Optimizing Sequential Morse Code Decoding Using Rooted Binary Tree Transversal

Peter Emmanuel Suwardy - 13525125
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
E-mail: peteresuwardy@gmail.com , 13525125@std.stei.itb.ac.id

***Abstract*—This electronic document is a "live" template and already defines the components of your paper [title, text, heads, etc.] in its style sheet. **CRITICAL: Do Not Use Symbols, Special Characters, or Math in Paper Title or Abstract.* (*Abstract*)**

Keywords—component; formatting; style; styling; insert (key words)

I. Introduction

Morse code is a system to represent letters of the alphabet, numbers, and punctuation marks using a series of dots, dashes and spaces. There are two Morse code system, one is invented by an American artist and inventor Samuel F.B. Morse in the United States during the 1830s, and the other is called the International Morse Code, which was devised by a conference of European nations in 1851 to account for letters with diacritic marks. Although both systems are similar, the International Morse Code is simpler and more precise and complete. The original morse code used a patterns of dots and spaces while the International one uses combinations of dots and dashes of constant length.

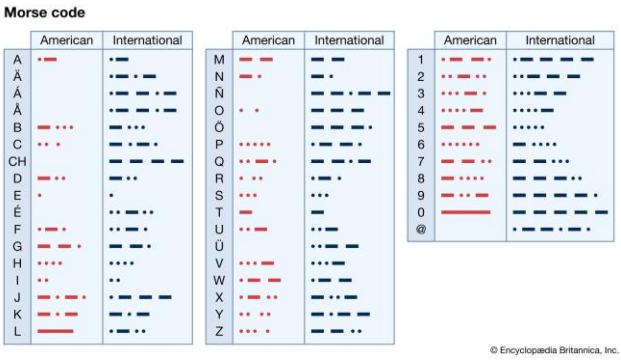

Image 1.1 Comparison of the two morse code system
Source: https://www.britannica.com/topic/International-Morse-Code, accessed on 15/6/2026

International Morse Code was used back in World War II and also in the Korean and Vietnam wars. It was mainly used by the shipping industries and communications in the military. In today's day and age, morse code can be used as an emergency survival communication, military and tactical operations, an educational tool, cybersecurity, and also an assistive technology for the disabilities.

In the modern computing system, processing serial data streams like Morse code is different with the standard keyboard whereas each keypress maps instantly to a specific character. To handle this, traditional software allocates a temporary memory buffer, waits for the mark such as a pause or a space, seals the stored buffer and runs a search algorithm to translate into a readable character. However, this approach is inefficient, while the signals are being transmitted, the processor is idle until the sequential check one by one. In the worst case scenario, the letter could be found at the end of the list.

This paper aims to apply Rooted Binary Tree structure to optimize sequential decoding process or Morse code. Using this approach, the decoding process is expected to eliminate post-stream latency and minimizing workload on system. Other than that, this paper will use an algorithmic complexity analysis using Big-O notation to provide mathematical proof regarding the performance efficiency of the tree structure compared with linear search.

II. Basic Theories

A. Graph and Tree Theorem

A.1 Graph

Graph is commonly used to represents discrete objects and the relations between them. A graph G is defined as G = (V, E) in which V is a non-empty set of the vertices $\{v_1, v_2, .., v_n\}$ and E is a set of edges that connects a pair of vertex $\{e_1, e_2, .., e_n\}$. Based on the directional orientation of the edges, a graph can be classified as undirected graph which is a graph whose edges do not have a directional orientation and directed graph or digraph whose every edge has a directional orientation.

