COMSATS University Islamabad, Sahiwal Campus

Semester Project Report

For

AVL Tree Based Dictionary System with Prefix Search

By

Ujala Safdar **CIIT/FA24-BCS-171/SWL**

Masoomah Shabbir **CIIT/FA24-BCS-125/SWL**

Submitted By:

Miss Aleema Imran

Bachelor of Science in Computer Science (2024-2028)

Title

AVL Tree Based Dictionary System with Prefix Search

1 Brief Overview of the Project

A dictionary is one of the most commonly used applications in computer systems, requiring fast and efficient search operations. This project focuses on implementing a Dictionary System using an AVL Tree**,** which is a self-balancing Binary Search Tree (BST). The AVL Tree ensures that the height of the tree remains balanced after each insertion, thereby guaranteeing efficient search and insertion operations.

In this system, each dictionary entry consists of a **word** and its corresponding **meaning**, stored as a node in the AVL Tree. Words are arranged in lexicographical (alphabetical) order**,** allowing fast retrieval. In addition to exact word searching, the system also supports prefix-based searching**,** where users can enter a partial word and retrieve all words starting with that prefix. This feature resembles real-world dictionary and autocomplete systems.

The project is implemented in **C++** using a menu-driven console interface**, m**aking it easy to use and understand. The system demonstrates the practical application of advanced data structures and highlights the performance benefits of self-balancing trees over traditional linear data structures.

2 Problem Statement

Traditional dictionary implementations often rely on linear data structures such as **arrays** or **linked lists**. While these structures are simple to implement, they become inefficient as the number of stored words increases. Searching for a word in a linear structure may require traversing the entire list, resulting in a time complexity of O(n)**.**

Moreover, maintaining sorted order in linear structures requires additional overhead, and implementing advanced features such as prefix-based searching becomes complex and time-consuming. These limitations result in poor performance, lack of scalability, and inefficient memory utilization.

To overcome these issues, there is a need for an efficient data structure that maintains sorted order automatically and provides fast insertion and search operations. The **AVL Tree**, with its self-balancing property, offers an optimal solution by ensuring logarithmic time complexity even for large datasets.

3 Objectives of the Project

The main objectives of this project are:

1. To design and implement an efficient dictionary system.
2. To use an AVL Tree for storing dictionary words and meanings.
3. To ensure fast insertion of new words into the dictionary.
4. To enable efficient searching of words.
5. To implement prefix-based search functionality.
6. To maintain a balanced tree structure using AVL rotations.
7. To improve performance compared to linear data structures.
8. To store dictionary data in sorted order automatically.
9. To demonstrate real-world applications of self-balancing trees.
10. To enhance understanding of AVL Trees and tree traversal techniques.

4 Data Structures Used and Justification

The primary data structure used in this project is the **AVL Tree**, which is a self-balancing Binary Search Tree. In an AVL Tree, the height difference between the left and right subtrees of any node is at most one. This balance condition ensures that the height of the tree remains minimal, leading to efficient operations.

The AVL Tree guarantees:

- **O(log n)** time complexity for insertion
- **O(log n)** time complexity for searching

Strings are used to store words and meanings, as they provide flexibility for handling textual data. The AVL Tree was selected over a simple BST because a standard BST can become skewed in worst-case scenarios, leading to degraded performance with **O(n)** complexity. AVL Trees avoid this issue by performing rotations automatically.

5 System Requirements Specification (SRS)

- ## Functional Requirements

- The system shall load a predefined set of dictionary words.
- The system shall allow users to insert new words with meanings.
- The system shall allow searching of words using full word input.
- The system shall support prefix-based searching.
- The system shall display matching words with meanings.
- The system shall provide a menu-driven console interface.
- The system shall handle invalid inputs gracefully.

6 Non-Functional Requirements

- The system shall be efficient and responsive.
- The system shall maintain AVL Tree balance automatically.
- The system shall be easy to use and understand.
- The system shall be memory efficient.
- The system shall be implemented using C++.
- The system shall be platform-independent.

Input Specifications

- **Word Input:** String
- **Meaning Input:** String
- **Prefix Input:** String
- **Menu Choice:** Integer

Output Specifications

- Display of inserted dictionary entries.
- Display of search results.
- Display of words matching a prefix.
- Success and error messages.

