Tries and Huffman Coding

Introduction to tries

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.

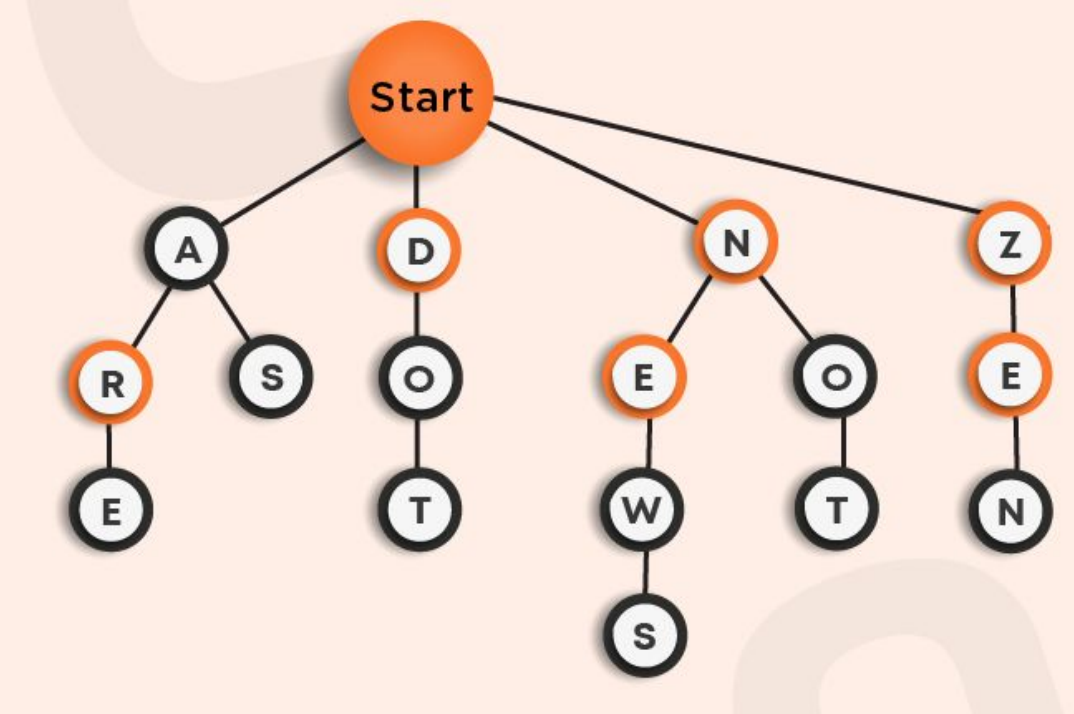
Suppose we want to insert string ***abc*** in our hashmap. To do so, first, we would need to calculate the hashcode for it, which would require the traversal of the whole string *abc*. Hence, the time taken will be the length of the entire string irrespective of the time taken to perform any of the above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

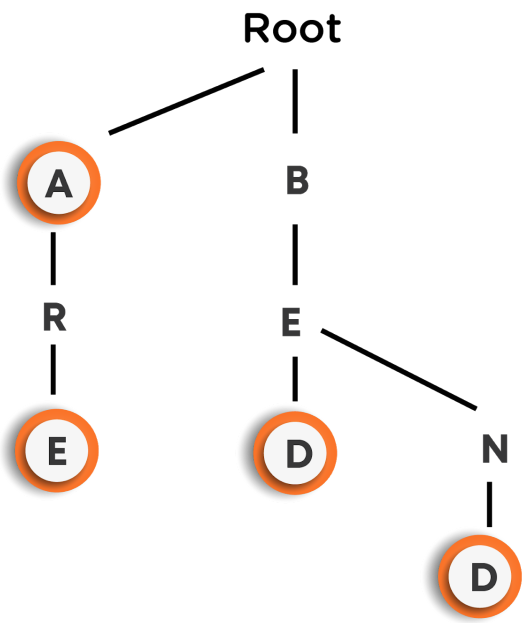
```
class TrieNode {
        public :
        char data;                              // To store data (character value: 'A' - 'Z')
        TreeNode **children;                    // To store the address of each child
        bool isTerminal;     // it will be TRUE if the word terminates at this character

        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 it 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,** 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-

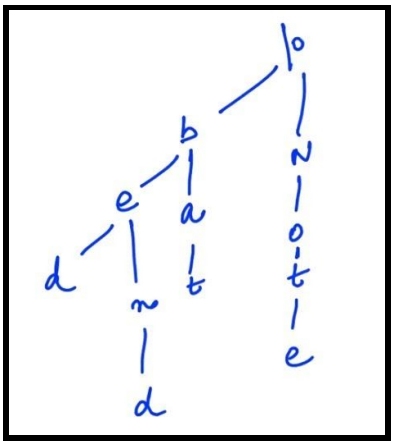
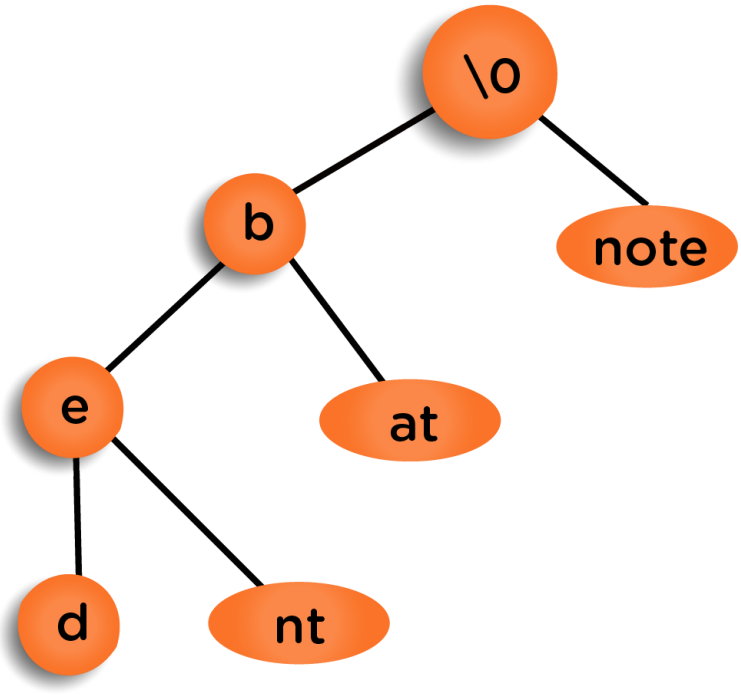


Figure - 1

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.

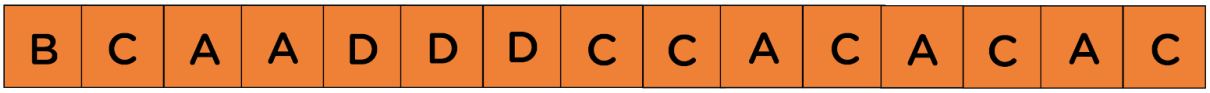
Huffman Coding

Huffman Coding is one approach followed for **Text Compression.** Text compression means reducing the space requirement for saving a particular text.

