



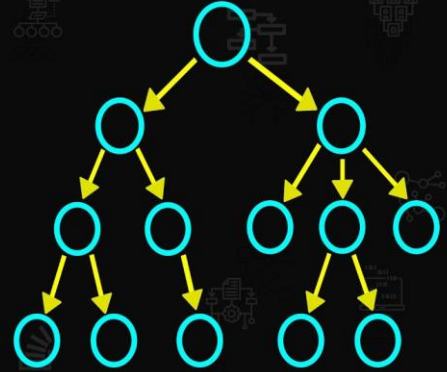
TRIE DATA STRUCTURE

BY ADRIAN SICAJU RUIZ, DANIEL MAIELLO, KENNY SULLIVAN

WHAT ARE TRIES?

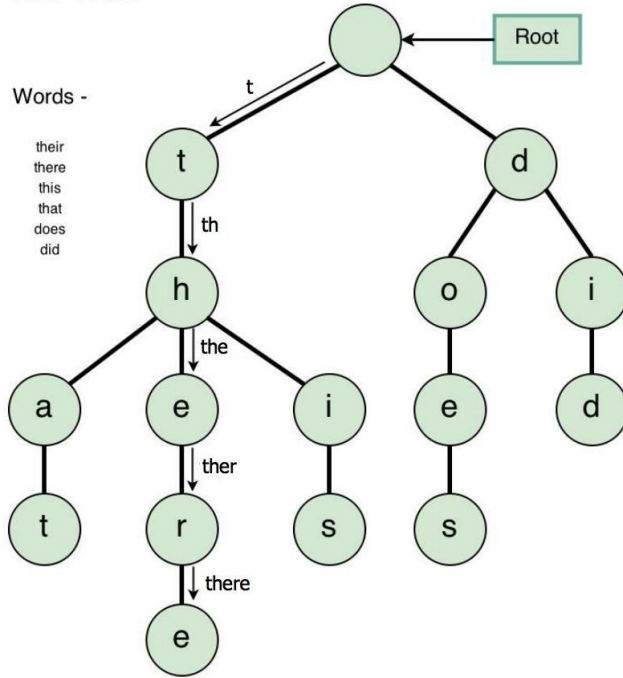
- Tries derive from the term Retrieval
- The idea was described in 1960 by Edward Fredkin
- A Trie is a kind of search digital or prefix tree used to locate specific keys within a set
- The tree is a hierarchical structure of nodes
- Often used to store characters
- Path from root, the node shares a word or a part of a word

WHAT
IS A
TREE?



 SIMPLE SNIPPETS

Trie Tree -



THE TRIE PROCESS

[illegible]

CAR
CARD
CARDS
COT
COTS
TRIE
TRIED
TRIES
TRY

CARD

CARDS

COT

COTS

TRIE

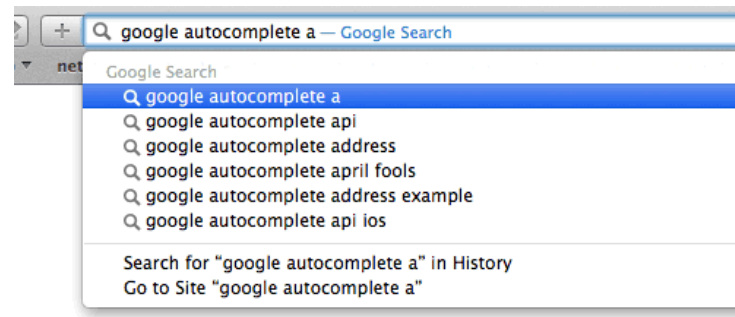
TRIED

TRIES

TRY

TRIE APPLICATIONS

- Autocomplete
 - Web Browsers
 - Search Engines
 - Source Code



It was a dark and stromy night

CONTINUED...

- Spell Check/Autocorrect
 - Word processors
 - Email

ALGORITHM

- Functions we implemented:
 - Insert
 - Read File
 - Search
 - AutoComplete

INSERT

- Insert { Ex: "People"
 - Start at blank root node
 - Check to see if child node with key "P" exists
 - If child does not exist, create a new node with key
 - If child does exist (or has been created) move to that node
 - Repeat until end of word is reached
 - Set final key node to a Leaf Node and set the word count to 1
 - If a repeat word is entered, leaf node word count increment by 1 to represent repetitions
- }

```
57 // Function to insert a word into the Trie
58 void Trie::insert(std::string data, Node* root) {
59
60     // Starting from the root node from constructor
61     Node* current = root;
62     //std::cout<<(*curr)<<std::endl;
63     // Loop to check each letter in the word to see if a child node of the key already exists
64     std::cout<< "Adding " << data << " to the trie. . . "<<std::endl;
65     for (unsigned int i = 0; i < data.length(); i++) {
66
67
68         // If a child node of that key does not already exist
69
70         if (current->character[data[i]] == nullptr) {
71
72             // Create a new node of that data
73             current->character[data[i]] = new Node(data[i]);
74         }
75
76         // If a child node of that key does exist, or has been created then,
77         // go to the next node.
78         current = current->character[data[i]];
79     }
80
81     // Once the end of the key word being inserted has been reached,
82     // mark the current node as a leaf
83
84     current->isLeaf = true;
85     current->counter += 1;
86 }
```


