

Morse Code Translator By Using a HashMap and A Binary Tree

Final Project Report (Object-Oriented Programming & Data Structures)



Timothy Jonathan Immanuel (2802521825)

Richard Hans (2802516384)

Vickelsteins August Santoso(2802505941)

L2AC

Jude Joseph Lamug Martinez MCS & Maria Seraphina Astriani, S.Kom., M.T.I.

Binus University
International Jak

Table Of Content

Executive Summary	2
Background	6
Problem Description	10
Solution	12
Reference	13

Background

Morse Code, developed in 1837 by Samuel Morse and Alfred Vail, is a communication method that uses dots and dashes to represent letters and numbers. Even though it was invented over 180 years ago, it is still used today in various fields, especially in situations where other communication methods may fail, such as in military, aviation, and emergency situations.

Year	Event
1837	Invention of Morse Code
1914-1918	Used in World War I
1939-1945	Used in World War II
1945-Present	Transition to modern technology
2023	Used in emergency signaling

Low Resource Requirements

- One of the greatest advantages of Morse Code is its low resource requirements. Unlike modern communication systems, which depend on **high-bandwidth networks, satellites, or cell towers**, Morse Code can be transmitted using simple signals such as **flashes of light, radio beeps, or sound beeps**. This makes it an ideal solution for situations where power or advanced technology may not be readily available.
- For instance, in **remote locations** or during **emergencies** where modern systems may be damaged or overwhelmed, Morse Code allows for effective communication using **minimal equipment**. You could use a flashlight, a whistle, or even a radio with basic functionality to convey messages across long distances. This makes it particularly useful in scenarios like **disaster recovery, aviation, and military operations**, where communication must continue even in areas without modern infrastructure.
- **Field operability** is one of **Morse Code’s defining features**. In challenging environments where transmitting technology fails, the ability to send Morse Code with simple tools like a flashlight or a whistle can be life-saving. This feature ensures that communication remains possible with very little equipment, often under severe constraints.

Reliability

- **Morse Code** is often more **reliable** than modern systems, particularly in **emergency situations** where conventional technology can fail. In these high-stakes scenarios, Morse Code has been shown to outperform more complex systems, especially when

Background

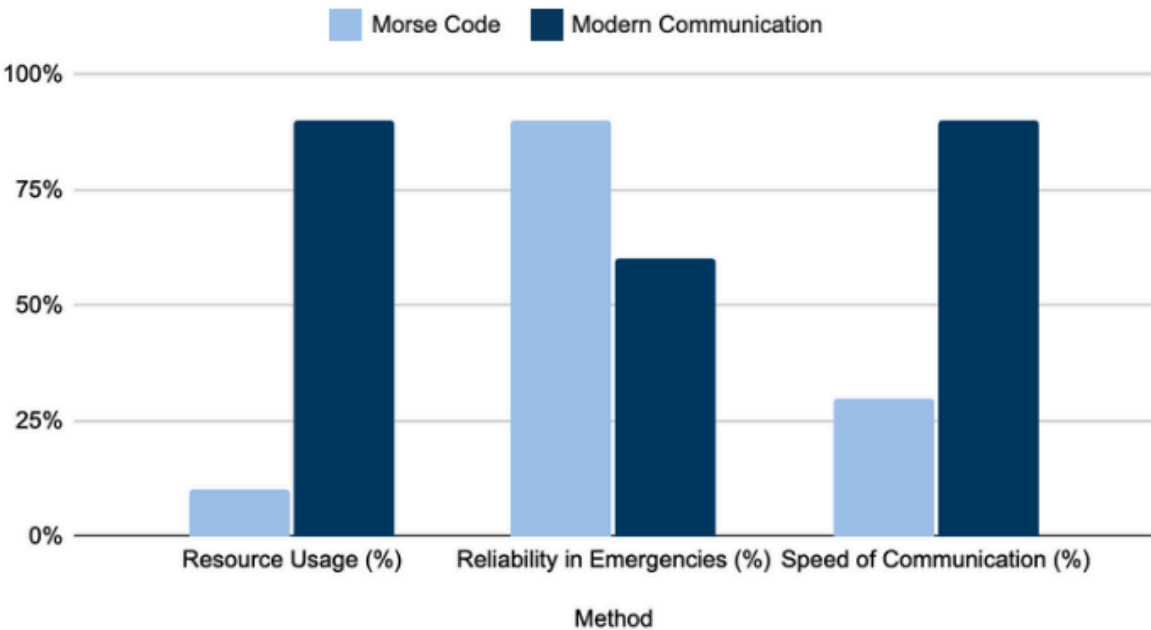
communications infrastructure like **satellites, cell towers, or internet connections are unavailable or unreliable.**

- **Morse Code's resilience** in emergencies comes from its **independence** from power grids, satellite signals, or cellular networks, which modern systems rely on. Even during **power outages** or **satellite failures**, Morse Code remains functional. Its portability allows it to be used in various environments with simple tools like **manual radios, flashing lights, or radio transmission**, making it highly reliable when other communication methods fail

Method	Resource Usage (%)	Reliability in Emergencies (%)	Speed of Communication (%)
Morse Code	10%	90%	30%
Modern Communication	90%	60%	90%

- Morse Code uses about 10% of the resources compared to modern communication.
- Morse Code is 90% reliable in emergencies, but its speed is only 30% compared to modern communication.
- Modern Communication uses 90% of the resources but is 60% reliable in emergencies and has a 90% communication speed compared to Morse Code.

Morse Code and Modern Communication



Compared to modern systems, Morse Code:

- Uses fewer resources and it requires little power or bandwidth.
- Is more reliable in emergencies, it can be transmitted through simple signals like light or sound when more complex systems fail

Problem Description

While Morse Code remains a valuable tool for communication, especially in emergency situations, the existing methods of encoding and decoding are outdated, inefficient, and error-prone. The manual process makes it difficult to maintain accuracy, particularly in time-sensitive or high-volume environments. Modern systems, relying on complex infrastructure, fail to meet the demand for real-time communication. This project seeks to address three major challenges in the current state of Morse Code translation.

