**DSA Project-3**

**1. Introduction to TRIE Advanced Data Structure**

A **TRIE** (pronounced as "try") is a tree-based data structure used for efficient retrieval of keys in datasets such as dictionaries, search engines, and autocomplete systems. Unlike other search trees, TRIE nodes do not store keys directly; instead, they store references to characters in a word, ensuring efficient storage and lookup times.

**Key Features of TRIE:**

* Stores characters of words hierarchically.
* Allows fast word lookup in **O(L)** time complexity, where **L** is the length of the word.
* Supports prefix-based searches efficiently.
* Commonly used in auto-suggest, spell-checkers, and IP routing.

**2. Important Operations on TRIE**

TRIE supports several fundamental operations that make it an efficient data structure for search-based applications.

**Insertion (insert(word))**

* Start from the root node.
* Traverse through the characters of the word, adding nodes if necessary.
* Mark the last node as the end of the word.
* **Time Complexity:** O(L), where **L** is the word length.

**Search (search(word))**

* Traverse through the TRIE based on the characters of the word.
* If a node does not exist at any point, return false.
* If the last node is marked as a word end, return true.
* **Time Complexity:** O(L).

**Prefix Search (startsWith(prefix))**

* Similar to the search operation but does not require the last node to be a word end.
* Used in autocomplete systems.
* **Time Complexity:** O(L).

**Deletion (delete(word))**

* Recursively remove nodes if they are not part of another word.
* **Time Complexity:** O(L).

**3. Complexity Analysis for Storing Large Amount of Words**

Comparing different data structures for storing words:

|  |  |  |  |
| --- | --- | --- | --- |
| **Data Structure** | **Insert Complexity** | **Search Complexity** | **Space Complexity** |
| **Array** | O(N) | O(N) | O(N \* L) |
| **Linked List** | O(N) | O(N) | O(N \* L) |
| **BST** | O(log N) | O(log N) | O(N \* L) |
| **TRIE** | O(L) | O(L) | O(N \* L) |

**Observations:**

* TRIE provides the fastest lookup time, making it ideal for large-scale search applications.
* While BSTs perform well, their performance may degrade in unbalanced scenarios.
* Arrays and linked lists are inefficient for large datasets due to linear search times.

**4. How to Use TRIE as a Map?**

TRIE can also be used as a **key-value map** where:

* **Keys** are stored as words in the TRIE.
* **Values** are stored in the end node of the corresponding word.
* This allows fast retrieval and update operations in **O(L)** time complexity.

**Example: TRIE as a Dictionary**

If we store:

* "apple" → 100
* "app" → 50

A search for "apple" will return 100, while "app" will return 50.

**5. Real-World Applications of TRIE**

TRIE is widely used in several real-world applications due to its efficiency.

**1. Auto-Suggest & Autocomplete**

* Used in search engines (Google, Bing) to suggest words while typing.
* Predictive text in mobile keyboards.

**2. Spell Checkers**

* Dictionary-based spell-checking applications use TRIE to detect incorrect words efficiently.

**3. IP Routing**

* TRIE is used in networking for fast prefix-based IP lookup in routers.

**4. Text Compression**

* Applications like LZW Compression use TRIE to replace repeated substrings with shorter codes.

**5. DNA Sequence Matching**

* TRIE helps in searching genome sequences quickly by matching substrings efficiently.

**Conclusion**

TRIE is a powerful data structure that excels in fast word lookups, prefix-based searching, and autocomplete functionalities. It significantly improves search efficiency compared to arrays, linked lists, and BSTs. With widespread applications in search engines, spell checkers, and networking, TRIE remains a fundamental tool in advanced computing systems.