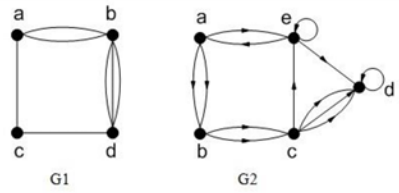

Image 2.1 Illustration of undirected graph G1 and digraph G2
Source:
https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2025-2026/20-Graf-Bagian1-2026.pdf, accessed on 15/6/2026

In formal graph terminology, data can be analyzed or evaluated through the graph configuration using the properties of paths and cycles. A path with the length n from an initial vertex v_0 to a destionation v_n in graph G is an alternating sequence of vertices and edges of the form $v_0, e_1, v_1, e_2, v_2, \ldots, v_{n-1}, e_n, v_n$ such that $e_1 = (v_0, v_1)$, $e_2 = (v_1, v_2)$, …, $e_n = (v_{n-1}, v_n)$ are edges of the graph G. Meanwhile, a cycle or a circuit is a path that starts and terminates at the exact same vertex, meaning $v_0 = v_n$.

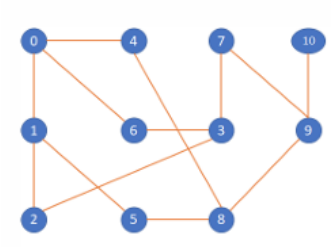

Image 2.2 Illustration of undirected graph to demonstrate path and circuit
Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2025-2026/20-Graf-Bagian1-2026.pdf, accessed on 15/6/2026

For example, the graph shown above has a path from the vertex 0 to destination 10 with the length of 5 that goes through the edges (0,6), (6,3), (3,7), (7,9), (9,10). The graph also contains multiple circuits, for example, one of the path that goes through these vertex 0, 4, 8, 5, 1, 0 is a circuit with the length of 5.

A.2 Tree

A tree is an undirected graph, that is connected and does not contain any circuit. According to the foundational graph theorem of G = (V, E) is a simple undirected graph with n vertices, a tree can also be defined if there exist a unique simple path between two distinct vertices in G, and a connected graph G contains m = n – 1 amount of edges.

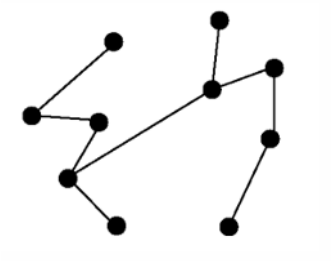

Image 2.3 Simple illustration of a tree
Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2025-2026/23-Pohon-Bag1-2026.pdf, accessed on 15/6/2026

A tree in which one of the vertex acts as a root and its edges are given a direction so that it became a digraph is called a rooted tree. There are multiple important terminologies within rooted tree such as:

1. Parent, Child, and Sibling
   If there is a directed edge from the vertex u to vertex v and also from vertex u to vertex w, then u is defined as the parent of v and w, while v and w is called the children of u. The vertices u and v themselves are sibling as they share the same parent vertex.
2. Degree, Internal Vertices, and Leaves
   A degree of a vertex is decided by how many subtree or children that vertex has. If a vertex has at least one child, it is called an internal vertices, else it will be called a leaf.
3. Path, Level, and Height
   A vertex is connected to another vertex if there exist a path that goes through an alternating sequence of vertices and edges. The exact depth of any vertex in a tree is tracked by its level, where the root sits at level 0 and the number increase along the rooted path. The height of the tree is taken from the highest or maximum level.

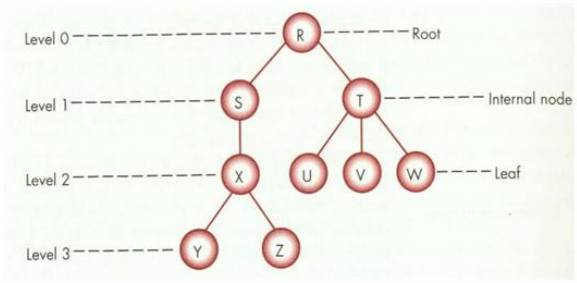

Image 2.4 Illustration of root, levels, internal node and leaf of a tree
Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2025-2026/24-Pohon-Bag2-2026.pdf, accessed on 15/6/2026

One type of rooted tree is binary tree. A binary tree is an m-ary tree with m = 2, which means every vertex can only have a maximum of two children. A binary tree is also an ordered tree, which means the order of its children is important, that's why the left child and the right child must be clearly differentiated.