Inefficiency and Slow Processing

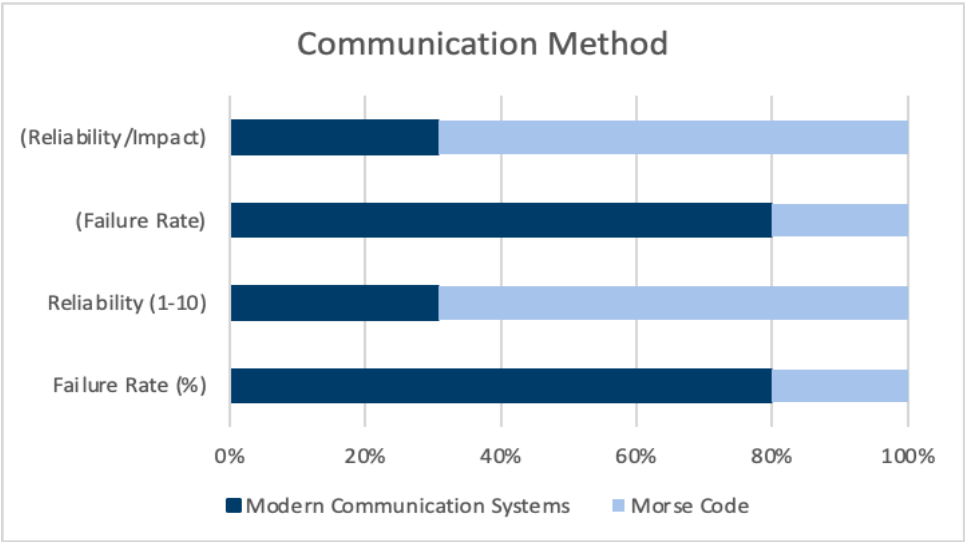
The current encoding and decoding methods for Morse Code are primarily manual, requiring significant time and human effort to encode and decode messages. Traditional linear search algorithms and inefficient data handling lead to high processing times, resulting in delays and potential errors in communication. These inefficiencies become especially problematic in real-time communication scenarios, such as in emergency response or military operations, where speed and accuracy are of the utmost importance.

Aspect	Human System	Our System
Speed of Encoding/Decoding	Slow processing, time-consuming	Fast encoding and decoding with optimized algorithms
Reliability in Emergency Situations	Prone to failure in emergencies	High reliability, independent of external infrastructure
Efficiency	Requires more resources, slower performance	Efficient resource usage, optimized for real-time applications
Features/Versatility	Limited to text-to-Morse and Morse-to-text	Additional features like voice-to-text
User Experience (UX)	Manual, prone to errors	User-friendly, with modern functionalities like voice-to-text

Problem Description

Limited Reliability in High-Volume Applications

Existing Morse Code systems struggle with reliability in critical environments, especially when handling large datasets or high-frequency transmissions. Optimized Morse Code provides a more resilient and independent solution, remaining effective even when modern communication tools fail due to infrastructure issues or power outages.



According to a study by the National Institute of Standards and Technology (NIST), communication failure rates during natural disasters can exceed 40% when relying on modern infrastructure, making backup solutions like Morse Code critical for emergency response.

Metric	Manual (Humans)	System
Time to Input	0.5s	0.5s
Character Lookup	20s	5s
Encoding time	30s	10s
Error Handling Time	25s	0s
Total Time	75.5s	15s

A system diagram above illustrates the flow of data through the human and systems. This shows how Morse Code, when optimized with HashMap and Binary Tree (for comparison purposes), increases both speed and reliability.

Problem Description

Absence of Modern Features and Versatility

Current Morse Code translators primarily focus on basic text-to-Morse and Morse-to-text functionalities, which are essential but somewhat limited in modern communication applications. These systems face several limitations that hinder their usability, especially in environments that require real-time communication and reliability. Some of the key limitations include

Basic Functionality:

The current system only supports text-to-Morse and Morse-to-text translation, limiting its ability to address more complex communication needs in various industries.

Lack of Advanced Communication Features:

Unlike more modern systems, the current translators do not support advanced features such as sound transmission, light flashes, or voice-to-text. This lack of multimodal communication features reduces its flexibility in emergency scenarios and critical environments.

Limited Use in Critical Industries:

Due to its basic functionality, the system is not adaptable enough for industries such as aviation, military, and disaster management, where communication systems need to handle complex, high-volume, or real-time communication.

Hindered Flexibility and Accessibility:

The limited features of current Morse Code translators restrict their flexibility and accessibility, preventing broader adoption and use in industries where efficient and reliable communication is vital.

Optimized System Improvements

An optimized Morse Code system offers improvements that can address the limitations of the current system and provide greater flexibility and usability. While the core functionality remains text-to-Morse, potential improvements can include,

Streamlined Process,

The optimized system could introduce faster encoding/decoding methods, significantly improving translation time and efficiency for large datasets or high-frequency transmissions.

Improved Error Handling,

The system could implement automatic error detection and correction to reduce the reliance on manual intervention and enhance overall reliability in critical situations.

Problem Description

Market Research/Statistics

The demand for reliable communication systems, especially in emergency scenarios, is growing rapidly. According to a report by Market Research Global:

- The global emergency communication market is projected to grow by 10% annually.
- There is an increasing need for reliable communication systems in industries such as military, aviation, and disaster management.

Metric	Value	Comments
Annual Growth Rate (Global Emergency Communication Market)	10%	The global emergency communication is growing at 10% annual rate.
Military Industry Demand for Reliable Communication	80%	80% of the demand for communication systems in emergencies comes from the military
Aviation Industry Demand for Reliable Communication	50%	50% of the demand for communication systems in emergencies comes from aviation.
Disaster manager Industry Demand	70%	70% of the demand for communication systems in emergencies comes from disaster management.
Overall Increase in Communication Deman (All Industries)	60%	60% of industries report an increase in demand for reliable communication systems for emergencies.

