M.Kumarasamy College of Engineering
NAAC Accredited Autonomous Institution
Approved by AICTE & Affiliated to Anna University
ISO 9001:2015 Certified Institution
Thalavapalayam, Karur - 639 113, TAMILNADU.

A Project Report on

SPELL CHECKER

Submitted in partial fulfilment of requirements for the award of the course

of

CGB1122 – DATA STRUCTURES

Under the guidance of

Mrs. Dr.K.POONGOTHAI

Associate Professor/IT

Submitted By

PRIYADARSHIKA G K (927624BAD073)

DEPARTMENT OF FRESHMAN ENGINEERING

M.KUMARASAMY COLLEGE OF ENGINEERING
(Autonomous)

KARUR – 639 113

MAY 2025

M. KUMARASAMY COLLEGE OF ENGINEERING

(Autonomous Institution affiliated to Anna University, Chennai)

KARUR – 639 113

BONAFIDE CERTIFICATE

Certified that this project report on **“SPELL CHECKER”** is the bonafide work of **PRIYADARSHIKA G K (927624BAD073)** who carried out the project work during the academic year 2024- 2025 under my supervision.

Signature	Signature
Dr.K.POONGOTHAI	**Dr. K.CHITIRAKALA, M.Sc., M.Phil.,Ph.D.,**
SUPERVISOR,	**HEAD OF THE DEPARTMENT,**
Department of Information and Technology,	Department of Freshman Engineering,
M. Kumarasamy College of Engineering,	M. Kumarasamy College of Engineering,
Thalavapalayam, Karur -639 113.	Thalavapalayam, Karur -639 113.

Submitted for the End Semester Review held on ___________________________

M.Kumarasamy
College of Engineering
NAAC Accredited Autonomous Institution
Approved by AICTE & Affiliated to Anna University
ISO 9001:2015 Certified Institution
Thalavapalayam, Karur - 639 113, TAMILNADU.

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

VISION OF THE INSTITUTION

To emerge as a leader among the top institutions in the field of technical education

MISSION OF THE INSTITUTION

- Produce smart technocrats with empirical knowledge who can surmount the global challenges
- Create a diverse, fully-engaged, learner-centric campus environment to provide quality education to the students
- Maintain mutually beneficial partnerships with our alumni, industry, and Professional associations

VISION OF THE DEPARTMENT

To produce competent industry relevant education, skillful research, technical and innovative computer science professionals acquaintance with managerial skills, human and social values.

MISSION OF THE DEPARTMENT

- To impart technical knowledge through innovative teaching, research, and consultancy.
- To develop and to promote student ability thereby to compete globally through excellence in education.
- To facilitate the development of academic-industry Collaboration.
- To produce competent engineers with professional ethics, technical competence and a spirit of innovation and managerial skills.

PROGRAM EDUCATIONAL OBJECTIVES (PEOs)

PEO 1: To acquire technical knowledge and proficiency required for the employment and higher education in the contemporary areas of computer science or management studies.

PEO 2: To apply their competency in design and development of innovative solutions for real-world problems.

PEO 3: To demonstrate leadership qualities with high ethical standards and collaborated with other industries for the socio-economical growth of the country.

PROGRAM OUTCOMES (POs)

Engineering students will be able to:

1. **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

11. **Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAM SPECIFIC OUTCOMES (PSOs)

1. **PSO1:** Ability to apply the analytical and business skills to provide sustainable solutions as an engineer/researcher for the real-time applications using Machine Learning, Internet of Things and Data analytics.
2. **PSO2:** Ability to practice ethical and human values with soft-skills qualities in computer science and business disciplines to emerge as an entrepreneur for the growth and development of the society.

