Tries

Introduction

Suppose we want to implement a word-dictionary using a C++ program and perform the following functions:

- Addition of the word(s)
- Searching a word
- Removal of the word(s)

To do the same, hashmaps can be thought of as an efficient data structure as the **average case time complexity** of insertion, deletion, and retrieval is O(1) for integer, character, float, and decimal values.

Let us discuss the time complexity of the same in case of strings.

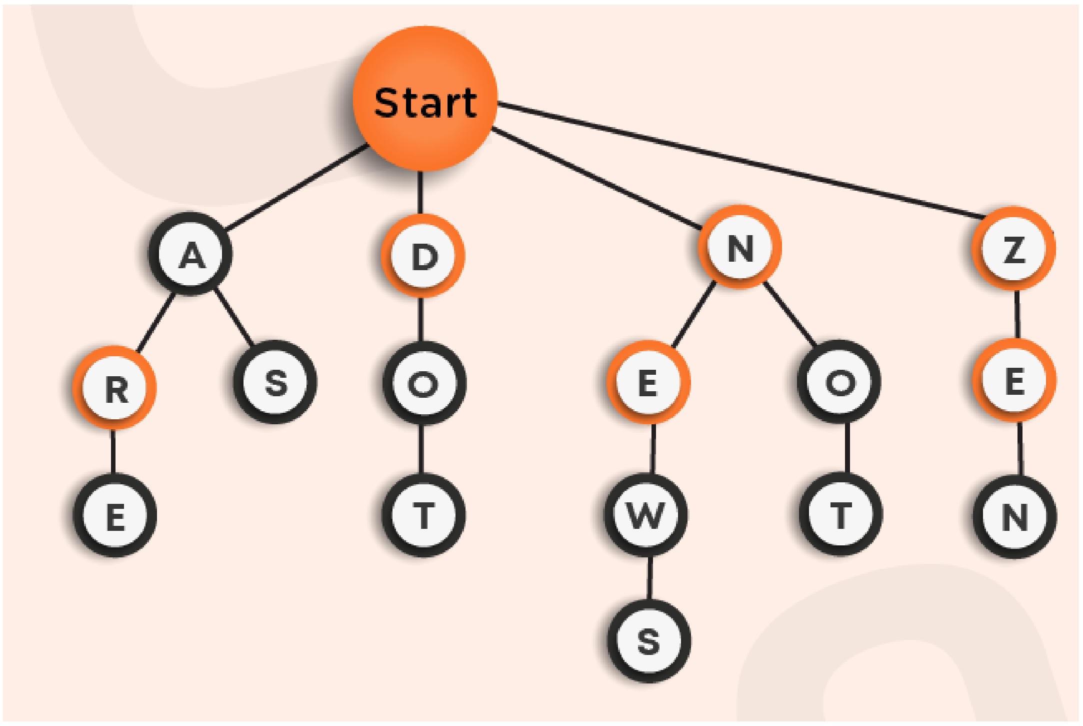
Suppose we want to insert string ***abc*** in our hashmap. To do so, first, we would need to calculate the hashcode for it, which would require the traversal of the whole string *abc*. Hence, the time taken will be the length of the entire string irrespective of the time taken to perform any of the above operations. Thus, we can conclude that the insertion of a string will be O(string_length).

To search a word in the hashmap, we again have to calculate the hashcode of the string to be searched, and for that also, it would require O(string_length) time.

Similarly, for removing a word from the hashmap, we would have to calculate the hashcode for that string. It would again require O(string_length) time.

For the same purpose, we can use another data structure known as **tries**.

Below is the visual representation of the trie:



Here, the node at the top named as the **start** is the root node of our **n-ary tree.**

Suppose we want to insert the word ***ARE*** in the trie. We will begin from the root, search for the first word ***A*** and then insert ***R*** as its child and similarly insert the letter ***E***. You can see it as the left-most path in the above trie.

Now, we want to insert the word ***AS*** in the trie. We will follow the same procedure as above. First-of-all, we find the letter ***A*** as the child of the root node, and then we will search for ***S***. If ***S*** was already present as the child of ***A***, then we will do nothing as the given word is already present otherwise, we will insert ***S***. You can see this in the above trie. Similarly, we added other words in the trie.

This approach also takes O(word_length) time for insertion.

Moving on to searching a word in the trie, for that also, we would have to traverse the complete length of the string, which would take O(word_length) time.

Let us take an example. Suppose we want to search for the word ***DOT*** in the above trie, we will first-of-all search for the letter ***D*** as the direct child of the start node and then search for ***O*** and ***T*** and, then we will return true as the word is present in the trie.

