

# **BSc (Hons) Artificial Intelligence and Data Science**

**Module: CM1602 Data Structures and Algorithms  
for Artificial Intelligence**

## **Individual Coursework Report**

**Module Leader: Ms. Malsha Fernando**

**RGU Student ID : 2506755**

**IIT Student ID : 20240281**

**Student Name : W.D. Nisitha Nimsara**

## Contents

2. Introduction.....	3
3. Algorithm selection justification.....	4
3.1 Merge Sort ( $O(n \log n)$ ).....	4
3.2 Fast/Slow Pointer ( $O(n)$ ) - finding middle in merge sort.....	6
3.3 Binary Search Tree Traversals ( $O(n)$ ).....	6
3.4 Undo Mechanism ( $O(1)$ ).....	7
4. Data Structure selection justification .....	8
4.1 Array (HospitalManager).....	8
4.2 Linked List (DiseaseRecordLinkedList).....	9
4.3 Queue (ReportQueue) .....	10
4.4 Binary Search Tree (SeverityBST) .....	11
4.5 Fixed-Size Stack (UndoManager).....	11
5. Test plans .....	12
6.Reference list .....	20

## 2. Introduction

The “HealthSense” Disease Outbreak Analysis System is a command-line prototype designed to support epidemiologists in analyzing patient outbreak data across multiple hospitals and regions. The system provides tools to:

- Add hospitals and maintain disease case records.
- Search disease data by name or case count.
- Sort hospital records by patient counts or weekly order.
- Track disease trends and identify synchronized peak outbreaks.
- Manage outbreak reports using a queue structure.
- Classify severity levels using a binary search tree (BST).
- Support undo functionality for recent operations.

The objective is to illustrate how basic algorithms and data structures can be used in a practical sense to tackle healthcare analytics problems with a focus on efficiency, scalability, and maintainability. This mimics the learning outcomes of CM1602 just by combining algorithmic thinking, complexity analysis, and applying our own purpose-built structures.

### 3. Algorithm selection justification

#### 3.1 Merge Sort ( $O(n \log n)$ )

```
11
12 //Sort by case count (highest to lowest).
13 @ public static void sortLinkedListByCount(DiseaseRecordLinkedList list) { 2 usages
14     list.head = mergeSortByCount(list.head);
15 }
16
17 // Recursive merge sort by case count
18 private static DiseaseRecordLinkedList.Node mergeSortByCount(DiseaseRecordLinkedList.Node head) { 3 usages
19     if (head == null || head.next == null) return head; // empty or 1 element
20     DiseaseRecordLinkedList.Node mid = getMiddle(head); // find middle
21     DiseaseRecordLinkedList.Node right = mid.next;
22     mid.next = null; // split list in two halves
23
24     DiseaseRecordLinkedList.Node leftSorted = mergeSortByCount(head);
25     DiseaseRecordLinkedList.Node rightSorted = mergeSortByCount(right);
26
27     return mergeByCount(leftSorted, rightSorted); // merge back together
28 }
29
30 // Merge two sorted lists by case count (descending)
31 private static DiseaseRecordLinkedList.Node mergeByCount(DiseaseRecordLinkedList.Node a, DiseaseRecordLinkedList.Node b) {
32     DiseaseRecordLinkedList.Node dummy = new DiseaseRecordLinkedList.Node(new DiseaseRecord("diseaseName: ",
33     DiseaseRecordLinkedList.Node tail = dummy;
34
35     while (a != null && b != null) {
36         if (a.data.caseCount >= b.data.caseCount) {
37             // copy node from a
38             tail.next = new DiseaseRecordLinkedList.Node(
39                 new DiseaseRecord(a.data.diseaseName, a.data.weekNumber, a.data.caseCount));
40             a = a.next;
41         } else {
42             // copy node from b
43             tail.next = new DiseaseRecordLinkedList.Node(
44                 new DiseaseRecord(b.data.diseaseName, b.data.weekNumber, b.data.caseCount));
45             b = b.next;
46         }
47         tail = tail.next;
48     }
49
50     // attach remaining nodes
51     while (a != null) {
52         tail.next = new DiseaseRecordLinkedList.Node(
53             new DiseaseRecord(a.data.diseaseName, a.data.weekNumber, a.data.caseCount));
54         a = a.next;
55         tail = tail.next;
56     }
57     while (b != null) {
58         tail.next = new DiseaseRecordLinkedList.Node(
59             new DiseaseRecord(b.data.diseaseName, b.data.weekNumber, b.data.caseCount));
60         b = b.next;
61         tail = tail.next;
62     }
63
64     return dummy.next;
65 }
66
```

