

Courses

Tutorials

Jobs

Practice

Contests



P

Deletion in a Trie

We had already discussed the representation of a Trie data structure, inserting a key and searching a key in a Trie. Let us look at the process of deleting a given key from a Trie.

During delete operation, we delete the key in a bottom-up manner using recursion. The following are possible cases which may occur while deleting the key from trie,

- **Case 1:** Key may not be present in the trie. In this case, the delete operation should not modify trie.
- **Case 2:** Key present as a unique key (no part of key contains another key (prefix), nor the key itself is a prefix of another key in trie). In this case, delete all the nodes of that key.
- **Case 3:** Key is a prefix key of another long key in the trie. In this case, simply unmark the leaf node.
- **Case 4:** Key present in the trie, having at least one other key as a prefix key. In this case, delete nodes from the end of key until first leaf node of longest prefix key.

Implementation:

C++

Java

```
// Java implementation of delete
// operations on Trie

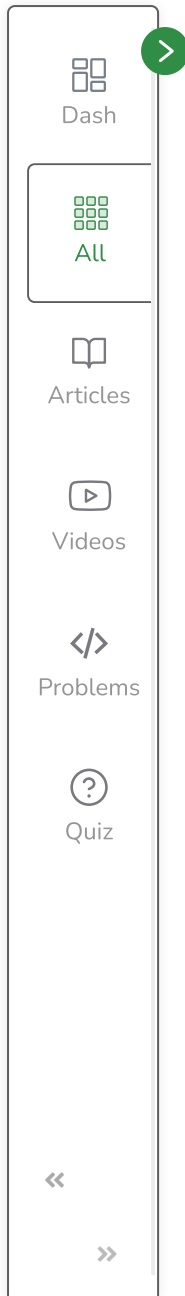
import java.util.*;
import java.io.*;
import java.lang.*;

public class Trie {
```


Dash
All
Articles
Videos
Problems
Quiz

<< Prev

Next >>



```
static int ALPHABATE_SIZE=26;

// Trie node
static class TrieNode
{
    TrieNode children[]=new TrieNode[ALPHABATE_SIZE];

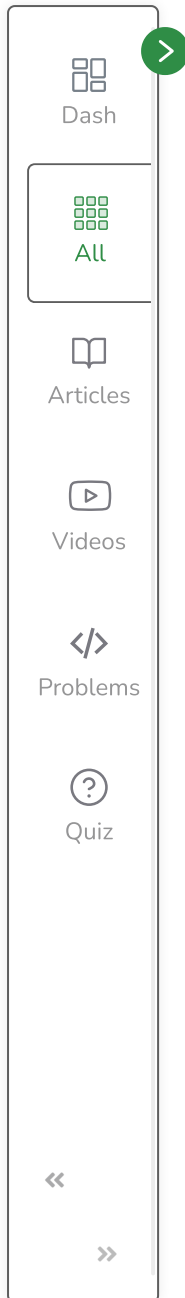
    // isEndOfWord is true if the node represents
    // end of a word
    boolean isEndOfWord;

    // Returns new trie node (initialized to NULLs)
    public TrieNode()
    {
        isEndOfWord=false;
        for(int i = 0; i<ALPHABATE_SIZE;i++)
        {
            children[i]=null;
        }
    }
};

static TrieNode root;

// If not present, inserts key into trie
// If the key is prefix of trie node, just
// marks leaf node
static void insert(String key)
{
    int level;
    int length=key.length();
    int index;
    TrieNode pCrawl=root;
    for(level = 0;level<length;level++)
    {
        index=key.charAt(level)-'a';
        if(pCrawl.children[index]==null)
        {
            pCrawl.children[index]=new TrieNode();
        }
        pCrawl=pCrawl.children[index];
    }
}
```





```

    }

    // mark last node as leaf
    pCrawl.isEndOfWord=true;
}

// Returns true if key presents in trie, else
// false
static boolean search(String key)
{
    int index;
    int length=key.length();
    int level;
    TrieNode pCrawl=root;

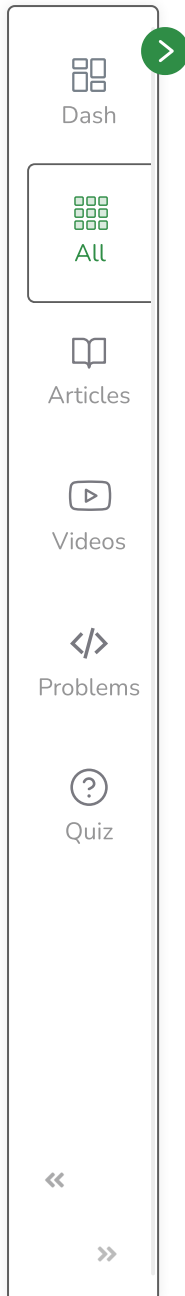
    for(level = 0;level<length;level++)
    {
        index=key.charAt(level)-'a';
        if(pCrawl.children[index]==null)
        {
            return false;
        }
        pCrawl=pCrawl.children[index];
    }

    if(pCrawl!=null && pCrawl.isEndOfWord)
    {
        return true;
    }
    else
    {
        return false;
    }
}

// Returns true if root has no children
// else false
static boolean hasNoChild(TrieNode currentNode)
{
    for(int level=0;level<currentNode.children.length;level++)
    {