Huffman Coding is a lossless data compression algorithm, ie. it is a way of compressing data without the data losing any information in the process. It is useful in cases where there is a series of frequently occurring characters.

Working of Huffman Algorithm:

Suppose, the given string is:



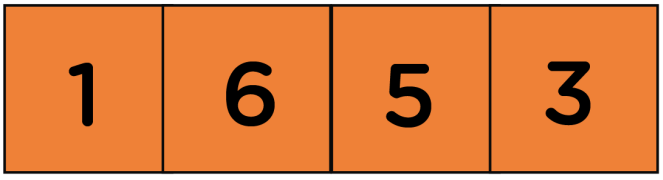
Intial String

Here, each of the characters of the string takes 8 bits of memory. Since there are a total of 15 characters in the string so the total memory consumption will be 15*8 = 120 bits. Let's try to compress its size using the Huffman Algorithm.

First-of-all, the Huffman Coding creates a tree by calculating the frequencies of each character of the string and then assigns them some unique code so that we can retrieve the data back using these codes.

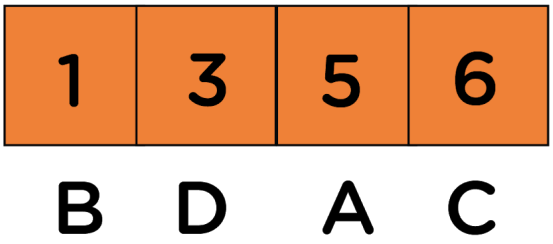
Follow the steps below:

1. Begin with calculating the frequency of each character value in the given string.



Frequency of String

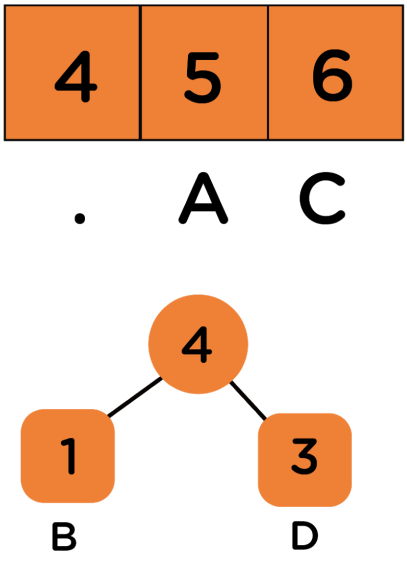
2. Sort the characters in ascending order concerning their frequency and store them in a priority queue, say **Q.**
3. Each character should be considered as



Characters sorted according to the frequency

a different leaf node.

