CS-221 DSA

SEMESTER PROJECT

SAAD

2022509

IMPLEMENTING TRIE DATA STRUCTURES:

SUMMARY:

In this project, we implement the Trie data structure in C++. Trie is a tree-like data structure that stores a dynamic set or associative array where the keys are usually strings. The keys are stored at the leaf nodes. Trie data structure provides efficient ways to search and insert strings in the set.

DETAILS:

- We create a TrieNode class that represents each node in the trie. Each node contains an unordered_map called 'children' that maps a character to the TrieNode that represents the character's child. Additionally, a boolean variable 'is_end_of_word' is used to mark the end of a word in the trie.

- We create a Trie class that encapsulates the trie. The class contains a private root TrieNode and a public insert method. The insert method takes a string as input and inserts it into the trie. The method iterates through each character in the string and updates the children of the current node accordingly. If the character is the last one in the string, the is_end_of_word property of the current node is set to true.

- We implement a search method in the Trie class. The search method takes a string as input and checks if it exists in the trie. The method iterates through each character in the string and navigates to the corresponding child of the current node. If any character in the string does not exist in the trie, the method returns false. If all characters are successfully traversed, the method returns the value of the is_end_of_word property of the current node.

- We implement a words_with_prefix method in the Trie class. The words_with_prefix method takes a string prefix as input and returns a vector of all words in the trie that have the given prefix. The method iterates through each character in the prefix and navigates to the corresponding child of the current node. Then, it performs a depth-first search (DFS) to collect all words under the node. The results are stored in a vector and returned to the caller.

- Finally, in the main function, we create a Trie object and insert several words into the trie. We then call the words_with_prefix method to find all words that have a specific prefix. The results are printed to the console.

CONCLUSION:

By implementing the Trie data structure in C++, we can efficiently search and insert strings in a dynamic set. This approach is particularly useful when dealing with large datasets and when searching for strings based on prefixes is a common operation.

CODE:

```
#include <iostream>
#include <string>
#include <unordered_map>
#include <vector>
#include <stack>
using namespace std;

// TrieNode class definition
class TrieNode {
public:
 unordered_map<char, TrieNode*> children;
 bool is_end_of_word;
 TrieNode() {
    is_end_of_word = false;
 } };
// Trie class definition
class Trie {
private:
 TrieNode* root;
public:
 Trie() {
   root = new TrieNode();
 }
 void insert(string word) {
   TrieNode* node = root;
   for (char c : word) {
     if (node->children.find(c) == node->children.end()) {
       node->children[c] = new TrieNode();
     }
     node = node->children[c];
   }
```

```
    node->is_end_of_word = true;
  }
  bool search(string word) {
    TrieNode* node = root;
    for (char c : word) {
      if (node->children.find(c) == node->children.end()) {
        return false;
      }
      node = node->children[c];
    }
    return node->is_end_of_word;
  }
vector<string> words_with_prefix(string prefix) {
  vector<string> results;
  TrieNode* node = root;
  for (char c : prefix) {
     if (node->children.find(c) == node->children.end()) {
      return results;
     }
     node = node->children[c];
  }
  // Perform a depth-first search to collect all words under the node
  stack<pair<TrieNode*, string>> dfs_stack;
  dfs_stack.push({node, prefix});
  while (!dfs_stack.empty()) {
     pair<TrieNode*, string> top = dfs_stack.top();
     dfs_stack.pop();
     if (top.first->is_end_of_word) {
       results.push_back(top.second);
     }
     for (auto it : top.first->children) {
       dfs_stack.push({it.second, top.second + it.first});
     }
  }
  return results;
```

```
}
int main() {
 Trie trie;
 trie.insert("apple");
 trie.insert("app");
 trie.insert("bat");
 trie.insert("batman");
 string prefix = "app";
 vector<string> results = trie.words_with_prefix(prefix);
 for (string word : results) {
    cout << word << endl;
 }
 return 0;
}
```