Figure 1- sortLinkedListByCount()

```

66
67 // by week number (ascending order).
68 @ public static void sortLinkedListByWeek(DiseaseRecordLinkedList list) { 2 usages
69     list.head = mergeSortByWeek(list.head);
70 }
71
72 // Recursive merge sort by week number
73 private static DiseaseRecordLinkedList.Node mergeSortByWeek(DiseaseRecordLinkedList.Node head) { 3 usages
74     if (head == null || head.next == null) return head; // base case
75     DiseaseRecordLinkedList.Node mid = getMiddle(head); // find middle
76     DiseaseRecordLinkedList.Node right = mid.next;
77     mid.next = null;
78
79     DiseaseRecordLinkedList.Node leftSorted = mergeSortByWeek(head);
80     DiseaseRecordLinkedList.Node rightSorted = mergeSortByWeek(right);
81
82     return mergeByWeek(leftSorted, rightSorted);
83 }
84
85 // Merge two sorted lists by week number (ascending)
86 private static DiseaseRecordLinkedList.Node mergeByWeek(DiseaseRecordLinkedList.Node a, DiseaseRecordLinkedList.Node b) {
87     DiseaseRecordLinkedList.Node dummy = new DiseaseRecordLinkedList.Node(new DiseaseRecord(" ", 0, 0));
88     DiseaseRecordLinkedList.Node tail = dummy;
89
90     while (a != null && b != null) {
91         if (a.data.weekNumber <= b.data.weekNumber) {
92             tail.next = new DiseaseRecordLinkedList.Node(
93                 new DiseaseRecord(a.data.diseaseName, a.data.weekNumber, a.data.caseCount));
94             a = a.next;
95         } else {
96             tail.next = new DiseaseRecordLinkedList.Node(
97                 new DiseaseRecord(b.data.diseaseName, b.data.weekNumber, b.data.caseCount));
98             b = b.next;
99         }
100         tail = tail.next;
101     }
102
103     // attach remaining nodes
104     while (a != null) {
105         tail.next = new DiseaseRecordLinkedList.Node(
106             new DiseaseRecord(a.data.diseaseName, a.data.weekNumber, a.data.caseCount));
107         a = a.next;
108         tail = tail.next;
109     }
110     while (b != null) {
111         tail.next = new DiseaseRecordLinkedList.Node(
112             new DiseaseRecord(b.data.diseaseName, b.data.weekNumber, b.data.caseCount));
113         b = b.next;
114         tail = tail.next;
115     }
116
117     return dummy.next;
118 }

```

Figure 2- sortLinkedListByWeek()

- Used to sort disease records either by case count (descending) or week number (ascending).
- Chosen because:
  - Works efficiently with linked lists (unlike quicksort which relies on random access).
  - Provides consistent worst-case performance of  $O(n \log n)$ , better than insertion or bubble sort for large datasets.
  - Stable sorting ensures records with equal counts/weeks retain order.

### 3.2 Fast/Slow Pointer (O(n)) - finding middle in merge sort

```
    }

    //Find the middle node of a linked list using the fast/slow pointer method.
    private static DiseaseRecordLinkedList.Node getMiddle(DiseaseRecordLinkedList.Node head) { 2 usages
        if (head == null) return head;
        DiseaseRecordLinkedList.Node slow = head, fast = head.next;
        while (fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;
        }
        return slow;
    }
}
```

Figure 3- getMiddle()

- Used to find the middle of a linked list during merge sort.
- Avoids multiple traversals or node counting.
- Efficient and widely recognized in linked list algorithms.