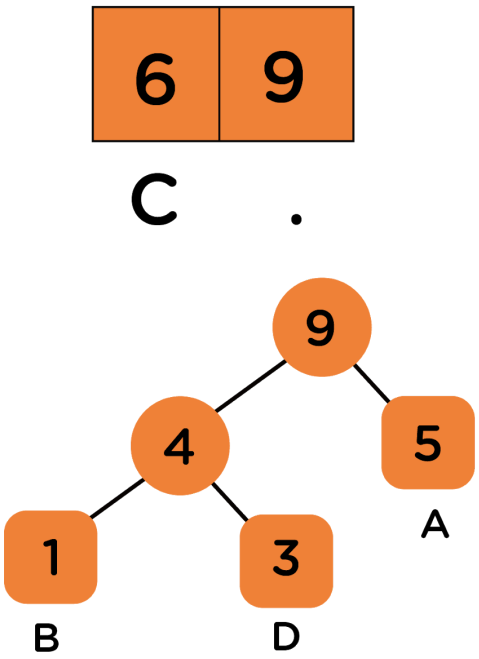
4. Make an empty node, say **z**. The left child of z marked as the minimum frequency and right child, the second minimum frequency. The value of z is calculated by summing up the first two frequencies.



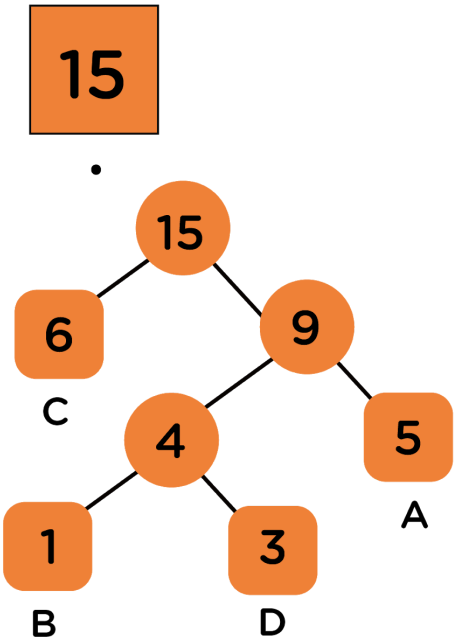
Getting the sum of the least numbers

Here, * denote the internal nodes.

5. Now, remove the two characters with the lowest frequencies from the priority queue Q and append their sum to the same.
6. Simply insert the above node z to the tree.
7. For every character in the string, repeat steps 3 to 5.

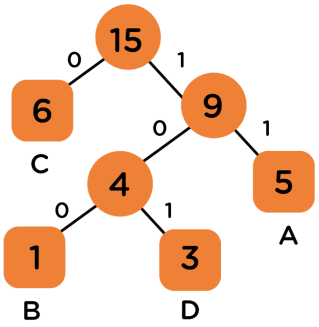


Repeat steps 3 to 5 for all the characters.



Repeat steps 3 to 5 for all the characters.

8. Assign 0 to the left side and 1 to the right side except for the leaf nodes.



Assign 0 to the left edge and 1 to the right edge

The size table is given below:

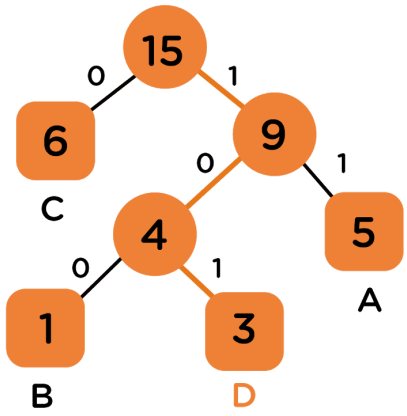
Character	Frequency	Code	Size
A	5	11	5*2 = 10

B	1	100	1*3 = 3
C	6	0	6*1 = 6
D	3	101	3*3 = 9
4*8 = 32 bits	15 bits		28 bits

Size before encoding: 120 bits

Size after encoding: 32 + 15 + 28 = 75 bits

To decode the code, simply traverse through the tree (starting from the root) to find the character. Suppose we want to decode 101, then:



Decoding

Time complexity:

In case of encoding, inserting each character into the priority queue takes O(log n) time. Therefore, for the complete array, the time complexity becomes O(nlog n).

Similarly, extraction of the element from the priority queue takes O(log n) time. Hence, for the complete array, the achieved time complexity is O(nlog n).

Applications of Huffman Coding:

- They are used for transmitting fax and text.
- They are used by conventional compression formats like PKZIP, GZIP, etc.