ABSTRACT

In the digital era, spell checking plays a crucial role in enhancing the quality and accuracy of written communication. This project presents a basic yet functional spell checker implemented in the C programming language using an AVL (Adelson-Velsky and Landis) tree data structure. The AVL tree is a type of self-balancing binary search tree that ensures consistently efficient performance for insertions, deletions, and lookups.The spell checker loads a predefined list of words (a dictionary) into the AVL tree, allowing fast verification of user-input words. When a user inputs a word, the system checks if it exists in the dictionary. If it is not found, the program searches the tree for words that share the same prefix and suggests up to five possible corrections. These suggestions help users correct their input based on the closest matching entries in the dictionary.The choice of the AVL tree ensures that all operations remain efficient regardless of the number of words stored, making the solution scalable. Additionally, by using in-order traversal during suggestion generation, the program presents words in lexicographical order, improving the usability of the suggestions. This project serves as a practical example of applying advanced data structures to solve common real-world problems such as spell checking.The system stores a predefined dictionary of words in an AVL tree, allowing fast insertion and lookup operations due to the tree's self-balancing property. When a user inputs a word, the program checks its presence in the dictionary. If the word is not found, the program suggests up to five alternative words that share the same prefix. The AVL tree ensures balanced and optimized performance, maintaining O(log n) complexity for search and insert operations. This makes the implementation suitable for basic spell-checking applications with quick response times and minimal memory usage.

M.Kumarasamy
College of Engineering
NAAC Accredited Autonomous Institution
Approved by AICTE & Affiliated to Anna University
ISO 9001:2015 Certified Institution
Thalavapalayam, Karur - 639 113, TAMILNADU.

ABSTRACT WITH POs AND PSOs MAPPING

ABSTRACT	POs MAPPED	PSOs MAPPED
This project presents a simple yet efficient spell checker implemented using an AVL tree data structure. The system stores a predefined dictionary of words in an AVL tree, allowing fast insertion and lookup operations due to the tree's self-balancing property. When a user inputs a word, the program checks its presence in the dictionary. If the word is not found, the program suggests up to five alternative words that share the same prefix. The AVL tree ensures balanced and optimized performance, maintaining O(log n) complexity for search and insert operations. This makes the implementation suitable for basic spell-checking applications with quick response times and minimal memory usage.	**PO1** **PO2** **PO3** **PO4** **PO5** **PO6** **PO7** **PO9** **PO11** **PO12**	**PSO1** **PSO2** **PSO3**

Note: 1- Low, 2-Medium, 3- High

SUPERVISOR **HEAD OF THE DEPARTMENT**

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Introduction

Spell checking is an essential feature in modern text processing and editing tools. It assists users in identifying and correcting typographical errors, thereby improving the clarity and professionalism of written communication. The underlying mechanism of a spell checker involves comparing user-input words against a predefined dictionary and suggesting corrections for misspelled words.

This project aims to implement a basic spell checker using the C programming language, utilizing an AVL tree as the primary data structure for storing and managing dictionary words. An AVL tree is a self-balancing binary search tree that maintains its height to ensure efficient operations, making it suitable for dynamic and performance-critical applications like spell checking.

The program reads a fixed list of dictionary words and inserts them into the AVL tree. Users can then input words interactively, and the program checks whether each word exists in the dictionary. If the word is correct, it is confirmed as such; if not, the program provides up to five suggestions based on prefix matching to help the user find the intended word.

1.2 Objective

To build a spell checker in C using an AVL tree that checks word correctness and suggests similar words efficiently. The AVL tree ensures fast insertions and lookups by maintaining balance. The program also helps users by offering up to five suggestions based on prefix matching.

1.3 Data Structure Choice

The primary data structure used in this project is the AVL Tree, a self-balancing binary search tree. It is chosen because it maintains balance after every insertion, ensuring that the height of the tree remains O(log n), which guarantees efficient search, insert, and suggestion operations.An AVL tree keeps words in sorted order, which is ideal for lexicographic operations like prefix-based suggestions. Unlike hash tables, which do not preserve order, the AVL tree allows in-order traversal to easily retrieve suggestions in alphabetical order. It is also more space-efficient than tries for moderate-sized dictionaries, making it a practical choice for a basic spell checker.

