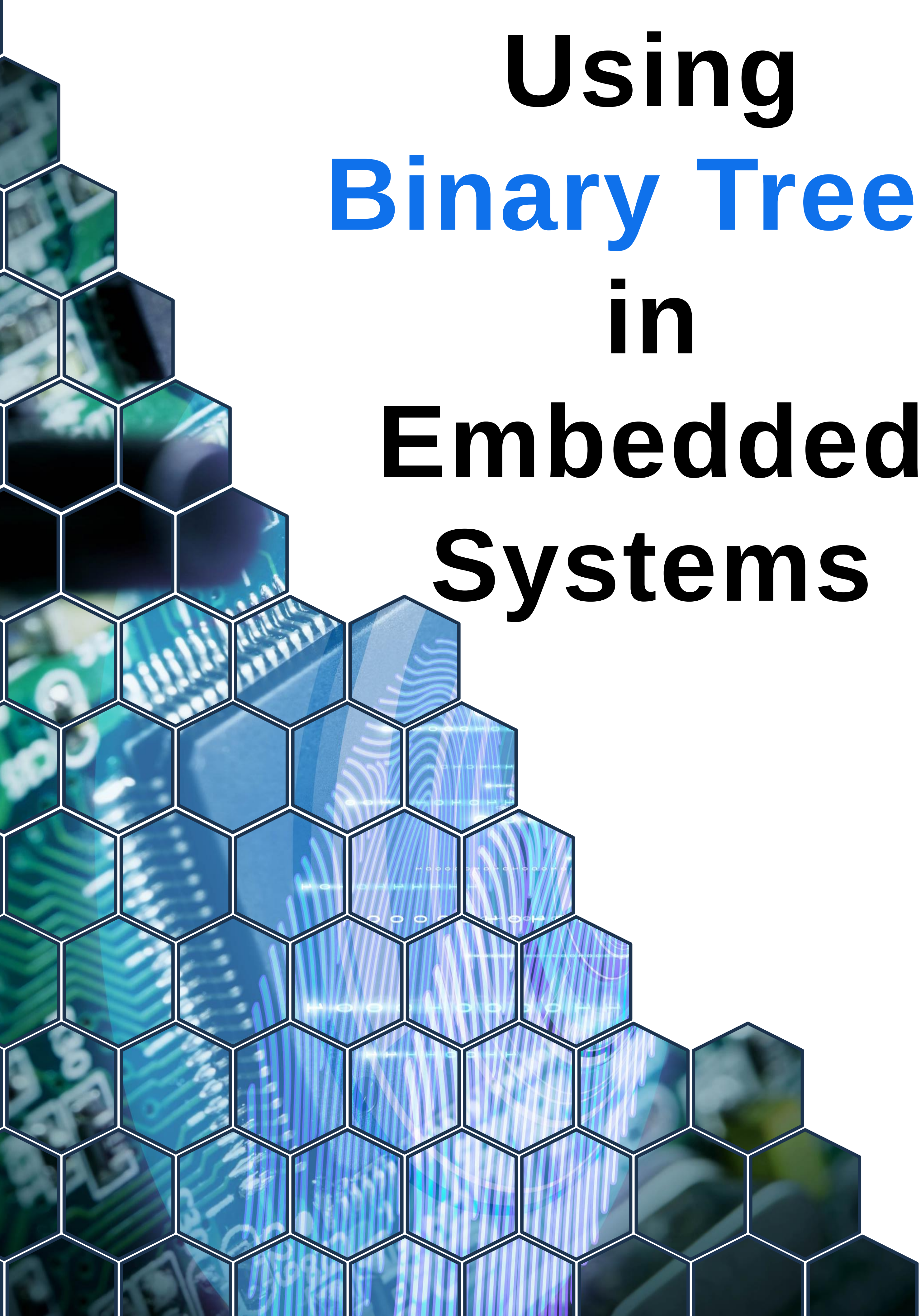
Using Binary Trees in Embedded Systems



