Assignment No-13

Mini Project

Problem Statement:

The problem is to implement an Auto Suggest Program using a Trie data structure. The program should allow users to enter a prefix and retrieve a list of suggested words that start with the given prefix.

Data structures, algorithms, header files used :

Data Structures:

- **Trie**: The main data structure used in this program is a Trie. It is implemented using a struct called **TrieNode**, which contains an array of child nodes representing the alphabet, and a flag **isEndOfWord** to mark the end of a word.
- **Array:** The children array in the TrieNode struct represents the child nodes of a Trie node. It is an array of TrieNode pointers, where each index corresponds to a letter of the alphabet.
- **Struct**: The TrieNode struct itself represents a node in the Trie data structure. It contains an array of child nodes, a boolean flag isEndOfWord to mark the end of a word, and other necessary fields and methods.
- **Vector**: The suggestions vector is used to store the suggestions retrieved from the Trie. It dynamically grows to accommodate the suggestions and provides a flexible container for storing and managing the results.
- **String**: The word and prefix variables are strings used to hold input words and prefixes, respectively. They are essential for performing operations such as inserting words into the Trie and searching for suggestions.
- **File**: The ifstream object is used to read words from a file. It allows reading data from external files in a structured manner.

Algorithms:

- Trie Insertion: The **insert** function implements the algorithm to insert a word into the Trie. It traverses the Trie, creating new nodes as needed, and marks the end of the inserted word by setting the **isEndOfWord** flag.
- Trie Traversal: The **traverseTrie** function recursively traverses the Trie, collecting suggestions by appending characters to the current prefix and storing words when encountering a node marked as the end of a word.
- Trie Search: The **search** function performs a prefix search in the Trie. It traverses the Trie based on the given prefix and calls the **traverseTrie** function to collect suggestions starting from the given prefix.

Header Files:

- **<iostream>**: Used for input/output operations.
- **<fstream>**: Used for file input/output operations.
- **<vector>**: Used to store the suggestions as a dynamic array.
- **<iomanip>**: Used to format the output by setting the width of the displayed strings.

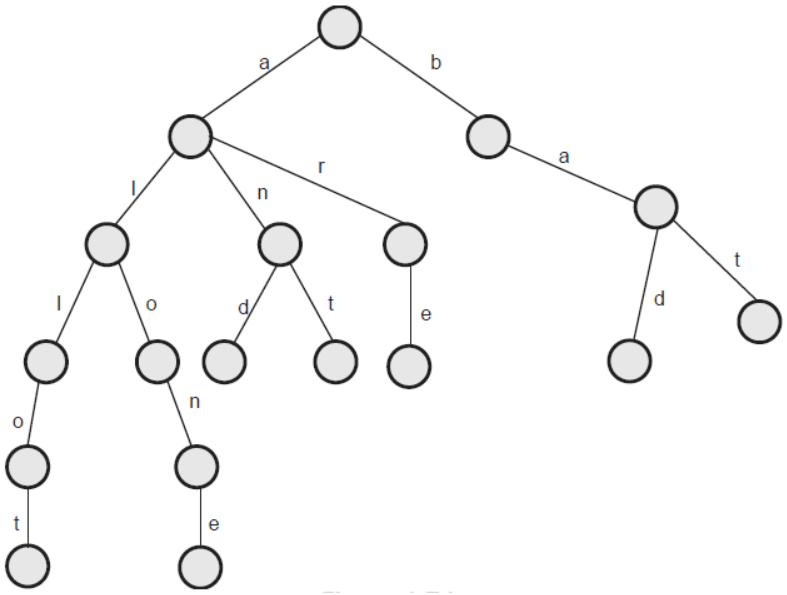
These data structures, algorithms, and header files are used in the program to implement the Auto Suggest functionality.

Theory:

The Trie data structure is an efficient way to store and retrieve strings. It is widely used in applications that involve searching and auto-completion. The Trie is a tree-like data structure where each node represents a character. By traversing the Trie, we can efficiently search for words that match a given prefix.

Trie Tree

Definition : A trie (derived from retrieval) is a multiway tree data structure used for storing strings over an alphabet. It is used to store a large amount of strings. The pattern matching can be done efficiently using tries. Following figure represents the trie.



The trie shows words like allot, alone, ant, and, are, bat, bad. The idea is that all strings sharing common prefix should come from a common node. The tries are used in spell checking programs.

• Preprocessing pattern improves the performance of pattern matching algorithm.
But if a text is very large then it is better to preprocess text instead of pattern for efficient search.

• A trie is a data structure that supports pattern matching queries in time proportional to the pattern size.

