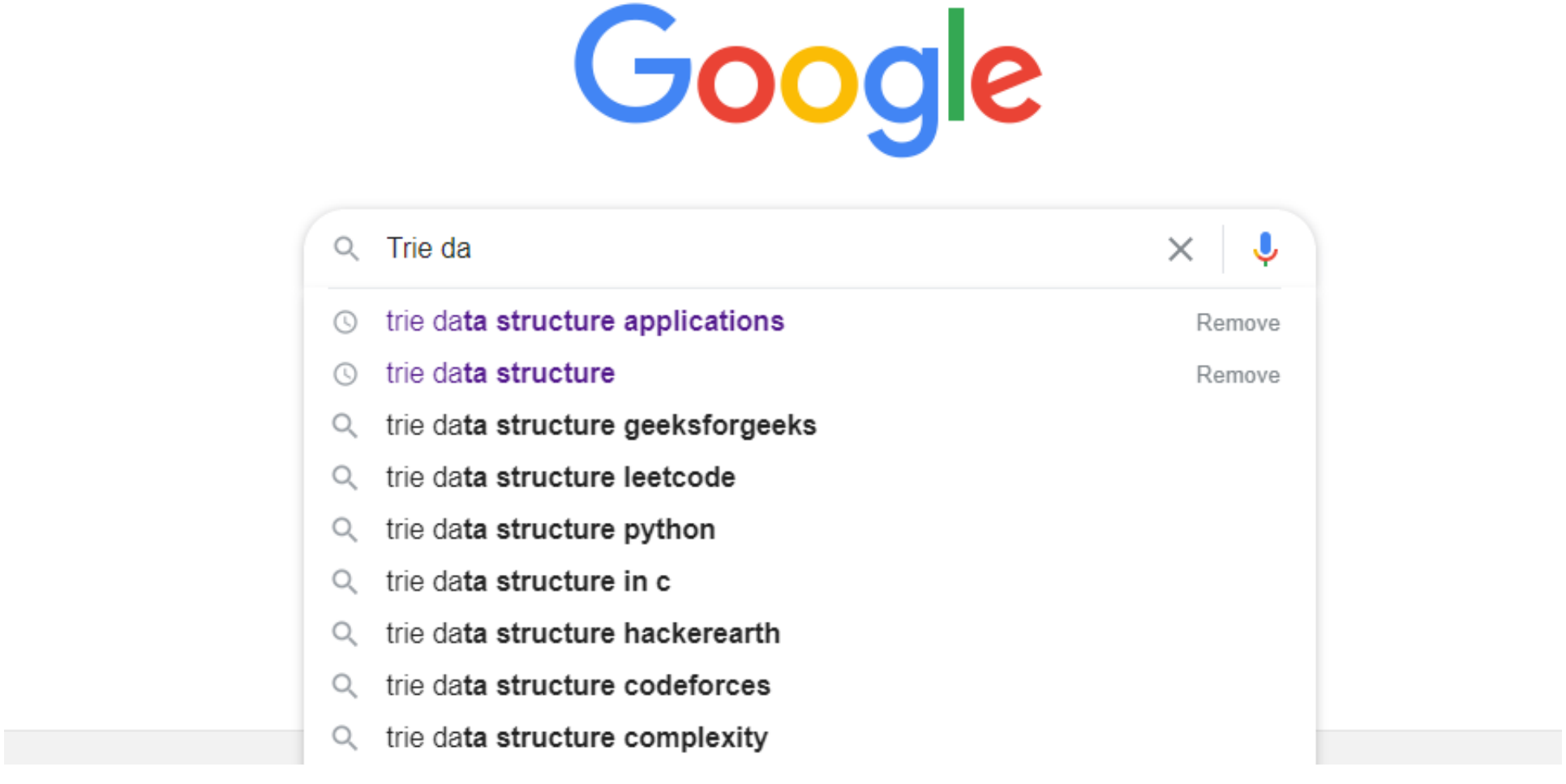
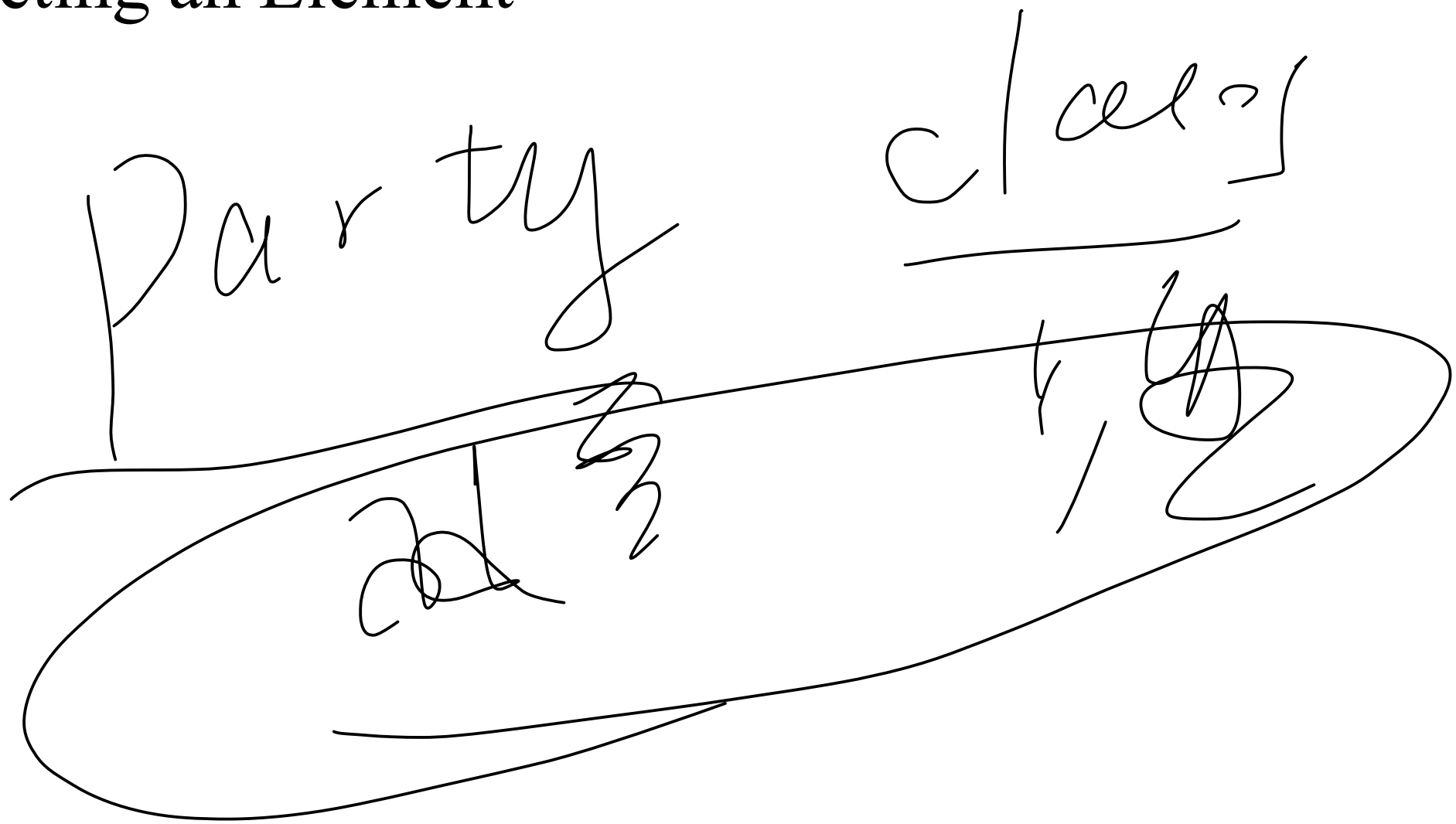
Topic 15 –
Trie Data Structure