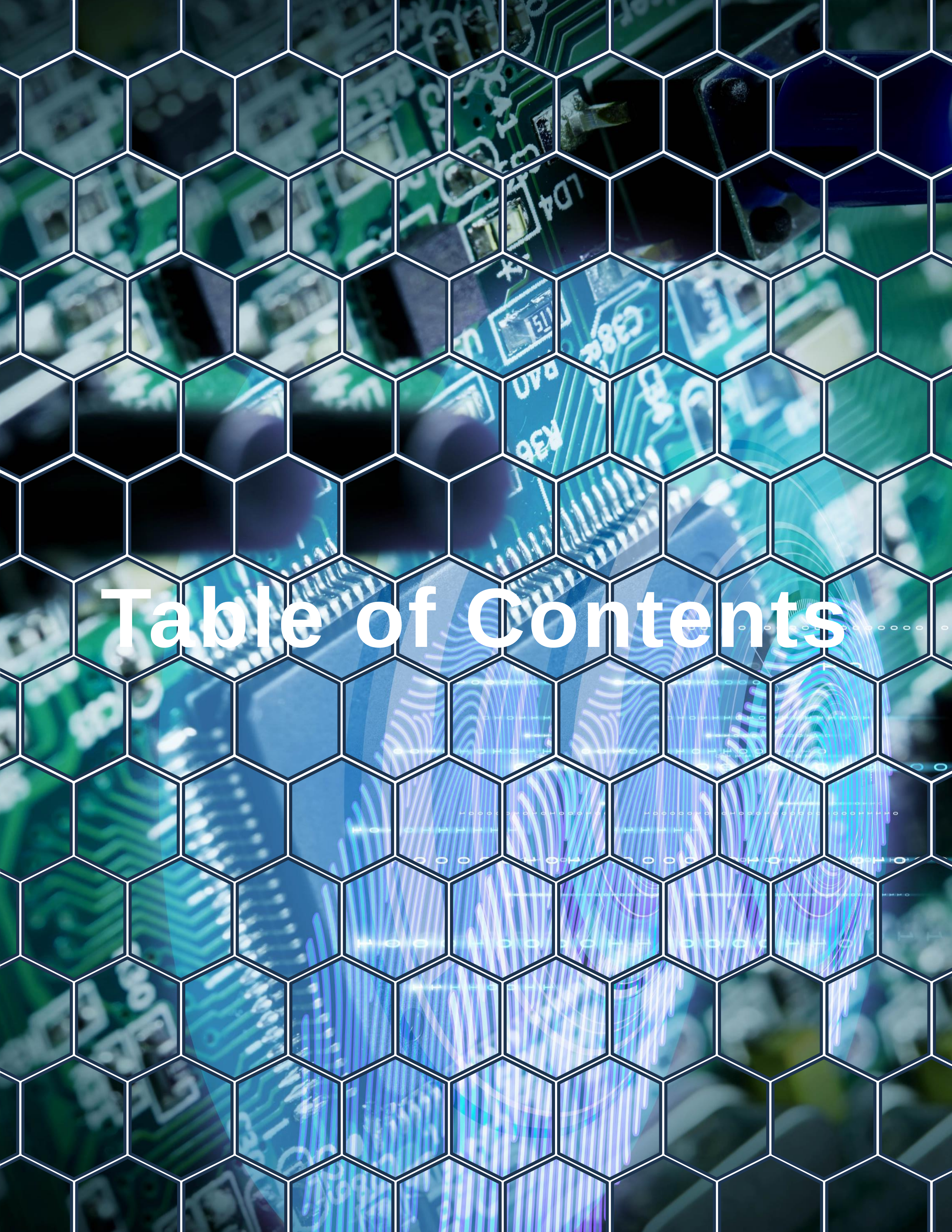
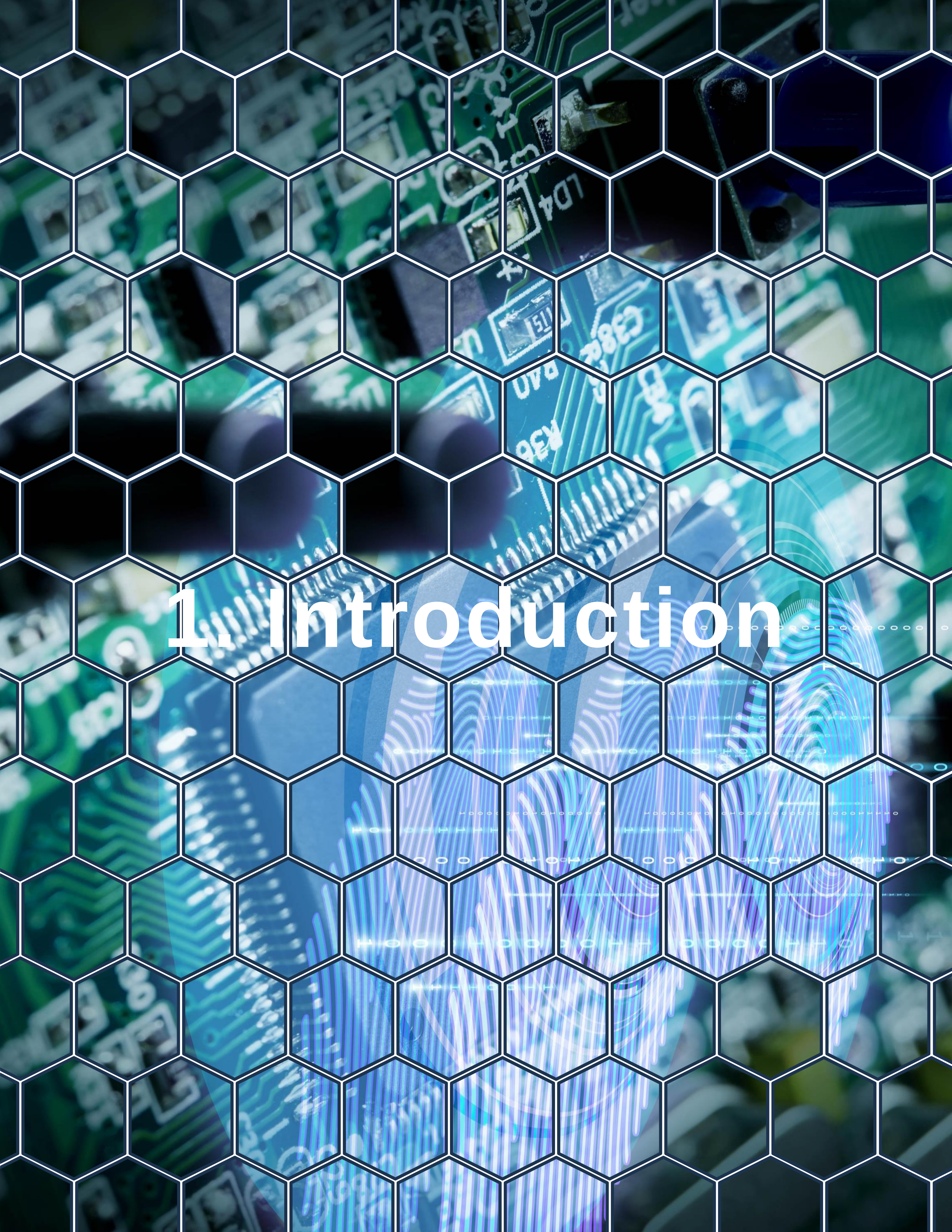


