

**BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI**  
***CS-F211: Data Structures and Algorithms***  
**Auxillary Lab-Sheet : Tries Data-structure, Huffman Encoding**

## **Introduction**

The Trie Data-structure is an ADT that is primarily used for storing, checking and retrieving data with keys that are strings defined over a sequence of alphabets (eg. the english letter, binary 0 or 1, etc). The data-structure has an underlying Tree, with nodes storing one of the characters belonging to the defined set of alphabets (and a pointer to the corresponding data item if it exists), along with other fields.

In this lab-sheet, we will be learning about tries and its variants. We will also be learning about “Huffman Encoding”, which is a data compression technique that involves using a min-heap (min-priority queue) to construct a binary tree, that contains the details of character encodings.

## **Topics to be covered in this lab-sheet**

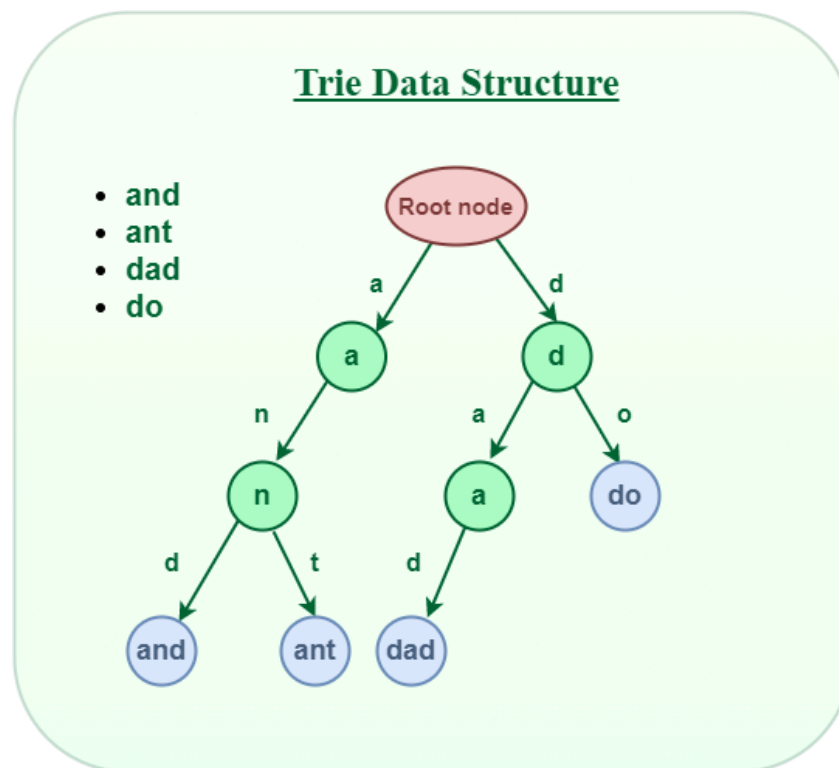
- Introduction to Tries
  - Operations on Tries
  - Code implementation
  - The Space-Time Trade-off
- Compressed Tries
- Suffix Tries
- Huffman Encoding

## Introduction to Tries

A Trie (reTRIEval) data structure is an n-ary tree that is used for storing and retrieving strings as described previously. It supports operations for inserting strings (and optionally data corresponding to that string as a key-value pair), along with searching for a string (whether it has been inserted or not) and deleting the string that has been previously inserted. A trie is also known as a digital tree or prefix tree.

Structure of a Trie: Every node of a Trie stores an alphabet, and a bool value isEndOfWord, along with an array (array size = number of alphabets that can be used for constructing the string) of pointers to its children nodes. The Root node does not hold any alphabet.