- **Trie**
- Finding Elements
- Inserting Elements
- Deleting an Element

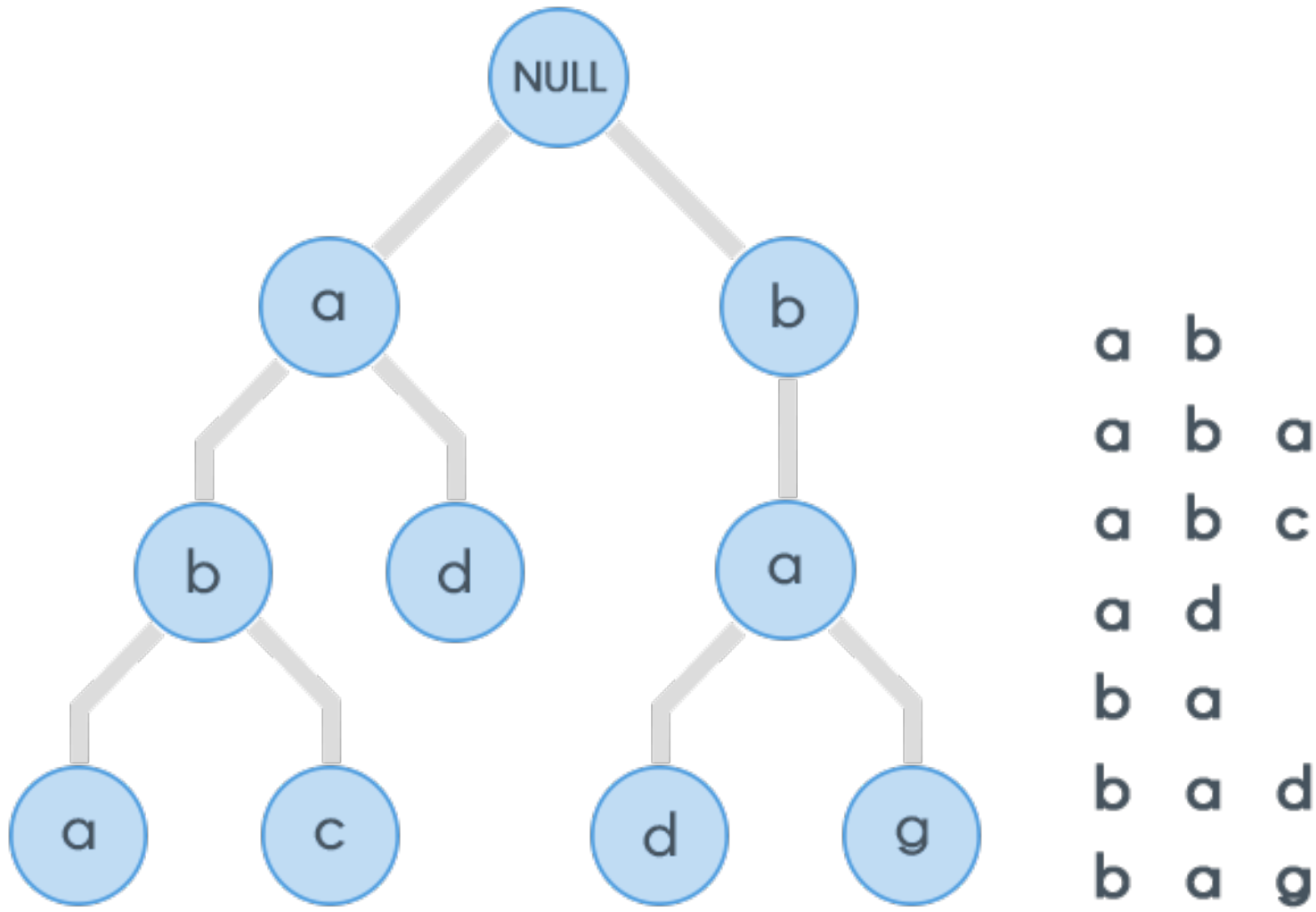


Prefix : What is prefix:

- The prefix of a string is nothing but any n letters $n \leq |S|$ that can be considered beginning strictly from the starting of a string.
- For example, the word “abacaba” has the following prefixes:
  - a
  - ab
  - aba
  - abac
  - abaca
  - abacab

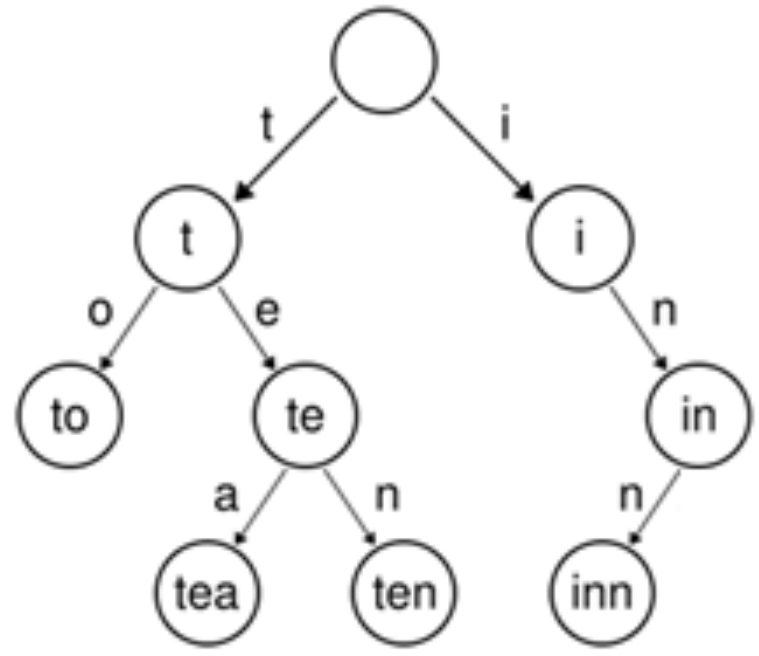
Prefix

- 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.



Trie (**prefix tree)**

- A Trie (pronounced try) or Prefix Tree is an ordered tree in which characters are stored at each node. Each path down the tree may represent a word.
- The maximum number of children of a node is equal to the size of the alphabet.
- Trie supports search, insert and delete operations in O(L) time where **L** is the length of the key.



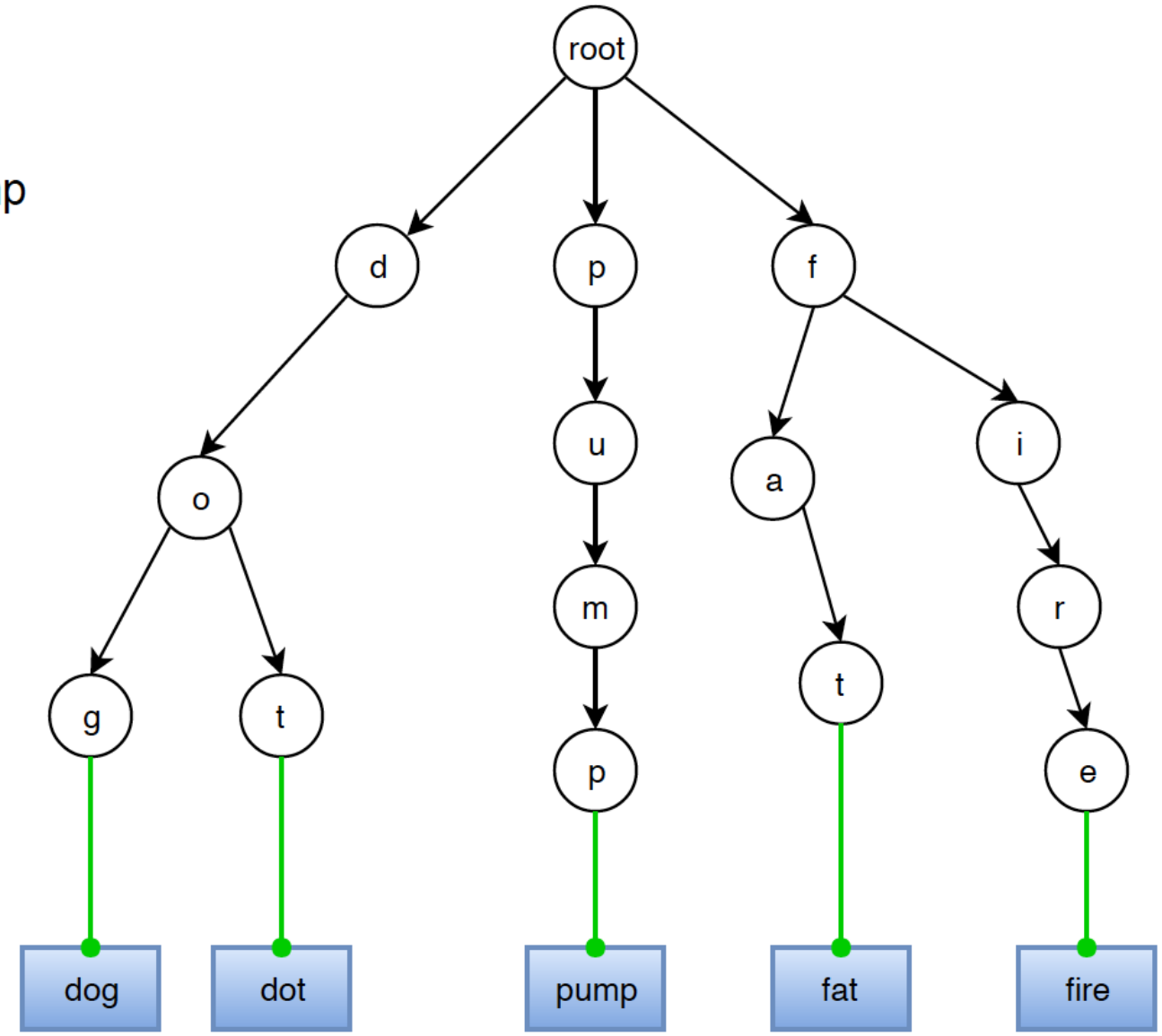
Trie (**prefix tree**)

- Very commonly, a trie is used to store the entire (English) language for quick prefix lookups.
- While a hash table can quickly look up whether a string is a valid word, however it cannot tell us if a string is a prefix of any valid words.
- A trie can do this very quickly.