Table of Contents

Table of Contents



1. Introduction

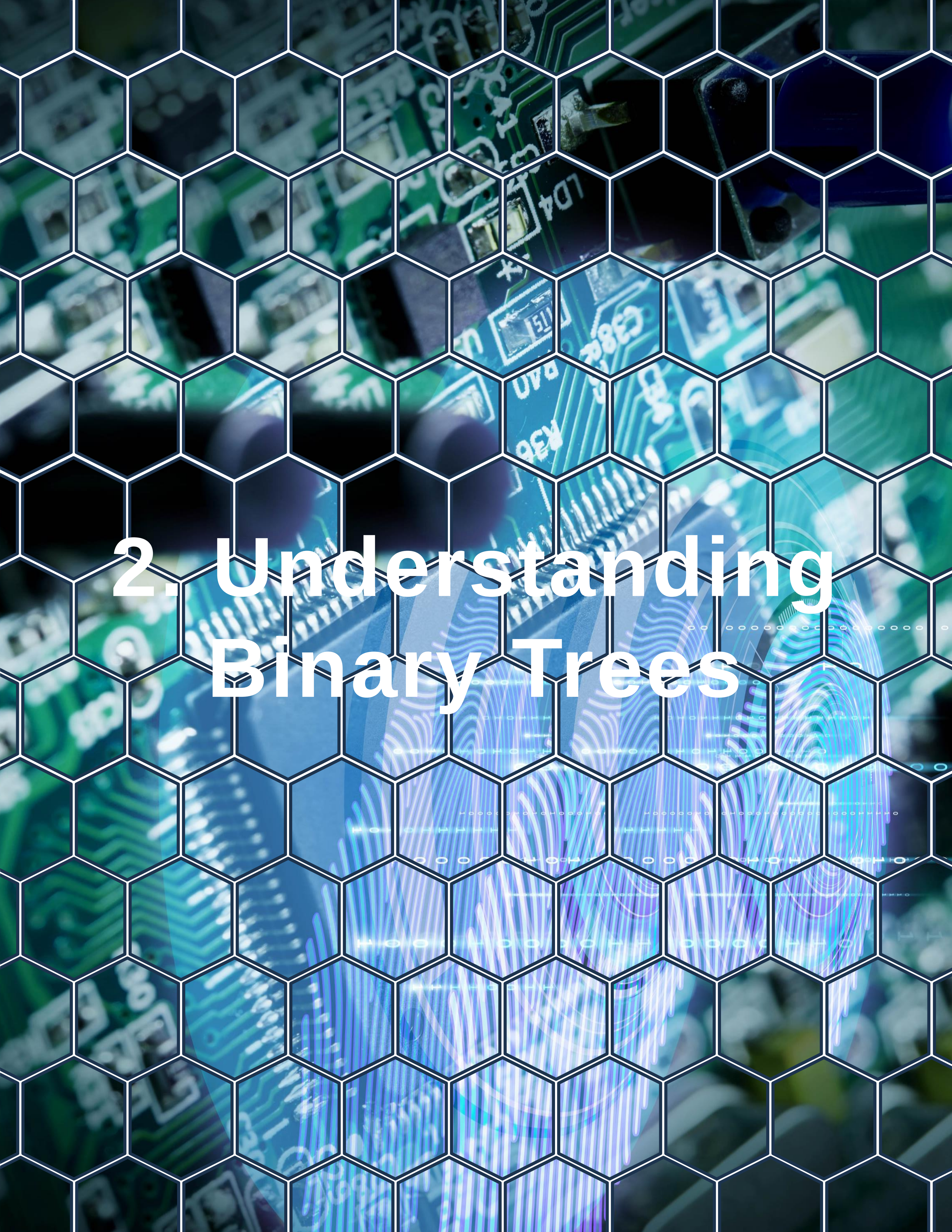
1. Introduction

In the realm of embedded systems, data structures play a crucial role in ensuring efficient data management and retrieval. Binary trees, a fundamental data structure, are widely utilized in various computing applications due to their hierarchical organization and balanced data access. However, leveraging binary trees in embedded systems presents unique challenges, especially given the limited memory and processing power inherent to microcontrollers (MCUs).

his article explores the application of binary

1. Introduction

This article explores the application of binary trees in embedded systems, focusing on their implementation, optimization, and practical use cases for MCUs.



2. Understanding Binary Trees

2. Understanding Binary Trees

A binary tree is a data structure where each node has at most two child nodes: a left child and a right child. The topmost node is called the root, and nodes with no children are known as leaf nodes.

The primary features of a binary tree include:

- **Hierarchical Structure**: Data is organized in levels, starting from a root node.
- **Nodes and Pointers**: Each node contains data and pointers to its child nodes.
- **Recursive Nature**: Binary trees are often defined recursively, making them suitable for algorithms that benefit from divide-and-

2. Understanding Binary Trees

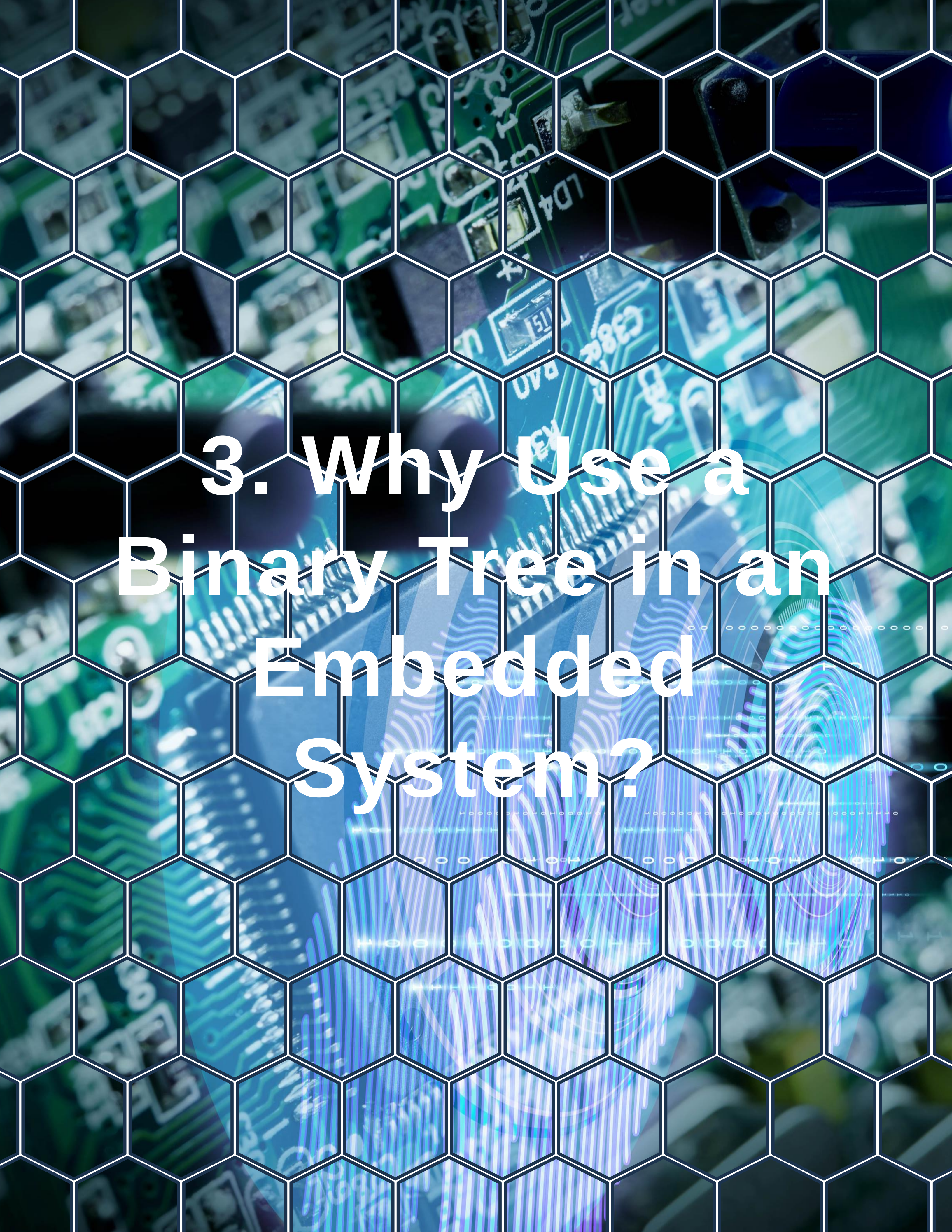
There are several types of binary trees, including:

- **Binary Search Trees (BST)**: A BST is a binary tree where the left child of a node contains a value less than the node’s value, and the right child contains a value greater than the node’s value. This property facilitates efficient searching.
- **Balanced Binary Trees**: These trees maintain a balanced structure, ensuring that the height of the tree is minimized, which is crucial for maintaining optimal performance.

2. Understanding Binary Trees

- **Heap Trees**: These are specialized binary trees used in priority queues, where the parent node is either greater than or less than its children, depending on the type of heap.

Binary trees can be utilized to represent hierarchical data, facilitate efficient searching (binary search trees), or provide a structure for dynamic data processing. They serve as the foundation for more complex data structures like heaps, tries, and balanced trees (e.g., AVL trees).



3. Why Use a Binary Tree in an Embedded System?

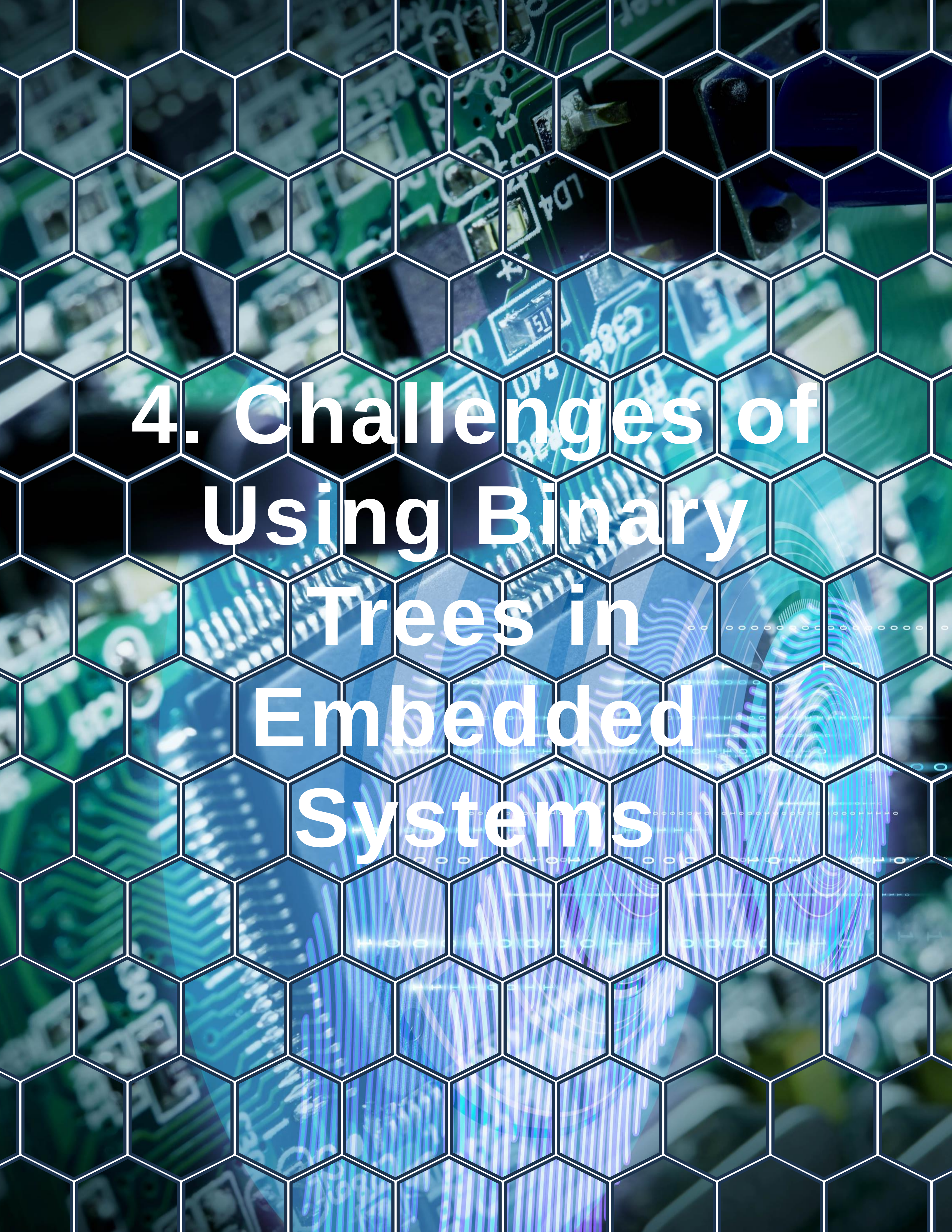
3. Why Use a Binary Tree in an Embedded System?