CHAPTER 2
PROJECT METHODOLOGY

2.1 AVL Tree

The development of the spell checker project followed a systematic approach, divided into several key stages:

1. Problem Analysis

Identified the need for an efficient spell checker.

Defined core functionalities: word verification and suggestion of alternatives for misspelled words.

2. Data Structure Selection

Evaluated various data structures (Trie, Hash Table, BST).

Chose AVL Tree for its balanced structure, efficient search time, and lexicographical traversal support.

3. Design and Implementation

Implemented the AVL Tree with insertion, rotation, and balancing functions in C.

Designed insert, search, and suggest functions to manage dictionary words and handle user input.

4. Dictionary Initialization

Loaded a predefined set of words into the AVL tree during program startup.

5. User Interaction

Built a loop to accept user-input words and check their correctness.

Provided up to five suggestions using prefix matching if the word was not found.

6. Testing and Debugging

Tested with various inputs to ensure accuracy of word checking and balance of the tree.Handled edge cases such as duplicate entries and invalid input.

7. Memory Management

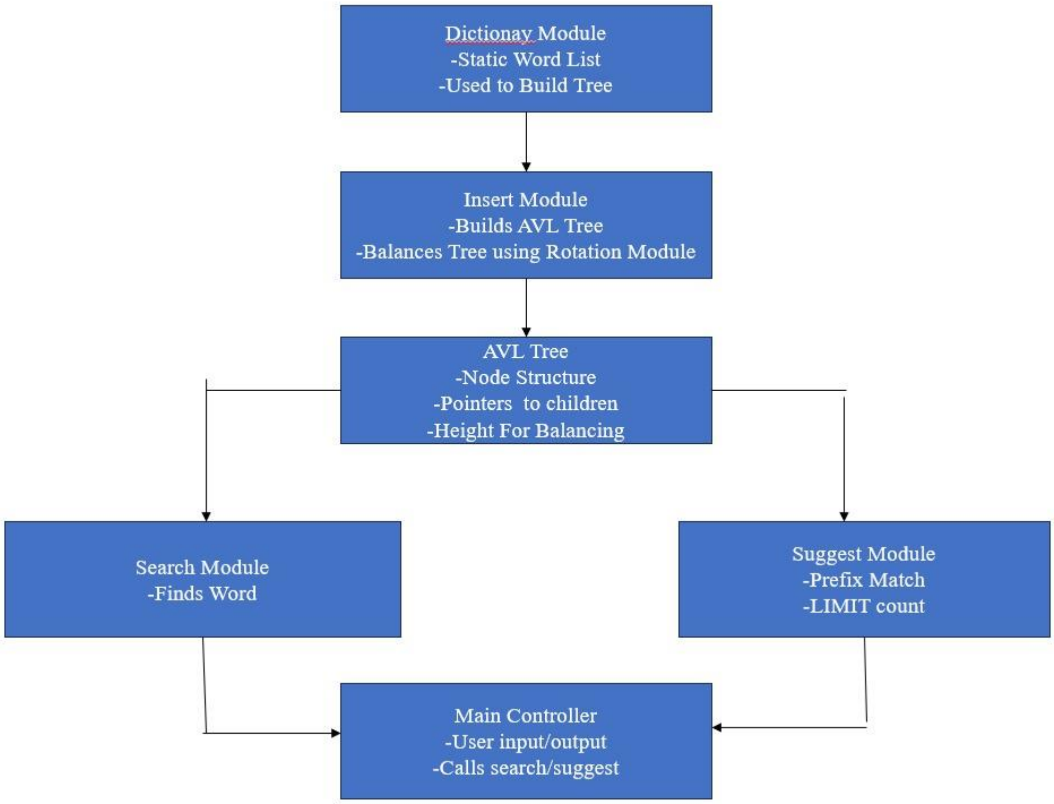
Ensured all dynamically allocated memory (nodes of the tree) is properly freed at the end.

8. Finalization

Refined the interface and output for better user experience.