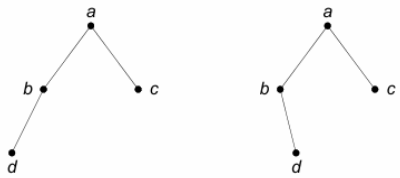

Image 2.5 Illustration of two different binary tree
Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2025-2026/24-Pohon-Bag2-2026.pdf, accessed on 15/6/2026

A binary tree can be classified as a full binary tree if every vertex has exactly two child except the leaves, as a complete binary tree if every level is filled except the last one and the leaves are sorted from left to right, and as a perfect binary tree if every level is completely filled.

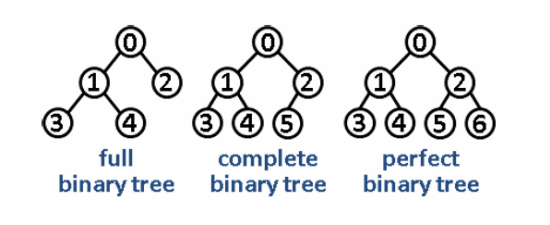

Image 2.6 Illustration of the comparison between full, complete and perfect binary tree
Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2025-2026/24-Pohon-Bag2-2026.pdf, accessed on 15/6/2026

One application of the binary tree is a binary search tree. A binary search tree (BST) is a type of binary tree, in which each vertex has a unique key and follows a specific ordering pattern or property. All vertex inside the left of the subtree's root contain a value less than the root, while all the vertex in the right side of the subtree's root contain a value greater than the root.Using this binary search tree makes searching, insertion and deletion of elements more efficient.

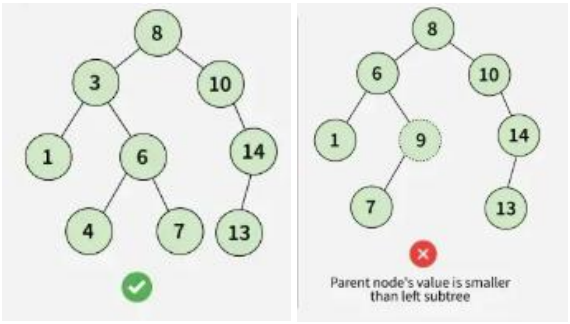

Image 2.7 Illustration of a correct BST (left image) and wrong BST (right image)
Source: https://www.geeksforgeeks.org/dsa/binary-search-tree-data-structure/, accessed on 16/6/2026

After a binary tree is constructed, the elements inside can be systematically visited using 3 different traversal techniques, which is preorder, inorder, and postorder.

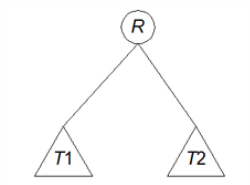

Image 2.8 Illustration for Travesring Binary Search Tree
Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2025-2026/24-Pohon-Bag2-2026.pdf, accessed on 15/6/2026

Preoder traversal starts at the root or R, then traverse to the left subtree or T1 and then traverse to the right subtree T2. Meanwhile inorder traversal starts at the left subtree or T1, then visit the root or R, and finally traverse the right subtree or T2. Lastly, postorder traversal starts with the left subtree or T1, and then traverse to the right subtree or T2, and then visit the root or R.

B. Algorithmic Complexity

An algoritm doesn't just need to be correct, it also needs to be efficient. A way to measure its efficiency is by comparing the time it takes to run the algorithm and also the space or memory it needs. Time complexity or T(n) is measured from the amount of computation steps in an algorithmas a function with input n. There are many operations inside an algorithm, such as read and write, algorithm, assignment, comparison, and etc. Time complexity can be classified into three categories:

- $T_{max}(n)$: Worst-case time complexity, the maximum execution time required.
- $T_{min}(n)$: Best-case time complexity, the minimum execution time required.
- $T_{avg}(n)$: Average-case time complexity, the average execution time required.

