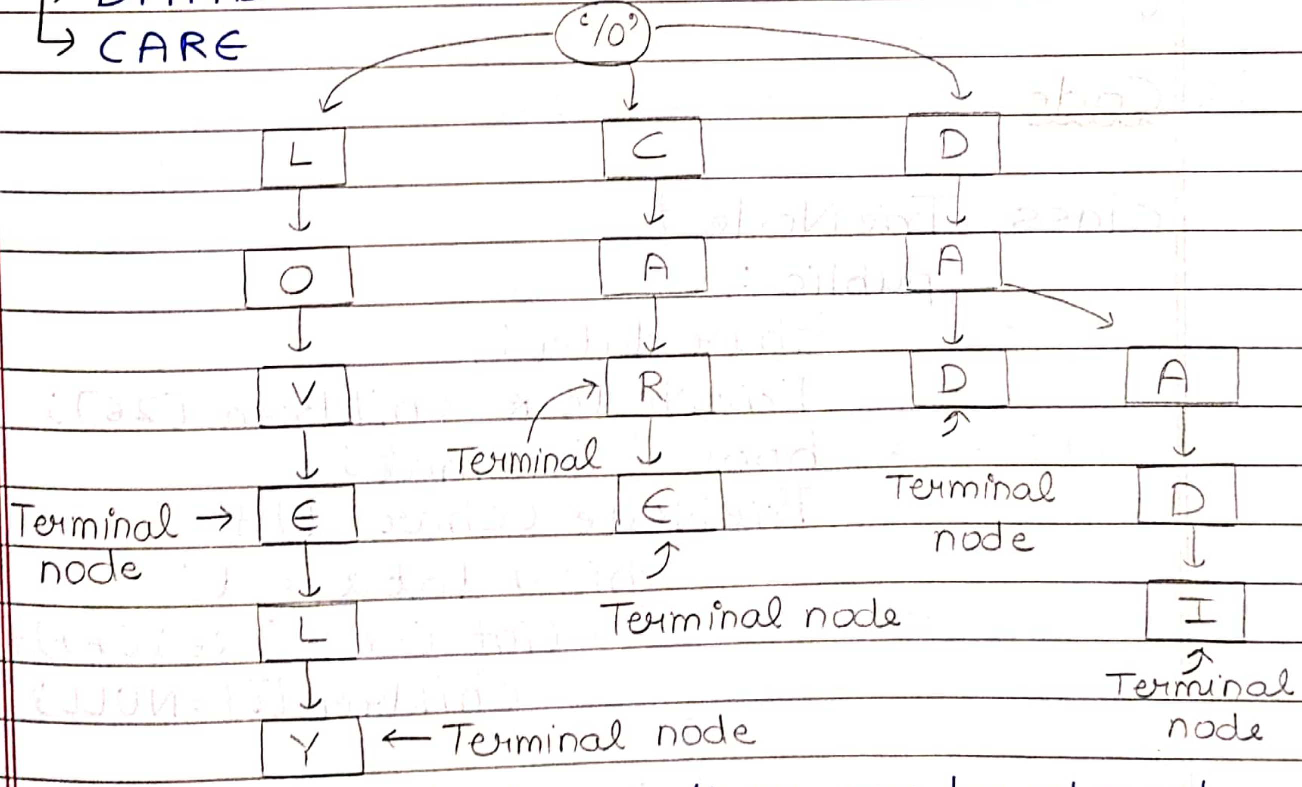
31/05/2023

Tries

It is a type of data structure. Its highly used in pattern searching as its time complexity will be proportional to the length of pattern.

Strings
- LOVE
- LOVELY
- DAD
- CAR
- DAADI
- CARE



Here in case of strings, there can be atmost 26 children.

Note → Auto suggestions in google search can be created with the help of tries.

* If node is present, then go to that node.
* If node is not present, then create it.
* As we traverse a string, then at the end we have to mark that node as terminal
* Deletion in trie

Suppose that we want to delete CAR, then simply mark R as the non-terminal node.

Insertion in tries
1) Node present ⇒ Go to that node
2) Node absent ⇒ Create that node and then go to that node.

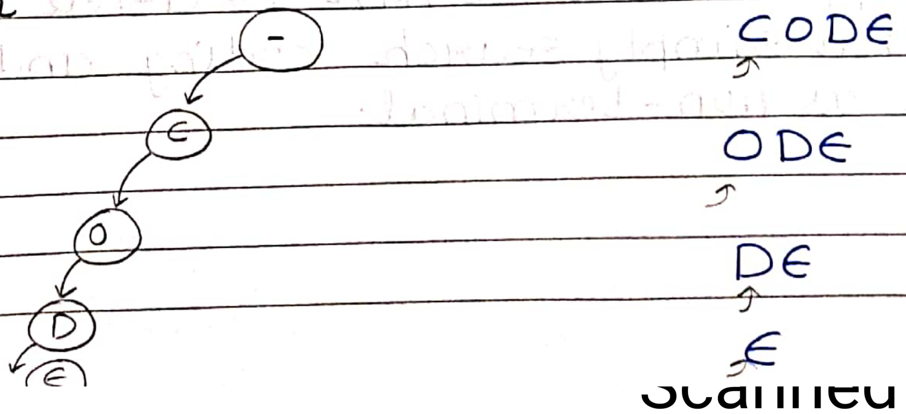
Code

```
class TrieNode {
        public:
            char data;
            TrieNode* children[26];
            bool isTerminal;
            TrieNode (char d) {
                this->data = d;
                for (int i=0; i<26; i++){
                    children[i]=NULL;
                }
                this->isTerminal = false;
            }
};

void insertWord (TrieNode* root, string word){
```

```
// Base case → No character left and hence
mark terminal
if (word.length () ==0){
        root → isTerminal =false;
        return;
}
// Fetch the index of character
 char ch = word [0];
 int index = ch - 'A'; //Play with capital
                               characters
 TrieNode * child;
 // Already present character
 if (root → children [index] != NULL) {
          //move to that node
          child = root → children [index];
 }
 else { //not present
   // Create node
   child = new TrieNode (ch);
   //move to that node
   root → children [index] = child;
 }
 // Recursion will handle
 insertWord (root, word.substr(1));
}
```

Dry run



Now no characters left and hence mark node E as terminal.

Time complexity = O(L)
↳ word.length()

Searching in tries

```
bool searchWord (TrieNode * root, string w){
    // Base case
    if(w.length() == 0) {        If terminal,
                                 then found
            return root → isTerminal;
    }
    // Fetch index
    char ch = word[0];
    int index = ch - 'a';
    TrieNode * child;
    // Children present & hence move
    if (root → children [index] != NULL)
            child = root → children [index];
    else
            return false;
    //recursive call
    return searchWord (child, word.substr(1));
}
                    ↳ (Important)
```

Deletion in tries
Suppose that we have to delete coding, then simply search coding and mark g as non-terminal.

Note → Time complexity of searching is O(L) where L is the length of word.