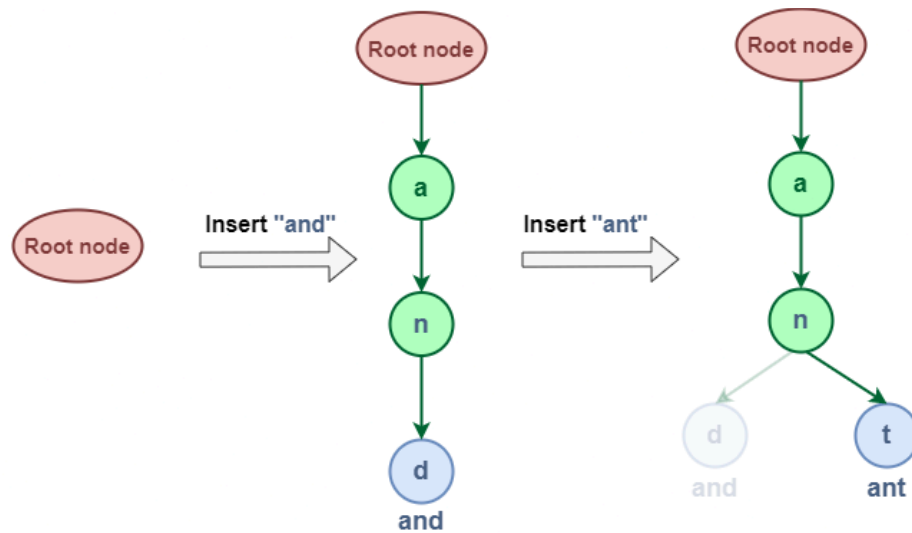
Note that all operations that are described below are linear in time i.e.  $O(n)$  where  $n$  is the size of the string passed as input to the operation, given that the size of the set of alphabets is a constant. The *worst* space complexity under the same assumption is also  $O(\Sigma(n))$ , where  $\Sigma(n)$  is the sum of lengths of all strings inserted into the trie so far.



Overview of a Trie

### Insertion in a Trie:

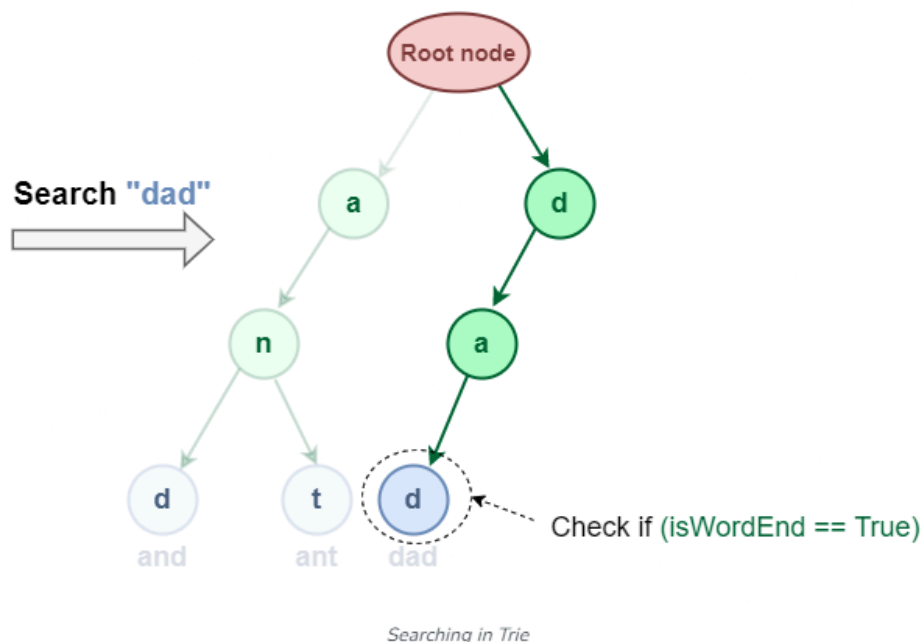
- Every character of the input key is inserted as an individual Trie node. Note that the children of a trie node are stored in an array of pointers (or references) to next-level trie nodes.
- The key character at a particular level acts as an index to the array of children.
- If the input key is new or an extension of the existing key, construct non-existing nodes for the new key, and mark isEndOfWord true for the last node.
- If the input key is a prefix of an existing key in Trie, simply mark the last node of the input key as the end of a word.
- The key length of the longest inserted string determines Trie depth



*Inserting in Trie*

### Search Operation in a Trie:

- Searching for a key is similar to the insert operation. However, It only compares the characters and moves down. The search can terminate due to the end of a string or lack of key in the trie.
- In the former case, if the isEndofWord field of the last node is true, then the key exists in the trie.
- In the second case, the search terminates without examining all the characters of the key, since the key is not present in the trie.
- Note that instead of returning true or false, if there is a data field defined in the node, then the function shall return the pointer to the data item, otherwise it shall return NULL.



