Deletion in a Trie

This lesson defines all the cases needed for deleting a word in Trie, along with implementing this functionality in Java.

We'll cover the following

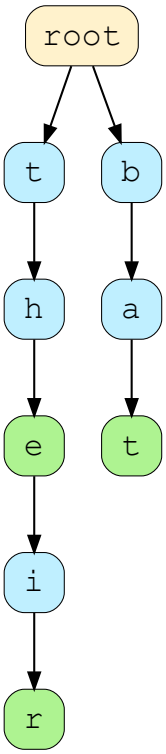
Deleting a word in Trie

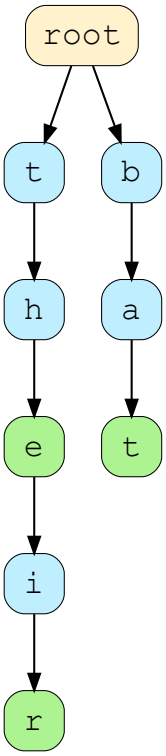
While deleting a word from a Trie, we make sure that the node that we are trying to delete does not have any further branches. If there are no branches, then we can easily remove the node. However, if the `node` contains further branches then this opens up a lot of the scenarios covered below.

Case 1: If the word to be deleted has no common subsequence

- If the word to be deleted has no common subsequence, then all the nodes of that word are deleted.

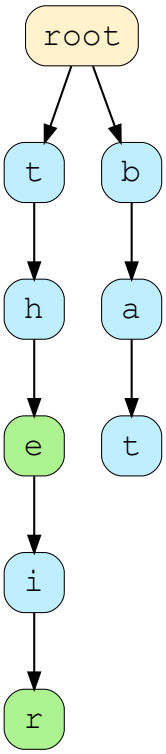
For example, in the figure given below, we have to delete all characters of “bat” in order to delete the word `bat`.





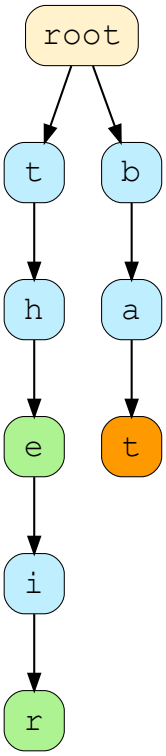
Deleting 'bat'

/



Unmark 't'

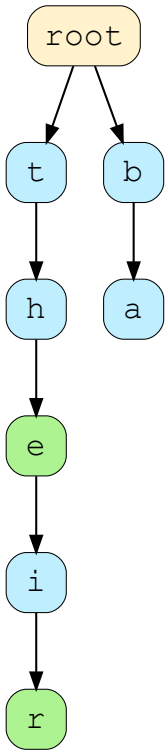
3 of 9



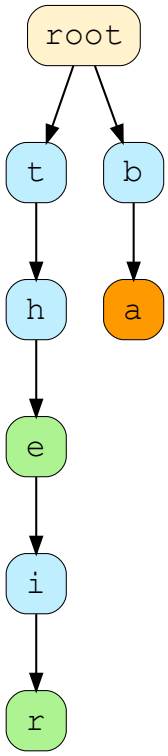
Delete 't'

4 of 9

/



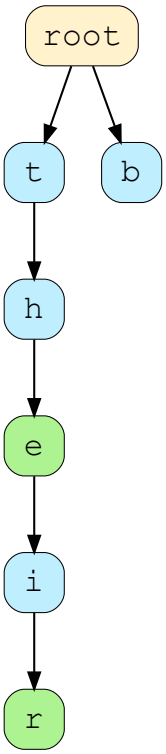
5 of 9



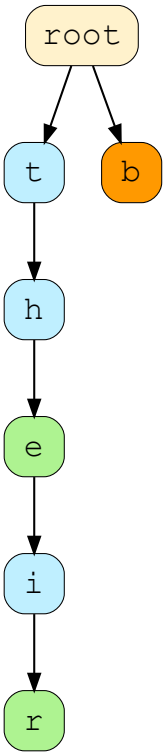
Delete 'a'

6 of 9

/



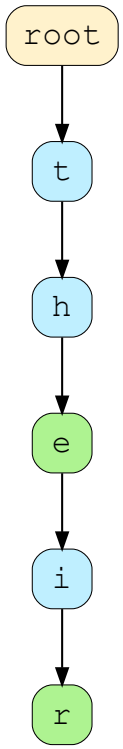
7 of 9



Delete 'b'

8 of 9

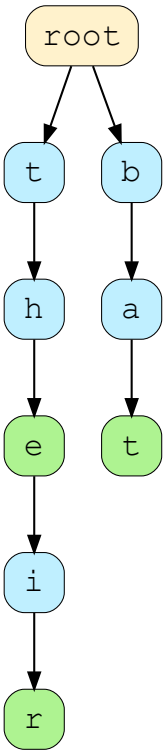
/



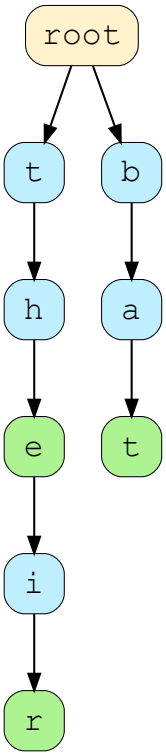
Case 2: If the word to be deleted is a prefix of some other word

- If the word to be deleted is a prefix of some other word, then the value of `isEndWord` of the last node of that word is set to `false`, and no node is deleted.