Consider another example ***AR***, which we want to search for in the given trie. Following the above approach, we would return true as the given word is present in it. However ideally, we should return false as the actual word was ***ARE*** and not ***AR***. To overcome this, it can be observed that in the above trie, some of the letters are marked with a bolder circle, and others are not. This boldness represents the termination of a word starting from the root node. Hence, while searching for a word, we will always be careful about the last letter that should be bolded to represent the termination.

Note: While insertion in a trie, the last letter of the word should be bolded as a mark of termination of a word.

In the above trie, the following are all possibilities that can occur:

- ARE, AS
- DO, DOT
- NEW, NEWS, NOT
- ZEN

Here, to remove the word, we will simply unbold the terminating letter as it will make that word invalid. For example: If you want to remove the string ***DO*** from the above trie by preserving the occurrence of string DOT, then we will reach ***O*** and then unbolden it. This way the word ***DO*** is removed but at the same time, another word ***DOT*** which was on the same path as that of word ***DO*** was still preserved in the trie structure.

For removal of a word from trie, the time complexity is still O(word_length) as we are traversing the complete length of the word to reach the last letter to unbold it.

When to use tries over hashmaps?

It can be observed that using tries, we can improve the space complexity.

For example: We have 1000 words starting from character ***A*** that we want to store. Now, if you try to hold these words using hashmap, then for each word, we have to store character ***A*** differently. But in case of tries, we only need to store the character ***A*** once. In this way, we can save a lot of space, and hence space optimization leads us to prefer tries over hashmaps in such scenarios.

In the beginning, we thought of implementing a dictionary. Let's recall a feature of a dictionary, namely **Auto-Search.** While browsing the dictionary, we start by typing a character. All the words beginning from that character appear in the search list. But this functionality can't be achieved using hashmaps as in the hashmap, the data stored is independent of each other, whereas, in case of tries, the data is stored in the form of a tree-like structure. Hence, here also, tries prove to be efficient over hashmaps.

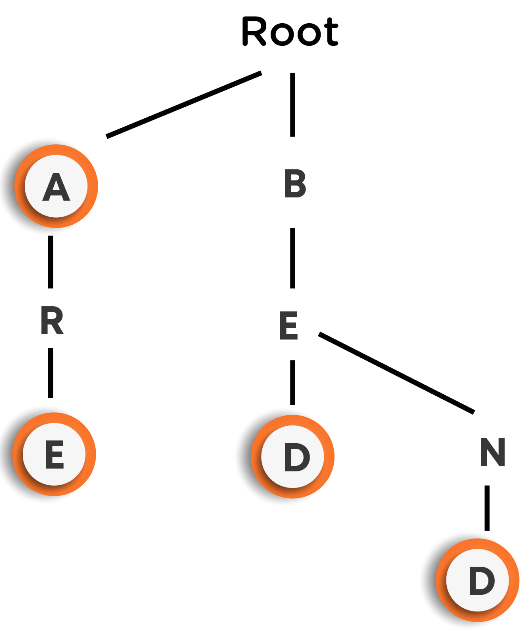
TrieNode class implementation

Follow the below-mentioned code (with comments)...

```
class TrieNode {
      public :
      char data;      // To store data (character value: 'A' - 'Z')
      TreeNode **children;   // To store the address of each child
      bool isTerminal;  // it will be TRUE if the word terminates at
this character
      TrieNode(char data) {  // Constructor for values initialization
            this -> data = data;
            children = new TrieNode*[26];
                           for(int i = 0; i < 26; i++) {
                                             children[i] = NULL;
                           }
            isTerminal = false;
      }
};
```

Insert function

To insert a word in a trie, we will use recursion. Suppose we have the following trie:



Now we want to insert the word ***BET*** in our trie.

Recursion says that we need to work on a smaller problem, and the rest will handle itself. So, we will do it on the root node.

Note: Practically, the functionality of bolding the terminal character is achieved using the boolean **isTerminal** variable for that particular node. If this value is true means that the node is the terminal value of the string, otherwise not.

Approach:

We will first search for letter ***B*** and check if it is present as the children of the root node or not and then call recursion on it. If ***B*** is present as a child of the root node as in our case it is, then we will simply recurse over it by shifting the length of the string by 1. In case, character ***B*** was not the direct child of the root node, then we have to create one and then call recursion on it. After the recursive call, we will see that be is now a root node, and the character ***E*** is now the word we are searching for. We will follow the same procedure as done in searching for character ***B*** against the root node and then move forward to the next character of the string, i.e., ***T,*** which happens to be the last character of our string. Now we will check character ***T*** as the child of character ***E***. In our case, it is not a child of character ***E***, so we'll create it. As ***T*** is the last character of the string, so we will mark its **isTerminal** value to **True**.

