Assignment 2, Task 2 Report  
**Word Completion Using AVL Trees**

**Course: COMP-8547**

**Instructor: DR. OLENA SYROTKINA**

**Student: FASEEH U REHMAN QURESHI**

**Student ID: 110188603**

**Date: 07th JUNE 2025**

# 1. Task Objective

## Word Completion

The objective of this task is to implement a word completion feature using a self-balancing search tree, specifically an AVL tree. The solution involves:

* Building a vocabulary from text data in CSV files.
* Inserting words into an AVL tree along with their frequency of occurrence.
* Allowing the user to enter a prefix and retrieving top-K word suggestions.
* Ranking and returning these suggestions based on word frequency using a min-heap.

# 2. Solution Explanation with Code Snippets

## 2.1 Reading and Cleaning Data

The CSV file is read line by line. The first line in CSV file is the header, which is skipped. Each remaining line is split by commas to extract columns. Every column is processed to extract individual words.

BufferedReader br = new BufferedReader(new FileReader("all\_laptops\_data.csv"));

String line;

br.readLine(); // Skip header

while ((line = br.readLine()) != null) {

String[] columns = line.split(",", -1);

for (String col : columns) {

String cleaned = col.toLowerCase().replaceAll("[^a-z0-9 ]", " ");

String[] words = cleaned.split("\\s+");

for (String word : words) {

if (!word.isEmpty()) {

// Clean up suffixes like "ca" or "â„¢"

if (word.endsWith("ca") || word.endsWith("â„¢")) {

word = word.substring(0, word.length() - 2);

}

tree.insert(word); // Insert into AVL Tree

}

}

}

}

### Explanation:

This code reads data from a CSV file named all\_laptops\_data.csv, skipping the header line and processing each subsequent line by splitting it into columns. For each column, it converts the text to lowercase and removes any non-alphanumeric characters except spaces using a regular expression. The cleaned text is then split into individual words. Each word is checked for specific unwanted suffixes like "ca" or "â„¢", which are removed by trimming the last two characters. Finally, all valid words are inserted into an AVL tree, which stores them along with their frequency of occurrence to support efficient word completion functionality later.

## 2.2 AVL Tree Structure and Node

Each AVL tree node holds a word, its frequency (count of appearances), and links to left and right children.

class AVLNode {

String word;

int frequency;

int height;

AVLNode left, right;

AVLNode(String word) {

this.word = word;

this.frequency = 1;

this.height = 1;

}

}

### Explanation:

This code defines an AVLNode class representing a node in an AVL tree used for word storage and frequency tracking. Each node contains a word string, an integer frequency that counts how many times the word appears (initialized to 1 for new words), and an integer height to help maintain the balance of the AVL tree. The node also has references to its left and right child nodes, which enable the binary tree structure. This setup allows the AVL tree to efficiently organize words and keep track of their occurrence counts for tasks like autocomplete suggestions.

## 2.3 AVL Tree Insertion and Balancing

Insertion in AVL trees is similar to binary search trees, but rebalancing is done after every insertion to maintain height difference (balance factor) of ±1.

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++; // Word exists; increment frequency

return node;

}

node.height = 1 + Math.max(height(node.left), height(node.right));

int balance = getBalance(node);

// Rebalance tree

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;

}

### Explanation:

This insert method adds a word into an AVL tree while maintaining its balanced structure. If the current node is null, a new AVLNode is created and returned. Otherwise, the method compares the input word with the current node’s word to determine whether to insert it in the left or right subtree. If the word already exists in the tree, its frequency is incremented. After insertion, the method updates the height of the current node and calculates its balance factor to check if the subtree is unbalanced. If it is, one of four rotations (left, right, left-right, or right-left) is applied to restore balance, based on the insertion pattern. This ensures the AVL tree maintains O(log n) time complexity for insertions.

## 2.4 Prefix-Based Word Collection

Words starting with the entered prefix are collected using an in-order traversal of the tree.

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(); // Maintain top k

}

if (prefix.compareTo(node.word) <= 0)

collectWordsWithPrefix(node.left, prefix, heap, k);

if (prefix.compareTo(node.word) >= 0)

collectWordsWithPrefix(node.right, prefix, heap, k);

}

### Explanation:

The collectWordsWithPrefix method recursively traverses an AVL tree to find and collect words that start with a given prefix. It uses a min-heap (PriorityQueue) to keep track of the top k most frequent matching words. If the current node’s word starts with the prefix, it is added to the heap; if the heap exceeds k elements, the least frequent one is removed to maintain only the top k. The traversal is optimized using lexicographic comparison: if the prefix is less than or equal to the current word, it explores the left subtree; if greater than or equal, it explores the right subtree. This approach efficiently narrows the search space and ensures the most relevant suggestions are retained.