### Deletion Operation in a Trie (recursive description):

Method 1: Traverse down the Trie to the required node, and mark isEndOfWord as false; (free the void\* data pointer if part of the data-structure).

Method 2: (Space optimisations - TrieNode data-structure will require changes) In the recursive function stack, after reaching the required node to delete in a recursive call manner, check if it has any children (by adding a childCount integer in the data-structure definition). If it doesn't, then delete the node and then return true. Else return false. Now, the calling function

that is operating on the parent of the deleted node gets the returned value on calling the function on its child. If the value is false, then return false. If the value received is true, decrement childCount of the parent by one. If the childCount is zero, then free the data in this parent node(along with its pointers) and return true. Otherwise return false.

**Basic code implementation of a Trie:** (reference: [geeksforgeeks](#))

```
#include<stdio.h>
#include<stdbool.h>
#include<string.h>
#include<stdlib.h>

#define ALPHABET_SIZE 128 //you can instead reduce the alphabet size and create
your own mapping of character to index in array

int CHAR_TO_INDEX(char c)
{
    //custom mapping if alphabet size reduced
    return (int)c;
}

// trie node
struct TrieNode
{
    struct TrieNode *children[ALPHABET_SIZE];
    // isEndOfWord is true if the node represents
    // end of a word
    bool isEndOfWord;
    // void* data; //instead of checking for the presence or absence of a string,
if we wish to assign some value with the key as a pair, then this could be
uncommented
};

// Returns new trie node (initialized to NULLs)
struct TrieNode *getNode(void)
{
    struct TrieNode *pNode = NULL;
```

```

    pNode = (struct TrieNode *)malloc(sizeof(struct TrieNode));
    if (pNode)
    {
        int i;
        pNode->isEndOfWord = false;
        for (i = 0; i < ALPHABET_SIZE; i++)
            pNode->children[i] = NULL;
    }
    return pNode;
}

// If not present, inserts key into trie
// If the key is prefix of trie node, just marks leaf node
void insert(struct TrieNode *root, const char *key)
{
    int level;
    int length = strlen(key);
    int index;
    struct TrieNode *pCrawl = root;
    for (level = 0; level < length; level++)
    {
        index = CHAR_TO_INDEX(key[level]);
        if (!pCrawl->children[index])
            pCrawl->children[index] = getNode();
        pCrawl = pCrawl->children[index];
    }
    // mark last node as leaf
    pCrawl->isEndOfWord = true;
}

// Returns true if key presents in trie, else false
bool search(struct TrieNode *root, const char *key)
{
    int level;
    int length = strlen(key);

```

```

    int index;
    struct TrieNode *pCrawl = root;
    for (level = 0; level < length; level++)
    {
        index = CHAR_TO_INDEX(key[level]);
        if (!pCrawl->children[index])
            return false;
        pCrawl = pCrawl->children[index];
    }
    return (pCrawl->isEndOfWord);
}

// Driver
int main()
{
    // Input keys (use only 'a' through 'z' and lower case)
    char keys[][8] = {"the", "a", "there", "answer", "any",
                     "by", "bye", "their"};
    char output[][32] = {"Not present in trie", "Present in trie"};
    struct TrieNode *root = getNode();
    // Construct trie
    int i;
    for (i = 0; i < sizeof(keys)/sizeof(keys[0]); i++)
        insert(root, keys[i]);
    // Search for different keys
    printf("%s --- %s\n", "the", output[search(root, "the")] );
    printf("%s --- %s\n", "these", output[search(root, "these")] );
    printf("%s --- %s\n", "their", output[search(root, "their")] );
    printf("%s --- %s\n", "thaw", output[search(root, "thaw")] );
    return 0;
}

```

Task 1: Copy the code given above in trieADT.c, and implement the delete operation as described in Method 2 above of “delete operation in a trie” using the template of the trie and its other operations implemented above. Test out your code on names1000.csv attached alongside the lab sheet. Also create a function freeTrieRecursive that frees the data, nodes and pointers of the trie recursively, to prevent memory leaks.

