K L UNIVERSITY

FRESHMAN ENGINEERING DEPARTMENT

A Project Based Lab Report

On

AVL Tree

SUBMITTED BY:

ID NUMBER	NAME
2400032687	G. YUGA KEERTHI

UNDER THE GUIDANCE OF

Dr. SK. RAZIA

Associate Professor

KL UNIVERSITY
Green fields, Vaddeswaram – 522 502
Guntur Dt., AP, India.

DEPARTMENT OF BASIC ENGINEERING SCIENCES

CERTIFICATE

This is to certify that the project based laboratory report entitled "AVL TREE" submitted by **G. YUGA KEERTHI** bearing Regd. No.2400032687 to the **Department of COMPUTER SICENCE ENGINEERING , KL University** in partial fulfillment of the requirements for the completion of a project based Laboratory in "DATA STRUCTURES(24SC1203)" course in I B Tech II Semester, is a Bonafide record of the work carried out by them under my supervision during the academic year 2024 – 2025.

PROJECT SUPERVISOR	HEAD OF THE DEPARTMENT
Dr. SK. RAZIA	Dr. D. HARITHA

ACKNOWLEDGEMENTS

It is great pleasure for me to express my gratitude to our honorable President. **Sri. Koneru Satyanarayana**, for giving the opportunity and platform with facilities in accomplishing the project-based laboratory report.

I express sincere gratitude to our Director **Dr. A. Jagdeesh** for his administration towards our academic growth. I express sincere gratitude to our Principal **Dr. V. Krishna Reddy** for his administration towards our academic growth.

I express sincere gratitude to our Coordinator and HOD-BES **Dr. D. Haritha** for her leadership and constant motivation provided for the successful completion of our academic semester. I record it as my privilege to deeply thank you for providing us with the efficient faculty and facilities to make our ideas into reality.

I express my sincere thanks to our project supervisor **Dr. SHAIK RAZIA** for his novel association of ideas, encouragement, appreciation, and intellectual zeal which motivated us to venture this project successfully.

Finally, we are pleased to acknowledge our indebtedness to all those who devoted themselves directly orindirectly to making this project a success.

G. Yuga Keerthi -2400032687

ABSTRACT

My Project entitled “AVL tree”. This project aims to implement an application that uses an AVL tree—a self-balancing binary search tree—to efficiently store and manage words from a given document, alongside their frequency counts. The primary goal is to construct an AVL tree where each node represents a unique word from the document, with a pointer to an integer that maintains the count of occurrences for that word.The AVL tree ensures logarithmic time complexity for insertion, deletion, and search operations, enabling the application to manage large textual datasets effectively. This implementation not only highlights the utility of AVL trees in managing structured textual data but also serves as a practical demonstration of efficient data organization and retrieval.

INDEX

INTRODUCTION

In the modern era of information technology, the volume of textual data processed daily is immense. From documents and articles to emails and logs, analyzing and processing text efficiently has become a necessity in many applications such as search engines, data mining, and content management systems.

One of the most fundamental operations in text processing is determining the frequency of each word used in a document. Word frequency analysis plays an important role in identifying keywords, performing statistical analysis, and even improving search engine optimization (SEO). Efficiently tracking and storing the frequency of words, especially in large documents, requires a suitable data structure that supports fast insertion, searching, and retrieval.

This project presents a solution by implementing an **AVL Tree** — a self-balancing binary search tree-to store and count the frequency of words in a given file. Each node in the AVL Tree contains:

- A unique word from the document
- A pointer to an integer representing the number of times that word appears

The AVL Tree maintains balance after every insertion operation by performing rotations, ensuring that the height of the tree is minimized and the performance of operations remains optimal. With AVL Trees, insertions, deletions, and look ups all operate in **O(log n)** time complexity.

The application works by reading the content of a file word by word. As each word is read:

- If the word does not exist in the tree, it is inserted as a new node with a count of 1.
- If the word already exists in the tree, its count is incremented by 1.
- After each insertion, the AVL Tree checks and restores balance if necessary.

The resulting data structure is a balanced tree that stores all the words in the document along with the frequency of each word. The application also includes functionality to print:

- All the words in alphabetical order with their corresponding counts (using in-order traversal)

- A specific word and its frequency (if required)

Importance of AVL Trees in This Context

Unlike hash tables that provide fast lookup but do not maintain order, AVL Trees store data in a sorted manner. This makes it easy to retrieve and display words in alphabetical order — a common requirement in reporting and analysis.