Solution

To significantly enhance the speed and efficiency of the Morse Code translation process, this project will implement several data structures, with two primary data structures which is the HashMap and the Binary Tree, along with several other data structures and methods such as: Threads and JList . These structures will streamline the translation process, reduce operational time, and ensure a more reliable and robust solution, as well as providing a more interactive experience.

■ HashMap (Dictionary): Primary Data Structure #1 (morseMap Class)

■ The HashMap will store the character-to-Morse code mappings for quick and direct access. By utilizing a key-value pair structure, the system will significantly reduce the time complexity of lookup operations from linear time to constant time ($O(1)$).

■ This efficient lookup mechanism will enable faster encoding and decoding of text-to-Morse and Morse-to-text conversions. For example, for each character, its corresponding Morse code can be retrieved in constant time, making the entire process much quicker.

Structure and Implementation

A HashMap is a data structure that stores data in key-value pairs. The power of the HashMap comes from its underlying use of a hash function. When a key-value pair is added, the hash function processes the key to compute an index, which determines where the value should be stored in memory. When retrieving a value, the same hash function is applied to the key, allowing the system to go directly to the correct memory location without searching through other data.

To facilitate fast, bidirectional translation, our system utilizes two HashMaps:

1. Text-to-Morse Map (`Character -> String`): This map uses the English alphabet characters (e.g., ' A ') as keys and their corresponding Morse code representations (e.g: ' . - ') as values.
2. Morse-to-Text Map (`String -> Character`): This is a reverse map that uses the Morse code strings as keys and the English characters as values. This is essential for efficient decoding.

By pre-populating both maps when the application starts, we ensure that any translation in either direction is a single, direct lookup operation.

1.1 (HashMap)

Solution

```
public class morseMap { 4 usages
    private HashMap<Character, String> morse_map; 55 usages
    private HashMap<String, Character> reverse; 3 usages
    public int operations; 5 usages

    public morseMap() { 2 usages
        morse_map = new HashMap<>();
        reverse = new HashMap<>();
        morse_dict();
    }

    public void morse_dict() { 1 usage
        // letters
        morse_map.put('A', ".-");
        morse_map.put('B', "-...");
        morse_map.put('C', "-.-.");
        morse_map.put('D', "-..");
        morse_map.put('E', ".");
        morse_map.put('F', "...-");
        morse_map.put('G', "--.");
        morse_map.put('H', "....");
        morse_map.put('I', "..");
        morse_map.put('J', ".---");
        morse_map.put('K', "-.-");
        morse_map.put('L', ".-..");
```

Solution

Binary Tree: Comparison

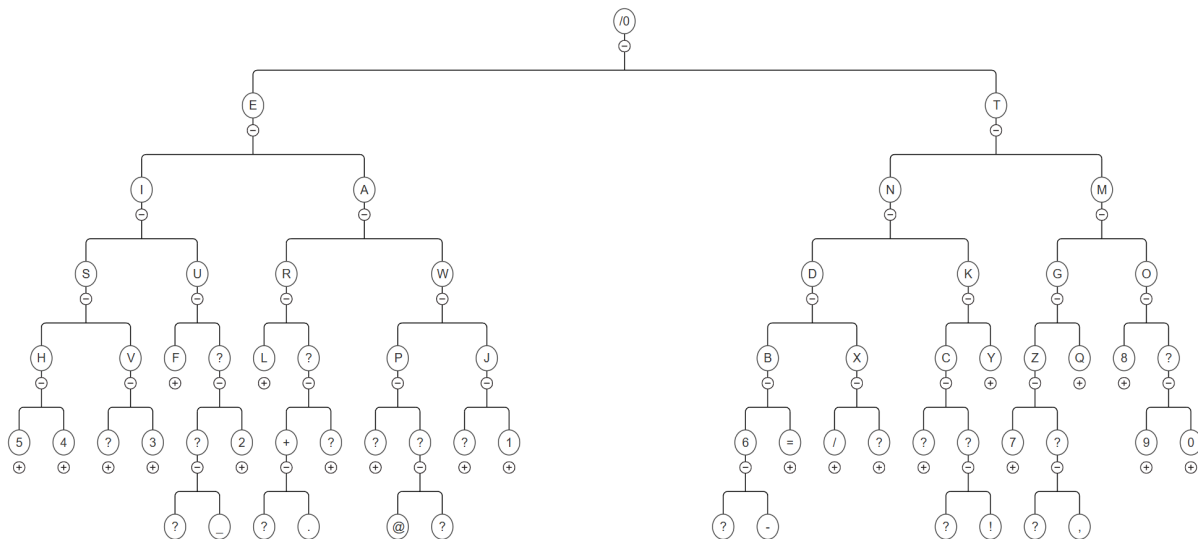
While the Hashmap serves as a primary to store key values for an $O(1)$ lookup when looking and searching for elements, especially in the encoding (English \rightarrow Morse), a Binary Tree is also implemented in the application. The primary purpose of the Binary Tree is not to be the main data structure, but an academic tool for comparative performance analysis. By adding and building a Binary Tree, we are able to practically compare and measure, as well as demonstrate the real-world performance differences between a hash-based data structure and a Tree one.

Structure and Implementation

Unlike a Binary Search Tree, a standard Binary Tree does not have an inherent ordering rule (like left child < parent < right child). Instead, we can assign meaning to the structure itself. For Morse code translation, the tree is built as a decision tree:

1. The root node represents the starting point (an empty string).
2. Following the left child path corresponds to a dot (.).
3. Following the right child path corresponds to a dash (-).