Home exercise 1: *Maximum XOR subarray problem:*

Given an array of integers, find the subarray with the maximum XOR value in **O(n)** time and **auxiliary space complexity**.

A subarray of the array  $a$  is a sequence  $a_l, a_{l+1}, \dots, a_r$  for some integers  $(l, r)$  such that  $1 \leq l \leq r \leq n$ . (XOR between two integers can be computed using the bitwise operator  $\wedge$  )

The XOR value of a subarray is defined as the the bitwise XOR of all elements inside that subarray

(Note that the XOR operation is *commutative* as well as *associative*, and that the *inverse* operation of XOR is XOR itself).

*Concepts required:*

- *Prefix-sum arrays (Think about how we can extend the logic to compute a subarray XOR of the given array in  $O(1)$  after  $O(n)$  precomputation)*
- *Tries over Alphabet =  $\{0, 1\}$ .*



### The Space-Time Trade-off:

Notice that the size of each node in a trie is proportional to the size of the set of alphabets that the trie operates on. To be precise, the array of pointers to its children that is initialised in a node has a size of  $8 * (\text{ALPHABET\_SIZE})$  bytes, regardless of how many of its children are NULL valued. In practice, this could lead to a lot of space being utilised and wasted as well. For example, if a trie is defined for alphabets belonging to all ASCII characters, then the space used would be  $128 * 8 = 1024$  bytes = 1KB!! And so, even if only a thousand nodes are initialised (which in practice is not a lot), the datastructure would take more than 1 MB of space.

Now, depending on the use-case of the trie (for example, if the underlying tree is **sparsely-populated**<sup>1</sup>), we may as well increase the number of children dynamically, so that there are no NULL entries. To implement this, we would need to **replace the fixed-size array children with**

- (a) a **dynamic array of children** whose size should increase with the child count;
- (b) an **integer to store the child count** of the current node; and
- (c) a **dynamic array of characters** whose size should also increase with the child count.

This array of characters would let us know *which child refers to which alphabet*. That is, at the same index in their corresponding array, we get to know what is the child pointer's value in the children array and what is the corresponding alphabet in the char array. We would then linearly iterate over the number of children in the *character* array to find a match for the required character transition, and if the required character exists, we index on to the pointer to the next child having the same index as the character in the *children* array.

The tradeoff in this case is the linear lookup, which in the worst case would be per-node  $O(\text{ALPHABET\_SIZE})$ , which would be a multiplicative factor in the total time complexity of operations. However, we are reducing the wasted space if the tree is not very densely populated. That is the trade-off between space optimizations for time complexity.  $\text{ALPHABET\_SIZE}$  is usually not greater than 50 or 100, so it may as well be considered a constant for the purpose of time complexity calculations, essentially keeping it the same. Also, if the tree is sparsely-populated<sup>1</sup>, then on average, the required iterations would be much less than  $\text{ALPHABET\_SIZE}$ .

[Task 2: Implement this space optimization described above in a new file trieADTv2.c by copying the contents from trieADT.c and making the required changes. Test out your code on names1000.csv attached alongside the labsheet.](#)

---

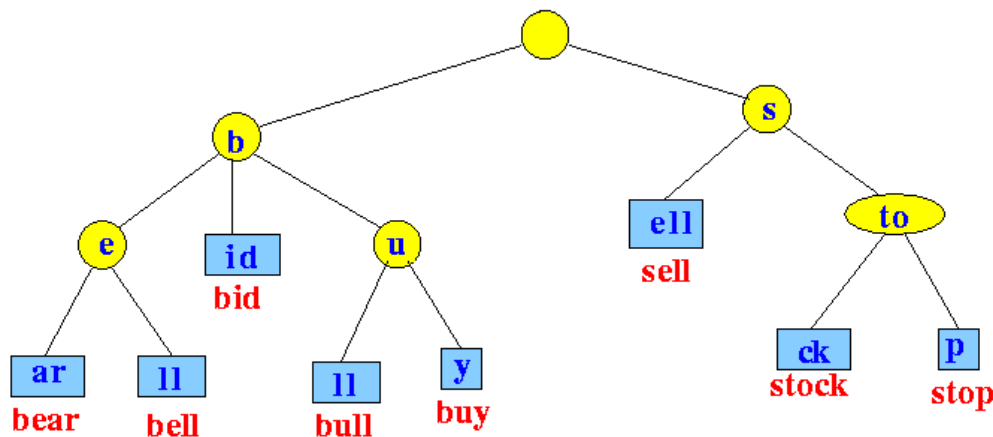
<sup>1</sup> A sparsely populated data-structure is one in which the ratio of the number of entries inserted into the data-structure, to the memory space that the data-structure is occupying, is much less than the maximum possible ratio. In terms of data-structures with underlying tree implementations (for example, tries), it often refers to the ratio of the average branching factor to the maximum branching factor being very low.

