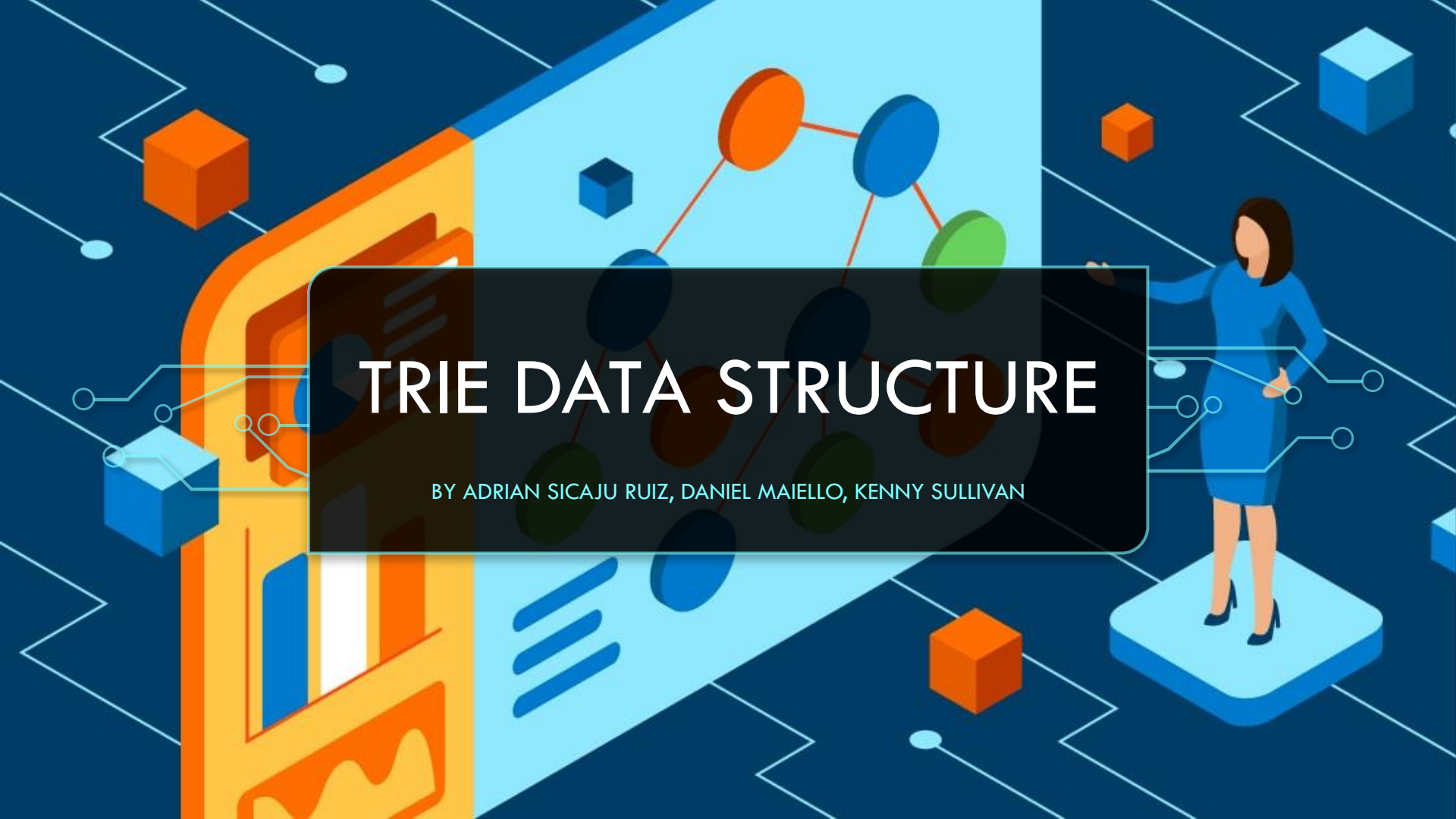
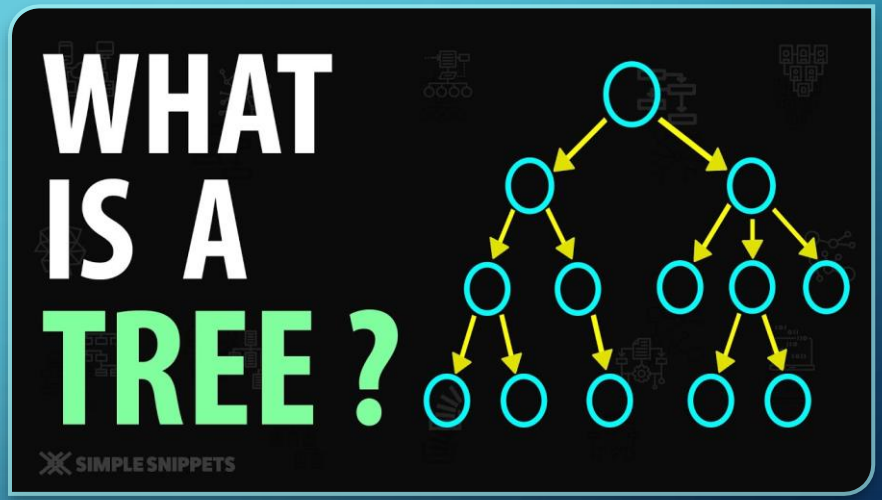