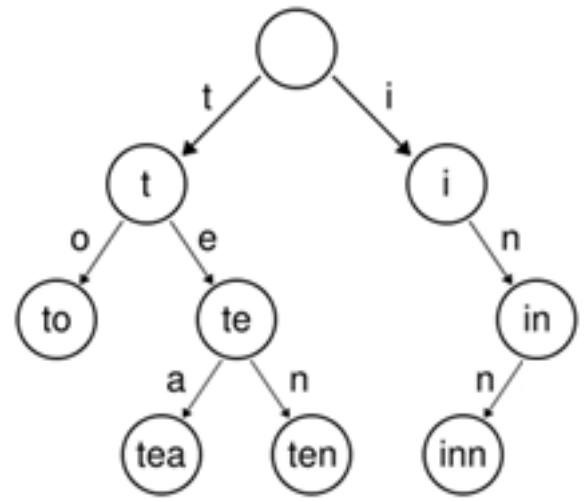
Trie (**prefix tree**)

- This trie stores five words: dog, dot, pump, fat, fire.
- Each node has a hashmap and a flag to indicate whether the current node is a leaf(a complete path for a word).

- dog
- dot
- pump
- fat
- fire



Trie (**prefix tree**)



- Tries is a tree that stores strings.
  - O(L) time where **L** is the length of the key.
- **Hashing : ** In hashing, we convert the key to a small value and the value is used to index data. Hashing supports search, insert and delete operations in O(L) time on average.
- **Self Balancing BST :** The time complexity of the search, insert and delete operations in a self-balancing Binary Search Tree (BST) (like Red-Black Tree, AVL Tree, Splay Tree, etc) is O(L * Log n) where n is total number words and L is the length of the word. The advantage of Self-balancing BSTs is that they maintain order which makes operations like minimum, maximum, closest (floor or ceiling) and kth largest faster. Please refer Advantages of BST over Hash Table for details.

Why Trie?

- With Trie, we can insert and find strings in ***O(L)*** time where *L* represent the length of a single word. This is obviously faster than BST. This is also faster than Hashing because of the ways it is implemented. We do not need to compute any hash function. No collision handling is required (like we do in open addressing and separate chaining)
- Another advantage of Trie is, we can easily print all words in alphabetical order which is not easily possible with hashing.
- We can efficiently do prefix search (or auto-complete) with Trie.

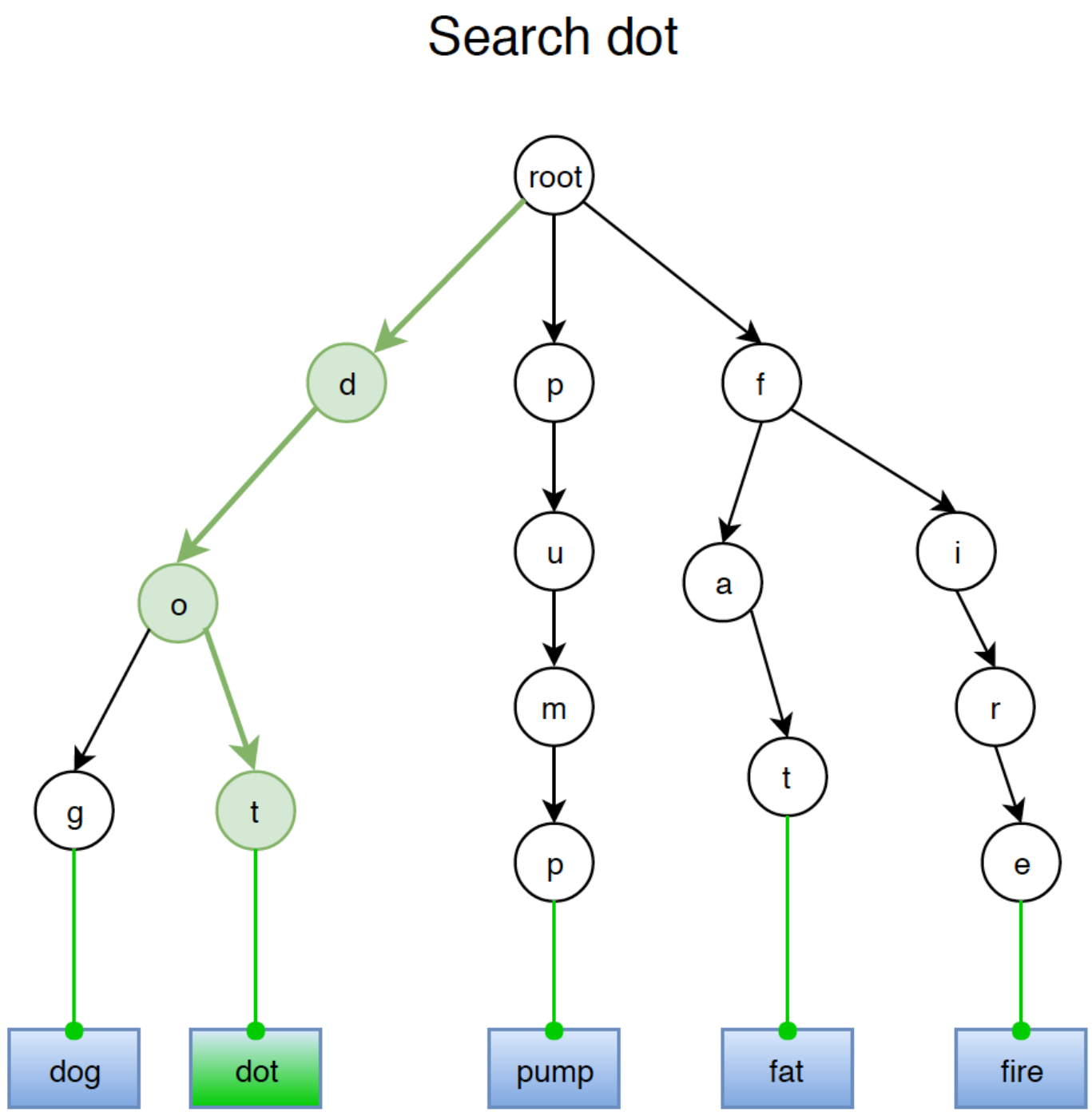
Disadvantage

- The main disadvantage of tries is that they need a lot of memory for storing the strings.
- **faster but require *huge memory* for storing the strings**

- Trie
- **Finding Elements**
  - Searching Prefix
  - Searching Entire Word
  - Searching Words with Same Prefix
- Inserting Elements
- Deleting an Element

Finding Elements

- Given a trie as follows, search word ‘dot’.



- There are two search approaches in trie.
    1. Find whether the given word exists.
    2. Find whether any word starts with the given prefix exists.
- Both approaches have the similar search pattern.
    1. To search a given word in Trie, we first convert the word to chars.
    2. Then start comparing each of them with trie node from root.
    3. If the current character is present in the node, move forward to its children.
    4. Recursively doing this until all of the characters are found.

Trie node

```
public class TrieNode {
    public Map<Character, TrieNode> children;
    public boolean leaf;

    public TrieNode() {
        children = new HashMap<Character, TrieNode>();
        leaf = false;
    }
}
```

Finding Elements

- Searching Prefix
- Searching Entire Word
  - Similar with prefix search, add additional check whether the node is leaf.
- Searching Words with Same Prefix
  - Return all words which start with the given prefix, check if the node is leaf.

Searching Prefix

```
// Return true if there is any word in trie that starts with the given prefix
public boolean startsWith(String prefix) {
    if (searchNode(prefix) == null) return false;
    else return true;
}

private TrieNode searchNode(String str) {
    TrieNode current = root;
    for (int i = 0; i < str.length(); i++) {
        char ch = str.charAt(i);
        if (current.children.containsKey(ch)) {
            current = current.children.get(ch);
        }
        else { return null; }
    }
    return current;
}
```

Searching Entire Word

```
// Return true if the word is in trie
public boolean search(String word) {
    TrieNode tn = searchNode(word);
    if (tn != null && tn.leaf) {
        return true;
    } else {
        return false;
    }
}
```

Searching Words with Same Prefix (1/2)

```
// Return all words which start with the given prefix
public List<String> searchWords(String prefix) {
    TrieNode current = root;
    StringBuilder sb = new StringBuilder();

    for (int i = 0; i < prefix.length(); i++) {
        char ch = prefix.charAt(i);
        if (!current.children.containsKey(ch)) { return null; }
        else {
            sb.append(ch);
            current = current.children.get(ch);
        }
    }

    List<String> list = new ArrayList<>();
    dfs(current, sb.toString(), list);
    return list;
}
```

Searching Words with Same Prefix (2/2)