For example, we will simply unmark `e` to delete the word `the` and show that it doesn't exist anymore.



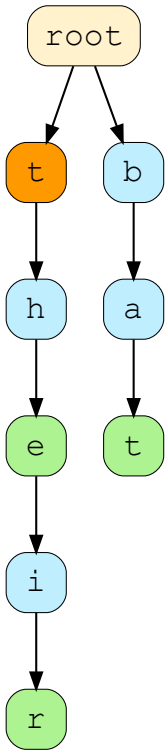
1 of 6



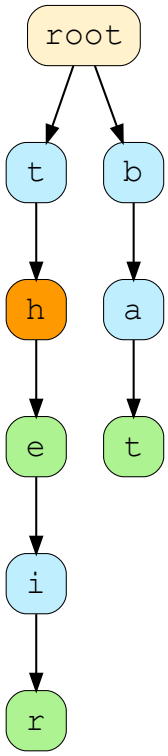
Deleting 'the'

2 of 6

/

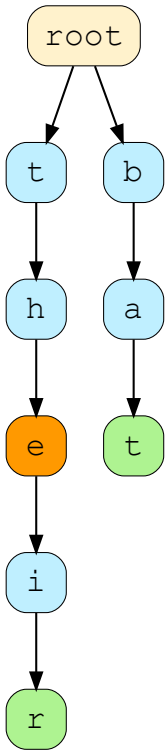


3 of 6

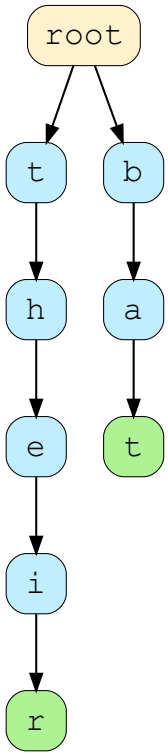


4 of 6

/



5 of 6



Unmark 'e'

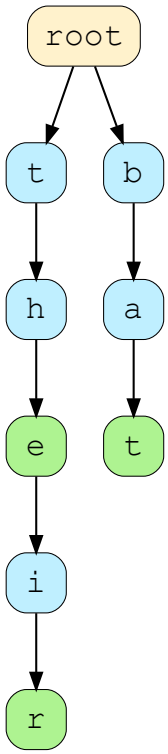
6 of 6

Case 3: If the word to be deleted has a common prefix

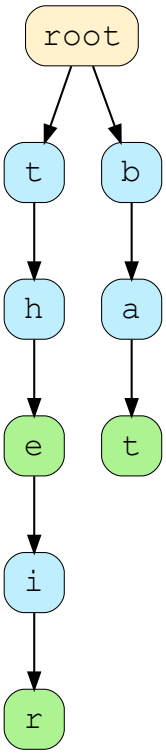
/

- If the word to be deleted has a common prefix and the last node of that word is also the leaf node (i.e. the last node of its branch), then this node is deleted along with all the higher-up nodes in its branch that do not have any other children and whose `isEndWord` is *false*.

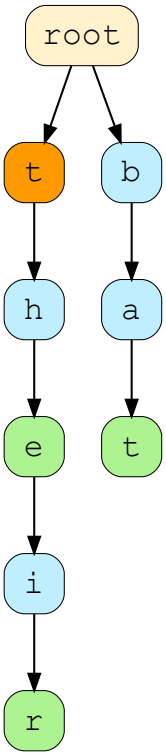
For example, we'll traverse the common path up to `the` and delete the characters "i" and "r" in order to delete `their`.

