AOA PROJECT

Saleem aman

SAP ID: 53526

Class Instructor

USMAN SHARIF

Riphah International University

Islamabad Pakistan

Project Report: Trie (Prefix Tree)

1. Introduction

A Trie, or Prefix Tree, is a tree-based data structure that stores strings efficiently. Each node represents a character, and paths from the root represent words or prefixes. Tries are particularly useful for operations like searching for words or prefixes, autocompletion, and spell checking.

2. Why Use a Trie?

- Efficient Operations: Tries offer fast search, insert, and prefix query operations, typically in O(L) time.
- Prefix-Based Retrieval: They allow quick retrieval of all words sharing a common prefix.
- Avoids Duplicates: Tries inherently prevent duplicate entries.

3. Core Operations in C++

Trie Node Structure:

```
struct TrieNode {
    TrieNode* children[26];
    bool isEndOfWord;

    TrieNode() {
        isEndOfWord = false;
        for (int i = 0; i < 26; ++i)
            children[i] = nullptr;
    }
} ;
```

Insert Function:

```
class Trie {
private:
    TrieNode* root;

public:
    Trie() {
        root = new TrieNode();
    }
```

```
    void insert(string word) {
        TrieNode* current = root;
        for (char ch : word) {
            int index = ch - 'a';
            if (!current->children[index])
                current->children[index] = new TrieNode();
            current = current->children[index];
        }
        c u r r e n t - > i s E n d O f W o r d   =   t r u e ;
    }
```

Search Function:

```
    bool search(string word) {
        TrieNode* current = root;
        for (char ch : word) {
            int index = ch - 'a';
            if (!current->children[index])
                return false;
            current = current->children[index];
        }
        r e t u r n   c u r r e n t - > i s E n d O f W o r d ;
    }
```

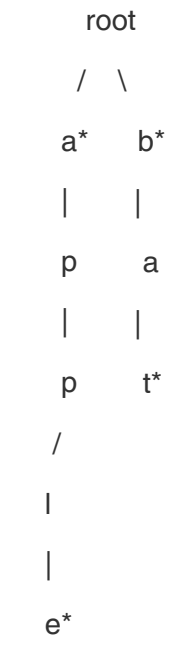
Prefix Check Function:

```
    bool startsWith(string prefix) {
        TrieNode* current = root;
        for (char ch : prefix) {
            int index = ch - 'a';
            if (!current->children[index])
                return false;
            current = current->children[index];
        }
        r e t u r n   t r u e ;
    }
} ;
```

Project Report: Trie (Prefix Tree)

4. Trie Diagram

Let's visualize the Trie after inserting the words: "apple", "app", and "bat":



* indicates end of a valid word.

5. Sample Output

```
int main() {
    Trie            trie;
    trie.insert("apple");
    trie.insert("app");
    trie.insert("bat");

 cout << trie.search("apple") << endl; // Output: 1   cout
<<  trie.search("app")  <<  endl;  //  Output:  1    cout  <<
trie.search("appl")  <<  endl;  //  Output:  0    cout  <<
trie.startsWith("ap")  <<  endl;  //  Output:  1    cout  <<
trie.startsWith("ba")  <<  endl;  //  Output:  1    cout  <<
trie.startsWith("cat") << endl; // Output: 0

    return 0;
}
```

Project Report: Trie (Prefix Tree)

6. Performance Analysis

Operation	Time Complexity	Space Complexity
Insert	O(L)	O(ALPHABET_SIZE x L)
Search	O(L)	O(1)
Prefix Check	O(L)	O(1)

L: Length of the word or prefix

ALPHABET_SIZE: Number of possible characters (e.g., 26 for lowercase English)

7. Applications of Trie

- Autocomplete Systems
- Spell Checkers
- IP Routing (Longest Prefix Match)
- DNA Sequencing
- Word Games (e.g., Boggle)

8. Conclusion

Tries are powerful data structures for efficient string storage and retrieval, especially when dealing with prefixes. While they may consume more memory than other data structures, their speed and effectiveness in search operations make them ideal for many real-world applications.