Moreover, as documents can be large and contain thousands of words, maintaining performance and efficiency is critical. The AVL Tree's balancing mechanism ensures that performance does not degrade as the number of words increases.

The key objectives of this project are:

- To implement an AVL Tree data structure from scratch
- To read and parse a file word by word
- To insert words into the AVL Tree with frequency counts
- To maintain balance in the tree dynamically
- To display the final list of words and their frequencies in sorted order

The application will consist of several key functionalities:

1. **Word Counting**: The program will read a specified text file and count the occurrences of each word, ignoring case and punctuation to ensure accurate results.
2. **AVL Tree Construction**: As words are read from the file, they will be inserted into the AVL tree. Each node in the tree will contain a word and a pointer to an integer that tracks the count of that word's occurrences.
3. **Word Reading**: A dedicated function will handle the reading of individual words from the file, ensuring that the program can process the text efficiently.
4. **Integer Comparison**: To facilitate the insertion and balancing of the AVL tree, a function will be implemented to compare the counts of words, allowing for proper ordering within the tree.
5. **Output Functions**: The application will include functions to print the entire list of words along with their counts, as well as the ability to print a specific word and its count, providing users with flexible options for viewing the results.

AIM

The aim of this project is to create an application that efficiently counts and organizes the words in a text document using an AVL tree. Specifically, the project seeks to:

1. **Count Word Frequencies**: Accurately count how many times each word appears in a given text file.
2. **Organize Words**: Store the words and their counts in a balanced AVL tree, which allows for quick access and retrieval.
3. **Provide Easy Access**: Enable users to view the list of words along with their counts and to look up specific words to see how many times they appear.

Overall, the goal is to develop a user-friendly tool that simplifies the process of analysing text data by leveraging efficient data structures.

ADVANTAGES

1. Efficient Searching & Insertion: AVL trees maintain a balanced structure, ensuring O(log n) time complexity for searching and inserting words.
2. Automatic Balancing: Unlike regular binary search trees, AVL trees self balance, preventing performance degradation in worst-case scenarios.
3. Accurate Word Count: The program correctly tracks occurrences of words, making it useful for text analysis.
4. Case Insensitivity: Converts words to lowercase before insertion, ensuring consistency in word counts.

DISADVANTAGES

1. **Higher Rotation Overhead**: Balancing the tree requires additional rotations, which can slow down insertion in some cases.
2. **Memory Usage**: Each node requires extra storage for height and balance factor, which might be excessive for large documents.
3. **Complexity:** Implementing and debugging AVL trees can be more challenging compared to simpler structures like hash maps.

FUTURE IMPLEMENTATION

1. Support for Larger Files: Optimize memory usage and implement efficient disk-based storage for very large files.
2. Parallel Processing: Multi-threading can be used to speed up file reading and AVL tree operations.
3. Improved Output Formatting: Enhancing the word count display (sorting by frequency, exporting results to a file, etc.).
4. Stop word Filtering: Implementing a feature to ignore common words (e.g., “the”, “and”, “is”) for more meaningful analysis.
5. Integration with Other Data Structures: Hybrid approaches (e.g., using tries for prefix-based search) can improve performance.

SYSTEM REQUIREMENTS

- **SOFTWARE REQUIREMENTS:**

  The major software requirements of the project are as follows:
  Language: Turbo-C.
  Operating system**:** Windows 10 or more.
  Technical requirements: Monitor, CPU, Keyboard, Mouse, etc...

