# Project Report: **Text Prediction Mechanism using Trie**

*Faculty:*
*Dr. Rishikesh CA*
*Team:*
*Tharun Gopinath (24BCE1086)*
*Neha Balakrishnan (24BCE5521)*

## 1. PROBLEM STATEMENT

Modern search engines and messaging applications offer **auto-complete** or **text prediction** features, where the system suggests possible queries or words as the user types.
The goal of this project is to **design and implement an efficient text prediction mechanism** that returns the most relevant completions for a given input prefix.

## 2. OBJECTIVE

- Build a **back-end mechanism** that predicts user input.
- Suggestions should be **fast, relevant, and scalable**.
- Use **Data Structures & Algorithms (DSA)** concepts, focusing on **Trie (Prefix Tree)**.

## 3. WHY TRIE?

A **Trie** (prefix tree) is an efficient data structure for storing and retrieving strings based on prefixes.

- **Fast Lookup**: Prefix searches in O(P), where P = length of prefix.
- **Space Efficient**: Stores common prefixes once.
- **Scalable**: Handles large datasets (search queries, dictionary words).
- **Customizable**: Each node can store extra info (e.g., frequency, top_k suggestions).

In comparison:

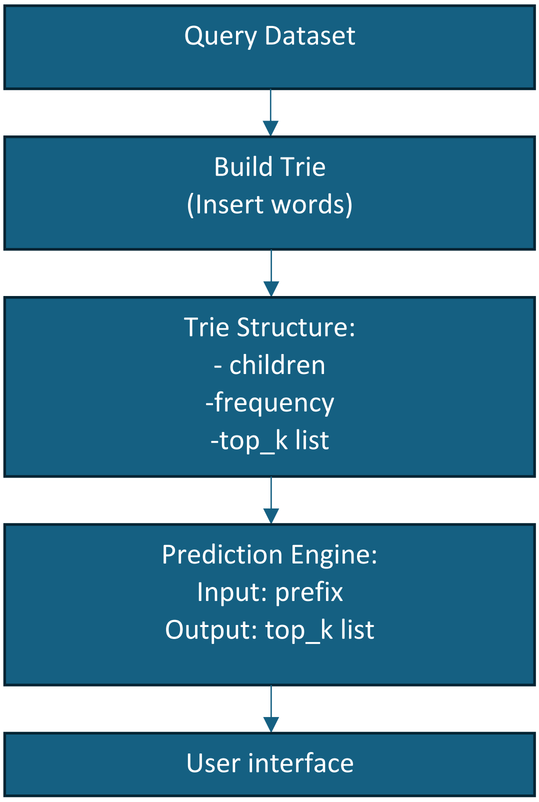
- **Binary Search** would require sorting + scanning. Good for small datasets, but slower for large ones.
- **HashMap** works for exact matches, but not for prefix-based search.

## 4. HIGH-LEVEL FLOW

1. **Data Collection**
   Collect query dataset (e.g., search logs, common dictionary words, phrases).
2. **Preprocessing**
   - Normalize text (lowercasing, removing extra spaces).
   - Store frequency/weight of each query.
3. **Trie Construction**
   - Insert each query into a Trie.

    - At each node, maintain a **list of top_k (e.g., top 5) completions** sorted by frequency.
4. **Prediction Phase (User Typing)**
    - User types prefix `"iph"`.
    - Traverse Trie character by character.
    - At the last node, directly return its `top_k` list.
5. **Cache Layer (Optional)**
    - If user repeats the same prefix, store it in cache for O(1) retrieval.

## 5. SYSTEM ARCHITECTURE (BACKEND)




## 6. DATA STRUCTURE DESIGN

Trie Node Structure:

```
class TrieNode:

    def __init__(self):

        self.children = {}        # dict of char → TrieNode

        self.is_end = False       # marks end of word

        self.frequency = 0        # frequency of word if end node

        self.top_k = []           # list of (word, freq) suggestions
```

# 7. ALGORITHM

## 7.1 INSERT WORD INTO TRIE

1. Start from root.
2. For each character in word:
   - If char not in children → create new node.
   - Move to that child.
   - Update `top_k` at this node with current word (maintain sorted list by frequency).
3. At end node → mark `is_end = True` and store frequency.

## 7.2 GET SUGGESTIONS

1. Start from root.
2. For each char in prefix:
   - If char not in children → return empty list.
   - Move to that child.
3. At last node → return its `top_k`.

# 8. PSEUDOCODE

## INSERT:

```
function insertWord(root, word, freq):

    node = root

    for char in word:

        if char not in node.children:

            node.children[char] = new TrieNode()

        node = node.children[char]

        updateTopK(node.top_k, word, freq)  # maintain top k suggestions

    node.is_end = True

    node.frequency = freq
```

## SEARCH SUGGESTION:

```
function getSuggestions(root, prefix, k):
```

```
node = root
for char in prefix:
    if char not in node.children:
        return []
    node = node.children[char]
return node.top_k[0:k]
```

## 9. EXAMPLE WALKTHROUGH

### DATASET:

```
"iphone 16 release date" (1200)
"iphone 15 price in india" (900)
"ipad pro 2025" (600)
"ipl 2025 schedule" (1500)
"ipl auction date" (1100)
```

### USER TYPES "IPH":

- Traverse: root → i → p → h.
- Node `"iph"` has `top_k = ["iphone 16 release date", "iphone 15 price in india"]`.
- Suggestions returned instantly.

## 10. COMPLEXITY ANALYSIS

- **Insertion**: $O(N \times L)$, where $N$ = number of words, $L$ = avg length of word.
- **Search (prediction)**: $O(P)$, where $P$ = prefix length.
- **Space Complexity**: $O(N \times L)$ in worst case, but reduced with shared prefixes.

## 11. EXTENSIONS / FUTURE WORK

- **NLP Integration**: Add language models for context-aware predictions.
- **Ranking Improvements**: Use frequency + recency + personalization.
- **Caching**: Frequently used prefixes can be cached for O(1) lookup.
- **Backend Scaling**: Store Trie in memory + persist in database (e.g., Redis).

## 12. CONCLUSION

This project demonstrates a **DSA-based implementation of text prediction** using Trie.

- Achieves **fast prefix-based lookups**.
- Demonstrates **real-world application of algorithms** in search engines & text editors.
- Can be extended with **NLP techniques** for advanced features.

## 13. COMPUTATIONAL FLOW OF SEARCH (DIAGRAM)