Asymptotic Time Complexity expressed using Big-O notation (O) is used to mathemathically quantify these variations. This approach focuses on how the execution time grows as input size approaches infinity. Big-O notation establish a theoritical upper bound on the growth rate. In data lookup operations, most algorithm usually fall into a contant time O(1), logarithmic time O(log n) or a linear time O(n).

III. . ANALYSIS AND DISCUSSION

First, the latin alphabet need to be translated first into the form of a binary tree by following the pattern of dots and dashes. The tree will expand into the left subtree if the pattern is a dot (·), and it will expand to the right subtree if the character is a dash (-).

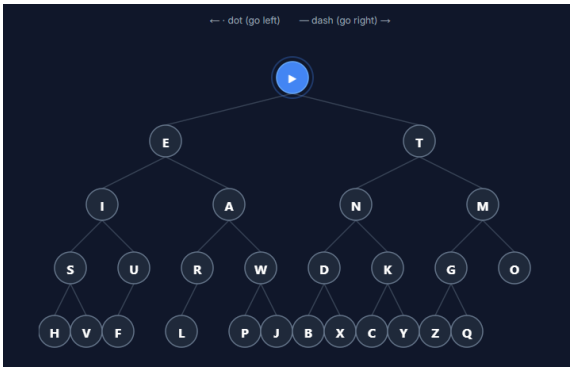

Image 3.1 Morse code in the form of binary tree
Source: https://morse.cool/tree

The core application is engineered in C programming language, using a pointer operation on a dynamically allocated tree structure. To compare the time between the two algorithm, the program will use <time.h>, and both of the structure need to be defined first. Other than that, an array of morseMapping which consist of both characters and binary is needed for the linear searching algorithm.

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

// =====================
// STRUCTS & DEFINITIONS
// =====================

// 1. Binary Tree Node Structure
typedef struct Node {
    char data;
    struct Node* left;  // Dot '.'
    struct Node* right; // Dash '-'
} Node;

// 2. Linear Search Mapping Structure
typedef struct {
    char character;
    const char* morseCode;
} MorseMapping;

const MorseMapping linearLookupTable[26] = {
    {'A', ".-"},   {'B', "-..."}, {'C', "-.-."}, {'D', "-.."},  {'E', "."},
    {'F', "..-."}, {'G', "--."},  {'H', "...."}, {'I', ".."},   {'J', ".---"},
    {'K', "-.-"},  {'L', ".-.."}, {'M', "--"},   {'N', "-."},   {'O', "---"},
    {'P', ".--."}, {'Q', "--.-"}, {'R', ".-."},  {'S', "..."},  {'T', "-"},
    {'U', "..-"},  {'V', "...-"}, {'W', ".--"},  {'X', "-..-"}, {'Y', "-.--"},
    {'Z', "--.."}
};
```

Image 3.2 Header and definitions part
Source: Author's implementation

```c
// ==========================
// BINARY TREE IMPLEMENTATION
// ==========================

Node* createNode(char character) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    if (newNode != NULL) {
        newNode->data = character;
        newNode->left = NULL;
        newNode->right = NULL;
    }
    return newNode;
}

Node* initializeMorseTree() {
    Node* root = createNode('\0'); // Root/Idle state

    // Level 1
    root->left = createNode('E');  root->right = createNode('T');
    // Level 2
    root->left->left = createNode('I'); root->left->right = createNode('A');
    root->right->left = createNode('N'); root->right->right = createNode('M');
    // Level 3
    root->left->left->left = createNode('S'); root->left->left->right = createNode('U');
    root->left->right->left = createNode('R'); root->left->right->right = createNode('W');
    root->right->left->left = createNode('D'); root->right->left->right = createNode('K');
    root->right->right->left = createNode('G'); root->right->right->right = createNode('O');
    // Level 4
    root->left->left->left->left = createNode('H'); root->left->left->left->right = createNode('V');
    root->left->left->right->left = createNode('F'); root->left->right->left->left = createNode('L');
    root->left->right->right->left = createNode('P'); root->left->right->right->right = createNode('J');
    root->right->left->left->left = createNode('B'); root->right->left->left->right = createNode('X');
    root->right->left->right->left = createNode('C'); root->right->left->right->right = createNode('Y');
    root->right->right->left->left = createNode('Z'); root->right->right->left->right = createNode('Q');

    return root;
}
```

Image 3.3 Binary tree implementation code
Source: Author's implementation

This part functions as the tree node construction and also for the alphabet path mapping. This module initializes memory dynamically with malloc and also hardcoded the tree hierarchy structural matching the Morse code depth layers.