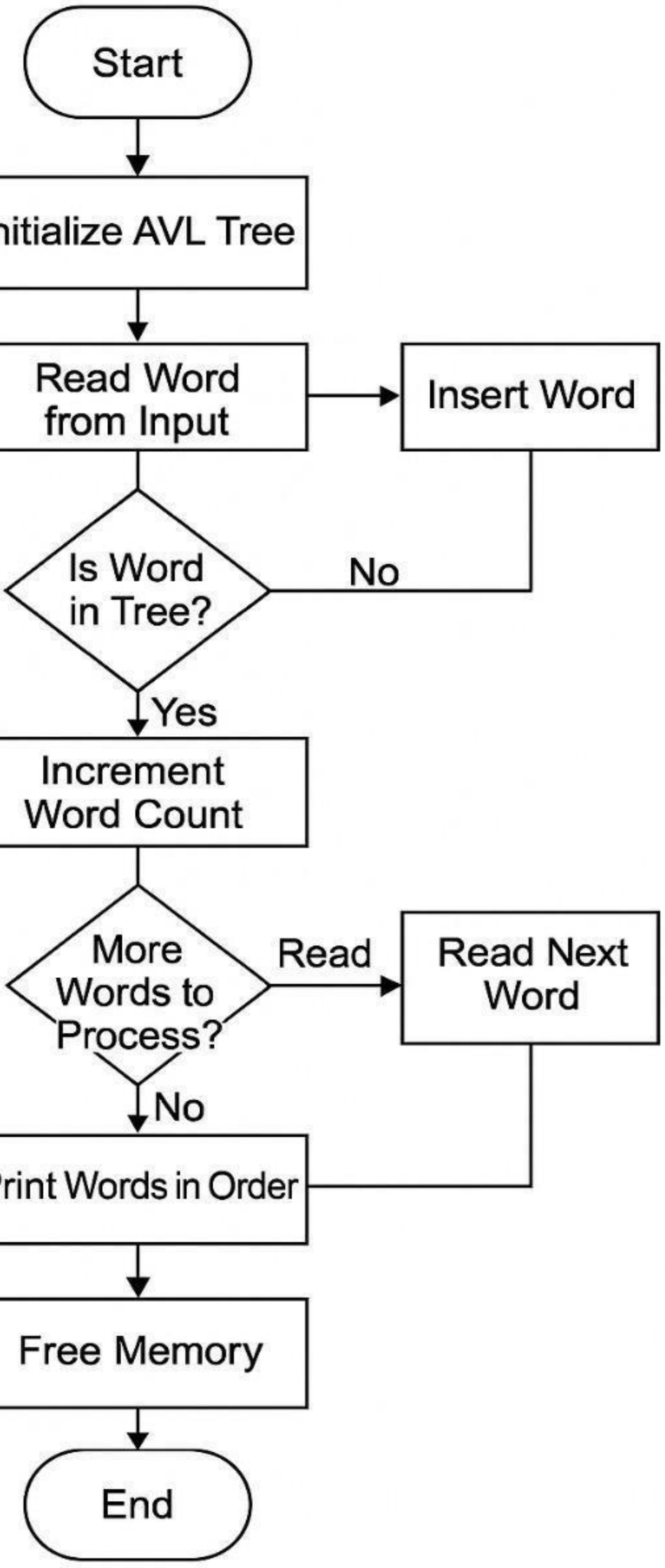
- **HARDWARE REQUIREMENTS:**

  The hardware requirements that map towards the software are as follows:

  RAM: 2 GB.

  Processor: i3

Flow Chart


Start
Initialize AVL Tree
Read Word from Input
Insert Word
Is Word in Tree?
No
Yes
Increment Word Count
More Words to Process?
Read
Read Next Word
No
Print Words in Order
Free Memory
End

Algorithm for Word Counting using AVL Tree

1. **Initialize AVL Tree**:
   - Create a structure for the AVL tree node that includes:
     - A string to store the word.
     - A pointer to an integer to store the count of occurrences.
     - Pointers to the left and right child nodes.
     - An integer to store the height of the node.
2. **Read File**:
   - Open the specified text file for reading.
   - If the file cannot be opened, display an error message and exit.
3. **Process Words**:
   - While there are more words in the file:

   1. Read the next word from the file.
   2. Normalize the word (e.g., convert to lowercase, remove punctuation).
   3. Insert the word into the AVL tree:
      - If the word already exists in the tree, increment its count.
      - If the word does not exist, create a new node with a count of 1 and insert it into the tree.
   4. After each insertion, update the height of the nodes and perform necessary rotations to maintain the AVL property.
4. **Insert Function**:
   - If the tree is empty, create a new node for the word and return it.
   - Compare the new word with the current node's word:
     - If the new word is less than the current node's word, recursively insert it into the left subtree.
     - If the new word is greater, recursively insert it into the right subtree.