## 2.5 Returning Top-K Suggestions

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;

}

### Explanation:

The getSuggestions method returns the top k word completions that start with a given prefix, ranked by their frequency of occurrence. It first initializes a min-heap (PriorityQueue) to collect candidate words by calling collectWordsWithPrefix, which populates the heap with matching words from the AVL tree. The heap ensures that only the k most frequent words are retained. The method then converts the heap into a list, sorts it in descending order of frequency (using a comparator), and returns this sorted list as the final set of suggestions. This ensures efficient and accurate word completion based on real usage frequency.

## 2.6 User Interaction Loop

Scanner sc = new Scanner(System.in);

System.out.println("Enter prefix: ");

String prefix = sc.nextLine().toLowerCase().trim();

List<WordFrequency> suggestions = tree.getSuggestions(prefix, 10);

**Explanation:**

This code snippet handles user interaction for the autocomplete feature. It first creates a Scanner object to read input from the console. The program prompts the user to "Enter prefix:" and captures the entered text, converting it to lowercase and trimming any leading or trailing spaces for consistency. It then calls the getSuggestions method on the AVL tree, passing in the cleaned prefix and requesting the top 10 word completions. The result is a list of suggested words based on the prefix, ranked by their frequency of occurrence in the dataset.

# 3. Screenshots and Output Explanations

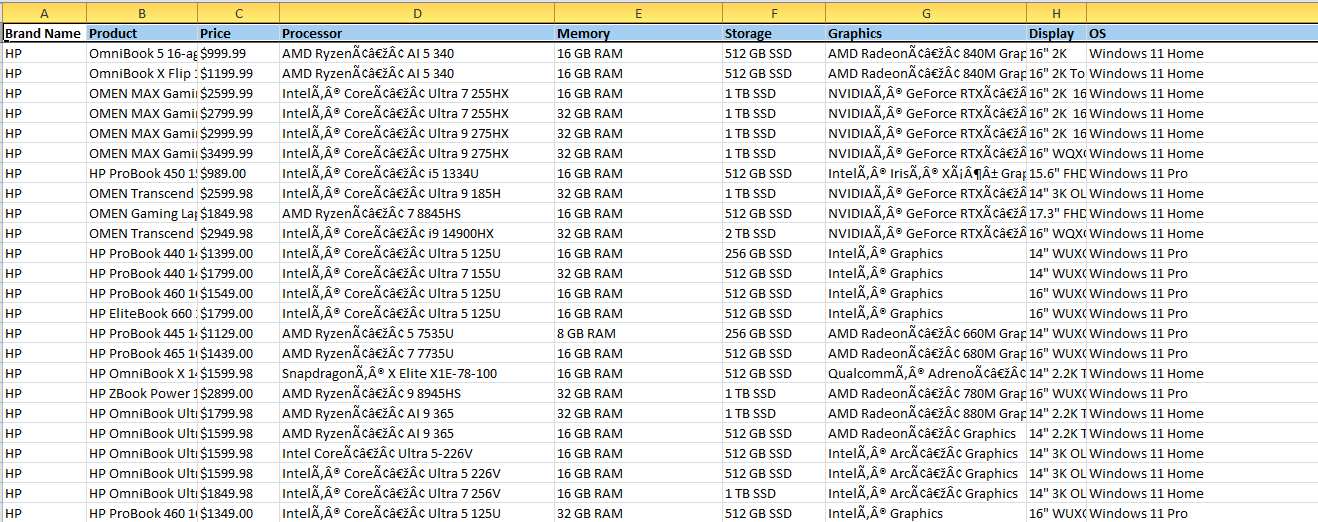


Fig 1: Data in CSV File  
*Explanation: This screenshot displays the content of the CSV file (hp\_laptops.csv) used as the input vocabulary source. Each row represents specifications and descriptions of HP laptops. During processing, all text columns are tokenized into individual words which are then cleaned and inserted into the AVL tree for building the vocabulary.*

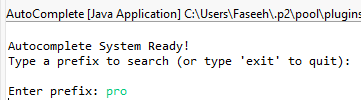


Fig 2: User Prompt for Prefix Input  
*Explanation: This screenshot shows the autocomplete system prompting the user to enter a prefix. The green text represents the user's input. The system uses this input to traverse the AVL tree and retrieve suggestions that match the entered prefix.*

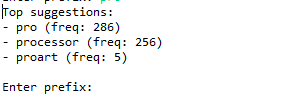


Fig 3: Autocomplete Suggestions  
*Explanation: The top suggestions based on the entered prefix and frequency ranking are displayed.*



Fig 4: No Suggestions Found  
*Explanation: This screenshot shows the system’s response when no words in the AVL tree match the user-entered prefix. This indicates that the prefix either doesn't exist in the vocabulary or has too few characters to match any stored word. The system gracefully informs the user with a "No suggestions found" message.*