Assumptions and Constraints

- Duplicate words are not allowed.
- The application runs in a console environment.
- The dictionary size depends on system memory.
- Input words consist of alphabetic characters only.

Data Storage, Access, and Management

- Dictionary entries are stored as nodes in an AVL Tree.
- Each node contains a word, meaning, height, and child pointers.
- Data access is performed using tree traversal and key comparison.
- AVL rotations ensure balanced structure after insertion.

Data Structure Implementation Plan

List of Data Structures Used

- AVL Tree
- Custom Node Structure
- Strings

Application of Data Structures

- AVL Tree stores dictionary entries.
- Nodes represent words and meanings.
- In-order traversal retrieves words in sorted order.
- Prefix search is implemented using traversal and string matching.

Justification for Selecting Each Data Structure

Data Structure	Justification
AVL Tree	Ensures balanced height and fast operations
String	Efficient handling of textual data
Node Structure	Encapsulates dictionary entries

Custom Data Structures

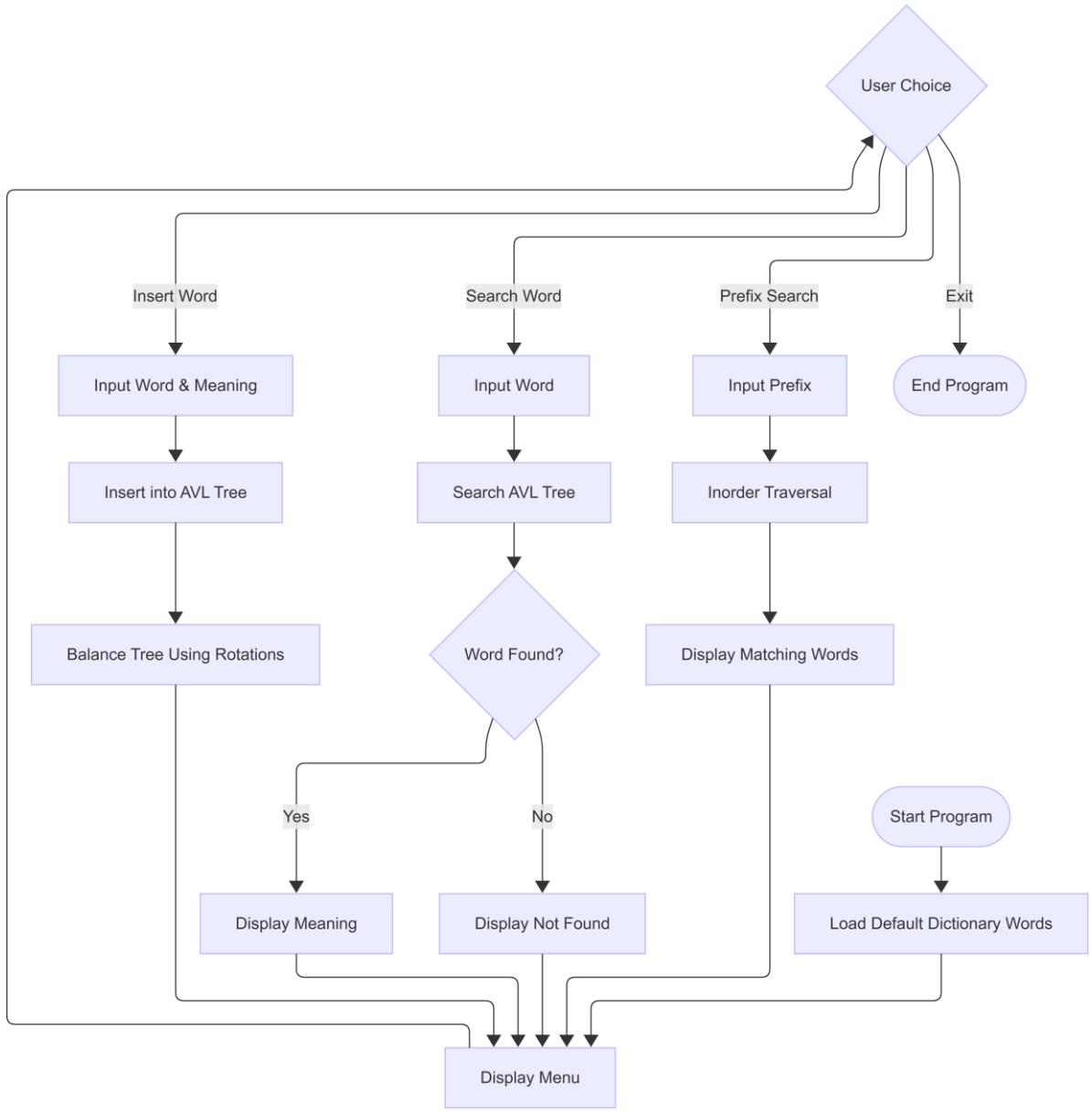
A custom **Node** structure is implemented with the following attributes:

- Word
- Meaning
- Height
- Left child pointer
- Right child pointer

Design Document

System Flow

1. Program starts
2. Dictionary is initialized
3. Menu is displayed
4. User selects an option
5. AVL Tree operations are executed
6. Results are displayed



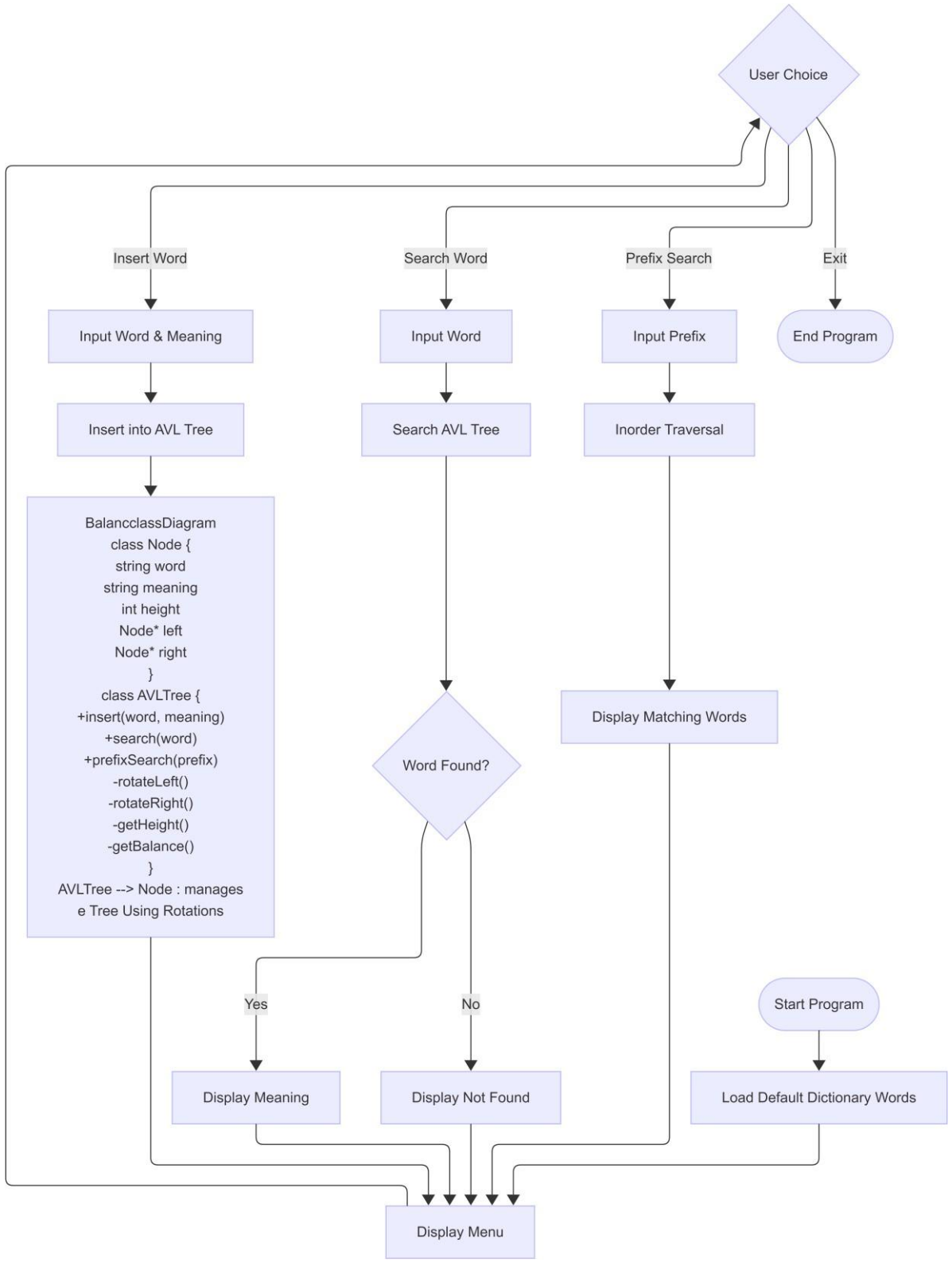
4.2 Class / Structure Design

Node Structure Design

The Node structure is the fundamental building block of the *AVL Tree Based Dictionary System*. Each node represents a single dictionary entry and is designed to store both the data and the structural information required to maintain the properties of an AVL Tree.