## Compressed Tries

Compressed tries are a modified version of the standard trie data-structure. A compressed trie-node struct contains an additional string variable, which contains the common prefix of all its children-nodes. This ensures that no node in the trie can have only one child.. it either has two or more branchings, or is a leaf node. The only situation that a trie-node can have one child in a compressed trie is if it is the end of the word, but is also a prefix of other strings inserted into the trie. For convenience, to mark the end of string character and simplify our data-structure, we may get rid of the `isEndOfWord` bool variable, and mark the end of a string with a character that won't appear in the alphabet set: `'\0'`. Now, with this modification, we have made sure that no node can have only one child.

Compressed Tries are also known as Radix Tries, where the radix is equal to the number of alphabets in the alphabet-set. PATRICIA Tries are Radix Tries with radix equal to 2, and strings are stored based on the binary encodings of the sequence of characters. Compressed Tries have a lot of applications in different forms, many to do with linear time algorithms for a string in a document, such as searching for string `s` in document `d`, or counting the number of occurrences of `s` in `d`, finding the most common substring appearing in `d`, etc. Most of these applications are implemented using suffix trees, which are a form of compressed tries.

Note that, while coding the compressed trie, there are a lot of considerations to be made to make sure that the compressed trie is logically correct. For example, while inserting a new string, it is possible that a leaf node splits into two, along with splitting the string that the node was storing between itself and one of its children. Describing these intricacies would take longer than figuring it out while coding it yourself, so I will leave the implementation as an exercise for you.



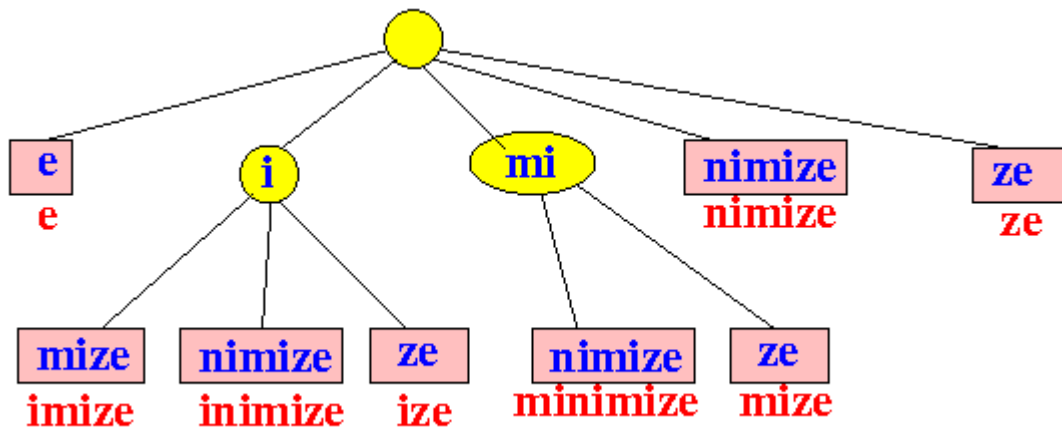
*Representation of a compressed trie, where the words  
"bear", "bell", "bid", "bull", "buy", "sell", "stock", "stop" have been inserted*

Home exercise 2: Implement the compressed tries data-structure described above in C. Test out your code on `names1000.csv` attached alongside the labsheet.

## Suffix Tries

For a given string  $s$  of length  $n$ , its corresponding suffix trie is a trie that has been populated with the suffixes of the string, including the string itself ( $\Rightarrow$  a total of  $n$  entries into the trie).