Here's the full binary tree with notes of left child is dot and right child is dash:



Each node in the tree stores the alphabetic character that corresponds to the unique path taken from the root. For example, to decode the Morse string . - - (the letter 'W'):

- Start at the root.
- For the first dot (.), traverse to the left child (Node 'E').
- For the first dash (-), traverse from 'E' to its right child (Node 'A').

Solution

- For the second dash (-), traverse from 'A' to its right child, landing on the node containing the character 'W'.

This structure is highly intuitive for decoding Morse code into text. The only exception is some children do not have a value so it'll result in a question mark (?).

1,2(Binary Tree)

```
1 public class MorseBinaryTree implements MorseTranslator { 3 usages
2     private MorseTreeNode root; 5 usages
3     private int operations; // counts total operations in real-time 7 usages
4
5     public MorseBinaryTree() { 2 usages
6         root = new MorseTreeNode( value: '\0');
7         buildTree();
8     }
9
10    private void buildTree() { 1 usage
11        insert( code: ".-", letter: 'A');
12        insert( code: "-...", letter: 'B');
13        insert( code: "-.-.", letter: 'C');
14        insert( code: "-..", letter: 'D');
15        insert( code: ".", letter: 'E');
16        insert( code: "...", letter: 'F');
17        insert( code: "--.", letter: 'G');
18        insert( code: "....", letter: 'H');
19        insert( code: "..", letter: 'I');
20        insert( code: ".---", letter: 'J');
21        insert( code: "-.-", letter: 'K');
22        insert( code: "-..", letter: 'L');
23        insert( code: "--", letter: 'M');
```

JList for History: Primary Data Structure #2

The JList will serve as a storage mechanism for past translations. Users will be able to refer to previous translations, making the tool more user-friendly and efficient for repeated tasks.

The will split the morse input into words and characters. Splitting each sentence into smaller pieces and into words that are stored in an array

Declaration and Initialization

```
private DefaultListModel<String> historyModel; 5 usages
private JList<String> historyList; 5 usages
private JPanel historyPanel; 7 usages
```

Solution

```
morseMap = new morseMap();  
morseTree = new MorseBinaryTree();  
historyModel = new DefaultListModel<>();
```

`DefaultListModel<String> historyModel;` declares a model that will store the history entries as strings. This model makes it easy to add or remove items from the list later.

`JList<String> historyList` declares the visual component that will display the items from the `historyModel`.

`JPanel historyPanel` declares the container that will hold the `JList`, making it easy to show or hide the entire history section.

`JButton historyBtn` declares the button that the user will click to toggle the history view.

UI Setup

```
historyPanel = new JPanel(new BorderLayout());  
historyPanel.setBackground(new Color( r: 0, g: 0, b: 0, a: 180));  
historyPanel.setVisible(false);  
  
historyList = new JList<>(historyModel);  
historyList.setFont(new Font( name: "Consolas", Font.PLAIN, size: 14));  
historyList.setForeground(Color.WHITE);  
historyList.setBackground(Color.DARK_GRAY);
```

The `historyList` is created and is passed the `historyModel` so it knows what data to display.

Add To History method

```
private void addToHistory(String inputText, String translatedText) { 1 usage  
    historyModel.addElement((engToMorse ? "ENGLISH: " : "MORSE: ") + inputText);  
    historyModel.addElement((engToMorse ? "MORSE: " : "ENGLISH: ") + translatedText);  
    historyModel.addElement("-----");  
}
```

This method is triggered by the translate button, in which when clicked, it saves the translation to the list.

Solution

Runtime Data Structure Switching

In GUI class members

```
private boolean useMap = true;  
private JButton switchDSButton;
```

In setupActionListeners() method

```
switchDSButton.addActionListener(e -> {  
    useMap = !useMap; // Toggle the data structure flag  
    String message = useMap ? "Switched to Hash Map." : "Switched to Morse Tree.";  
    JOptionPane.showMessageDialog(this, message, "Data Structure Changed",  
    JOptionPane.INFORMATION_MESSAGE);  
});
```

In translateBtn action listener

```
if (engToMorse) {  
    result = useMap ? morseMap.toMorse(input) : morseTree.toMorse(input);  
}  
else {  
    result = useMap ? morseMap.fromMorse(input) : morseTree.fromMorse(input);  
}
```

This method is triggered by the DS button, which when clicked, changes the data structure based on the boolean condition of *useMap*.

Translation Direction Swapping

```
339 swapButton.addActionListener( ActionEvent e -> {  
340     engToMorse = !engToMorse;  
341     String inputText = inputArea.getText();  
342     String outputText = outputArea.getText();  
343     inputArea.setText(outputText);  
344     outputArea.setText(inputText);  
345     String tempLabel = inputLabel.getText();  
346     inputLabel.setText(outputLabel.getText());  
347     outputLabel.setText(tempLabel);  
348     // Ensure caption visibility is consistent with both engToMorse and toggleCaptions state  
349     toggleCaptions.setVisible(engToMorse);  
350     captionScroll.setVisible(engToMorse && toggleCaptions.isSelected());  
351 };
```

This method is used to quickly swap the direction of translation from English to morse code and the other way.

Solution

File Importing

```
429 importBtn.addActionListener( ActionEvent e -> {
430     JFileChooser fileChooser = new JFileChooser();
431     if (fileChooser.showOpenDialog( parent: this) == JFileChooser.APPROVE_OPTION) {
432         File selectedFile = fileChooser.getSelectedFile();
433         try {
434             String content = new String(java.nio.file.Files.readAllBytes(selectedFile.toPath())).trim();
435             inputArea.setText(content);
436             translateBtn.doClick();
437         } catch (IOException ex) {
438             JOptionPane.showMessageDialog( parentComponent: this, message: "Failed to read file.", title: "Error", J
439         }
440     }
441 });
```

This method imports files and translates the String in the file to morse code.