TRIE DATA STRUCTURE

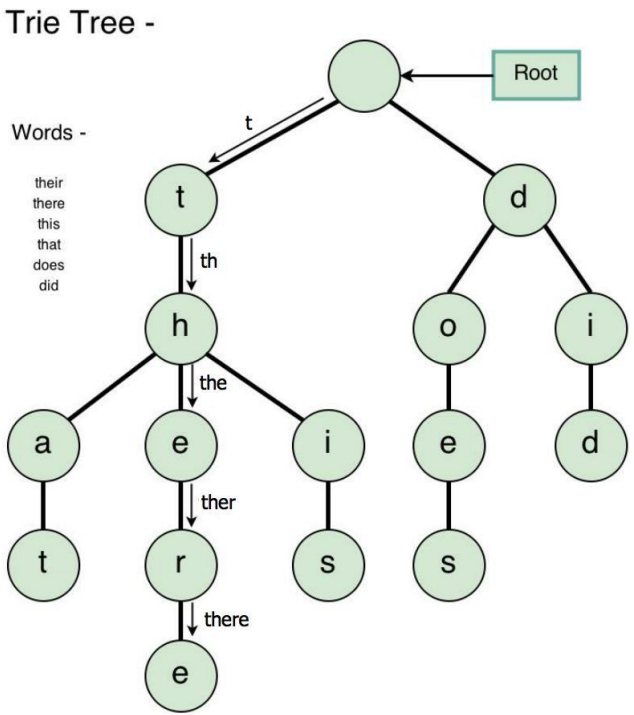
BY ADRIAN SICAJU RUIZ, DANIEL MAIELLO, KENNY SULLIVAN