### 3.3 Binary Search Tree Traversals (O(n))

```
private void inOrder(Node n) { 3 usages
    if (n == null) return;
    inOrder(n.left);
    System.out.println(" " + n.data);
    inOrder(n.right);
}
```

Figure 4- inOrder()

```
private void preOrder(Node n) { 3 usages
    if (n == null) return;
    System.out.println(" " + n.data);
    preOrder(n.left);
    preOrder(n.right);
}
```

Figure 5- preOrder()

```
private void postOrder(Node n) { 3 usages
    if (n == null) return;
    postOrder(n.left);
    postOrder(n.right);
    System.out.println(" " + n.data);
}
```

Figure 6- *postOrder()*

- In-order traversal used to produce sorted severity ranking (low to high).
- Pre-order and post-order give alternative structural views.
- BST insertion ensures flexible classification and efficient retrieval.

### 3.4 Undo Mechanism (O(1))

```
public class UndoManager { 3 usages
    private Operation[] stack = new Operation[3]; // holds up to 3 operations 8 usages
    private int size = 0; // current count of stored operations 6 usages

    // Push a new operation onto the undo stack.
    public void push(Operation op) { 3 usages
        if (size == 3) {
            // shift everything left: discard oldest, make space for new at top
            stack[0] = stack[1];
            stack[1] = stack[2];
            stack[2] = op;
        } else {
            stack[size++] = op; // place at next free slot
        }
    }

    // if there's anything to undo.
    public boolean canUndo() { 1 usage
        return size > 0;
    }

    //Undo the most recent operation.
    public void undo() { 1 usage
        if (size == 0) return; // nothing to undo
        Operation top = stack[size - 1]; // get last pushed
        top.undo(); // run its undo action
        stack[--size] = null; // remove it from stack
    }
}
```

Figure 7- *UndoManager* class

- Implemented using a fixed-capacity stack (array-based).
- Each operation stores an undo/redo function using Java Runnable.
- This allows constant-time push and pop while maintaining recent operation history.

## 4. Data Structure selection justification

### 4.1 Array (HospitalManager)

```
// Manages a collection of Hospital objects.
public class HospitalManager { 3 usages
    private Hospital[] hospitals = new Hospital[100]; // array to hold hospitals 5 usages
    private int count = 0; // how many hospitals are currently stored 5 usages

    //Add a new hospital
    public void addHospital(String name, String region) { 4 usages
        if (getHospitalByName(name) != null) return; // avoid duplicates by name
        if (count >= hospitals.length) {
            System.out.println("Max hospitals reached."); // no space left
            return;
        }
        hospitals[count++] = new Hospital(name, region); // add hospital and increment count
    }

    //Find a hospital by its name
    public Hospital getHospitalByName(String name) { 4 usages
        for (int i = 0; i < count; i++) {
            if (hospitals[i].name.equalsIgnoreCase(name)) return hospitals[i];
        }
        return null; // no match found
    }

    // Return an array containing all hospitals currently stored.
    public Hospital[] getAllHospitals() { 5 usages
        Hospital[] active = new Hospital[count];
        System.arraycopy(hospitals, srcPos: 0, active, destPos: 0, count);
        return active;
    }
}
```

Figure 8- HospitalManager class

- Used to maintain hospital objects (capacity = 100).
- Benefits:
  - **O(1)** indexing for retrieval.
  - Easy implementation for a known upper bound.
- Limitation: Not dynamically resizable (acceptable for prototype scale).

## 4.2 Linked List (DiseaseRecordLinkedList)

```
//Add a new record to the end of the list.
public void append(DiseaseRecord record) {
    Node node = new Node(record);
    if (head == null) {
        head = node; // list was empty, new node becomes head
        return;
    }
    // walk to the last node
    Node cur = head;
    while (cur.next != null) cur = cur.next;
    cur.next = node; // attach new node at end
}

//Find the first node that matches the given disease name
public Node searchByDisease(String disease) { 1 usage
    Node cur = head;
    while (cur != null) {
        if (cur.data.diseaseName.equalsIgnoreCase(disease)) return cur;
        cur = cur.next;
    }
    return null; // not found
}
```

Figure 9- `append()` , `searchByDisease()`

- Stores disease history records for each hospital.
- Benefits:
  - Efficient **append** operations.
  - Supports flexible traversal for searching and trend analysis.
  - Pairs well with merge sort.
- Limitation: Linear search cost ( $O(n)$ ), but manageable for moderate dataset sizes.