The Node structure encapsulates dictionary data, tree relationships, and balancing information, enabling efficient storage, retrieval, and self-balancing behavior.

Class Diagram



Class Diagram

Sequence Diagram

A Sequence Diagram is a Unified Modeling Language (UML) diagram that illustrates how different components of a system interact with each other over time. It focuses on the **order of** interactions between objects, showing the sequence in which messages are exchanged to perform a specific operation. Sequence diagrams are particularly useful for understanding system behavior during runtime and for visualizing the flow of control among system components.

In the AVL Tree Based Dictionary System with Prefix Search, the sequence diagram represents the interaction between the **user**, the menu interface, the **AVL Tree**, and the node structure during dictionary operations such as insertion, word search, and prefix-based search.

Purpose of the Sequence Diagram in This Project

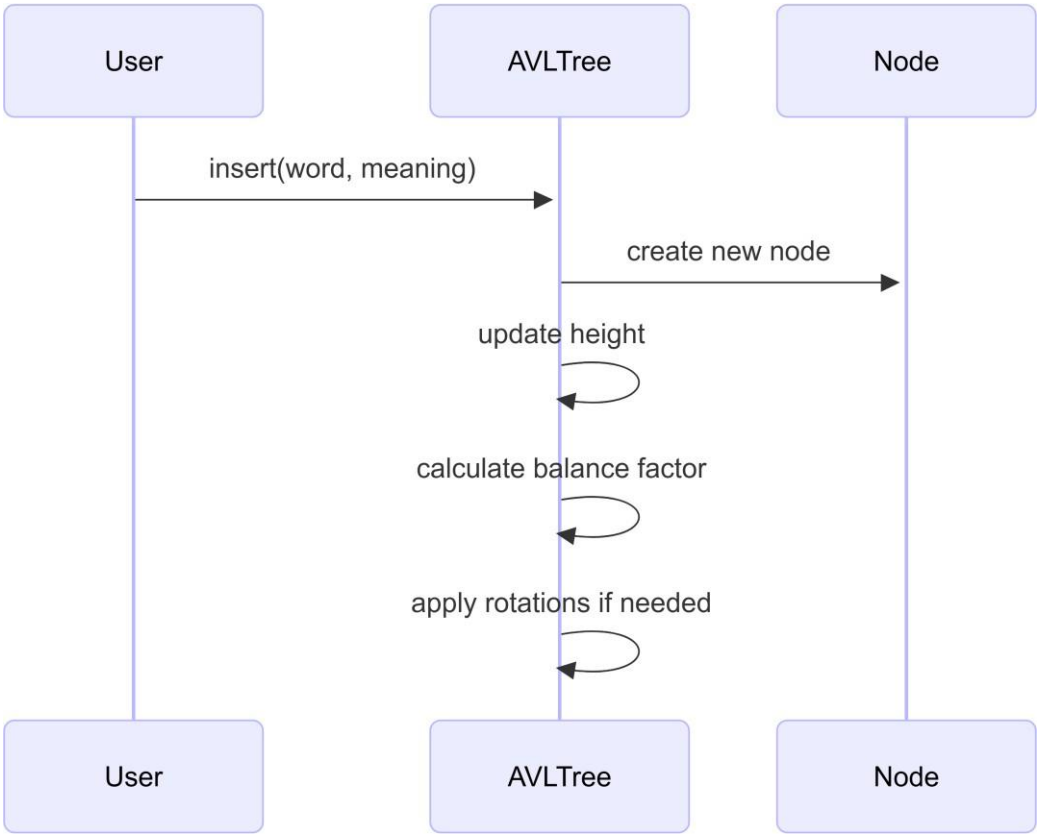
The sequence diagram is used to:

- Demonstrate how user requests are processed step by step.
- Show interaction between different modules of the system.
- Explain how data structures (AVL Tree and Node) are integrated.
- Visualize the execution order of operations.
- Improve understanding of system behavior for implementation and debugging.