***A Suffix Tree is a suffix trie that has been compressed. Note that instead of storing the substring in the node variable, we may just store the indices that correspond to the start and end of the substring of  $s$ . This will help improve the space and time complexity of the algorithms that we will be using for suffix trees specifically.***



*Suffix Tree of the word "minimise". Note that instead of the substring, you will be storing the start and end indices corresponding to it in the input string ie "minimise"*


Searching for string  $s$  in document  $d$ , or counting the number of occurrences of  $s$  in  $d$ , finding the most common substring appearing in  $d$  - can all be implemented efficiently using suffix trees. Once populated with the suffixes of the document  $d$ , both suffix trees and suffix tries (with some modifications) can achieve the same algorithmic time complexity for most purposes. However, it is the time complexity of populating these data structures with the suffix strings that differentiates them.

Space optimization consideration-wise, suffix trees are the clear winner, as they are compressed.

How to construct a Suffix Trie? We would simply iterate over the suffixes of the given string and insert them one by one into the trie. Note that the time and space complexity of this algorithm would be  $O(n^2)$ , as we would require at least  $n \cdot (n+1)/2$  iterations, and the space complexity in the worst case would happen when for example all the characters of the string are distinct (of course, this is not the best example as that practically limits the length of the string to the size of the alphabet-set. The upper-bound is  $O(n^2)$  in space in general when there are very few common nodes used while inserting). Note that since we have to create nodes in the quadratic order in the worst case, we can not reduce the time complexity further for suffix tries.

How to construct a Suffix Tree? We could do it in a similar manner as described above. That would lead to the same time complexities as described above. **But there is a better way.**

The **Ukkonen algorithm** can achieve this suffix tree construction in  $O(n)$  time and space complexity! This algorithm is beyond the scope of your syllabus. But here is a youtube video explaining the algorithm for those interested:

 [Ukkonen's Algorithm](#)

Task 3: Construct a suffix trie using the standard trie as an abstraction. Modify the underlying standard trie data-structure so that each node can store the value of the number of strings that can be derived from it in an integer `stringCount` (for example, if the words `e-n1-d` and `e-n1-d-i-n2-g` have been inserted, then `stringCount` of node corresponding to the letter `e` should be 2, of node corresponding to the letter `n1` should be 2, of node corresponding to `d` should be 2, and of nodes corresponding to `i`, `n2`, `g` should be one each).

Home exercise 3: Write an algorithm using the suffix trie constructed above, to construct a suffix trie for a document `d` (`names1000.csv`) in  $O(n^2)$ , and search for the number of occurrence of the substring `s` (for example, “`mar`” - case-insensitive) in the document in  $O(n)$ . You can confirm your answer by looking for the number of occurrences of the substring using Ctrl-F on `vscode`.

## Huffman Encoding

Huffman encoding is a lossless data compression technique, whose essence lies in that “commonly occurring characters should be encoded with smaller number of bits than characters appearing less frequently”.

We will need to use the min-heap data-structure that you have designed in the previous labs for implementing huffman encoding, with some modifications. The output will be the construction of a binary tree, containing the encoding information, according to which we would encode each character in bits based on their frequency.

### Steps to build Huffman Tree:

- Define binary tree node such that it has a field that can store a character
- Define a struct `Element` such that it stores an integer corresponding to frequency of the item it is pointing to, along with a pointer to a binary node.
- Define a max heap that store items of struct `Element`, which compares `Elements` based on their frequency field.

Algorithm for building the tree:

1. For each character in the input file `in_file`, calculate the frequency with which the characters appear in it. Create binary tree nodes for each character, and a struct `Element` containing a pointer to that binary tree node along with the frequency of the character in the binary tree-node. Add all these struct `Elements` into the min-heap.
2. Extract two `Elements` with the minimum frequencies from the min-heap, and remove them from the min-heap. Create a new binary tree node, making the left child point to

the binary tree-node from the first extracted Element. Similarly make the right child point to the binary tree-node from the second extracted Element. Create a new struct Element containing a pointer to this binary tree-node, and whose frequency field is the sum of the frequency fields of the previously extracted Elements.

3. Add this new Element into the min-heap.
4. Repeat steps 2 and 3 until there is only one Element in the min-heap. The tree node of this last Element will be the root of the Huffman Encoding Tree.