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

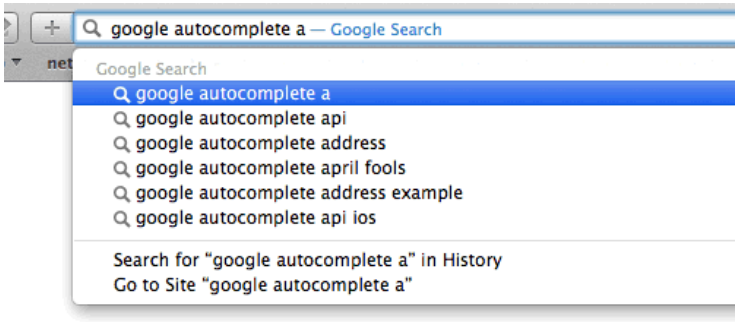


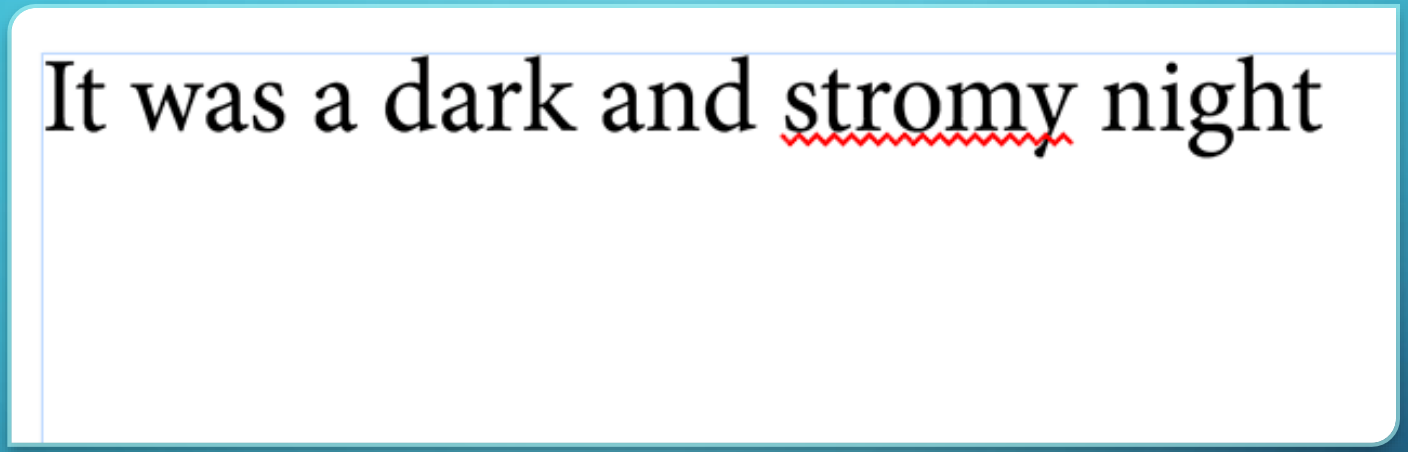
THE TRIE PROCESS



TRIE APPLICATIONS

- Autocomplete





CONTINUED…

- Spell Check/Autocorrect

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
- }

```
// Iterative function to search a key in a Trie. It returns true
// if the key is found in the Trie; otherwise, it returns false
bool Trie::search(std::string data, Node* root)
{
    // return false if Trie is empty
    if (root == nullptr) {
        std::cout<<"The trie is empty"<<std::endl;
        return false;
    }

    Node* current = root;

    //std::cout<<(*curr)<<std::endl;

    std::cout<<"Searching for "<< data<< " within trie"<<std::endl;

    for (unsigned int i = 0; i < data.length(); i++)

    {
        //Traverese to the next node
        current = current->character[data[i]];
        // if the string is invalid, reaching the end of a path in the Trie
        if (current == nullptr) {
            std::cout<<"Word was not found in trie"<<std::endl;
            return false;
        }
    }

    // return true if the current node is a leaf and the
    // end of the string is reached
    std::cout << data<< ": "<<current->counter << std::endl;
    return current->isLeaf;
}
```

```
int Trie::auto_comp(std::string data, Node* root){
    std::cout << "Looking for prefix 'un'" << std::endl;
    Node* current = root;

    //For loop to traverse through the string, data, passed into to check if each character is found in the trie
    for(unsigned int height = 0; height < data.length(); height++){
        //Used to indicate the "level" at which point the character is found in the trie
        int idx = data[height];
        //Conditional to check if the character at index level exists if not it's not in the trie
        if(!current->character[idx]){
            std::cout<< "No words found for prefix"<<std::endl;
            return 0;
        }

        //sets current to the character at index level
        current = current->character[idx];
    }

    //boolean used to check if the node is the end of the word
    bool end_of_word = (current->isLeaf == true);

    //boolean to check if the current node has a child
    bool child = haveChildren(current);

    //Conditional to check if the end of the prefix is the end of the word
    //meaning no more words are found to match it
    if(end_of_word && !child){
        std::cout<< data << std::endl;
        std::cout<< "No more words matching prefix"<< std::endl;
        return -1;
    }

    //Conditional to check if there is a child node and call recursive function
    if(child){
        std::string prefx = data;
        auto_comp_recur(current, prefx);

        return 1;
    }
```

```
void Trie::auto_comp_recur(Node* root, std::string prefx){
    //base case if root character is leaf node
    if (root ->isLeaf){
        std::cout<< "Found: " <<prefx<<std::endl;
    }

    //base case if the root has no more children
    if(!haveChildren(root)){
        return;
    }

    //Loop to iterate through the trie and pop remaining characters for
    for(int i = 0; i < CHAR_SIZE; i++){
        //Conditional if the charcater exists then push it onto string
        if(root->character[i]){

            prefx.push_back(i);

            auto_comp_recur(root->character[i], prefx);
            //Remove charcater off of prefix for next autocomplete word
            prefx.pop_back();

        }
    }
}
```

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
- }