Sequence Diagram for Insertion Operation

During insertion, the interaction proceeds as follows:

1. The user selects the **Insert Word** option from the menu.
2. The menu collects the word and its meaning from the user.
3. The menu sends an insert request to the AVL Tree.
4. The AVL Tree compares the word with existing nodes to find the correct position.
5. A new node is created and inserted following Binary Search Tree rules.
6. The height of affected nodes is updated.
7. The balance factor is calculated.
8. If imbalance is detected, appropriate AVL rotations are performed.
9. The insertion result is returned to the menu.
10. The menu displays a success message to the user.



Sequence Diagram

4.3 Pseudocode

Insertion Algorithm

```
#include <iostream>
#include <string>
using namespace std;
// Node Structure
struct Node {
    string word;
    string meaning;
    Node* left;
    Node* right;
```

```
    int height;

    Node(string w, string m) {
        word = w;
        meaning = m;
        left = right = NULL;
        height = 1;
    }
};

// Utility function to get height
int height(Node* node) {
    if (node == NULL)
        return 0;
    return node->height;
}
// Utility function to get maximum of two values
int max(int a, int b) {
    return (a > b) ? a : b;
}
// Right rotation
Node* rightRotate(Node* y) {
    Node* x = y->left;
    Node* T2 = x->right;
```

```
    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = max(height(y->left), height(y->right)) + 1;
    x->height = max(height(x->left), height(x->right)) + 1;
    return x;
}
// Left rotation
Node* leftRotate(Node* x) {
    Node* y = x->right;
    Node* T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    // Update heights
    x->height = max(height(x->left), height(x->right)) + 1;
    y->height = max(height(y->left), height(y->right)) + 1;

    return y;
}
```

```
// Get balance factor
int getBalance(Node* node) {
    if (node == NULL)
        return 0;
    return height(node->left) - height(node->right);
}

// AVL Insert function
Node* insert(Node* root, string word, string meaning) {

    // Step 1: Insert node like BST
    if (root == NULL)
        return new Node(word, meaning);

    if (word < root->word)
        root->left = insert(root->left, word, meaning);
    else if (word > root->word)
        root->right = insert(root->right, word, meaning);
    else
        return root; // Duplicate words not allowed

    // Step 2: Update height
    root->height = 1 + max(height(root->left), height(root->right));
    // Step 3: Compute balance factor
    int balance = getBalance(root);
```

```
    // Step 4: Perform rotations if needed
    // Left-Left Case
    if (balance > 1 && word < root->left->word)
        return rightRotate(root);
    // Right-Right Case
    if (balance < -1 && word > root->right->word)
        return leftRotate(root);
    // Left-Right Case
    if (balance > 1 && word > root->left->word) {
        root->left = leftRotate(root->left);
        return rightRotate(root);
    }

    // Right-Left Case
    if (balance < -1 && word < root->right->word) {
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }

    return root;
}
```

Git hub Drive Link

ujalasafdar/Dictionary_Searching-System

Prefix Search Algorithm

- Traverse tree in-order
- Match prefix
- Display results

```
// Prefix search using inorder traversal
```

```
void prefixSearch(Node* root, string prefix) {
    if (root == NULL)
        return;
    // Traverse left subtree
    prefixSearch(root->left, prefix);

    // Check prefix match
    if (root->word.find(prefix) == 0) {
        cout << root->word << " : " << root->meaning << endl;
    }
    // Traverse right subtree
    prefixSearch(root->right, prefix);
}
```

4.4 Feature to Data Structure Mapping

Feature	Data Structure
Insert	AVL Tree
Search	AVL Tree
Prefix Search	Tree Traversal

6. Conclusion

This project demonstrates the effective use of an AVL Tree to implement a dictionary system with fast insertion and search operations. The self-balancing nature of the AVL Tree ensures optimal performance, while prefix-based searching enhances usability. The project provides practical experience in advanced data structures and highlights their real-world applications.

7. References

http://geeksforgeeks.org/dsa/introduction-to-avl-tree/

https://www.geeksforgeeks.org/dsa/what-is-avl-tree-avl-tree-meaning/

https://www.geeksforgeeks.org/dsa/insertion-in-an-avl-tree/