```c
// Tree Decoding
void decodeWithTree(Node* root, const char* morseSequence) {
    Node* cursor = root;
    int i = 0;
    while (morseSequence[i] != '\0') {
        if (morseSequence[i] == '.') {
            if (cursor->left != NULL) cursor = cursor->left;
        } else if (morseSequence[i] == '-') {
            if (cursor->right != NULL) cursor = cursor->right;
        } else if (morseSequence[i] == ' ' || morseSequence[i] == '/') {
            if (cursor != root && cursor->data != '\0') {
                volatile char sink = cursor->data; // Prevent compiler from optimizing away
            }
            cursor = root;
        }
        i++;
    }
    if (cursor != root && cursor->data != '\0') {
        volatile char sink = cursor->data;
    }
}
```

Image 3.5 Binary Tree Decoding
Source: Author's implementation

This tree parsing routine uses a cursor pointer to go left or right children addresses by checking if it's a dot or a dash.

```c
// ============================
// LINEAR SEARCH IMPLEMENTATION
// ============================
void decodeWithLinearSearch(const char* morseSequence) {
    char currentToken[10];
    int tokenIdx = 0;
    int i = 0;

    while (morseSequence[i] != '\0') {
        if (morseSequence[i] != ' ' && morseSequence[i] != '/') {
            currentToken[tokenIdx++] = morseSequence[i];
        } else {
            currentToken[tokenIdx] = '\0';
            if (tokenIdx > 0) {
                // Sequential Array Search Routine
                for (int j = 0; j < 26; j++) {
                    if (strcmp(linearLookupTable[j].morseCode, currentToken) == 0) {
                        volatile char sink = linearLookupTable[j].character; // Found
                        break;
                    }
                }
            }
            tokenIdx = 0;
        }
        i++;
    }

    // Process trailing token
    currentToken[tokenIdx] = '\0';
    if (tokenIdx > 0) {
        for (int j = 0; j < 26; j++) {
            if (strcmp(linearLookupTable[j].morseCode, currentToken) == 0) {
                volatile char sink = linearLookupTable[j].character;
                break;
            }
        }
    }
}
```

Image 3.6 Linear search program
Source: Author's implementation

This implementation process the incoming Morse code stream character by character using a loop, extracting into a buffer array. If the program meets the character space ( ) or slash (/), the function will appened a null-terminator to solidify the token and then using another loop to sequentially search in the linearLookupTable

```c
// ==========================
// MAIN BENCHMARKING
// ==========================
int main() {
    Node* morseTree = initializeMorseTree();

    // The base Morse sequence used for profiling
    const char* pattern = "- .... . / --.- ..- .. -.-. -.- / -... .-. --- .-- -. / ...- --- -..- / .--- ..- -- .-
    // Dynamically construct an ultra-long sequence to allow CPU clock measurement accuracy
    size_t repetitions = 50000;
    size_t patternLen = strlen(pattern);
    char* largeWorkload = (char*)malloc(patternLen * repetitions + 1);
    largeWorkload[0] = '\0';
    for (size_t r = 0; r < repetitions; r++) {
        strcat(largeWorkload, pattern);
    }
```

Image 3.7 Main code (1)
Source: Author's implementation

This part of the code functions to make the Morse code being tested multiple thousand time and make the workload larger. That way, the time difference to compare the binary tree system and linear search can be more clear.