Following will be the three steps of recursion:

- **Small Calculation:** We will check if the root node has the first character of the string as one of its children or not. If not, then we will create one.
- **Recursive call:** We will tell the recursion to insert the remaining string in the subtree of our trie.
- **Base Case:** As the length of the string becomes zero, or in other words, we reach the NULL character, then we need to mark the **isTerminal** for the last character as True.

Follow the code below, along with the comments…

```
class Trie {
      TrieNode *root;

      public :
      Trie() {
            root = new TrieNode('\0');
      }

      void insertWord(TrieNode *root, string word) {
            // Base case
            if(word.size() == 0) {
                  root -> isTerminal = true;
                  return;
            }
            // Small Calculation
            int index = word[0] - 'a';   // As for 'a' refers to index
0, 'b' refers to index 2 and so on, so to reach the correct index we
will do so
            TrieNode *child;
            if(root -> children[index] != NULL) {   // If the first
character of string is already present as the child node of the root
node
                  child = root -> children[index];
            }
            else {        // If not present as the child then creating
one.
                  child = new TrieNode(word[0]);
                  root -> children[index] = child;
            }

            // Recursive call
            insertWord(child, word.substr(1));
      }
      // For user
      void insertWord(string word) {
            insertWord(root, word);
      }
};
```

Search in tries

Objective: Create a search function which will get a string as its argument to be searched in the trie and returns a boolean value. **True** if the string is present in the trie and **False** if not.

Approach: We will be using the same process as that used during insertion. We will call recursion over the root node after searching for the first character as its child. If the first character is not present as one of the children of the root node, then we will simply return false; otherwise, we will send the remaining string to the recursion. When we reach the empty string, then check for the last character's **isTerminal** value; if it is **true,** it means that word exists in our trie, and we will return true from there otherwise, return false.

Try to code it yourselves, and for the code, refer to the solution tab of the same.

Tries implementation: Remove

Objective: To delete the given word from our trie.

Approach: First-of-all we need to search for the word in the trie, and if found, then we simply need to mark the **isTerminal** value of the last character of the word to **false** as that will simply denote that the word does not exist in our trie.

Check out the code below: (Nearly same as that of insertion, just a minor changes which are explained along side)

```
void removeWord(TrieNode *root, string word) {
            // Base case
            if(word.size() == 0) {
                  root -> isTerminal = false;
                  return;
            }

            // Small calculation
            TrieNode *child;
            int index =  word[0] - 'a';
            if(root -> children[index] != NULL) {
                  child = root -> children[index];
            }
            else {
                  // Word not found
                  return;
            }

            removeWord(child, word.substr(1));
// Suppose if the character of the string doesn't have any child and
is a part of the word to be deleted, then we can simply delete that
node also as it is not referencing to any other word in the trie

            // Removing child Node if it is useless
            if(child -> isTerminal == false) {
                for(int i = 0; i < 26; i++) {
                    if(child -> children[i] != NULL) {
                        return;
                    }
                }
                delete child;
                root -> children[index] = NULL;
            }
        }

            // For user
             void removeWord(string word) {
            removeWord(root, word);
        }
```

Types of tries

There are two types of tries:

- **Compressed tries:**
  - Majorly, used for space optimization.
  - We generally club the characters if they have at most one child.
  - **General rule:** Every node has at least two child nodes.

Refer to the figure below:

Suppose our regular trie looks like this-

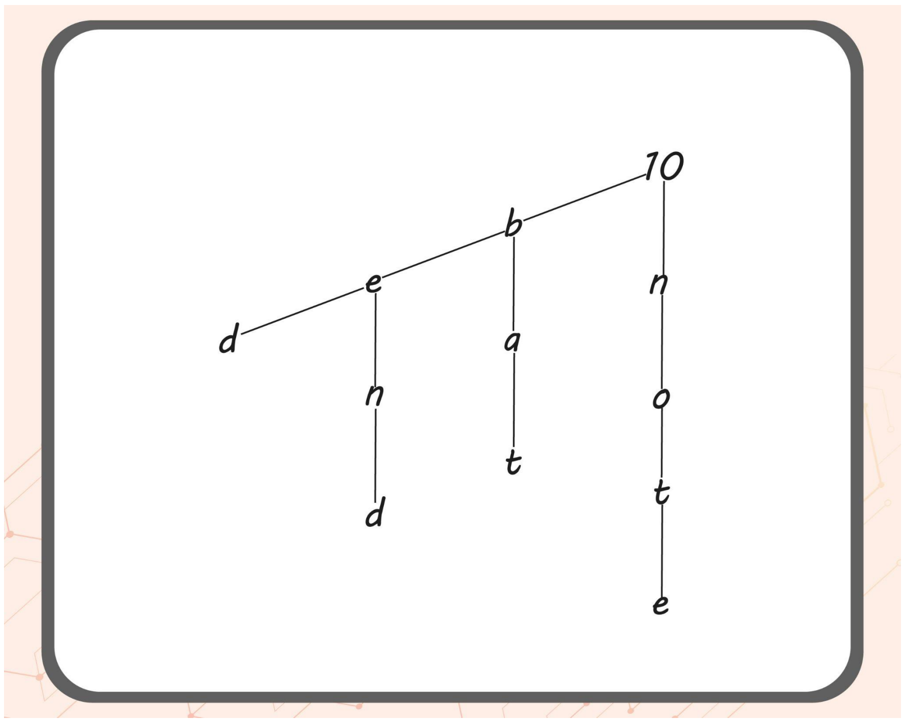
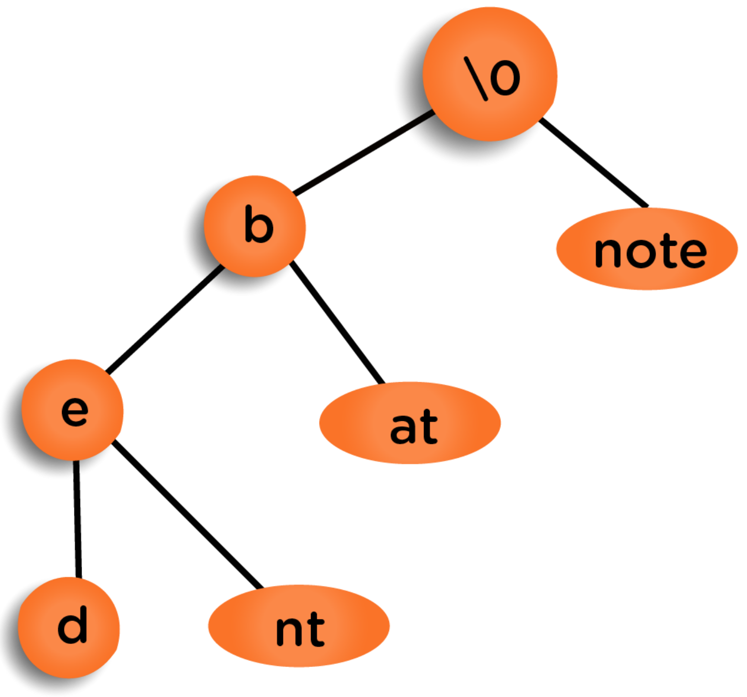


Figure - 1

Its compressed trie version will look as follows:



- **Pattern matching:**
    - Used to match patterns in the trie.
        - Example: In the figure-1 (shown above), if we want to search for pattern **ben** in the trie, but the word **bend** was present instead, using the normal search function, we would return false, as the last character **n**'s isTerminal property was false, but in this trie, we would return true.
    - To overcome this problem of the last character's identification, just remove the **isTerminal** property of the node.
    - In the figure-1, instead of searching for the pattern **ben**, we now want to search for the pattern **en**. Our trie would return false if **en** is not directly connected to the root. But as the pattern is present in the word **ben**, it should return true. To overcome this problem, we need to

attach each of the prefix strings to the root node so that every pattern is encountered.

- **For example:** for the string **ben,** we would store **ben**, **en**, **n** in the trie as the direct children of the root node.

Maximum XOR Subarray

Problem Statement: We are given an array and we have to find a subarray that has the maximum XOR possible.

Explanation:

Example:

Input: arr[] = {1, 2, 3, 4}

Output: 7

The subarray {3, 4} has maximum XOR value

Basic Approach: The basic approach to the solution of this problem includes calculating the XOR of every possible two elements. This would take O(N^2) time.

Trie Approach:

Algorithm:

Maintain a trie data structure which can handle two types of queries:

1. Insert an element (bitwise representation) into trie.
2. For each element Y:
    a. Find the maximum XOR of Y with all the numbers that have been inserted till now.
    b. Insert integers as you go, and keep finding the maximum XOR and keep track of the overall maximum.