Verified performance and correctness of the application.

2.2 Block Diagram



CHAPTER 3
MODULES

3.1 Dictionary Initialization Module

Loads a predefined set of words into the AVL tree.Ensures uniqueness by skipping duplicate entries.Validates words before insertion (e.g., no empty strings).

3.2 AVL Tree Operations Module

Node Creation: Allocates memory and initializes new nodes.

Insertion: Places words in the correct position while keeping the tree balanced.

Rotations: Handles left, right, left-right, and right-left rotations to maintain balance.

Height & Balance Factor Management: Automatically updates height and balance after every operation.

3.3 Search Module

Compares user input with words in the AVL tree.Uses recursive traversal for efficient searching.Returns search status to the main program.

3.4 Suggestion Module

Traverses the tree in-order to preserve lexicographical order.Uses prefix matching to find relevant suggestions.Limits output to the top 5 matching words.Stops traversal early once the suggestion limit is reached.

3.5 User Interaction Module

Provides a text-based interface for users.Accepts word inputs in a loop until “exit” is typed.Displays whether the word is correct or incorrect.

hows suggestions if the word is not found.

3.6 Input Validation Module

Sanitizes user input (e.g., converts to lowercase, removes non-alphabetic characters).

Prevents malformed or invalid words from affecting the program.

3.7 Memory Management Module

Frees all AVL tree nodes recursively after use.Ensures there are no memory leaks at program termination.

3.8 Output Display Module

Formats and displays output messages to the user.Ensures clarity in spelling results and suggestions.

CHAPTER 4

RESULTS AND DISCUSSION

4.1 Results

4.1.1 Exact match and prefix suggestion

```
Enter words (type 'exit' to quit):

Word: trie
Correct

Word: tri
Incorrect
  Suggestion: trick
  Suggestion: trie

Word: tr
Incorrect
  Suggestion: tree
  Suggestion: trick
  Suggestion: trie

Word: exit
```

4.1.2 Multiple suggestions for prefix

```
Enter words (type 'exit' to quit):

Word: world
Correct

Word: worr
Incorrect
  No suggestions found.

Word: wor
Incorrect
  Suggestion: world

Word: exit
```

4.1.3 Single or no suggestions

```
Enter words (type 'exit' to quit):

Word: che
Incorrect
  Suggestion: check
  Suggestion: checker
  Suggestion: cheese

Word: cheese
Correct

Word: choi
Incorrect
  Suggestion: choice

Word: exit
```

4.1.4 Deep prefix and ordered results

```
Enter words (type 'exit' to quit):

Word: str
Incorrect
  Suggestion: structure

Word: spell
Correct

Word: spe
Incorrect
  Suggestion: spell

Word: spek
Incorrect
  No suggestions found.

Word: exit
```

4.2 Discussion

This C program implements a simple yet efficient spell checker using an AVL tree, which is a self-balancing binary search tree. The AVL tree ensures that insertions and lookups are performed in O(log n) time by maintaining balance through rotations after each insertion. Words from a predefined dictionary are inserted into the tree, and the user can continuously input words to check their correctness. If a word is found in the tree, it is deemed correct; otherwise, the program suggests up to five alternatives that begin with the same prefix, aiding users in correcting potential spelling errors. These suggestions are generated using an in-order traversal of the tree to maintain lexicographic order. The use of AVL trees ensures efficient searching and balanced performance even as the dictionary grows. The code is a solid demonstration of how advanced data structures like AVL trees can be applied to practical problems such as spell checking, while also highlighting the use of recursion, string manipulation, and dynamic memory allocation in C programming.

CHAPTER 5
CONCLUSION

This project successfully demonstrates the implementation of a basic spell checker using an AVL tree in the C programming language. By leveraging the self-balancing nature of AVL trees, the program ensures efficient insertion, searching, and suggestion retrieval operations, even as the size of the dictionary grows.The spell checker not only verifies whether a word exists in the dictionary but also provides helpful suggestions based on prefix matching, enhancing user experience. Through this project, key programming concepts such as dynamic memory management, tree-based data structures, recursion, and string manipulation were effectively applied.Overall, the project highlights the importance of choosing the right data structure for specific problem-solving tasks and showcases how foundational computer science principles can be used to build practical, real-world applications.