```c
    printf("==================================================\n");
    printf("   ALGORITHMIC PERFORMANCE LOOKUP EXPERIMENT\n");
    printf("==================================================\n");
    printf("Input Morse Code Pattern:\n%s\n\n", pattern);

    // Live execution output demo to print what the pattern translates to
    printf("Decoded Translation Verification:\n");
    Node* demoCursor = morseTree;
    int k = 0;
    while (pattern[k] != '\0') {
        if (pattern[k] == '.') {
            if (demoCursor->left != NULL) demoCursor = demoCursor->left;
        } else if (pattern[k] == '-') {
            if (demoCursor->right != NULL) demoCursor = demoCursor->right;
        } else if (pattern[k] == ' ' || pattern[k] == '/') {
            if (demoCursor != morseTree && demoCursor->data != '\0') {
                printf("%c", demoCursor->data);
            }
            demoCursor = morseTree;
            if (pattern[k] == '/') printf(" ");
        }
        k++;
    }
    printf("\n--------------------------------------------------\n");
    printf("Workload String Size: %lu characters\n\n", (unsigned long)strlen(largeWorkload));
```

Image 3.8 Main code (2)
Source: Author's implementation

This part of the main code will print out the binary and the translated version and also the workload ran by each algorithm.

```c
    // ------------------------------------------------------
    // BENCHMARK 1: Linear Scanning Array Search
    // ------------------------------------------------------
    clock_t startLinear = clock();
    decodeWithLinearSearch(largeWorkload);
    clock_t endLinear = clock();
    double timeLinear = ((double)(endLinear - startLinear)) / CLOCKS_PER_SEC;
    printf("1. Linear Array Sequential Search\n");
    printf("   Total Execution Time: %f seconds\n\n", timeLinear);

    // ------------------------------------------------------
    // BENCHMARK 2: Optimized Binary Tree Traversal
    // ------------------------------------------------------
    clock_t startTree = clock();
    decodeWithTree(morseTree, largeWorkload);
    clock_t endTree = clock();
    double timeTree = ((double)(endTree - startTree)) / CLOCKS_PER_SEC;
    printf("2. Ordered Binary Tree Cursor Shifting\n");
    printf("   Total Execution Time: %f seconds\n\n", timeTree);

    // ------------------------------------------------------
    // EFFICIENCY CALCULATION
    // ------------------------------------------------------
    printf("--------------------------------------------------\n");
    printf("Performance Gain: The Binary Tree was %.2fx FASTER\n", timeLinear / timeTree);
    printf("==================================================\n");

    free(largeWorkload);
    freeTree(morseTree);
    return 0;
}
```

Image 3.9 Main code (3)
Source: Author's implementation

This part of the main code will start running both algorithm and outputting the time needed to run that many workload. The time taken by each will then be compared and will be calculated on which one is faster by how much.

A. Testing Results

The program that have been explained above will be use to multiple different test cases:

1. Morse Input: .
   Translation: E

```
==================================================
   ALGORITHMIC PERFORMANCE LOOKUP EXPERIMENT
==================================================
Input Morse Code Pattern:
.

Decoded Translation Verification:
E
--------------------------------------------------
Workload String Size: 100000 characters

1. Linear Array Sequential Search
   Total Execution Time: 0.001000 seconds

2. Ordered Binary Tree Cursor Shifting
   Total Execution Time: 0.000000 seconds

--------------------------------------------------
Performance Gain: The Binary Tree was infx FASTER
==================================================
```

Image 3.10 Test Case 1
Source: Author's implementation

2. Morse Input: -.-. --- -.. .
   Translation: CODE

```
==================================================
   ALGORITHMIC PERFORMANCE LOOKUP EXPERIMENT
==================================================
Input Morse Code Pattern:
-.-. --- -.. .