Binary trees offer specific advantages in embedded systems, particularly for applications requiring efficient data management and rapid look-up times. The hierarchical structure of a binary tree allows:

- **Efficient Search Operations**: Ideal for scenarios where data needs to be frequently accessed or modified.
- **Organized Data Storage**: Data can be sorted and organized effectively.
- **Hierarchical Data Representation**: Supports applications like decision-making processes, classification, and more.

3. Why Use a Binary Tree in an Embedded System?

Embedded systems often handle tasks that involve frequent data sorting, retrieval, and decision-making, such as sensor data processing, device control logic, and real-time analysis. Binary trees can be employed to manage these tasks efficiently, especially when memory usage is optimized.



4. Challenges of Using Binary Trees in Embedded Systems

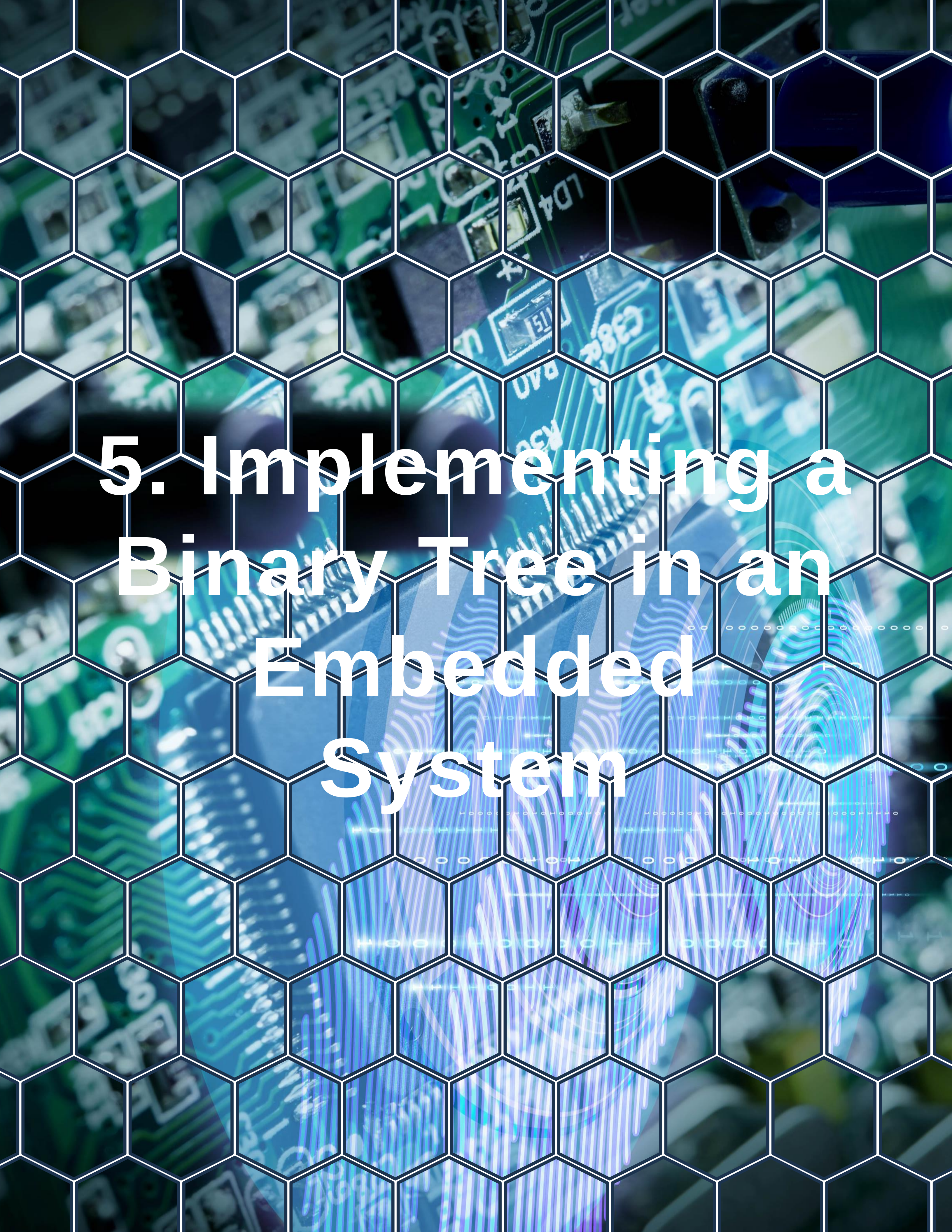
4. Challenges of Using Binary Trees in Embedded Systems

While binary trees offer numerous benefits, their implementation in embedded systems presents several challenges:

- **Memory Constraints**: Microcontrollers often have limited RAM, making it essential to manage memory efficiently. Binary trees can consume significant memory, especially if not implemented carefully.
- **Memory Fragmentation**: Frequent dynamic memory allocation and deallocation can lead to memory fragmentation, which can degrade performance over time.

4. Challenges of Using Binary Trees in Embedded Systems

- **Real-Time Performance**: Embedded systems often require real-time performance, and the recursive nature of binary tree operations can introduce latency.
- **Complexity**: Implementing and maintaining binary trees can be more complex than simpler data structures like arrays or linked lists.