REFERENCES

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, Introduction to Algorithms, 3rd Edition, MIT Press, 2009.

[2] Brian W. Kernighan and Dennis M. Ritchie, The C Programming Language, 2nd Edition, Prentice Hall, 1988.

[3] GeeksforGeeks – AVL Tree (Self-Balancing BST)

[4] TutorialsPoint, C Programming

[5] Wikipedia – Spell Checker

[6] Programiz – AVL Tree

APPENDIX

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX 100
#define LIMIT 5

typedef struct Node {
char word[MAX];
struct Node *l, *r;
int h;
} Node;

int max(int a, int b) { return a > b ? a : b; }
int height(Node* n) { return n ? n->h : 0; }

Node* newNode(const char* w) {
Node* n = (Node*)malloc(sizeof(Node));
strcpy(n->word, w); n->l = n->r = NULL; n->h = 1;
return n;
}

Node* rotRight(Node* y) {
Node *x = y->l, *T = x->r;
x->r = y; y->l = T;
y->h = max(height(y->l), height(y->r)) + 1;
x->h = max(height(x->l), height(x->r)) + 1;
```

```
return x;
}

Node* rotLeft(Node* x) {
Node *y = x->r, *T = y->l;
y->l = x; x->r = T;
x->h = max(height(x->l), height(x->r)) + 1;
y->h = max(height(y->l), height(y->r)) + 1;
return y;
}

int balance(Node* n) { return n ? height(n->l) - height(n->r) : 0; }

Node* insert(Node* n, const char* w) {
if (!n) return newNode(w);
int cmp = strcmp(w, n->word);
if (cmp < 0) n->l = insert(n->l, w);
else if (cmp > 0) n->r = insert(n->r, w);
else return n;

n->h = 1 + max(height(n->l), height(n->r));
int b = balance(n);

if (b > 1 && strcmp(w, n->l->word) < 0) return rotRight(n);
if (b < -1 && strcmp(w, n->r->word) > 0) return rotLeft(n);
if (b > 1 && strcmp(w, n->l->word) > 0) { n->l = rotLeft(n->l); return
rotRight(n); }
if (b < -1 && strcmp(w, n->r->word) < 0) { n->r = rotRight(n->r); return
rotLeft(n); }
```

```
return n;

}

int search(Node* root, const char* w) {
if (!root) return 0;
int cmp = strcmp(w, root->word);
if (cmp == 0) return 1;
return cmp < 0 ? search(root->l, w) : search(root->r, w);
}

void suggest(Node* root, const char* pre, int* count) {
if (!root || *count >= LIMIT) return;
suggest(root->l, pre, count);
if (strncmp(root->word, pre, strlen(pre)) == 0) {
printf("  Suggestion: %s\n", root->word);
(*count)++;
}
suggest(root->r, pre, count);
}

void freeTree(Node* root) {
if (!root) return;
freeTree(root->l); freeTree(root->r); free(root);
}

int main() {
const char* dict[] = {"hello", "world", "spell", "checker", "tree",
```

```
"data", "structure", "check", "cheese", "choice", "trick", "trie"};
int n = sizeof(dict) / sizeof(dict[0]);
Node* root = NULL;
for (int i = 0; i < n; i++) root = insert(root, dict[i]);

char word[MAX];
printf("Enter words (type 'exit' to quit):\n");
while (1) {
printf("\nWord: "); scanf("%s", word);
if (strcmp(word, "exit") == 0) break;
if (search(root, word)) printf("Correct\n");
else {
printf("Incorrect\n");
int count = 0;
suggest(root, word, &count);
if (!count) printf("  No suggestions found.\n");
}
}

freeTree(root);
return 0;

}
```