- Update the height of the current node.
- Calculate the balance factor (height of left subtree - height of right subtree).
- If the balance factor is greater than 1 or less than -1, perform appropriate rotations (left, right, left-right, or right-left) to balance the tree.

5. **Print All Words**:
    - Implement an in-order traversal of the AVL tree to print all words along with their counts in sorted order.
6. **Print Specific Word**:
    - Implement a search function that takes a word as input and traverses the AVL tree to find and print the count of that specific word.
7. **Close File**:
    - After processing all words, close the file.
8. **End Program**:
    - Provide an option to exit the program or to process another file.

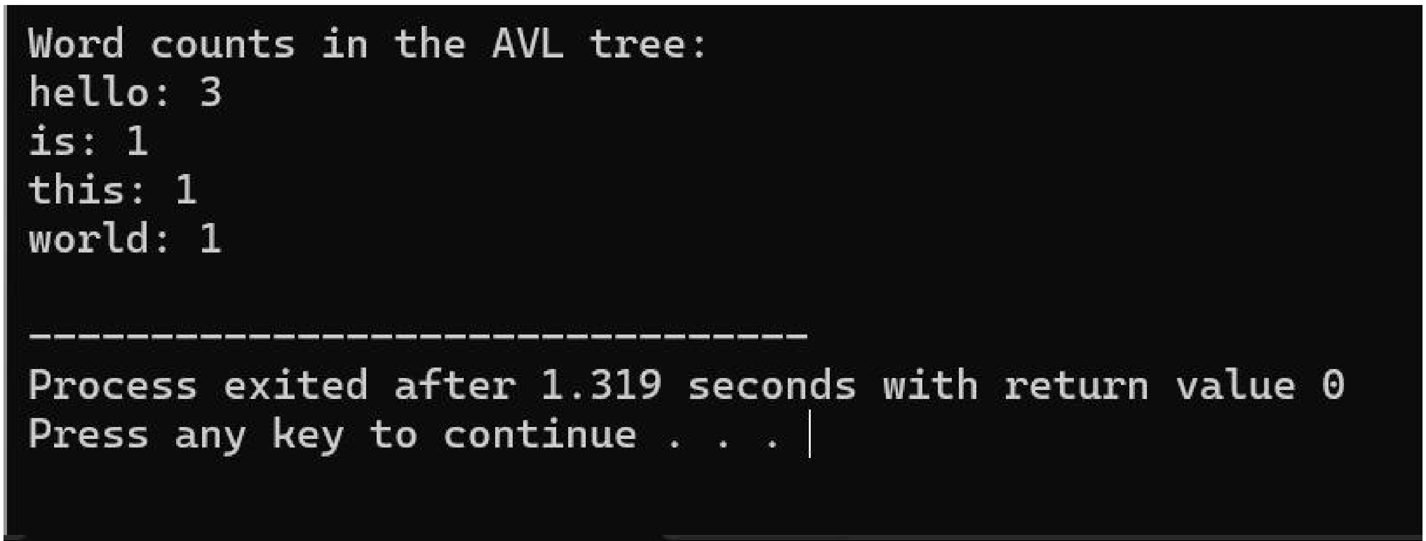
IMPLEMENTATION

```
#include  <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
typedef  struct  AVLTreeNode
   { char* word;
   int count;
   struct AVLTreeNode* left;
   struct AVLTreeNode* right;
   int height;
} AVLTreeNode;
AVLTreeNode* createNode(const char* word) {
   AVLTreeNode* newNode = (AVLTreeNode*)malloc(sizeof(AVLTreeNode));
   newNode->word = strdup(word);
   newNode->count = 1;
   newNode->left = newNode->right = NULL;
   newNode->height = 1;
   return newNode;
}
int getHeight(AVLTreeNode* node)
   { if (node == NULL) return 0;
   return node->height;
}
int getBalance(AVLTreeNode* node)
   { if (node == NULL) return 0;
   return getHeight(node->left) - getHeight(node->right);
}
AVLTreeNode* rightRotate(AVLTreeNode* y)
   { AVLTreeNode* x = y->left;
   AVLTreeNode* T2 = x->right;
   x->right = y;
   y->left = T2;
   y->height = 1 + fmax(getHeight(y->left), getHeight(y->right));
   x->height = 1 + fmax(getHeight(x->left), getHeight(x->right));
   return x;
```

```
}
AVLTreeNode* leftRotate(AVLTreeNode* x)
   { AVLTreeNode* y = x->right;
   AVLTreeNode* T2 = y->left;
   y->left = x;
   x->right = T2;
   x->height = 1 + fmax(getHeight(x->left), getHeight(x->right));
   y->height = 1 + fmax(getHeight(y->left), getHeight(y->right));
   return y;
}
AVLTreeNode* insert(AVLTreeNode* node, const char* word)
   { if (node == NULL)
      return createNode(word);
   if (strcmp(word, node->word) < 0)
      node->left = insert(node->left, word);
   else if (strcmp(word, node->word) > 0)
      node->right = insert(node->right, word);
   else {
      node->count++;
      return node;
   }
   node->height = 1 + fmax(getHeight(node->left), getHeight(node->right));
   int balance = getBalance(node);
   if (balance > 1 && strcmp(word, node->left->word) < 0)
      return rightRotate(node);
   if (balance < -1 && strcmp(word, node->right->word) > 0)
      return leftRotate(node);
   if (balance > 1 && strcmp(word, node->left->word) > 0)
      { node->left = leftRotate(node->left);
      return rightRotate(node);
   }
   if (balance < -1 && strcmp(word, node->right->word) < 0)
      { node->right = rightRotate(node->right);
      return leftRotate(node);
   }
   return node;
}
void inorder(AVLTreeNode* root)
   { if (root != NULL) {
        inorder(root->left);
        printf("%s: %d\n", root->word, root->count);
        inorder(root->right);
    }
}
void freeTree(AVLTreeNode* root)
    { if (root != NULL) {
        freeTree(root->left);
        freeTree(root->right);
        free(root->word);
        free(root);
    }
}
int main() {
    AVLTreeNode* root = NULL;
    const char* words[] = {"hello", "world", "hello", "this","is", "hello"};
    int n = sizeof(words) / sizeof(words[0]);
    int i;
    for (i = 0; i < n; i++) {
        root = insert(root, words[i]);
    }
    printf("Word counts in the AVL tree:\n");
    inorder(root);
    freeTree(root);
    return 0;
}
```