5. Implementing a Binary Tree in an Embedded System

5. Implementing a Binary Tree in an Embedded System

Implementing a binary tree in an embedded system requires careful consideration of memory management and performance. Below is a basic implementation of a binary search tree in C, tailored for microcontrollers:

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

// Define the structure for a binary tree node
typedef struct Node {
    int data;
    struct Node* left;
    struct Node* right;
} Node;

// Function to create a new node
Node* createNode(int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    if (newNode == NULL) {
        // Handle memory allocation failure
```

5. Implementing a Binary Tree in an Embedded System

```c
// Function to create a new node
Node* createNode(int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    if (newNode == NULL) {
        // Handle memory allocation failure
        return NULL;
    }
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

// Function to insert a node in the BST
Node* insertNode(Node* root, int data) {
    if (root == NULL) {
        return createNode(data);
    }
    if (data < root->data) {
        root->left = insertNode(root->left, data);
    } else if (data > root->data) {
        root->right = insertNode(root->right, data);
    }
    return root;
}

// Function to search for a node in the BST
Node* searchNode(Node* root, int data) {
```

5. Implementing a Binary Tree in an Embedded System

```c
// Function to search for a node in the BST
Node* searchNode(Node* root, int data) {
    if (root == NULL || root->data == data) {
        return root;
    }
    if (data < root->data) {
        return searchNode(root->left, data);
    }
    return searchNode(root->right, data);
}

// Function to perform an in-order traversal of the BST
void inOrderTraversal(Node* root) {
    if (root != NULL) {
        inOrderTraversal(root->left);
        printf("%d ", root->data);
        inOrderTraversal(root->right);
    }
}

int main() {
    Node* root = NULL;
    root = insertNode(root, 50);
    insertNode(root, 30);
    insertNode(root, 20);
    insertNode(root, 40);
    insertNode(root, 70);
    insertNode(root, 60);
```

5. Implementing a Binary Tree in an Embedded System

```c
int main() {
    Node* root = NULL;
    root = insertNode(root, 50);
    insertNode(root, 30);
    insertNode(root, 20);
    insertNode(root, 40);
    insertNode(root, 70);
    insertNode(root, 60);
    insertNode(root, 80);

    printf("In-order traversal of the BST: ");
    inOrderTraversal(root);
    printf("\n");

    Node* found = searchNode(root, 40);
    if (found != NULL) {
        printf("Node with data 40 found!\n");
    } else {
        printf("Node with data 40 not found.\n");
    }

    return 0;
}
```

5. Implementing a Binary Tree in an Embedded System

This code demonstrates the basic operations of a binary search tree, including node creation, insertion, search, and traversal. Note that in an embedded system, dynamic memory allocation (malloc) should be used cautiously to avoid memory fragmentation.

6. Optimizing Binary Tree Performance

6. Optimizing Binary Tree Performance

To optimize the performance of binary trees in embedded systems:

- **Use Static Memory Allocation**: Where possible, allocate memory statically to prevent fragmentation.
- **Minimize Dynamic Allocations**: Reduce the number of dynamic memory allocations to avoid heap fragmentation.
- **Implement Balanced Trees**: Utilize self-balancing trees (like AVL or Red-Black trees) to maintain efficiency, ensuring that tree operations remain $O(\log n)$.

Memory Pooling: Implement memory

6. Optimizing Binary Tree Performance

- **Minimize Dynamic Allocations**: Reduce the number of dynamic memory allocations to avoid heap fragmentation.
- **Implement Balanced Trees**: Utilize self-balancing trees (like AVL or Red-Black trees) to maintain efficiency, ensuring that tree operations remain $O(\log n)$.
- **Memory Pooling**: Implement memory pools to manage node allocations more efficiently, reducing fragmentation.

7. Practical Use Cases in Embedded Systems

7. Practical Use Cases in Embedded Systems

Binary trees are well-suited for several embedded applications, including:

- **Data Classification**: Organizing sensor data into categories for analysis.
- **Decision-Making**: Implementing decision trees for control systems.
- **Routing Tables**: Efficient data retrieval in networking applications.
- **Symbol Tables**: Managing variables and functions in embedded compilers and interpreters.

However, in cases where only simple data

7. Practical Use Cases in Embedded Systems

- **Decision-Making**: Implementing decision trees for control systems.
- **Routing Tables**: Efficient data retrieval in networking applications.
- **Symbol Tables**: Managing variables and functions in embedded compilers and interpreters.

However, in cases where only simple data retrieval is needed, hash tables or arrays might be more appropriate due to their simpler structure and potentially lower memory usage.

8. Real-life Binary Tree Implementation

8. Real-life Binary Tree Implementation

Consider a real-world example where a binary tree is used to manage sensor data in a home automation system. Each sensor reading is inserted into the binary tree based on its timestamp, allowing efficient retrieval of the latest or specific range of data:

```
/**
 * @file sensor_tree.c
 * @brief Binary Tree implementation for managing sensor data
 * @date 2024-02-06
 */

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <stdbool.h>
#include <time.h>

/**
 * @struct SensorData
 * @brief Structure to hold sensor measurement data
 */
typedef struct {
    uint32_t timestamp;      // Unix timestamp
    float temperature;       // Temperature in Celsius
```

8. Real-life Binary Tree Implementation

```c
/**
 * @struct SensorData
 * @brief Structure to hold sensor measurement data
 */
typedef struct {
    uint32_t timestamp;      // Unix timestamp
    float temperature;       // Temperature in Celsius
    float humidity;         // Humidity percentage
    float pressure;         // Pressure in hPa
    uint8_t sensor_id;      // Unique sensor identifier
} SensorData;

/**
 * @struct SensorNode
 * @brief Node structure for the binary tree
 */
typedef struct SensorNode {
    SensorData data;
    struct SensorNode* left;
    struct SensorNode* right;
} SensorNode;

/**
 * @brief Creates a new sensor node with the given data
 */
SensorNode* createSensorNode(const SensorData* data) {
    SensorNode* newNode = (SensorNode*)malloc(sizeof(SensorNode));
    if (newNode == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);
    }
    newNode->data = *data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

/**
 * @brief Inserts sensor data into the binary tree based on timestamp
 */
```

8. Real-life Binary Tree Implementation

```
/**
 * @brief Inserts sensor data into the binary tree based on timestamp
 */
SensorNode* insertSensorData(SensorNode* root, const SensorData* data) {
    if (root == NULL) {
        return createSensorNode(data);
    }

