# 1. Objective of Task

The main objective of this task is to build an Autocomplete System that suggests the most frequently occurring words based on a given prefix. This is achieved by reading and preprocessing data from a CSV file, inserting each cleaned word into an AVL Tree (self-balancing), maintaining word frequencies, and querying the tree using a prefix to retrieve the top-k most frequent suggestions. A min-heap (priority queue) is used to efficiently track and return these suggestions.

# 2. Code Snippets with their Explanations

## a. Word Cleaning and Insertion from CSV

String cleaned = col.toLowerCase().replaceAll("[^a-z0-9 ]", " ");  
String[] words = cleaned.split("\\s+");  
  
for (String word : words) {  
 if (!word.isEmpty()) {  
 word = word.trim();  
 if (word.endsWith("ca") || word.endsWith("â„¢")) {  
 word = word.substring(0, word.length() - 2);  
 }  
 if (!word.isEmpty()) {  
 tree.insert(word);  
 }  
 }  
}

This block reads strings from the CSV, cleans them, and inserts words into the AVL Tree. It removes noise, converts text to lowercase, trims unwanted suffixes, and avoids inserting empty or invalid strings. Cleaned words are inserted into the AVL Tree for later retrieval.

## b. AVL Tree Insertion with Self-Balancing

AVLNode insert(AVLNode node, String word) {  
 if (node == null)  
 return new AVLNode(word);  
  
 int cmp = word.compareTo(node.word);  
 if (cmp < 0) node.left = insert(node.left, word);  
 else if (cmp > 0) node.right = insert(node.right, word);  
 else {  
 node.frequency++;  
 return node;  
 }  
  
 node.height = 1 + Math.max(height(node.left), height(node.right));  
 int balance = getBalance(node);  
  
 if (balance > 1 && word.compareTo(node.left.word) < 0)  
 return rotateRight(node);  
 if (balance < -1 && word.compareTo(node.right.word) > 0)  
 return rotateLeft(node);  
 if (balance > 1 && word.compareTo(node.left.word) > 0) {  
 node.left = rotateLeft(node.left);  
 return rotateRight(node);  
 }  
 if (balance < -1 && word.compareTo(node.right.word) < 0) {  
 node.right = rotateRight(node.right);  
 return rotateLeft(node);  
 }  
  
 return node;  
}

This recursive function inserts words into the AVL Tree and ensures balance using rotations. If the word exists, it increases its frequency. Otherwise, it adds a new node and adjusts height and balance factor. All four rotation cases (LL, RR, LR, RL) are handled for tree balancing.

## c. Collecting Prefix Matches (Recursive Traversal)

void collectWordsWithPrefix(AVLNode node, String prefix, PriorityQueue<WordFrequency> heap, int k) {  
 if (node == null) return;  
  
 if (node.word.startsWith(prefix)) {  
 heap.offer(new WordFrequency(node.word, node.frequency));  
 if (heap.size() > k) heap.poll();  
 }  
  
 if (prefix.compareTo(node.word) <= 0)  
 collectWordsWithPrefix(node.left, prefix, heap, k);  
  
 if (prefix.compareTo(node.word) >= 0)  
 collectWordsWithPrefix(node.right, prefix, heap, k);  
}

This method traverses the AVL Tree recursively to find words that start with the given prefix. It adds them to a priority queue (min-heap), ensuring the queue holds at most k highest frequency matches. It prunes traversal paths that cannot contain valid matches.

## d. Getting Final Suggestions Sorted by Frequency

List<WordFrequency> getSuggestions(String prefix, int k) {  
 PriorityQueue<WordFrequency> heap = new PriorityQueue<>();  
 collectWordsWithPrefix(root, prefix, heap, k);  
  
 List<WordFrequency> result = new ArrayList<>(heap);  
 result.sort((a, b) -> Integer.compare(b.frequency, a.frequency));  
 return result;  
}

This method collects words matching the prefix into a heap and sorts them by descending frequency. It utilizes the previously defined recursive function and ensures the results are user-friendly by returning the most frequent words first.

# 3. Conclusions (Including Complexity)

This project successfully implements an efficient autocomplete system using an AVL Tree. The AVL Tree ensures O(log n) insertions and searches, keeping the system scalable. The use of a min-heap limits memory usage while efficiently tracking top-k frequent suggestions. Traversal is optimized to avoid unnecessary paths.

Time Complexity Summary:  
- Insertion into AVL Tree: O(log n)  
- Frequency Update: O(log n)  
- Prefix Search: O(log n + m), where m is the number of matches  
- Heap insertions: O(m log k)  
- Sorting top-k: O(k log k)