Audio Tone Generation

```
634 private void playBeep(int duration) { 3 usages
635     try {
636         float frequency = 800f;
637         int sampleRate = 44100;
638         byte[] buf = new byte[duration * sampleRate / 1000];
639         for (int i = 0; i < buf.length; i++) {
640             double angle = 2.0 * Math.PI * i / (sampleRate / frequency);
641             buf[i] = (byte) (Math.sin(angle) * 127);
642         }
643         AudioFormat af = new AudioFormat(sampleRate, sampleSizeInBits: 8, channels: 1, signed: true, bigEndian: false);
644         SourceDataLine sdl = AudioSystem.getSourceDataLine(af);
645         sdl.open(af);
646         sdl.start();
647         sdl.write(buf, off: 0, buf.length);
648         sdl.drain();
649         sdl.stop();
650         sdl.close();
651     } catch (Exception e) {
652         System.err.println("Audio playback error: " + e.getMessage());
653     }
654 }
```

This method generates a certain tone that will be used for the sound feature.

Character Limit and Counter

In GUI class members

```
private JLabel charCountLabel;
private final int MAX_CHARS = 1000;
```

In GUI constructor

```
((AbstractDocument) inputArea.getDocument()).setDocumentFilter(new CharacterLimitFilter());
charCountLabel = new JLabel(); updateCharCount();
inputArea.getDocument().addDocumentListener(new DocumentListener() {
    @Override public void insertUpdate(DocumentEvent e) { updateCharCount(); }
    @Override public void removeUpdate(DocumentEvent e) { updateCharCount(); }
    @Override public void changedUpdate(DocumentEvent e) { updateCharCount(); } });
```


Solution

CharacterLimitFilter inner class

```
class CharacterLimitFilter extends DocumentFilter {  
    @Override  
    public void replace(FilterBypass fb, int offset, int length, String text, AttributeSet attrs) throws  
        BadLocationException {  
        int currentLength = fb.getDocument().getLength();  
        int newLength = currentLength - length + (text == null ? 0 : text.length());  
        if (newLength <= MAX_CHARS) {  
            super.replace(fb, offset, length, text, attrs);  
        }  
        else {  
            Toolkit.getDefaultToolkit().beep();  
        }  
    }  
}
```

This feature prevent users to input String longer than 1000 words and give a real time feedback (a sound) to notify that the String is too long.

Threads

(Flash Morse Thread)

```
if (lightsBtn.getText().equals("Flash")) {  
    stopFlash = false;  
    lightsBtn.setText("Stop");  
    new Thread(() -> flashMorse(morseText, (int) dotSpinner.getValue())).start();  
} else {  
    stopFlash = true;  
}
```

This Thread snippet handles the flashing of the morse code based on the sequence. When the "Flash" button (*lightsBtn*) is clicked, this code creates a new Thread. The *flashMorse* method is executed on this new thread, which continuously changes the background color of the *flashLight* panel to simulate Morse code flashes (yellow for a signal, dark gray for no signal).

(Play Morse Sound)

Solution

```
if (soundBtn.getText().equals("Sound")) {
    stopSound = false;
    soundBtn.setText("Stop");
    new Thread(() -> playMorseSound(morseText, (int) dotSpinner.getValue())).start();
} else {
    stopSound = true;
}
```

When the "Sound" button (*soundBtn*) is clicked, a new Thread is created to execute the *playMorseSound* method. This method generates sound beeps for each dot and dash in the Morse code.

(Sync)

```
if(syncBtn.getText().equals("Sync")) {
    stopSync = false;
    syncBtn.setText("Stop");
    new Thread(() -> performSync(morseText, (int) dotSpinner.getValue())).start();
} else {
    stopSync = true;
}
```

When the "Sync" button (*syncBtn*) is clicked, a new Thread is started to run the *performSync* method. This method orchestrates both the *flashLight* and *playBeep* actions simultaneously for each Morse code element.

Solution

Comparison of time & space complexity

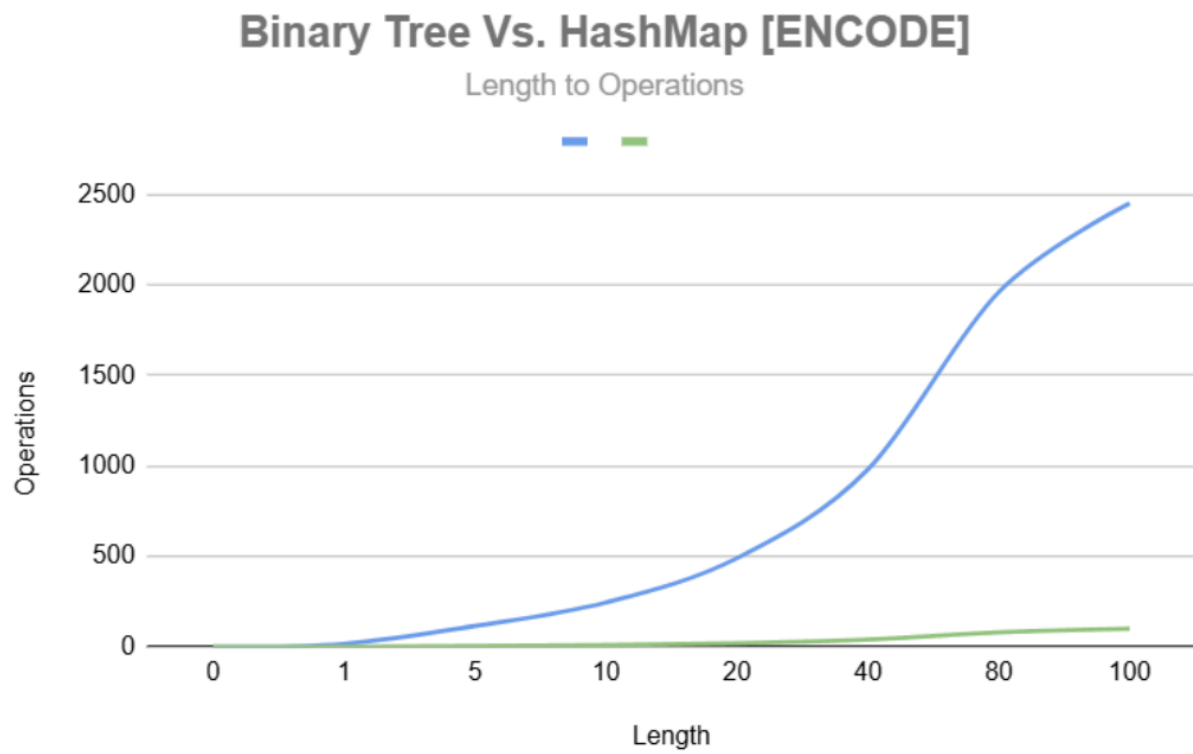
For the time and space complexity of our code, it provides us with clear vision of how our code is performing:

Time Complexity

Letter Inputs (A & B)		ENCODE			
Table1					
Input	Data Structure	Length	Operations	Runtime (Se	
None	Both	0	0	0	
A	Binary Tree	1	17	0.0012251 s	
ABABA	Binary Tree	5	115	0.0019443 s	
ABABABABAB	Binary Tree	10	245	0.0010857 s	
ABABABABABABABABABAB	Binary Tree	20	490	0.0038853 s	
ABABABABABABABABABABAE	Binary Tree	40	980	0.0027194 s	
ABABABABABABABABABABAE	Binary Tree	80	1960	0.003701 s	
ABABABABABABABABABABAE	Binary Tree	100	2450	0.004671 s	

Table2					
Input	Data Struct	Length	Operations	Runtime (S	
None	Both	0	0	0	
A	HashM...	1	1	0.000210 s	
ABABA	HashM...	5	5	0.000026 s	
ABABABABAB	HashM...	10	10	0.000054 s	
ABABABABABABABABABAB	HashM...	20	20	0.000091 s	
ABABABABABABABABABABABA	HashM...	40	40	0.000076 s	
ABABABABABABABABABABABA	HashM...	80	80	0.000094 s	
ABABABABABABABABABABABA	HashM...	100	100	0.000117 s	

Solution



Solution



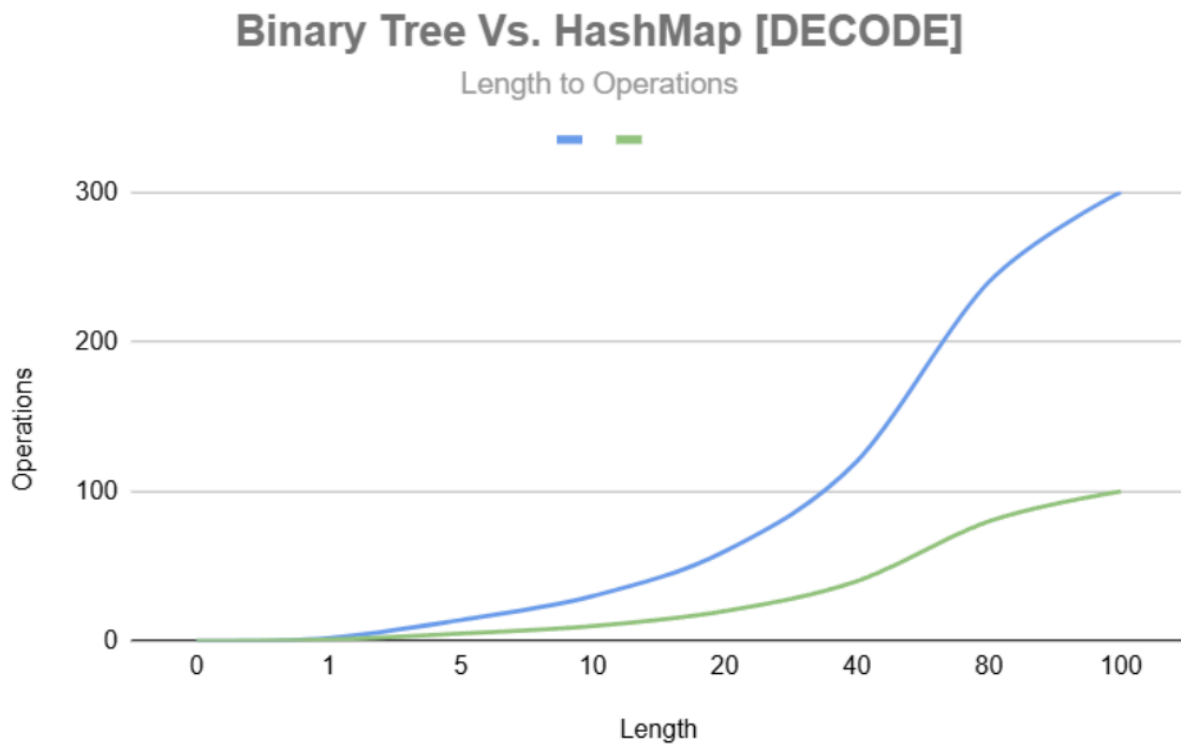
Table3 						DECODE				
Input		Data Structure	Length	Operations	Runtime (Se					
None		Both	0	0	0					
.		Binary Tree	1	2	0.0000092 s					
.		Binary Tree	5	14	0.0000209 s					
.		Binary Tree	10	30	0.0000287 s					
.		Binary Tree	20	60	0.000045 s					
.		Binary Tree	40	120	0.00006 s					
.		Binary Tree	80	240	0.0002113 s					
.		Binary Tree	100	300	0.0002039 s					

Table4 										
Input		Data Struct	Length	Operations	Runtime (S					
None		Both	0	0	0					
.		HashM...	1	1	0,000072 s					
.		HashM...	5	5	0,000175 s					
.		HashM...	10	10	0,000216 s					
.		HashM...	20	20	0,000269 s					
.		HashM...	40	40	0,000310 s					
.		HashM...	80	80	0,000425 s					
.		HashM...	100	100	0,000518 s					

Solution



As you can see for encoding and decoding, HashMap consistently performs fewer operations than Binary Tree. Even time wise, HashMap performs 40x better than Binary Tree if we look at the table where we encode 100 letters. Hashmap would take 0.000117 seconds while Binary Tree takes about 0.004671 seconds. Even though Hashmap performed better, both of these data structures would have $O(n)$ as their time complexity. This is based on the length of operations that shows a linear growth which indicates $O(n)$.