READ FILE

- ReadFile {
 - Takes in a text file
 - Breaks file into individual lines
 - Breaks each individual line into words separated by whitespace
 - Calls Insert on each individual word
- }

```
25 // Function to read input file and store words in tree
26 void Trie::readFile(std::string file_name){
27
28     //Create a stream object of File_Name
29     std::ifstream file(file_name);
30
31     // Create string object of line
32     std::string line;
33
34     // Loop to look at each line in the file
35     while(std::getline(file, line)){
36
37         // Create a stream object that contains each line in the file
38         std::istringstream stream(line);
39
40         // Create a string object for each word
41         std::string key;
42
43         // Loop to insert each word into the trie
44         while(stream >> key){
45
46             insert(key, this->root);
47
48         }
49         // After each word, continue to next word till line is done
50     }
51     //After each line, continue to next line till file is done
52 }
53 }
```

SEARCH

- Search { Ex: Cow
 - Start at blank root node
 - Check to see if node has child with key "C"
 - If child exist, repeat until end of word is reached
 - If node does not have child with the next key, return false
 - If end of word is reached and last key node is a leaf node, return true
 - If end of word is reached and last key node is not a leaf node, return false
- }

```
91 // Iterative function to search a key in a Trie. It returns true
92 // if the key is found in the Trie; otherwise, it returns false
93 bool Trie::search(std::string data, Node* root)
94 {
95     // return false if Trie is empty
96     if (root == nullptr) {
97         std::cout<<"The trie is empty"<<std::endl;
98         return false;
99     }
100
101     Node* current = root;
102
103     //std::cout<<(*curr)<<std::endl;
104
105     std::cout<<"Searching for "<< data<< " within trie"<<std::endl;
106
107     for (unsigned int i = 0; i < data.length(); i++)
108     {
109         //Traverse to the next node
110         current = current->character[data[i]];
111         // if the string is invalid, reaching the end of a path in the Trie
112         if (current == nullptr) {
113             std::cout<<"Word was not found in trie"<<std::endl;
114             return false;
115         }
116     }
117
118     // return true if the current node is a leaf and the
119     // end of the string is reached
120     std::cout << data<< ": "<<current->counter << std::endl;
121     return current->isLeaf;
122 }
123 }
```

```

141 int Trie::auto_comp(std::string data, Node* root){
142     std::cout << "Looking for prefix 'un'" << std::endl;
143     Node* current = root;
144     //For loop to traverse through the string, data, passed into to check if each character is found in the trie
145     for(unsigned int height = 0; height < data.length(); height++){
146         //Used to indicate the "level" at which point the character is found in the trie
147         int idx = data[height];
148         //Conditional to check if the character at index level exists if not it's not in the trie
149         if(!current->character[idx]){
150             std::cout << "No words found for prefix" << std::endl;
151             return 0;
152         }
153         //sets current to the character at index level
154         current = current->character[idx];
155     }
156     //boolean used to check if the node is the end of the word
157     bool end_of_word = (current->isLeaf == true);
158     //boolean to check if the current node has a child
159     bool child = haveChildren(current);
160     //Conditional to check if the end of the prefix is the end of the word
161     //meaning no more words are found to match it
162     if(end_of_word && !child){
163         std::cout << data << std::endl;
164         std::cout << "No more words matching prefix" << std::endl;
165         return -1;
166     }
167     //Conditional to check if there is a child node and call recursive function
168     if(child){
169         std::string prefix = data;
170         auto_comp_recur(current, prefix);
171     }
172     return 1;
173 }
174
175 void Trie::auto_comp_recur(Node* root, std::string prefix){
176     //base case if root character is leaf node
177     if (root -> isLeaf){
178         std::cout << "Found: " << prefix << std::endl;
179     }
180     //base case if the root has no more children
181     if(!haveChildren(root)){
182         return;
183     }
184     //Loop to iterate through the trie and pop remaining characters for
185     for(int i = 0; i < CHAR_SIZE; i++){
186         //Conditional if the character exists then push it onto string
187         if(root->character[i]){
188             prefix.push_back(i);
189
190             auto_comp_recur(root->character[i], prefix);
191             //Remove character off of prefix for next autocomplete word
192             prefix.pop_back();
193         }
194     }
195 }
196
197 }
198
199 }
200
201 }
202
203 }
204
205 }
206
207 }
208
209 }

```

AUTOCOMPLETE

- Auto-Complete { Ex: Un
- Start by searching for the Prefix in the trie
- If the prefix does not exist, return nothing
- If the prefix does exist, check to see if the prefix' leaf node ("n") has children
- If not, return that word as the only existence of that prefix
- If it does have children, return all words that contain that prefix
- }



ANY QUESTIONS?