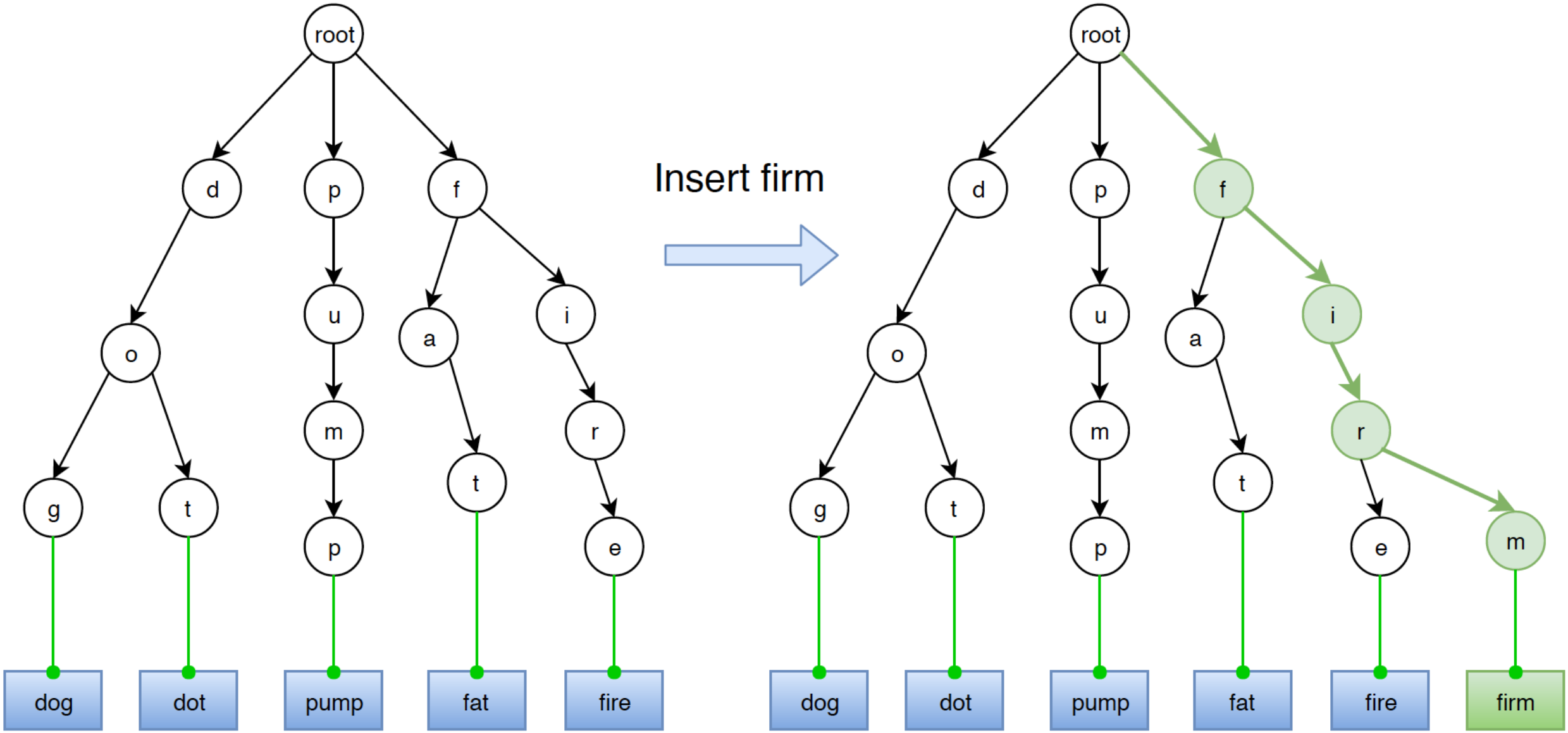
- Return all words which start with the given prefix, check if the node is leaf.

```
private void dfs(TrieNode node, String prefix, List<String> list)
{
    if (node.leaf) { list.add(prefix); }
    for (Map.Entry<Character, TrieNode> entry :
node.children.entrySet()) {
        dfs(entry.getValue(), prefix + entry.getKey(), list);
    }
}
```

- Trie
- Finding Elements
    - Searching Prefix
    - Searching Entire Word
    - Searching Words with Same Prefix
- **Inserting Elements**
- Deleting an Element

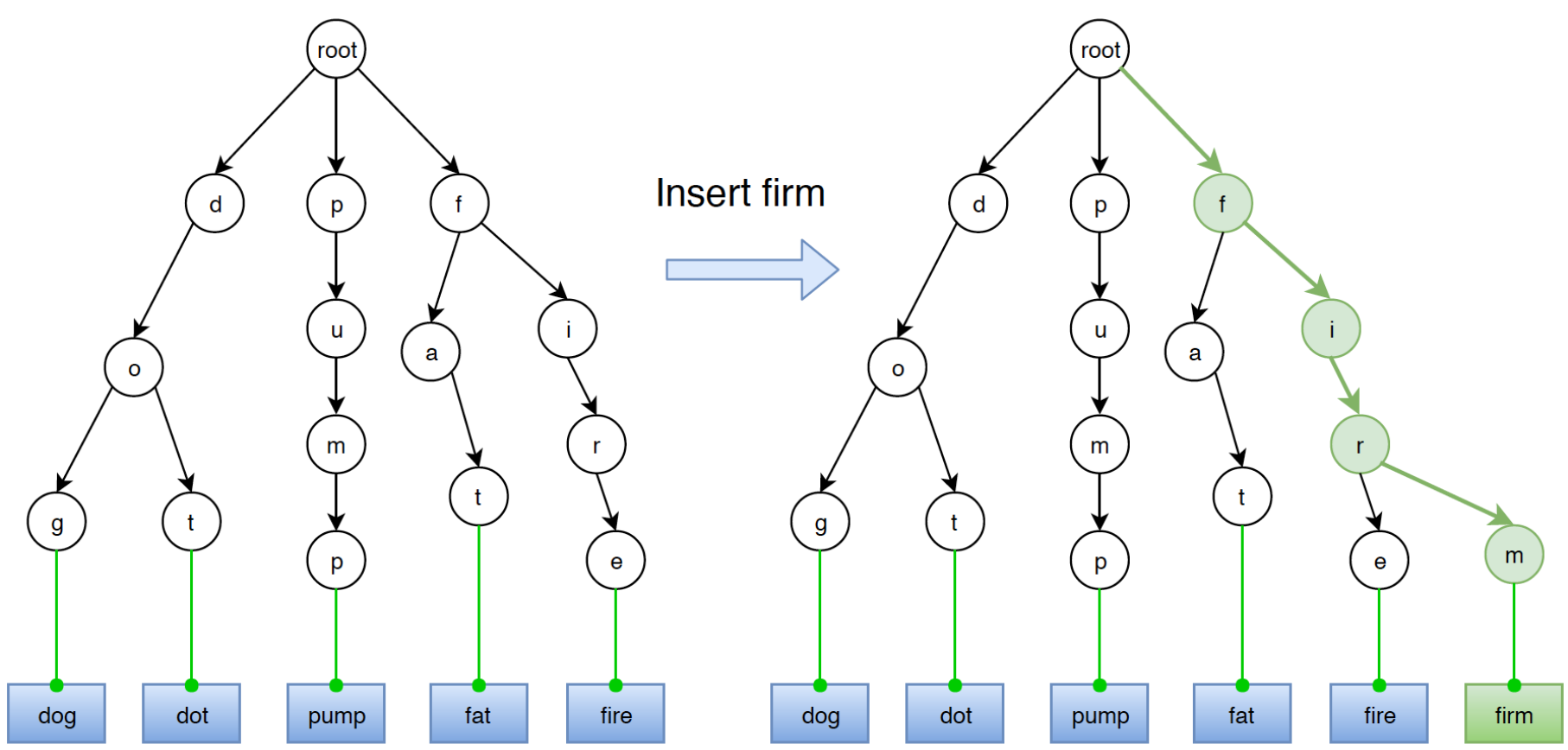
Inserting Elements

- Given a trie as follows, insert new word '**firm**' into this trie.