Solution

Space Complexity

ENCODE				
Table5				
Input	Data Structure	Length	Space	
None	Both	0	55	
A	Binary Tree	1	55	
ABABA	Binary Tree	5	55	
ABABABABAB	Binary Tree	10	55	
ABABABABABABABABABAB	Binary Tree	20	55	
ABABABABABABABABABABAE	Binary Tree	40	55	
ABABABABABABABABABABAE	Binary Tree	80	55	
ABABABABABABABABABABAE	Binary Tree	100	55	

Table7				
Input	Data Structure	Length	Space	
None	Both	0	100	
A	HashMap	1	100	
ABABA	HashMap	5	100	
ABABABABAB	HashMap	10	100	
ABABABABABA	HashMap	20	100	
ABABABABABA	HashMap	40	100	
ABABABABABA	HashMap	80	100	
ABABABABABA	HashMap	100	100	

Solution

Binary Tree Vs. HashMap [ENCODE]

Length to Space

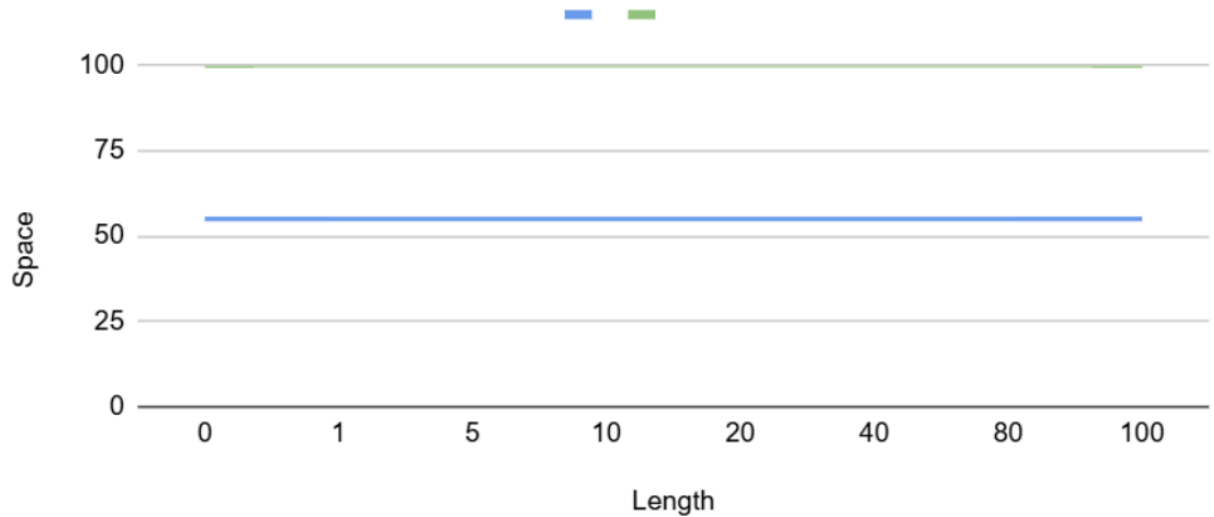
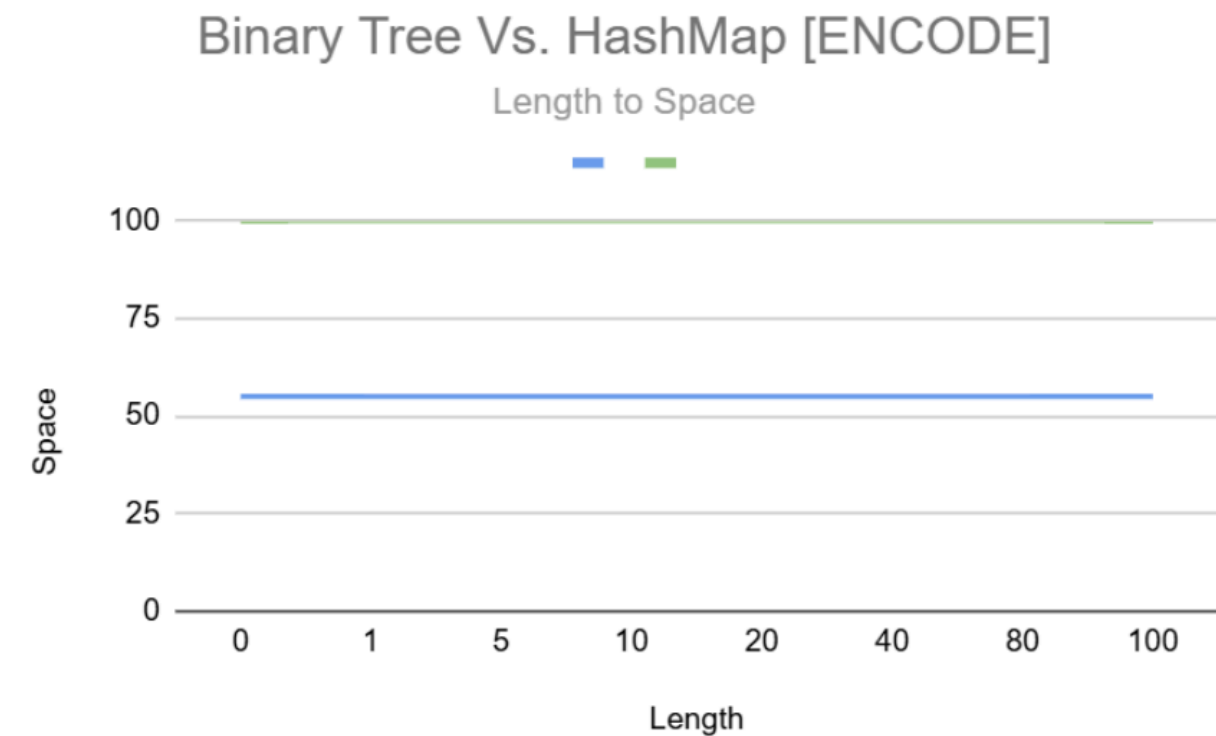