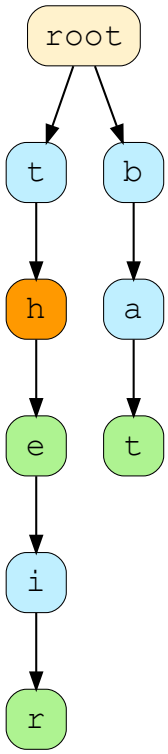


1 of 15

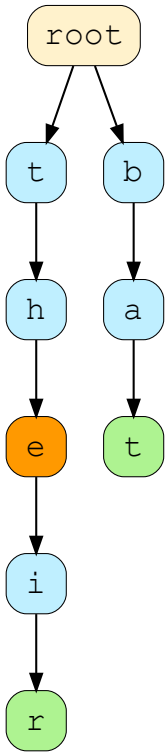


Deleting 'their'



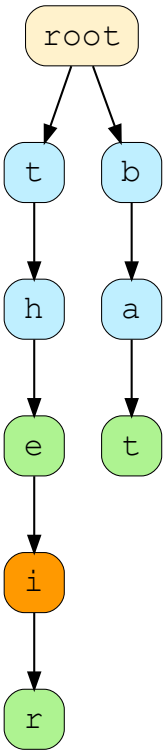


4 of 15

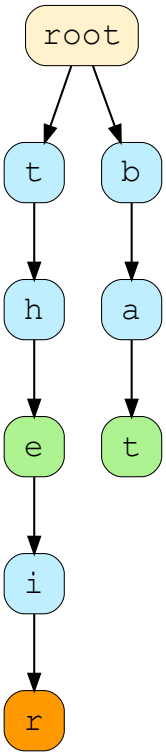


5 of 15

/



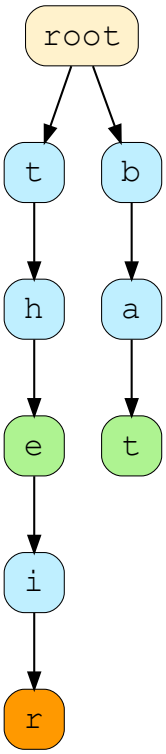
6 of 15



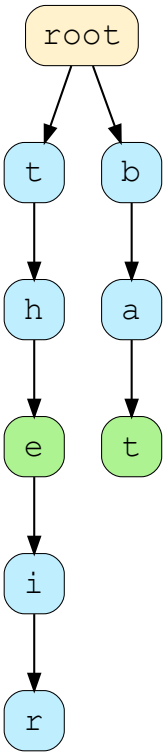
Word found!

7 of 15

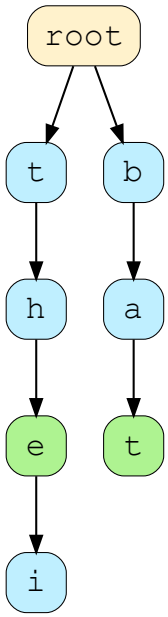
/



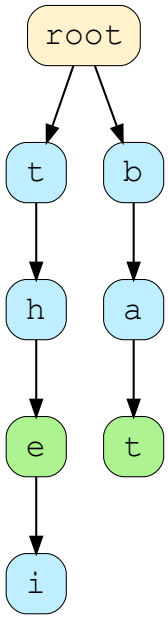
Unmark 'r'



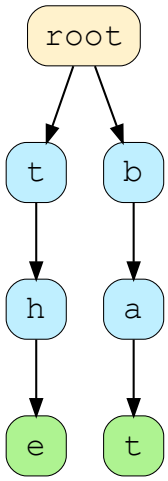
/



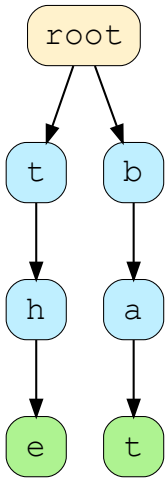
Delete 'r'



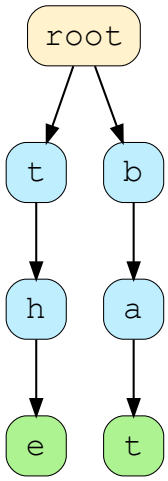
Delete 'r'



Delete 'i'

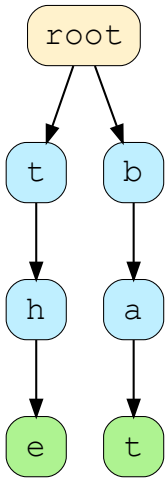


Delete 'i'



Do not delete 'e' as its been marked as the end of word

14 of 15



Do not delete 'e' as its been marked as the end of word

15 of 15

Implementation

The following is the code snippet, so try to understand the code. If you don't understand any point, you can read the explanation below.