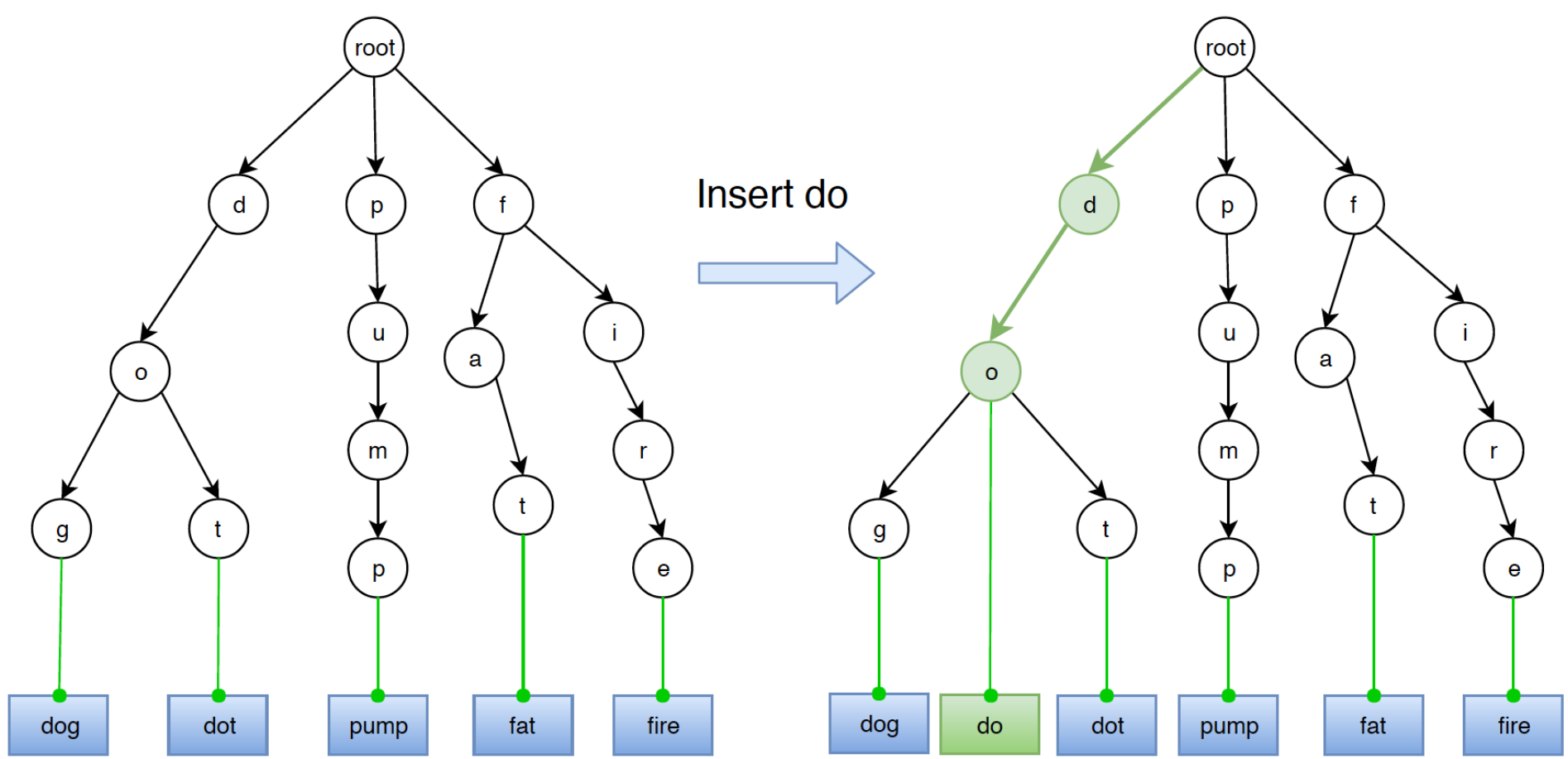
Inserting Elements

- We start searching the given word from **root** till we cannot find one particular character.
- Then we construct new trie nodes recursively for the rest characters.
- In the end, set the leaf attribute of the last node to true.



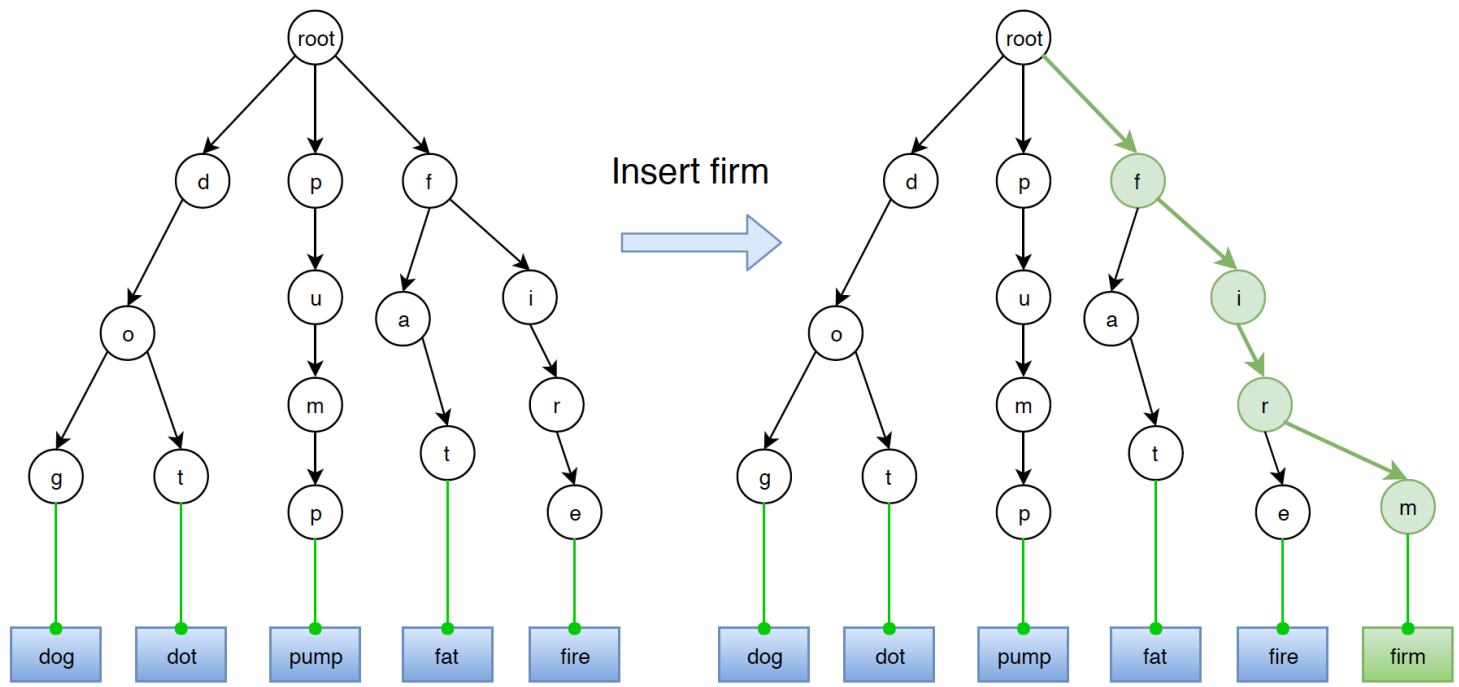
Inserting Elements

- One case needs to be noticed here.
- If the new word(eg. ‘do’) is prefix of other words(word ‘do’ is prefix of word ‘dot’), we just need to mark the last node(eg. node ‘o’) of the new word as leaf without creating any new node.
- Even though node ‘o’ has children, it is marked as leaf since the path from root to node ‘o’ represents word ‘do’.



Inserting Elements

1. Set a current node as a root node
2. Set the current letter as the first letter of the word
3. If the current node has already an existing reference to the current letter (through one of the elements in the “children” field), then set current node to that referenced node. Otherwise, create a new node, set the letter equal to the current letter, and also initialize current node to this new node
4. Repeat step 3 until the key is traversed



Inserting Elements

- Below is the implementation of the insert method.

```
// Insert a word into trie
public void insert(String word) {
    TrieNode current = root;

    for (int i = 0; i < word.length(); i++) {
        char ch = word.charAt(i);
        if (!current.children.containsKey(ch)) {
            current.children.put(ch, new TrieNode());
        }
        current = current.children.get(ch);
    }

    current.leaf = true;
}
```

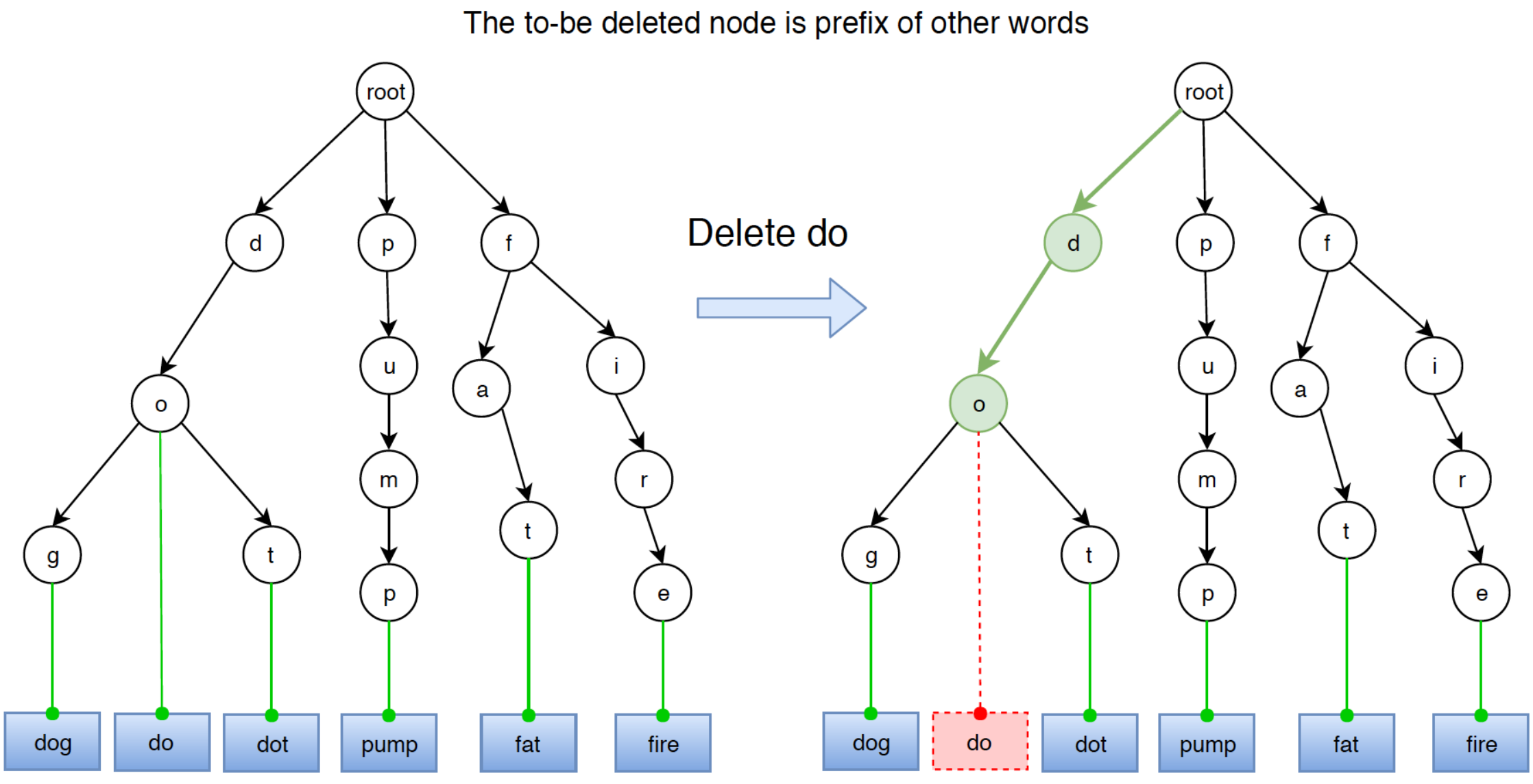
- Trie
- Finding Elements
- Inserting Elements
- **Deleting an Element**
    1. Word is prefix of other words.
    2. Word has prefix of other words.
    3. Word is unique.