Decoded Translation Verification:
CODE
--------------------------------------------------
Workload String Size: 750000 characters

1. Linear Array Sequential Search
   Total Execution Time: 0.005000 seconds

2. Ordered Binary Tree Cursor Shifting
   Total Execution Time: 0.002000 seconds

--------------------------------------------------
Performance Gain: The Binary Tree was 2.50x FASTER
==================================================
```

Image 3.11 Test Case 2
Source: Author's implementation

3. Morse Input: …. . .-.. --- / .-- --- .-. .-.. -..
   Translation: HELLO WORLD

```
==================================================
   ALGORITHMIC PERFORMANCE LOOKUP EXPERIMENT
==================================================
Input Morse Code Pattern:
.... . .-.. .-.. --- / .-- --- .-. .-.. -..

Decoded Translation Verification:
HELLO WORLD
--------------------------------------------------
Workload String Size: 2200000 characters

1. Linear Array Sequential Search
   Total Execution Time: 0.019000 seconds

2. Ordered Binary Tree Cursor Shifting
   Total Execution Time: 0.004000 seconds

--------------------------------------------------
Performance Gain: The Binary Tree was 4.75x FASTER
==================================================
```

Image 3.12 Test Case 3
Source: Author's implementation

4. Morse Input: - … . / --.- ..- .. -.-. -.- / -... .-. --- .-- -. / ..-. --- -..- / .--- ..- -- .--. ... / --- ...- . .-. / - .... . / .-.. .- --.. -.-- / -.. --- --. /
Translation: THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG

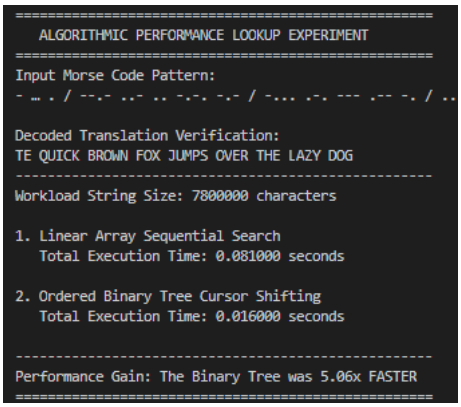

Image 3.13 Test Case 4
Source: Author's implementation

5. Morse Input: --.. --.. --.. --.. --.. --.. --.. --.. --.. --..
Translation: ZZZZZZZZZZ

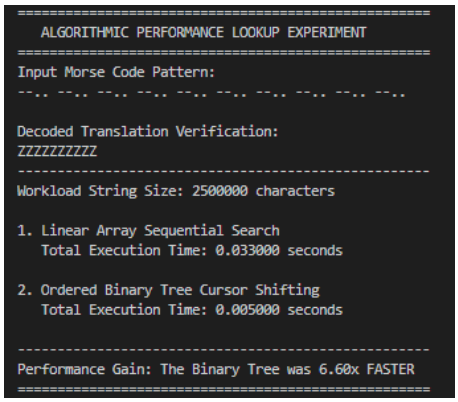

Image 3.14 Test Case 5
Source: Author's implementation

6. Morse Input: . . . . . .
Translation: EEEEEEEEEE

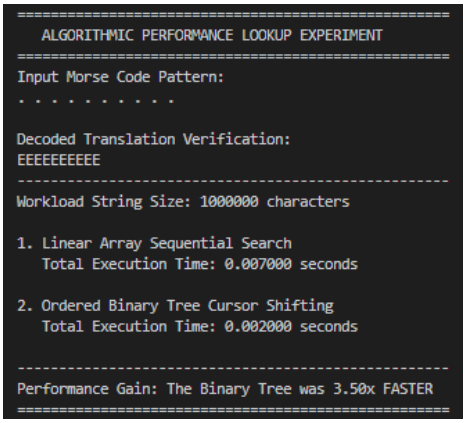

Image 3.15 Test Case 6
Source: Author's implementation

B. Analysis

Test Cases	Linear Runtime	Tree Runtime
Minimum Boundary (E)	0.001000 s	0.000000 s
Single Word (CODE)	0.005000 s	0.002000 s
Standard Phrase (HELLO WORLD)	0.019000 s	0.004000 s
Maximum Depth/Worst Case (ZZZZZZZZZZ)	0.033000 s	0.005000 s
Minimum Depth/Best Case (EEEEEEEEEE)	0.007000 s	0.002000 s
Full Alphabet Pangram (THE QUICK BROWN...)	0.081000 s	0.016000 s

Table 3.16 Test Data

From all the test data that have been gathered, it can be seen in all the test cases that the binary tree runtime is always faster than the linear runtime. The binary runtime is faster by an estimation around 2.5x up to more than 6x than the linear search runtime. For example, test case 6 is testing Minimum Depth Case, where E is at the very first layer of the tree and make the runtime fast, while test case 5 is testing the Maximum Depth Case, where Z is at the very end of a tree, yet it is still faster than the linear search runtime.