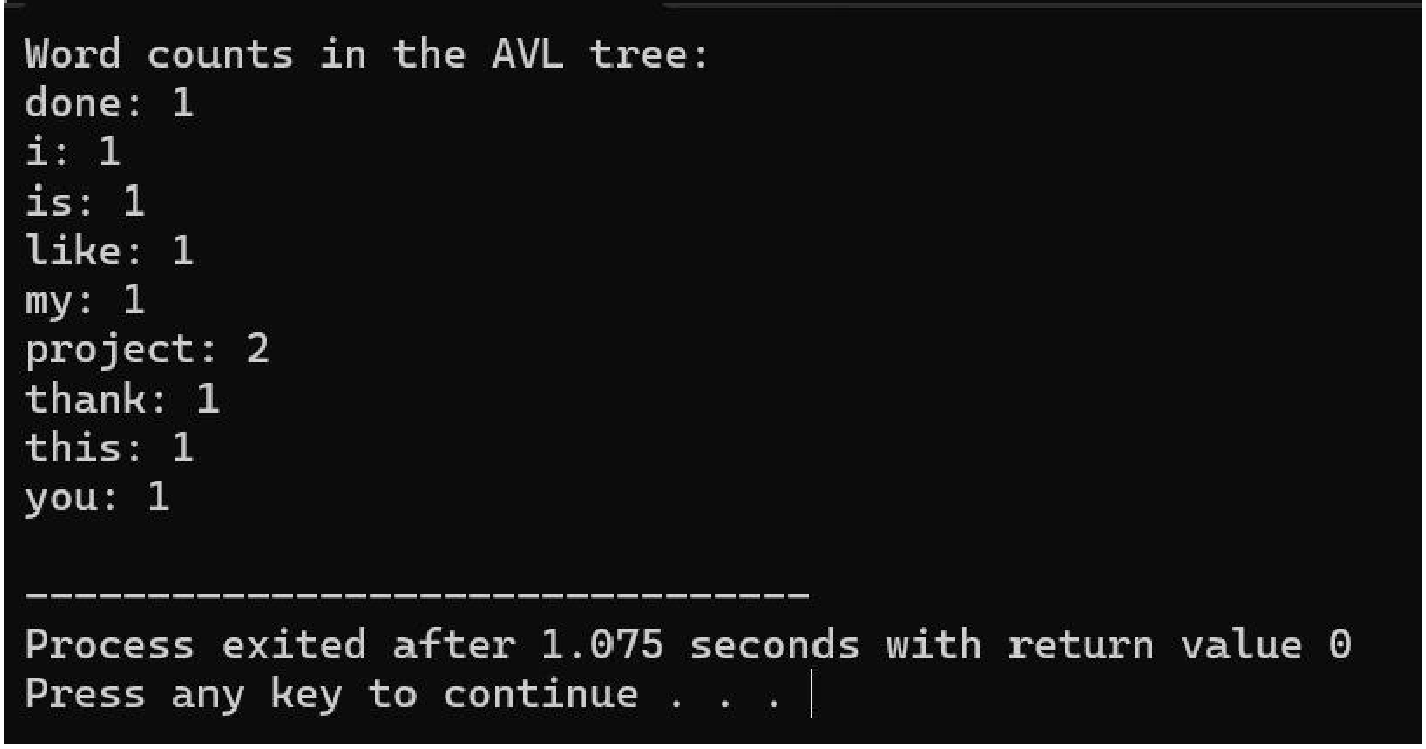
OUTPUTS

Screen Shots:



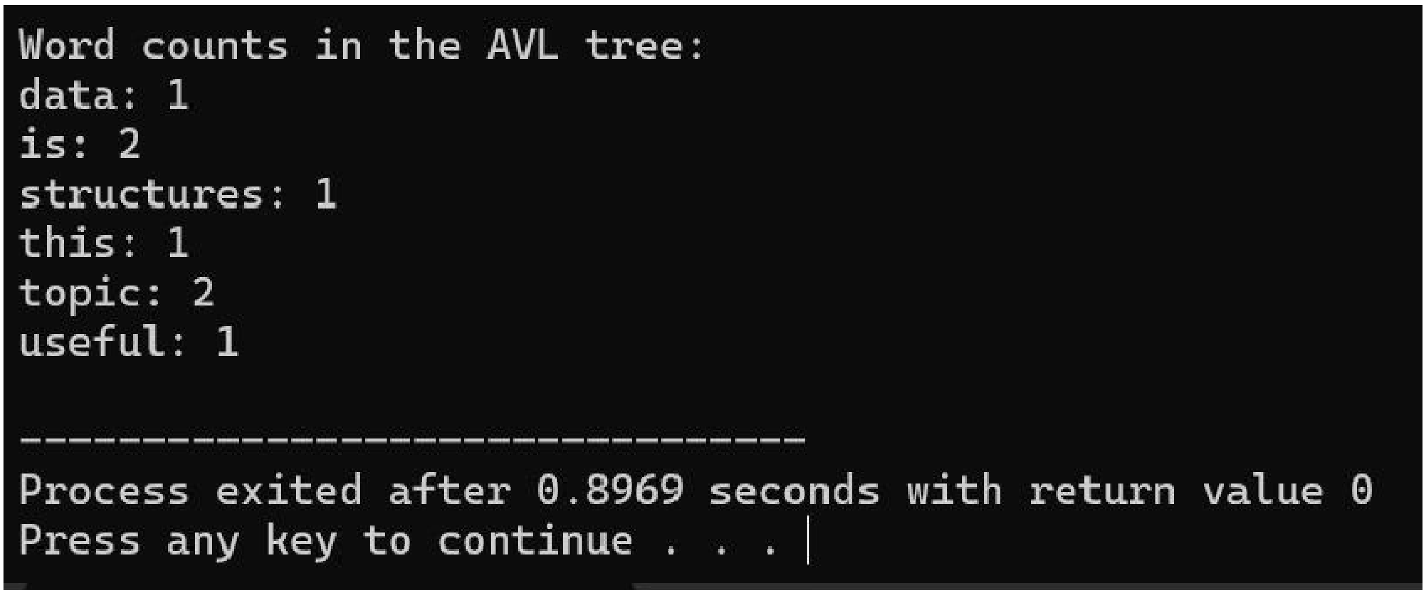
```
Word counts in the AVL tree:
hello: 3
is: 1
this: 1
world: 1

--------------------------------
Process exited after 1.319 seconds with return value 0
Press any key to continue . . .
```