Deleting an Element

- There are three cases when deleting a word from Trie.
  1. Word is prefix of other words.
  2. Word has prefix of other words.
  3. Word is unique, neither it is prefix of other words, nor it has prefix of other words.

Word Is Prefix of Other Words

- Word ‘do’ is the prefix of word ‘dot’ and ‘dog’.

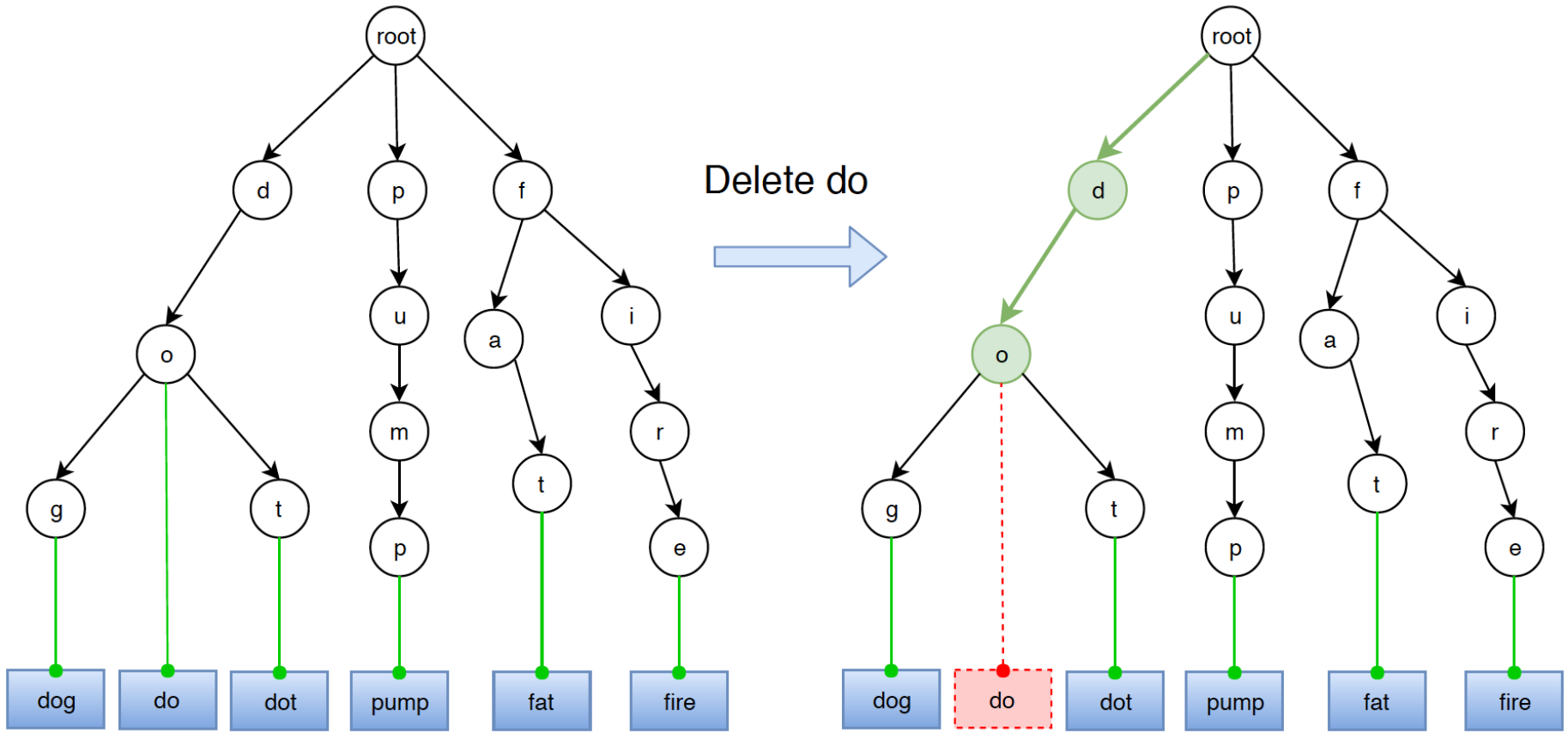


- The solution is easy, just unmark the leaf node. The leaf node for word ‘do’ is node ‘o’.

Word Is Prefix of Other Words

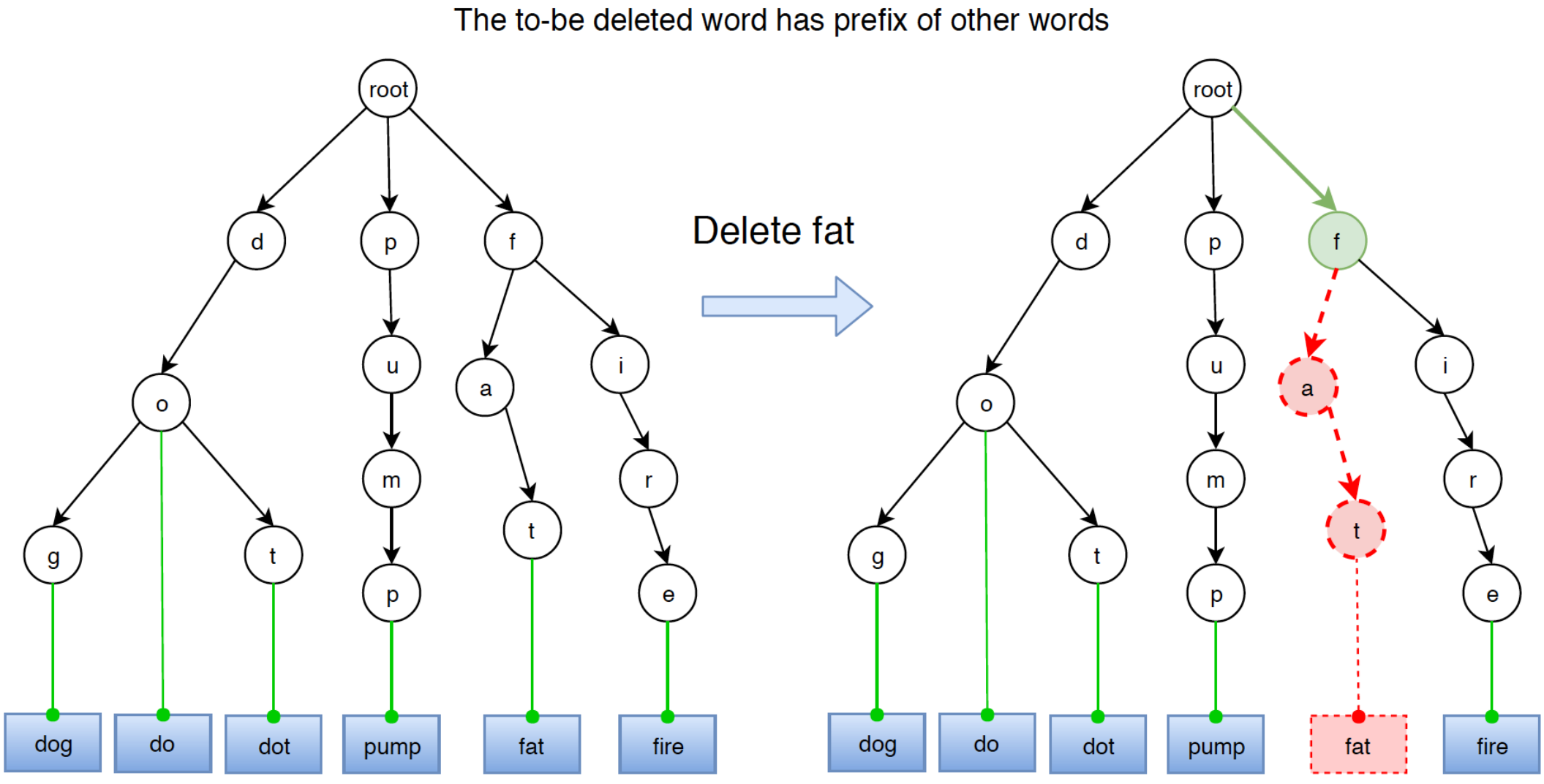
```
// case 1: The to-be deleted word is
// prefix of another long word in trie.
if (current.children.size() > 0) {
        current.leaf = false;
        return true;
}
```

第一种最简单，只消把叶结点的 leaf 修改为 false 即可.