Table8					DECODE				
Input		Data Structure		Length		Space			
None		Both		0		55			
.		Binary Tree		1		55			
.		Binary Tree		5		55			
.		Binary Tree		10		55			
.		Binary Tree		20		55			
. .		Binary Tree		40		55			
. .		Binary Tree		80		55			
. .		Binary Tree		100		55			

Solution

Table6					
Input		Data Structure	Length	Space	
None		Both	0	100	
.		HashMap	1	100	
.-		HashMap	5	100	
.-... -.-... -.-		HashMap	10	100	
.-... -.-... -.-... -.-		HashMap	20	100	
.-... -.-... -.-... -.-... -.-		HashMap	40	100	
.-... -.-... -.-... -.-... -.-... -.-		HashMap	80	100	
.-... -.-... -.-... -.-... -.-... -.-... -.-		HashMap	100	100	



Both data structures result in a constant $O(1)$ growth for space complexity because in this instance, the morse code is constant with nothing else to add or remove. While both are constant,

Solution

Binary Tree performed well since it can go from English to morse and one way another. However, HashMap needs to create a reverse map so it can allow users to translate from morse code to English.

■ Algorithm:

HashMap

■ In the instance of Hashmap, we use **hashing** to encode and decode the morse code. Hashing works as a function that stores keys and values together that can be used to retrieve data. For instance, storing "A" would go together with it's morse code which is "-.", and then if we want to retrieve its value we just need to put the keys in and then the morse code would be available for us.

Binary Tree

■ To decode in a binary tree, you need to use Depth-First Search(DFS). The reason behind this is that in a tree, you can't look up English letters. That's why we need to use DFS to navigate the path from the root, to the left subtree, then right subtree. It needs to travel through the entirety of the left subtree to make sure it didn't miss the letter. For other instance, encoding would be easier because we can use Path Traversal, where in path traversal, the code acts as a direction. The dots would imply us to go down to the left child and dash would lead us to the right child.

Solution

Class Diagram:



Key Benefits of the Solution:

Speed Optimization: The use of **HashMap** ensures quick lookups and rapid encoding/decoding, drastically reducing the time needed to translate large volumes of text into Morse code and vice versa.

Efficiency: Storing previous translations in a **String List** will allow users to easily access and reuse past translations, increasing the system's overall efficiency.

Solution

Multi-Modal Communication: The solution supports audio-based Morse code transmission through the use of String Lists, enabling the system to generate sound sequences (beeps) for communication, which is particularly valuable in emergency situations.

Comprehensive Translation Experience: By handling both letters and numbers, the HashMap or the Binary Tree ensures a complete translation experience for a wide range of inputs, covering all alphanumeric characters in Morse code. The Binary Tree will serve as a complimentary choice for users to measure the difference.

Flexibility: The system is designed to be adaptable, allowing future extensions like the audio-based morse code feature or light signaling, ensuring the tool remains relevant and versatile for various modes of communication

Overall Conclusion:

To conclude, based on the time and space complexity, HashMap performs a lot better when compared to Binary Tree time complexity wise. It performs 40x faster if we compare the encoding process. Even though it performed better, both of these data structures growth is linear which makes it $O(n)$. On the other hand, Binary Tree has better space complexity because HashMap provides the reverse map for it to decode from morse code to English. Which took almost twice the space Binary Tree used. Both of these still have $O(1)$ as space complexity because morse code is a constant size.

References

Oracle. (n.d.). How to use lists. In The Java™ Tutorials—Creating a GUI with Swing. Retrieved June 15, 2025 [[LINK](#)]

Flash (Panel flashing with background thread + SwingUtilities.invokeLater)
Stack Overflow user. (2014, March 9). Update Java Swing component from a background thread. [[LINK](#)]

Sound (Generating tones with SourceDataLine)
Oracle Forums. (2007). Example code to generate audio tone. Retrieved June 15, 2025, [[LINK](#)]

Flash (Panel flashing with background thread + SwingUtilities.invokeLater)
Stack Overflow user. (2014, March 9). Update Java Swing component from a background thread. Retrieved June 15, 2025, [[LINK](#)]

Sound (Generating tones with SourceDataLine)
Oracle Forums. (2007). Example code to generate audio tone. Retrieved June 15, 2025, [[LINK](#)]

Import (Reading files via JFileChooser)
Oracle. (n.d.). How to use file choosers. In The Java Tutorials Creating a GUI with Swing. Retrieved June 15, 2025, [[LINK](#)]

Sync (Combining flash + sound; coordination using shared timing)
Baeldung. (2023, February 10). Simple Morse Code translation in Java. Retrieved June 15, 2025, [[LINK](#)]

Translate (English ↔ Morse via HashMap)
Stack Overflow user. (2014, February 7). Morse code translator with HashMaps in Java. Retrieved June 15, 2025, [[LINK](#)]

Thread Safety / Swing Concurrency

Oracle. (n.d.). Event dispatch thread. In Java™ Tutorials—Concurrency in Swing.

Retrieved June 15, 2025, [[LINK](#)]