### 4.3 Queue (ReportQueue)

```
public class ReportQueue { 2 usages
    // Internal node for the queue (singly linked)
    static class Node { 5 usages
        OutbreakReport data; // the report stored here 4 usages
        Node next;           // next node in the queue 3 usages

        Node(OutbreakReport data) { 1 usage
            this.data = data;
        }
    }

    Node front, rear; // front = next to dequeue, rear = last enqueued 9 usages

    // Add a report to the end of the queue.
    public void enqueue(OutbreakReport report) { 1 usage
        Node node = new Node(report);
        if (rear == null) {
            // queue is empty, new node is both front and rear
            front = rear = node;
            return;
        }
        // attach to end and move rear pointer
        rear.next = node;
        rear = node;
    }

    // Remove and return the report at the front.
    public OutbreakReport dequeue() { no usages
        if (front == null) return null; // nothing to remove
        OutbreakReport data = front.data;
        front = front.next; // move front forward
        if (front == null) rear = null; // queue became empty, reset rear too
        return data;
    }
}
```

Figure 10- ReportQueue class

- Manages outbreak reports per region in **FIFO** order.
- Justification:
  - Outbreak alerts must be processed in the order they arrive.
  - Linked-list implementation allows **O(1)** enqueue and dequeue.

## 4.4 Binary Search Tree (SeverityBST)

```
//Binary search tree ordered by case count (severity)
public class SeverityBST { 3 usages
    // Node inside the BST, holds one severity record and left/right children
    static class Node { 8 usages
        SeverityRecord data; // the severity record (hospital name, disease, count) 5 usages
        Node left, right;    // left = lower severity, right = equal or higher severity 5 usages

        Node(SeverityRecord data) { 1 usage
            this.data = data;
        }
    }

    Node root; // root of the tree 6 usages

    // Insert a new record into the BST by its caseCount. If caseCount is equal or greater, it goes to the right
    public void insert(SeverityRecord rec) { 1 usage
        root = insertRec(root, rec);
    }
}
```

Figure 11- SeverityBST class

- Stores severity records keyed by case count.
- Benefits:
  - Natural ordering of hospitals by severity.
  - Traversals provide flexibility (sorted vs. structural view).
- Limitation: Not self-balancing (worst-case  $O(n)$ ), but acceptable for small datasets.

## 4.5 Fixed-Size Stack (UndoManager)

```
public class UndoManager { 3 usages
    private Operation[] stack = new Operation[3]; // holds up to 3 operations 8 usages
    private int size = 0; // current count of stored operations 6 usages
}
```

Figure 12- UndoManager class

- Tracks the last 3 operations for undo.
- Benefits:
  - Enforces bounded history, simple to manage.
  - Demonstrates stack behavior explicitly.
- Limitation: Limited to 3 actions (design choice to simplify prototype).

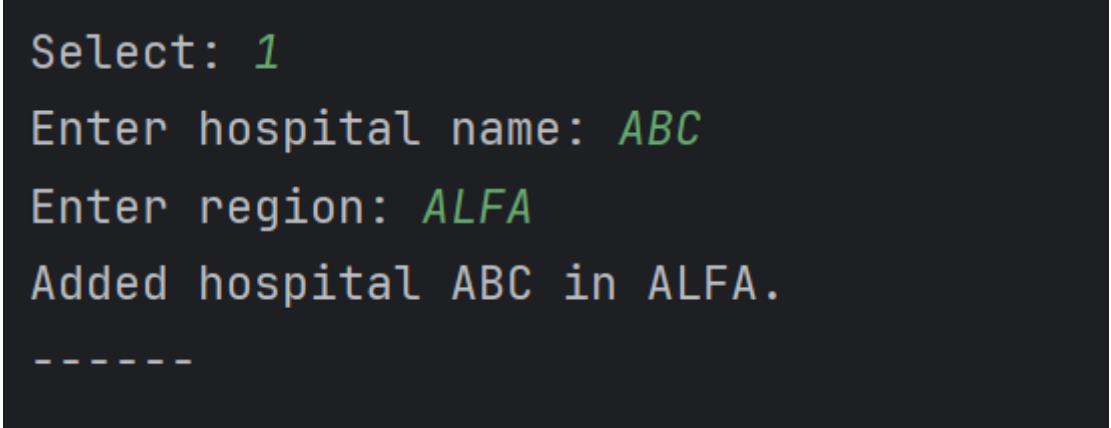
## 5. Test plans

### 5.1 Add hospital

Inputs: hospital name : ABC, Region: ALFA

Expected output: Hospital Added

Actual output:

A terminal window with a dark background showing the execution of a program. The text is as follows:

```
Select: 1
Enter hospital name: ABC
Enter region: ALFA
Added hospital ABC in ALFA.
-----
```

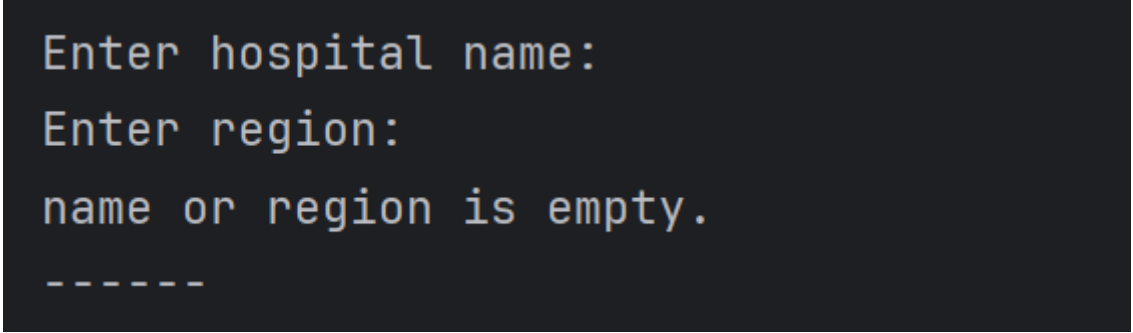
*Figure 13- Test case 1*

### 5.2 Add hospital – Empty fields

Inputs: hospital name : ABC, Region: null

Expected output: Showing fields empty

Actual output:

A terminal window with a dark background showing the execution of a program. The text is as follows:

```
Enter hospital name:
Enter region:
name or region is empty.
-----
```

*Figure 14- Test case 2*

### 5.3 Append disease record

Inputs: hospital name : ABC, Name: FFF, Week No:2, Case count:6

Expected output: Disease Added

Actual output:

```
10: EXEC
Select: 2
Hospital name: ABC
Disease name: FFF
Week number (integer): 2
Case count: 6
Added record to ABC: FFF week 2 count 6
-----
```

Figure 15- Test case 3

### 5.4 Append disease record – Wrong format

Inputs: hospital name : ABC, Name: FFF, Week No:XX or Case count:YY

Expected output: Invalid input

Actual output:

```
Select: 2
Hospital name: ABC
Disease name: FFF
Week number (integer): XX
Invalid input
-----|
```

Figure 16- Test case 4

## 5.5 Search by disease name

Inputs: Disease name to search: FFF

Expected output: Show disease information

Actual output:

```
15. EXIT
Select: 3
Disease name to search: FFF
Hospital ABC has record: FFF (week 2, count 6)
-----
```

Figure 17- Test case 5

## 5.6 Search by disease name – Wrong

Inputs: Disease name to search: FFS

Expected output: Not showing disease information

Actual output:

```
Select: 3
Disease name to search: FFS
No occurrences found.
-----
```

Figure 18- Test case 6

## 5.7 Search hospitals by patient count

Inputs: Minimum case count to filter: 10, Disease name (or blank for any): Null

Expected output: Cases that has more than 10 cases with their hospitals

Actual output:

```
Select: 4
Minimum case count to filter: 10
Disease name (or blank for any):
Hospital NHSL has Dengue cases 120 in week 10
Hospital NHSL has Dengue cases 180 in week 11
Hospital NHSL has Dengue cases 250 in week 12
Hospital NHSL has COVID-19 cases 90 in week 11
Hospital KTH has Leptospirosis cases 45 in week 10
Hospital KTH has Leptospirosis cases 60 in week 11
Hospital KTH has Influenza cases 75 in week 12
Hospital THJ has Cholera cases 40 in week 12
Hospital THJ has Dengue cases 95 in week 13
-----
```

Figure 19- Test case 7

## 5.8 Sort disease records for a hospital by case count (Merge Sort)

Inputs: Hospital name to sort by case count: NHSL

Expected output: Diseases sorted by case count.