```





```

        if(currentNode.children[level]!=null)
        {
            return false;
        }
    }

    return true;
}

static boolean removeUtil(TrieNode currentNode,String key,
                           int level,int length)
{
    // If tree is empty
    if(currentNode==null)
    {
        System.out.println("Does not exist");
        return false;
    }

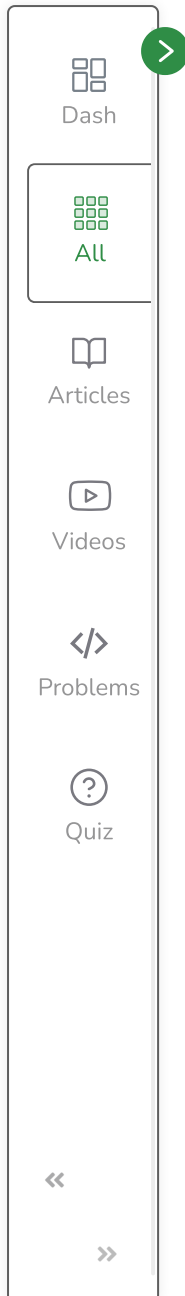
    // If last character of key is being processed
    if(level==length)
    {
        // This node is no more end of word after
        // removal of given key
        currentNode.isEndOfWord=false;

        // If given is not prefix of any other word
        if(hasNoChild(currentNode))
        {
            return true;
        }
        else
        {
            return false;
        }
    }
    else{

        // If not last character, recur for the child
        // obtained using ASCII value

```





```

TrieNode childNode =
currentNode.children[key.charAt(level) - 'a'];

boolean childDeleted =
removeUtil(childNode, key, level + 1, length);

    if(childDeleted)
    {
        // If root does not have any child
        //(its only child got
        // deleted), and it is not end of another word.
        return (currentNode.isEndOfWord
        &&hasNoChild(currentNode));
    }

    return false;
}

// Recursive function to delete a key
// from given Trie
static void remove(String key)
{
    int length=key.length();

    if(length>0)
    {
        removeUtil(root, key, 0, length);
    }
}

// Driver Code
public static void main(String[] args)
{
    // Input keys (use only 'a' through 'z'
    // and lower case)
    root=new TrieNode();

    String keys[]= {"the", "a", "there",
                    "answer", "any", "by", "bye", "their",
                    "hero", "heroplane"};

```





Dash



All



Articles



Videos



Problems



Quiz



```
// Construct trie
for (int i = 0; i < keys.length; i++)
{
    insert(keys[i]);
}

// Search for different keys
if(search("the") == true)
    System.out.println("Yes");
else
    System.out.println("No");

if(search("these") == true)
    System.out.println("Yes");
else
    System.out.println("No");

remove("heroplane");

if(search("hero") == true)
    System.out.println("Yes");
else
    System.out.println("No");
    }
}
```

Output

Yes

No

Yes



Time Complexity: The time complexity of the deletion operation is $O(n)$ where n is the key length.

Auxiliary Space: $O(n*m)$, where n is the key length of the longest word and m is the total no of words.

Mark as Read

 Report An Issue

If you are facing any issue on this page. Please let us know.