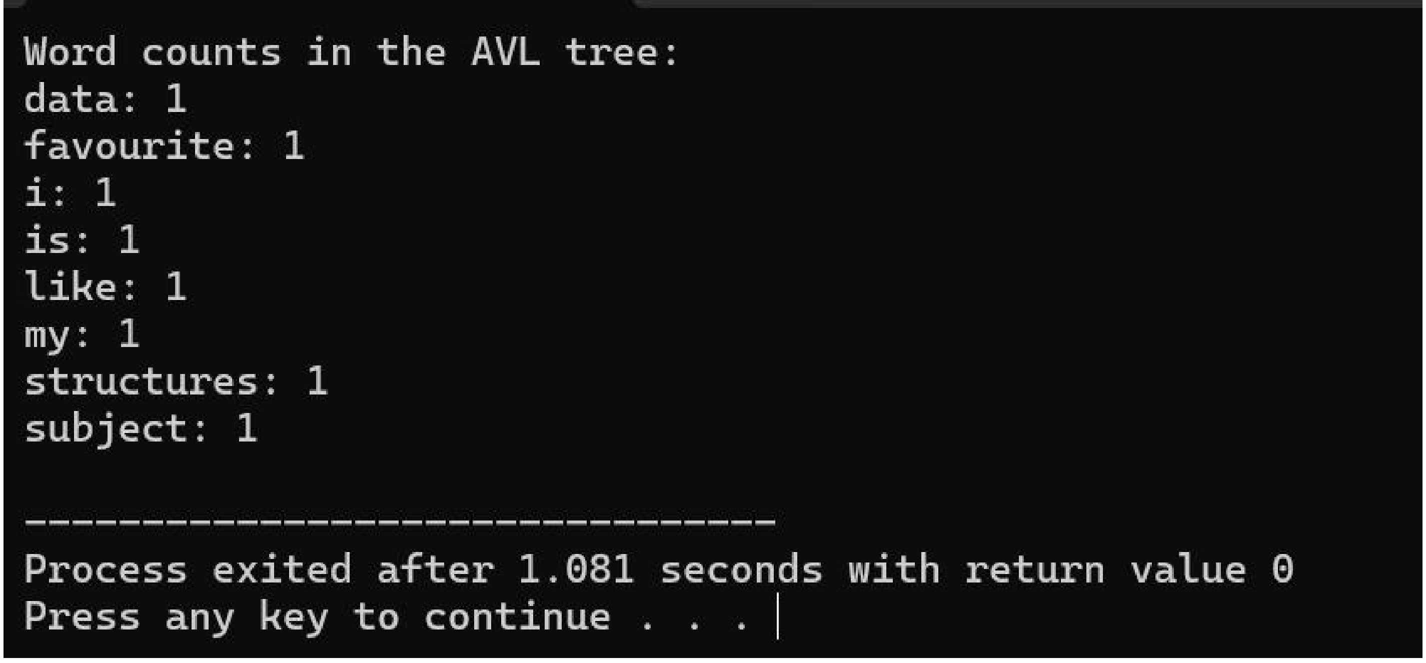
```
Word counts in the AVL tree:
done: 1
i: 1
is: 1
like: 1
my: 1
project: 2
thank: 1
this: 1
you: 1

--------------------------------
Process exited after 1.075 seconds with return value 0
Press any key to continue . . .
```



```
Word counts in the AVL tree:
data: 1
is: 2
structures: 1
this: 1
topic: 2
useful: 1

--------------------------------
Process exited after 0.8969 seconds with return value 0
Press any key to continue . . .
```



```
Word counts in the AVL tree:
data: 1
favourite: 1
i: 1
is: 1
like: 1
my: 1
structures: 1
subject: 1

--------------------------------
Process exited after 1.081 seconds with return value 0
Press any key to continue . . .
```

CONCLUSION

In this project, we developed a compact AVL tree application to efficiently count the occurrences of words in a text document. By utilizing an AVL tree, we ensured that the data structure remains balanced, allowing for quick insertions and lookups, which is essential for processing large volumes of text.