Word Has Prefix of Other Words

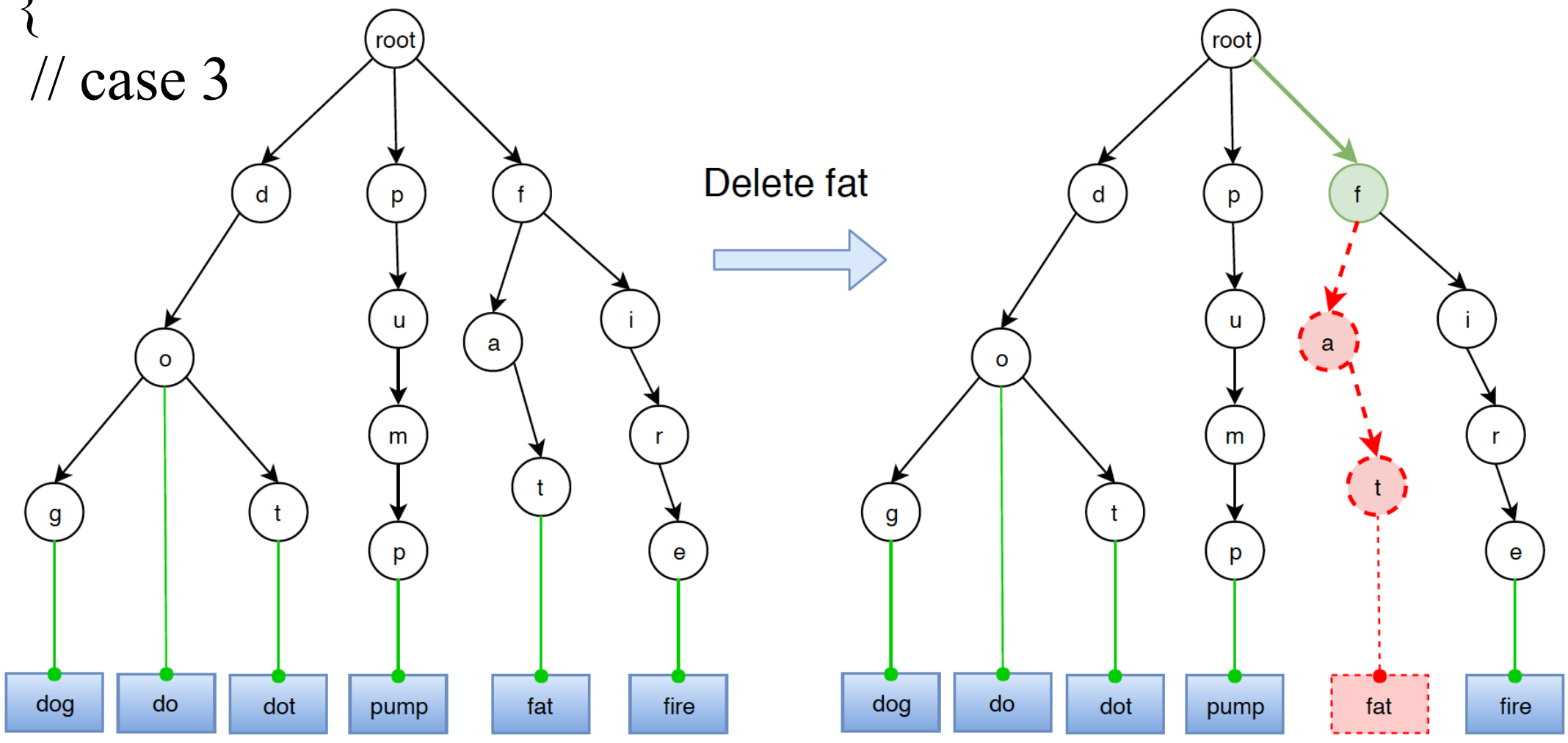
- Word ‘fat’ has same prefix with word ‘fire’. They share the prefix ‘f’.



- If word has prefix of other words, then delete nodes from prefix to end of the word.

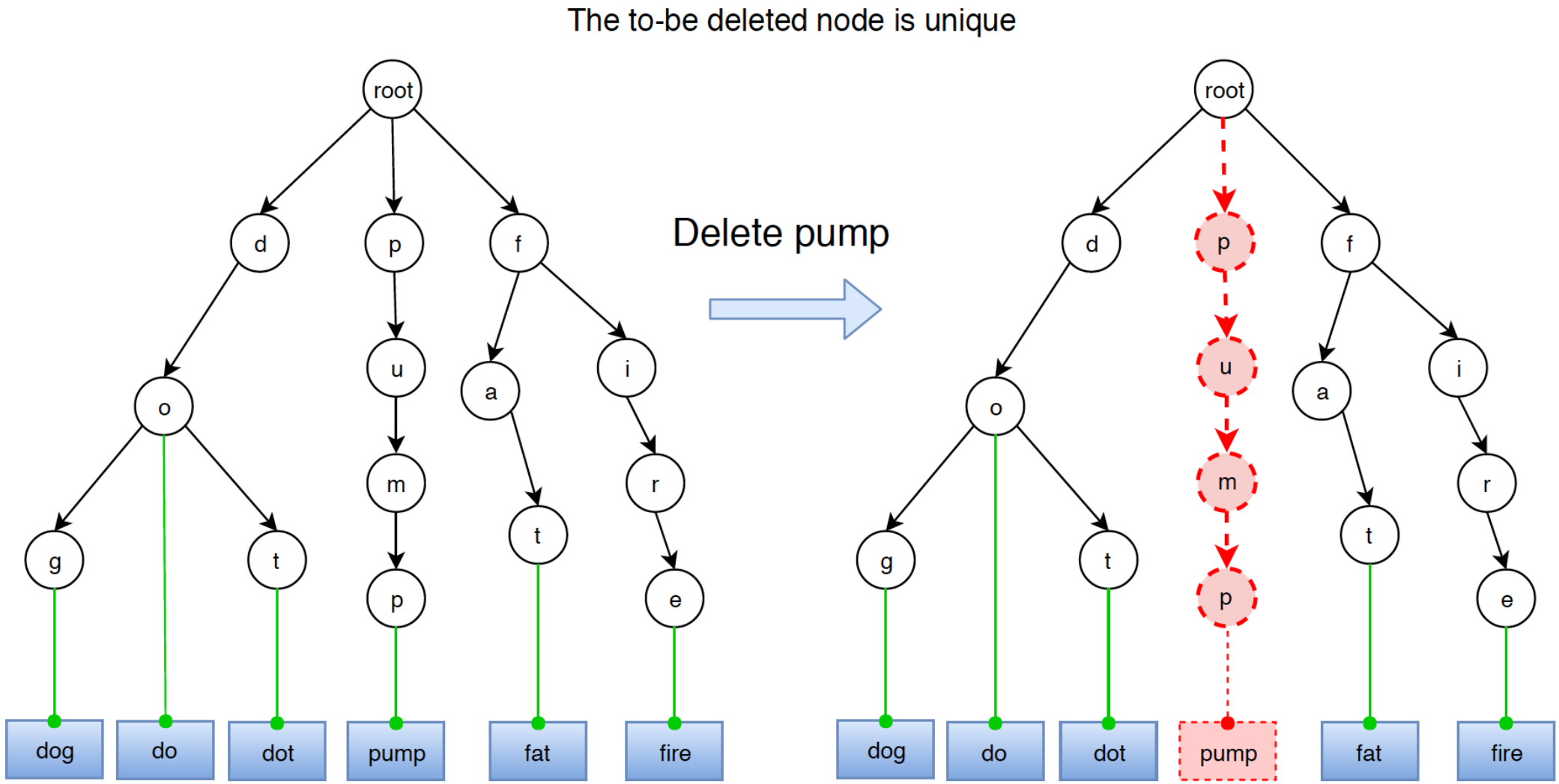
Word Has Prefix of Other Words

```
if (lastBranchNode != null) {
        // case 2: The to-be deleted word has other words as prefix
        lastBranchNode.children.remove(lastBrachChar);
        return true;
}
else {
        // case 3
}
```