    // Insert based on timestamp
    if (data->timestamp < root->data.timestamp) {
        root->left = insertSensorData(root->left, data);
    } else {
        root->right = insertSensorData(root->right, data);
    }

    return root;
}

/**
 * @brief Finds sensor data for a specific timestamp
 */
SensorNode* findByTimestamp(SensorNode* root, uint32_t timestamp) {
    if (root == NULL || root->data.timestamp == timestamp) {
        return root;
    }

    if (timestamp < root->data.timestamp) {
        return findByTimestamp(root->left, timestamp);
    }
    return findByTimestamp(root->right, timestamp);
}

/**
 * @brief Collects sensor data within a specified time range
 */
void findDataInRange(SensorNode* root, uint32_t start_time, uint32_t end_time,
                     SensorData* results, int* count, int max_results) {
    if (root == NULL || *count >= max_results) {
        return;
    }
```

8. Real-life Binary Tree Implementation

```c
/**
 * @brief Collects sensor data within a specified time range
 */
void findDataInRange(SensorNode* root, uint32_t start_time, uint32_t end_time,
                     SensorData* results, int* count, int max_results) {
    if (root == NULL || *count >= max_results) {
        return;
    }

    // Inorder traversal to get sorted timestamps
    findDataInRange(root->left, start_time, end_time, results, count, max_results);

    if (root->data.timestamp >= start_time && root->data.timestamp <= end_time) {
        results[*count] = root->data;
        (*count)++;
    }

    findDataInRange(root->right, start_time, end_time, results, count, max_results);
}

/**
 * @brief Finds the most recent sensor reading
 */
SensorNode* findMostRecent(SensorNode* root) {
    if (root == NULL || root->right == NULL) {
        return root;
    }
    return findMostRecent(root->right);
}

/**
 * @brief Deletes old sensor data before the specified timestamp
 */
SensorNode* deleteOldData(SensorNode* root, uint32_t cutoff_time) {
    if (root == NULL) {
        return NULL;
    }

    // First delete from left subtree
    root->left = deleteOldData(root->left, cutoff_time);
```

8. Real-life Binary Tree Implementation

```c
/**
 * @brief Deletes old sensor data before the specified timestamp
 */
SensorNode* deleteOldData(SensorNode* root, uint32_t cutoff_time) {
    if (root == NULL) {
        return NULL;
    }

    // First delete from left subtree
    root->left = deleteOldData(root->left, cutoff_time);

    // If current node is too old, delete it and return right subtree
    if (root->data.timestamp < cutoff_time) {
        SensorNode* temp = root->right;
        free(root);
        return deleteOldData(temp, cutoff_time);
    }

    // Process right subtree
    root->right = deleteOldData(root->right, cutoff_time);
    return root;
}

/**
 * @brief Prints sensor data in a formatted way
 */
void printSensorData(const SensorData* data) {
    time_t timestamp = (time_t)data->timestamp;
    struct tm* timeinfo = localtime(&timestamp);
    char timestr[26];
    strftime(timestr, sizeof(timestr), "%Y-%m-%d %H:%M:%S", timeinfo);

    printf("Timestamp: %s\n", timestr);
    printf("Sensor ID: %d\n", data->sensor_id);
    printf("Temperature: %.2f°C\n", data->temperature);
    printf("Humidity: %.2f%%\n", data->humidity);
    printf("Pressure: %.2f hPa\n", data->pressure);
    printf("------------------------\n");
}

/**
```

8. Real-life Binary Tree Implementation

```c
/**
 * @brief Prints sensor data in a formatted way
 */
void printSensorData(const SensorData* data) {
    time_t timestamp = (time_t)data->timestamp;
    struct tm* timeinfo = localtime(&timestamp);
    char timestr[26];
    strftime(timestr, sizeof(timestr), "%Y-%m-%d %H:%M:%S", timeinfo);

    printf("Timestamp: %s\n", timestr);
    printf("Sensor ID: %d\n", data->sensor_id);
    printf("Temperature: %.2f°C\n", data->temperature);
    printf("Humidity: %.2f%%\n", data->humidity);
    printf("Pressure: %.2f hPa\n", data->pressure);
    printf("------------------------\n");
}

/**
 * @brief Frees all memory used by the tree
 */
void deleteSensorTree(SensorNode* root) {
    if (root != NULL) {
        deleteSensorTree(root->left);
        deleteSensorTree(root->right);
        free(root);
    }
}

// Example usage
int main() {
    SensorNode* root = NULL;