main.java

Trie.java

TrieNode.java

```
class Trie {

    private TrieNode root; //Root Node

    //Constructor
    Trie() {
        root = new TrieNode();
    }

    //Function to get the index of a character 't'
    public int getIndex(char t) {
        return t - 'a';
    }

    //Function to insert a key in the Trie
    public void insert(String key) {
        //Null keys are not allowed
        if (key == null) {
            return;
        }
        key = key.toLowerCase(); //Keys are stored in lowercase
        TrieNode currentNode = this.root;
        int index = 0; //to store character index

        //Iterate the Trie with given character index,
        //If it is null, then simply create a TrieNode and go down a level
        for (int level = 0; level < key.length(); level++) {
            index = getIndex(key.charAt(level));

            if (currentNode.children[index] == null) {
                currentNode.children[index] = new TrieNode();
            }
            currentNode = currentNode.children[index];
        }
        //Mark the end character as leaf node
        currentNode.markAsLeaf();
    }

    //Function to search given key in Trie
    public boolean search(String key) {

        if (key == null) return false; //Null Key

        key = key.toLowerCase();
        TrieNode currentNode = this.root;
```

```
46          int index = 0;
47
48          //Iterate the Trie with given character index,
49          //If it is null at any point then we stop and return false
50          //We will return true only if we reach leafNode and have traversed the
51          //Trie based on the length of the key
52
53          for (int level = 0; level < key.length(); level++) {
54              index = getIndex(key.charAt(level));
55              if (currentNode.children[index] == null) return false;
56              currentNode = currentNode.children[index];
57          }
58          if (currentNode.isEndWord) return true;
59
60          return false;
61      }
62
63      //Helper Function to return true if currentNode does not have any children
64      private boolean hasNoChildren(TrieNode currentNode){
65          for (int i = 0; i < currentNode.children.length; i++){
66              if ((currentNode.children[i]) != null)
67                  return false;
68          }
69          return true;
70      }
71
72      //Recursive function to delete given key
73      private boolean deleteHelper(String key, TrieNode currentNode, int length, int le
74      {
75          boolean deletedSelf = false;
76
77          if (currentNode == null){
78              System.out.println("Key does not exist");
79              return deletedSelf;
80          }
81
82          //Base Case: If we have reached at the node which points to the alphabet at t
83          if (level == length){
84              //If there are no nodes ahead of this node in this path
85              //Then we can delete this node
86              if (hasNoChildren(currentNode)){
87                  currentNode = null;
88                  deletedSelf = true;
89              }
90              //If there are nodes ahead of currentNode in this path
91              //Then we cannot delete currentNode. We simply unmark this as leaf
92              else
93              {
94                  currentNode.unMarkAsLeaf();
95                  deletedSelf = false;
96              }
97          }
98          else
99          {
100             TrieNode childNode = currentNode.children[getIndex(key.charAt(level))];
101             boolean childDeleted = deleteHelper(key, childNode, length, level + 1);
102             if (childDeleted)
```

```
103             {
104                 //Making children pointer also null: since child is deleted
105                 currentNode.children[getIndex(key.charAt(level))] = null;
106                 //If currentNode is leaf node that means currntNode is part of anothe
107                 //and hence we can not delete this node and it's parent path nodes
108                 if (currentNode.isEndWord){
109                     deletedSelf = false;
110                 }
111                 //If childNode is deleted but if currentNode has more children then 
112                 //So, we cannot delete currenNode
113                 else if (!hasNoChildren(currentNode)){
114                     deletedSelf = false;
115                 }
116                 //Else we can delete currentNode
117                 else{
118                     currentNode = null;
119                     deletedSelf = true;
120                 }
121             }
122             else
123             {
124                 deletedSelf = false;
125             }
126         }
127         return deletedSelf;
128     }
129 
130     //Function to delete given key from Trie
131     public void delete(String key){
132         if ((root == null) || (key == null)){
133             System.out.println("Null key or Empty trie error");
134             return;
135         }
136         deleteHelper(key, root, key.length(), 0);
137     }
138 
139 }
```

Explanation

In `main()` function, we first check if the string we want to delete is present in our Trie. If the search results are positive then the `Delete("key")` function is called.

It takes in a **“key”** of type `String` and then checks if either the **Trie is empty** or the **key is null**; if any of the cases is true, it simply returns from the function.

If the **Trie is not empty** and the **"key" is also not null**, then `deleteHelper()` is called to delete the key. It is a recursive function and takes in a **"key"** of type `String`, `root` of Trie, `length` of the key, and an integer indicating `level` as an argument.

It goes through all the cases explained above, while the base case for this recursive function is when the level becomes equal to the length of the key; i.e we've reached the last node in a Trie path, indicating the last character for the particular word. At this point, we check if the last node has any further children or not. If it does then we simply `unMark` it (i.e set `isEndWord` to `false`). On the other hand, if the last node doesn't contain any children, then we simply set it to *null* and move up in Trie to check for the remaining nodes of the path.

And now that we have covered all the nitty gritty details of Trie including its implementation in Java, let's try to solve some practice questions using Trie in the next lessons!

Back

Search in a Trie

Next

Challenge 1: Total Number of Words i...

Mark as Completed

74% completed, meet the criteria and claim your course certificate!

Claim Certificate

Report an Issue

Ask a Question (https://discuss.educative.io/tag/deletion-in-a-trie__trie-advanced-trees__data-structures-for-coding-interviews-in-java)