Word Is Unique

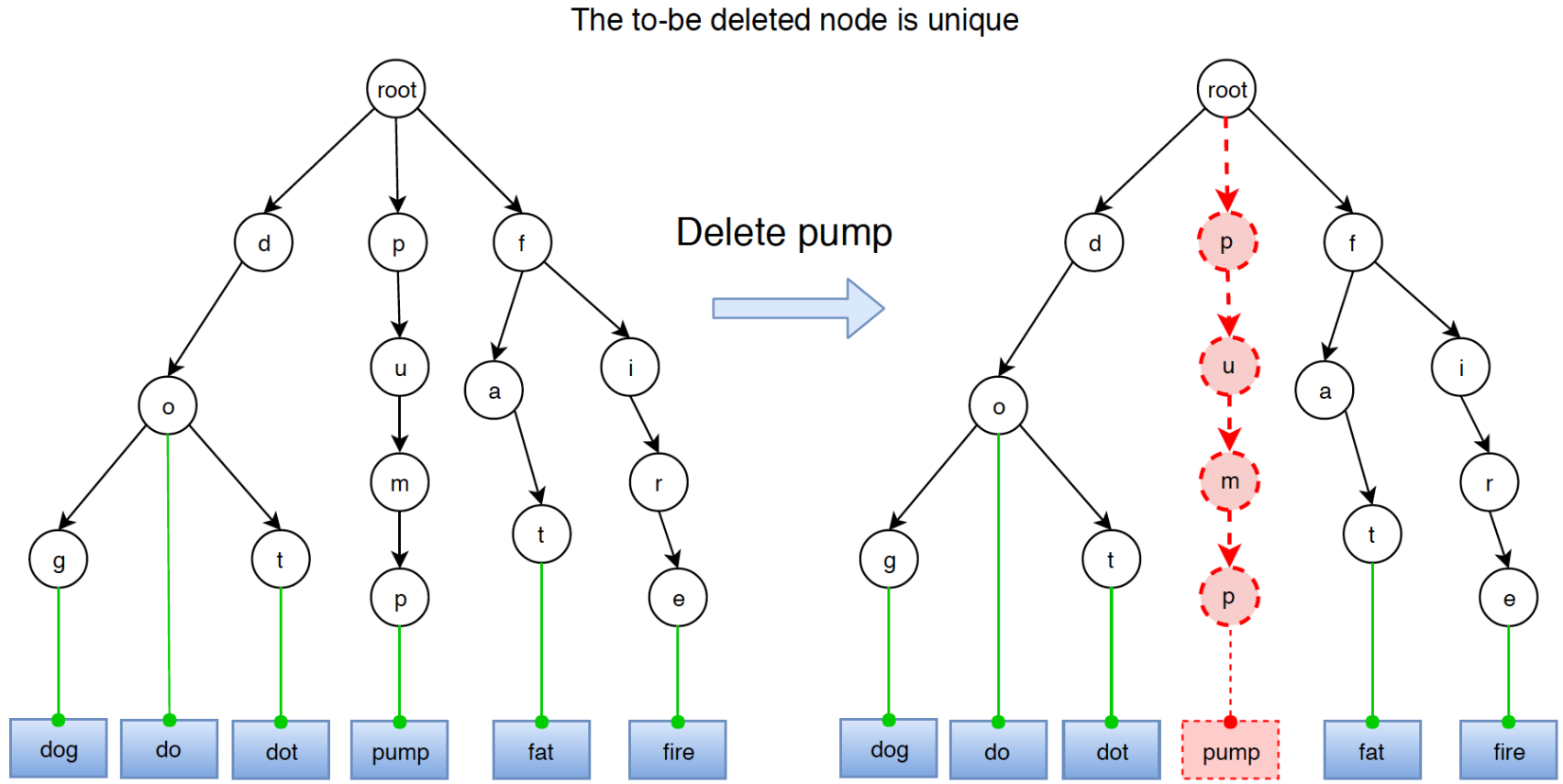
- Word ‘pump’ is a standalone word. It doesn’t share any prefix with others.



- If word neither is prefix of other words, nor has prefix of other words, then just delete all the nodes.

Word Is Unique

```
if (lastBranchNode != null) {
        // case 2:
} else {
        // case 3: The to-be deleted word present as unique word
        root.children.remove(word.charAt(0));
        return true;
}
```

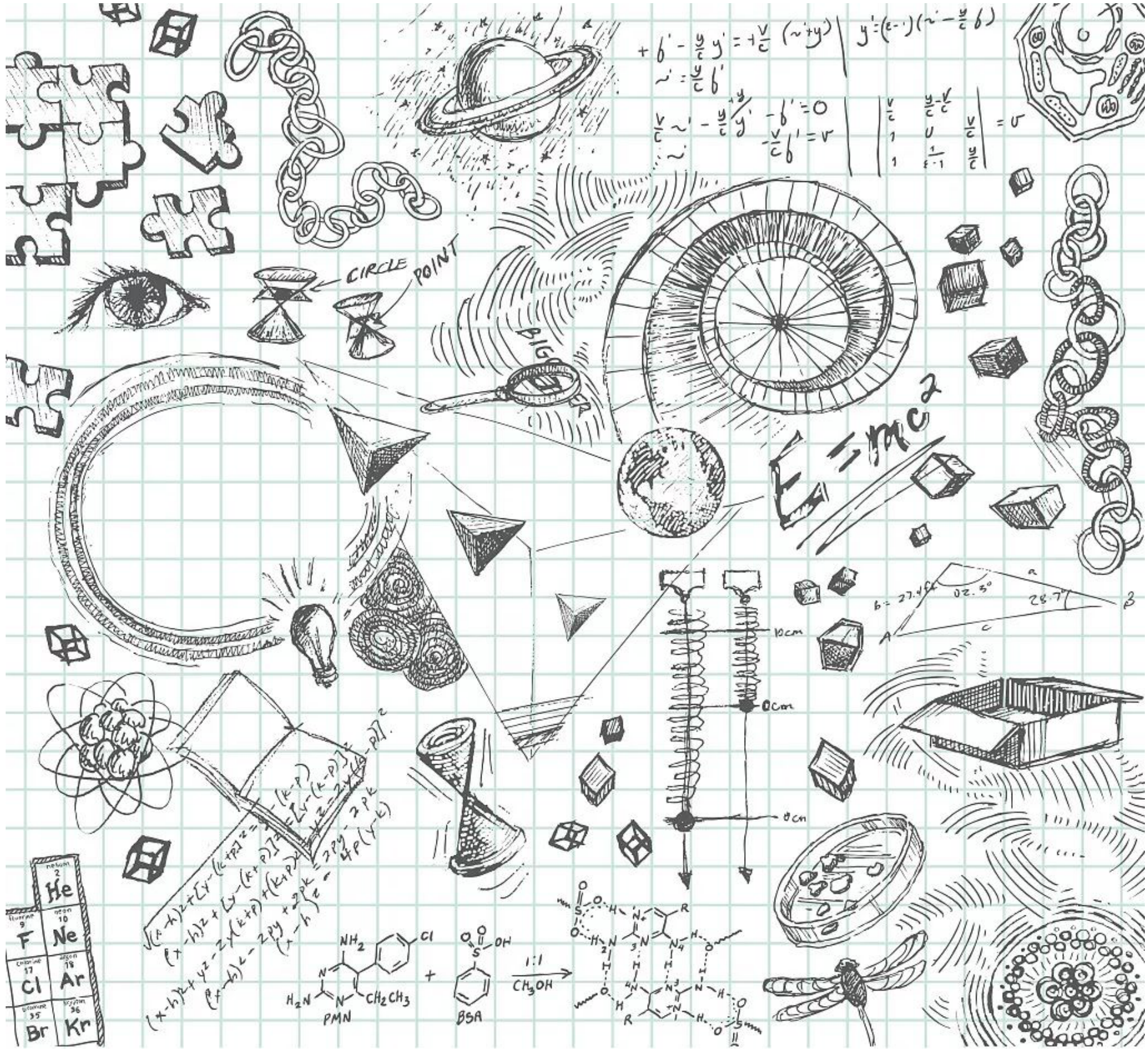


Implementation

```
public boolean delete(String word) {
    TrieNode current = root;
    TrieNode lastBranchNode = null;
    Character lastBrachChar = null;

    for (int i = 0; i < word.length(); i++) {
        char ch = word.charAt(i);
        if (current.children.containsKey(ch)) {
            if (current.children.size() > 1) {
                lastBranchNode = current;
                lastBrachChar = ch;
            }
            current = current.children.get(ch);
        } else { // word not found
            return false;
        }
    }

    if (current.children.size() > 0) { // case 1: word is prefix of another word }
    if (lastBranchNode != null) { // case 2}
    else { // case 3: unique word }
}
```


CIRCLE
POINT
BIG
E=mc2
10cm
0cm
8cm
PMN
BSA
1:1
CH3OH