#### Steps for using the tree for getting the encoding scheme:

1. Create an integer array *“encoding”* of size ALPHABET\_SIZE = 128 (you can modify this if you wish to have a different mapping of characters to indices). For convenience, let this be declared globally
2. Declare a function void recursiveTraverse that:
  - a. takes as input an integer binary\_encoding, and a binary tree node tree\_node
  - b. if tree\_node == NULL, return;
  - c. if the tree\_node is not a leaf node, then:
    - i. call recursiveTraverse with parameters (binary\_encoding\*2, tree\_node->left)
    - ii. call recursiveTraverse with parameters (binary\_encoding\*2 + 1, tree\_node->right)
  - d. assign encoding[(int)tree\_node->character]  $\leftarrow$  binary\_encoding
3. call recursiveTraverse(0, root of Huffman Encoding Tree)

#### Steps for using the encoding scheme:

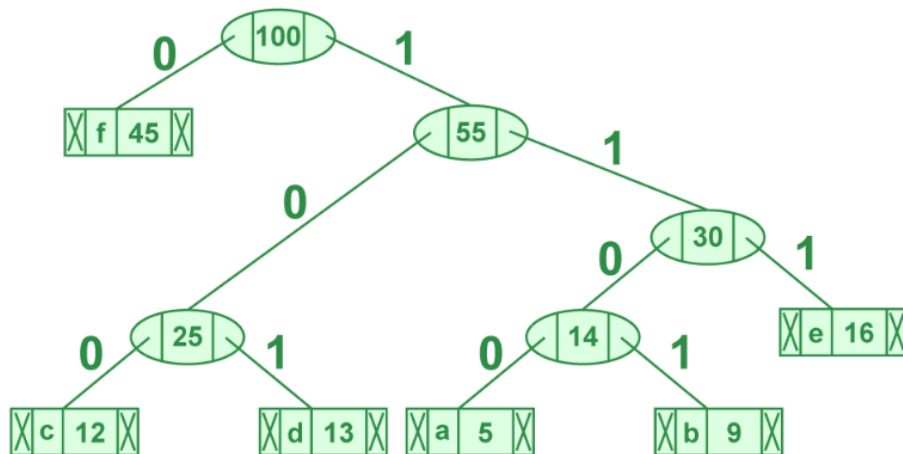
1. Create and open an output file data\_compressed in **BINARY WRITE MODE**<sup>2</sup>, that will contain the compressed form of in\_file.
2. For each character c in the input file in\_file, get the value of encoding[(int)c], convert it into its bit representation, remove the trailing zeroes from it, and write **BITWISE** the bit representation in data\_compressed

#### Steps for decoding the compressed file:

1. Initialise a tree\_node pointer pointing to the root of the Huffman Encoding Tree. Read from the data\_compressed file bit-by-bit.
2. Upon reading 0, tree\_node  $\leftarrow$  tree\_node->left. Upon reading 1, tree\_node  $\leftarrow$  tree\_node->right.
3. At any point of time, if the tree\_node is a leaf node, or if the current bit that was read is a 0 and the node does not have a left child (similarly for 1 and right child), then print the character stored in tree\_node, reinitialise tree\_node to root of the Huffman Encoding Tree, and for the current bit (before reading the next bit) move to the left or right child depending on the bit value. Then continue steps 2 and 3.
4. If all the bits are read, then print the character stored in tree\_node currently. You have finished decoding the encoded file.

---

<sup>2</sup> You will have to look up how to write into a file in BINARY WRITE MODE. You should not write the characters '1' and '0' instead, because that defeats the purpose of the compression



*Steps to print code from HuffmanTree*

The codes are as follows:

character	code-word
f	0
c	100
d	101
a	1100
b	1101
e	111

*Huffman Encoding Tree and encoding array for the following initial frequencies of letters:*

character	Frequency
a	5
b	9
c	12
d	13
e	16
f	45

**Task 4:** Implement the code for Huffman Encoding using the above given pseudo code. Test out your code on names1000.csv attached alongside the labsheet. Confirm that the output on the console after the decoding of the data\_compressed file created is the same as the initial input.

**Contributors:** @SIDLAD (SIDDHARTH S SHAH)