Advantages of tries

1. In tries the keys are searched using common prefixes. Hence it is faster. The lookup of keys depends upon the height in case of binary search tree.
2. Tries take less space when they contain a large number of short strings. As nodes are shared between the keys.
3. Tries help with longest prefix matching, when we want to find the key.

Comparison of tries with hash table

1. Looking up data in a trie is faster in worst case as compared to imperfect hash table.
2. There are no collisions of different keys in a trie.
3. In trie if single key is associated with more than one value then it resembles buckets in hash table.
4. There is no hash function in trie.
5. Sometimes data retrieval from tries is very much slower than hashing.
6. Representation of keys a string is complex. For example, representing floating point numbers using strings is really complicated in tries.
7. Tries always take more space than hash tables.
8. Tries are not available in programming tool it. Hence implementation of tries has to be done from scratch.

Applications of tries

1. Tries has an ability to insert, delete or search for the entries. Hence they are used in building dictionaries such as entries for telephone numbers, English words.

2. Tries are also used in spell-checking softwares.

Functions :

1. **createNode()**:
    - This function creates a new TrieNode dynamically using the **new** keyword.
    - It initializes the **isEndOfWord** flag to false and sets all child pointers in the **children** array to **nullptr**.
    - Finally, it returns the newly created TrieNode.
2. **charToIndex(char c)**:
    - This function takes a lowercase character **c** as input.
    - It converts the character to its corresponding index in the **children** array.
    - The conversion is done by subtracting the ASCII value of 'a' from the ASCII value of the character.
    - The result is the index in the range 0 to 25 (representing the 26 lowercase English alphabets).

- The function returns the calculated index.

3. **insert(TrieNode* root, const string& word)**:
    - This function is used to insert a word into the Trie.
    - It takes the root node of the Trie and the word to be inserted as input.
    - It starts from the root node and iterates over each character in the word.
    - For each character, it calculates the index using the **charToIndex** function.
    - If the child node at the calculated index is **nullptr**, a new node is created using **createNode()**.
    - The function then moves to the child node and continues the process for the next character.
    - After iterating over all characters, the **isEndOfWord** flag of the last node is set to true, indicating the end of the inserted word.
4. **traverseTrie(TrieNode* node, const string& prefix, vector<string>& suggestions)**:
    - This function is a recursive helper function used by the **search** function.
    - It takes a TrieNode pointer, a prefix string, and a vector of suggestions as input.
    - If the current node is marked as the end of a word, it adds the prefix to the **suggestions** vector.
    - Then, it iterates over the children nodes of the current node.
    - For each non-null child node, it calculates the corresponding character and appends it to the prefix.
    - It recursively calls itself with the child node, updated prefix, and suggestions vector.
    - This process continues until all nodes in the Trie have been traversed.
5. **search(TrieNode* root, const string& prefix, vector<string>& suggestions)**:
    - This function is used to search for words with a given prefix in the Trie.
    - It takes the root node of the Trie, the prefix to search, and a vector to store the suggestions as input.
    - It starts from the root node and iterates over each character in the prefix.
    - For each character, it calculates the index using the **charToIndex** function.
    - If the child node at the calculated index is **nullptr**, it means no words exist with the given prefix, and the function returns.

- If the prefix is found in the Trie, it calls the **traverseTrie** function with the current node, prefix, and suggestions vector.

6. **main()**:
    - The main function is the entry point of the program.
    - It creates the root node of the Trie using **createNode()**.
    - It reads words from a file, inserts them into the Trie using the **insert** function.
    - Then, it enters a loop that allows the user to enter a prefix or 'q' to quit.
    - For each prefix entered, it calls the **search** function to retrieve suggestions.
    - If suggestions are found, it displays them to the user.
    - Finally, it prints a goodbye message and terminates the program.

These functions work together to implement the Auto Suggest Program using a Trie data structure. They handle Trie node creation, insertion of words, searching for suggestions, and traversing the Trie to collect suggestions with a given prefix.

Applications:

The Auto Suggest Program using a Trie data structure has various applications, including:

- Auto-complete functionality in text editors and search engines.
- Spell checkers and word suggestion features in word processors and messaging applications.
- Keyword suggestions in programming IDEs and code editors.
- Address or location suggestions in maps and navigation applications.

Conclusion :

We have successfully implemented the Auto Suggest program using the Trie data structure. Through this mini project, we have gained practical knowledge and experience in working with Tries and their application in providing word suggestions based on prefixes. We have learned how to insert words into a Trie, search for matching prefixes, and traverse the Trie to collect suggestions. Additionally, we have developed skills in file handling, user input handling, and basic program structure. This mini project has served as a valuable learning opportunity and has enhanced our understanding of data structures and algorithms.