Actual output:

```
Select: 5
Hospital name to sort by case count: NHSL
Sorted disease records by case count for NHSL:
    Dengue week 12 count 250
    Dengue week 11 count 180
    Dengue week 10 count 120
    COVID-19 week 11 count 90
-----
```

Figure 20- Test case 8

### 5.9 Sort disease records for a hospital by week (chronological)

Inputs: Hospital name to sort by week: NHSL

Expected output: Diseases sorted by Week.

Actual output:

```
Select: 6
Hospital name to sort by week: NHSL
Sorted disease records chronologically for NHSL:
  Dengue week 10 count 120
  Dengue week 11 count 180
  COVID-19 week 11 count 90
  Dengue week 12 count 250
-----
```

Figure 21- Test case 9

### 5.10 Detect synchronized peaks for a disease

Inputs: Disease name: Dengue, Minimum hospitals overlapping peak (e.g., 1): 1

Expected output: How many hospitals were synchronized

Actual output:

```
Select: 8
Disease name: Dengue
Minimum hospitals overlapping peak (e.g., 1): 1
Synchronized peak weeks for disease Dengue:
  Week 11: 1 hospitals showing peak
  Week 12: 1 hospitals showing peak
-----
```

Figure 22- Test case 10

### 5.11 Enqueue outbreak report

Inputs: Region: Western, Hospital: NHSL, Summary: Dengue cases were lowed beggi...

Expected output: Report enqueued.

Actual output:

```
Select: 9
Region: Western
Hospital: NHSL
Summary: Dengue cases were lowed begging of the week- 13
Report enqueued.
-----
```

Figure 23- Test case 11

### 5.12 View outbreak queue for region

Inputs: Region to view queue: Western

Expected output: Outbreak reports for Western region.

Actual output:

```
Select: 10
Region to view queue: Western
Outbreak reports for Western:
[Hospital: NHSL | Region: Western] Dengue cases rising sharply in Colombo (Week 12)
[Hospital: NHSL | Region: Western] Dengue cases were lowed begging of the week- 13
-----
```

Figure 24- Test case 12

### 5.13 View/build severity BST (with traversals)

Inputs: 1-inorder

Expected output: In-order traversal (low to high severity):

Actual output:

```
Select: 11
Severity BST built from current peak counts.
Choose traversal: 1-inorder 2-preorder 3-postorder
1
In-order traversal (low to high severity):
  Influenza (KTH) cases: 75 - Severe
  Dengue (THJ) cases: 95 - Severe
  Dengue (NHSL) cases: 250 - Severe

-----
```

Figure 25- Test case 13

### 5.14 View/build severity BST (with traversals)

Inputs: 2-inorder

Expected output: Pre-order traversal

Actual output:

```
Select: 11
Severity BST built from current peak counts.
Choose traversal: 1-inorder 2-preorder 3-postorder
2
Pre-order traversal:
  Dengue (NHSL) cases: 250 - Severe
  Influenza (KTH) cases: 75 - Severe
  Dengue (THJ) cases: 95 - Severe

-----
```

Figure 26- Test case 14

### 5.15 View/build severity BST (with traversals)

Inputs: 3-inorder

Expected output: Post-order traversal

Actual output:

```
Select: 11
Severity BST built from current peak counts.
Choose traversal: 1-inorder 2-preorder 3-postorder
3
Post-order traversal:
  Dengue (THJ) cases: 95 - Severe
  Influenza (KTH) cases: 75 - Severe
  Dengue (NHSL) cases: 250 - Severe

-----
```

*Figure 27- Test case 5*

## 6.Reference list

- Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C., 2009. *Introduction to Algorithms*. 3rd ed. Cambridge, MA: MIT Press.
- Weiss, M.A., 2012. *Data Structures and Algorithm Analysis in Java*. 3rd ed. Boston: Pearson.
- Goodrich, M.T., Tamassia, R. and Goldwasser, M.H., 2014. *Data Structures and Algorithms in Java*. 6th ed. Hoboken, NJ: Wiley.
- GeeksforGeeks, n.d. *Merge Sort for Linked Lists*. [online] Available at: <https://www.geeksforgeeks.org/merge-sort-for-linked-list/> [Accessed 1 August 2025].
- Oracle, n.d. *The Java™ Tutorials*. [online] Available at: <https://docs.oracle.com/javase/tutorial/> [Accessed 1 August 2025].