    // Create some sample sensor readings
    SensorData readings[] = {
        {1707235200, 22.5, 45.0, 1013.2, 1},  // 2024-02-06 12:00:00
        {1707235500, 23.1, 44.5, 1013.1, 1},  // 2024-02-06 12:05:00
        {1707235800, 23.4, 44.2, 1013.0, 1},  // 2024-02-06 12:10:00
        {1707236100, 23.6, 44.0, 1012.9, 1},  // 2024-02-06 12:15:00
        {1707236400, 23.8, 43.8, 1012.8, 1}   // 2024-02-06 12:20:00
    };
```

8. Real-life Binary Tree Implementation

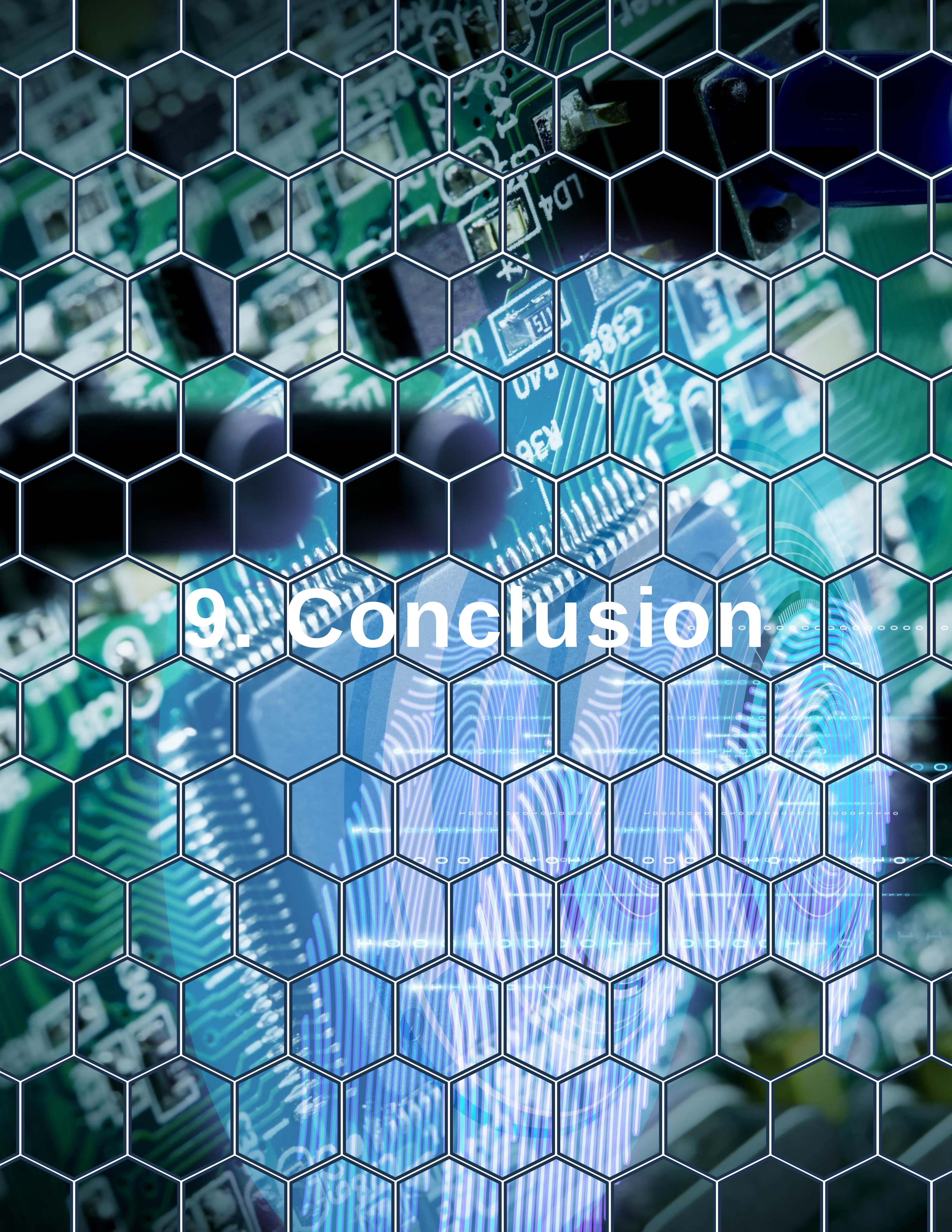
```c
// Example usage
int main() {
    SensorNode* root = NULL;

    // Create some sample sensor readings
    SensorData readings[] = {
        {1707235200, 22.5, 45.0, 1013.2, 1},  // 2024-02-06 12:00:00
        {1707235500, 23.1, 44.5, 1013.1, 1},  // 2024-02-06 12:05:00
        {1707235800, 23.4, 44.2, 1013.0, 1},  // 2024-02-06 12:10:00
        {1707236100, 23.6, 44.0, 1012.9, 1},  // 2024-02-06 12:15:00
        {1707236400, 23.8, 43.8, 1012.8, 1}   // 2024-02-06 12:20:00
    };

    // Insert readings into the tree
    for (int i = 0; i < 5; i++) {
        root = insertSensorData(root, &readings[i]);
    }

    // Find most recent reading
    SensorNode* recent = findMostRecent(root);
    printf("Most recent reading:\n");
    printSensorData(&recent->data);

    // Find readings in a time range
    printf("\nReadings between 12:05:00 and 12:15:00:\n");
    SensorData rangeResults[10];
    int count = 0;
    findDataInRange(root, 1707235500, 1707236100, rangeResults, &count, 10);

    for (int i = 0; i < count; i++) {
        printSensorData(&rangeResults[i]);
    }

    // Delete old data
    root = deleteOldData(root, 1707235800);  // Delete readings before 12:10:00

    printf("\nAfter deleting old data (before 12:10:00):\n");
    SensorData newRangeResults[10];
    count = 0;
    findDataInRange(root, 0, UINT32_MAX, newRangeResults, &count, 10);
```

8. Real-life Binary Tree Implementation

```
    // Delete old data
    root = deleteOldData(root, 1707235800);  // Delete readings before 12:10:00

    printf("\nAfter deleting old data (before 12:10:00):\n");
    SensorData newRangeResults[10];
    count = 0;
    findDataInRange(root, 0, UINT32_MAX, newRangeResults, &count, 10);

    for (int i = 0; i < count; i++) {
        printSensorData(&newRangeResults[i]);
    }

    // Clean up
    deleteSensorTree(root);

    return 0;
}
```

The example usage demonstrates:

- Inserting multiple readings
- Retrieving the most recent data
- Querying data within a time range
- Cleaning up old data
- Memory management



9. Conclusion

9. Conclusion

Binary trees offer a structured and efficient way to manage data in embedded systems, particularly when data retrieval and organization are critical. However, their implementation requires careful consideration of memory management, balancing techniques, and CPU utilization to overcome the inherent constraints of MCUs. By applying best practices, such as minimizing dynamic allocations and employing memory pools, embedded developers can effectively integrate binary trees into their applications.

10. Additional Resources

10. Additional Resources

For further reading and deeper understanding, consider the following resources:

Books:

- "Introduction to Algorithms" by Thomas H. Cormen et al.
- "Data Structures and Algorithm Analysis in C" by Mark Allen Weiss.

Online Courses:

- Coursera’s "Data Structures and Algorithms" specialization.
- edX’s "Embedded Systems Essentials" course.

Websites:

10. Additional Resources

Websites:

- GeeksforGeeks (https://www.geeksforgeeks.org/)
- Embedded.com (https://www.embedded.com/)

By leveraging these resources and applying the insights from this article, embedded systems developers can harness the full potential of binary trees in their projects.