Let's say our number Y is b1,b2...bn, where b1,b2.. are binary bits. We start from b1. Now for the XOR to be maximum, we'll try to make the most significant bit 1 after taking XOR. So, if b1 is 0, we'll need a 1 and vice versa. In the trie, we go to the required bit side. If a favorable option is not there, we'll go to the other side. Doing this all over for i=1 to n, we'll get the maximum XOR possible.

Sub - XOR

Problem Statement: Given an array of positive integers, you have to print the number of subarrays whose XOR is less than K.

Explanation:

Example:

Input: arr[] = {8, 9, 10, 11, 12}, k=3

Output: 4

Sub-arrays [1:3], [2:3], [2:5], [4:5] have xor

values 2, 1, 0, 1 respectively.

We will calculate all of the prefix XOR values i.e. arr[1:i] for all i. It can be verified that the xor of a subarray arr[l:r] can be written as (arr[1:l-1] xor arr[1:r]), where arr[i, j] is the xor of all the elements with index such that, i <= index <= j.

If xor[i, j] represents the xor of all elements in the subarray a[i, j], then at an index i what we have is, a trie which has xor[1:1], xor[1:2].....xor[1:i-1] already inserted. Now we somehow count how many of these (numbers in trie) are such that its xor with xor[1:i] is smaller than k. This will cover all the subarrays ending at the index i and having xor i.e. xor[j, i] <=k;

Now the problem remains, how to count the numbers with xor smaller than k. So, for example take the current bit of the ith index element is p, current bit of number k be q and the current node in trie be node.

Take the case when p=1, k=1. Then if we go to the right child the current xor would be 0 (as the right child means 1 and p=1, 1(xor)1=0).As k=1, all the numbers that are to the right child of this node would give xor value smaller than k. So, we would count the numbers that are right to this node.

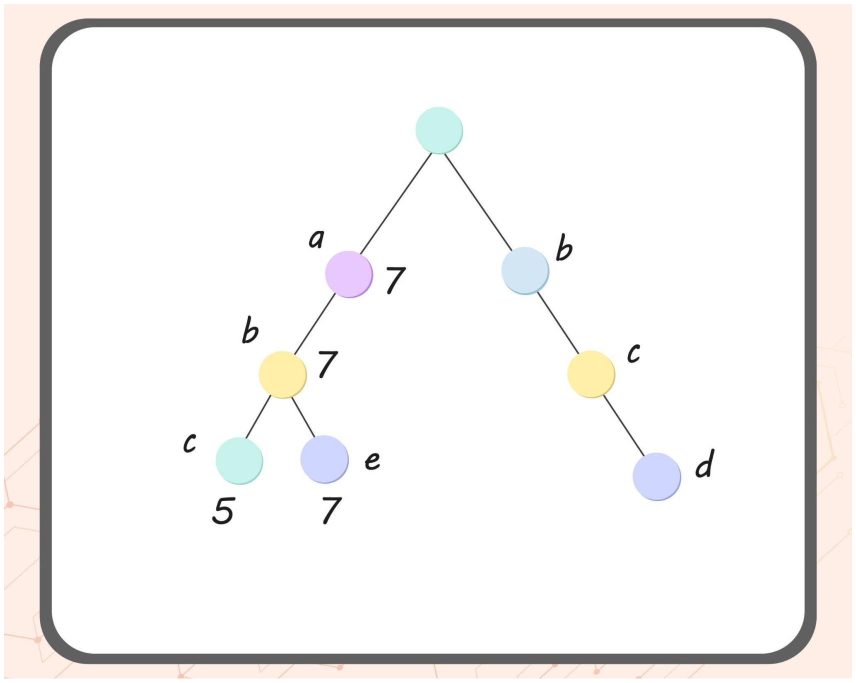
If we go to the left child, the current xor would be 1 (as the left child means 0 and p=1, 0(xor)1=1). So, if we go to the left child we can still find a number with xor smaller than k, therefore we move on to the left child.

Search Engine

Problem Statement: We are given N strings along with their weights. We get Q queries and in each query we get a prefix and we have to find the string with the best weight that contains that prefix.

Explanation:

We need the node in the trie to store the optimal weight among the prefixes that is formed by its children. Suppose we are given strings “abc”, “bcd”, “abe” with weights 5, 6, 7 respectively. Let’s take a look at the image given below to see how the trie is formed for this example:



Now we get the query along with the prefix “ab” so “ab” occurs in “abc” and “abe”. So as shown in the diagram at node a and b in the LHS, we store the optimal weight as the answer which in this case is 7 from the string “abe”.