For the sequential array search algorithm, the search time complexity is strictly linear, and can be denoted as O(n), where n represents the total size of the alphabet being tested. This algorithm has its best case if the alphabet is the first of the look up table, and its worst case is the very back of the lookup table. In contrast, the ordered Binary Search Tree works under a strictly bounded time complexity of O(h), where h is the maximum heigh of the tree layout. Because the test case uses the standard alphabet, the maximum height of the tree is $h \leq 4$.

IV. Conclusion

Based on the results and algorithm complexity assessments, it can be concluded that the optimized Binary Search Tree implementation is vastly a more superior structural alternative to the standard sequential linear array mapping for decoding Morse code. By transforming from a loop dependent array search with linear time complexity into an active direct pointer with effective

constant time complexity, the binary tree cuts down on redundant processing operations. The binary tree systematically outspace the linear lookup framework by an efficiency factor scaling from 2.5x up to 6.60x.Although the hierarchical node require a slightly larger memory space to hold its child pointers, this is a good tradeoff by the significant drop in the runtime execution.

ACKNOWLEDGMENT

The author would like to express their gratitude and appreciation to:

1. Almight God, for His blessings and guidance,
2. The author's parents, for the encouragement
3. The lecturers of Discrete Mathematics (IF1220) course, for the knowledge and insights
4. All other parties who have directly or indirectly supported the author during the process of writing this paper

REFERENCES

[1] The Editors of Encyclopaedia Britannica, "International Morse Code," Encyclopedia Britannica, 2026. [Online]. Available: https://www.britannica.com/topic/International-Morse-Code, accessed on 15 June 2026.

[2] R. Munir, "Graf Bagian 1," Lecture Notes on IF2120 Matematika Diskrit, STEI-ITB, 2026. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2025-2026/20-Graf-Bagian1-2026.pdf, accessed on 15 June 2026.

[3] R. Munir, "Pohon Bagian 1," Lecture Notes on IF2120 Matematika Diskrit, STEI-ITB, 2026. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2025-2026/23-Pohon-Bag1-2026.pdf, accessed on 15 June 2026.

[4] R. Munir, "Pohon Bagian 2," Lecture Notes on IF2120 Matematika Diskrit, STEI-ITB, 2026. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2025-2026/24-Pohon-Bag2-2026.pdf, accessed on 15 June 2026.

[5] R. Munir, "Kompleksitas Algoritma Bagian 1," Lecture Notes on IF2120 Matematika Diskrit, STEI-ITB, 2026. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2025-2026/25-Kompleksitas-Algoritma-Bagian1-2026.pdf, accessed on 15 June 2026.

[6] R. Munir, "Kompleksitas Algoritma Bagian 2," Lecture Notes on IF2120 Matematika Diskrit, STEI-ITB, 2026. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2025-2026/26-Kompleksitas-Algoritma-Bagian2-2026.pdf, accessed on 15 June 2026.

[7] GeeksforGeeks, "Binary Search Tree Data Structure," GeeksforGeeks DSA Reference Guide, 2026. [Online]. Available: https://www.geeksforgeeks.org/dsa/binary-search-tree-data-structure/, accessed on 16 June 2026.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 19 Juni 2026

Peter Emmanuel Suwardy, 13525125