COMP47500 – Advanced Data Structures in Java

Title: Browser Navigation History – Stack (based on Linked List)

Assignment Number: 1 **Group Number: 3**
Date: 25/02/2025

Assignment Type of Submission:				
Group	**Yes**	**List all group members' details:**	**% Contribution Assignment Workload**	**Area of contribution**
		Lucas George Sipos 24292215	20%	Code Implementation, Video Explanation, UML Diagram
		Firose Shafin 23774279	20%	Report: Formatting, Correction & Citation Report: Problem Domain Description
		Aasim Shah 24203773	20%	Report: Demonstration, Experiments, Analysis & Conclusion
		Gokul Sajeevan 24206477	20%	Report: Design Details
		Sachin Sampras M 24200236	20%	Report: Theoretical Foundations & Data Structures

Table of Contents

1. Problem Domain Description

Modern web browsers provide intuitive navigation features, allowing users to traverse backward and forward through their browsing history seamlessly. This project simulates such navigation by implementing core functionalities—visiting pages, navigating backward, and moving forward—using two distinct approaches:

- List-Based Navigation: Leverages Java's 'LinkedList' to manage backward and forward histories.
- Stack-Based Navigation: Relies on a custom 'MyStack' data structure to model the Last-In-First-Out (LIFO) behaviour inherent to browser history.

The goal is to evaluate how these data structures address challenges such as state synchronization, performance under load, and user-centric error handling, while mirroring real-world browser behaviour.

The primary challenge addressed here is efficiently managing and maintaining the navigation history such that users can:

1. Visit a new page: When a user navigates to a new page, the current page needs to be stored in the history, and any "forward" history should be cleared to avoid inconsistencies.
2. Go back to a previous page: Users should be able to return to the most recently visited page using the back button. This action involves retrieving the most recent page from the "back" stack and storing the current page into a "forward" stack for potential forward navigation.
3. Go forward to a subsequent page: If a user has navigated back, they should have the ability to move forward to the next page by retrieving it from the "forward" stack while ensuring the current page is stored back in the "back" stack.

2. Theoretical Foundations & Data Structures

2.1. Theoretical Foundations of Browser Navigation History

The core theoretical concepts behind browser navigation history revolve around efficiently maintaining and manipulating a sequence of visited pages. To ensure that all relevant methodologies are available for a smooth user experience, the system must:

- Store the current page when navigating to a new one.
- Maintain a history of previously visited pages for backward navigation.
- Clear the forward history when a new page is visited after backtracking.
- Allow forward navigation only if backward navigation has occurred prior.

These requirements align closely with the principles of the Stack Abstract Data Type (ADT), which follows the LIFO approach.

2.2. Data Structures

To implement the browser navigation system, two distinct data structures were chosen:

1. LinkedList
2. Stack

These data structures were selected based on their ability to efficiently manage the navigation history while ensuring optimal performance and memory usage.

2.2.1 List Based Navigation

In the list-based approach, Java's LinkedList (doubly linked structure) was used to maintain the back and forward histories. The choice of LinkedList was influenced by the following factors:

- **Doubly Linked Structure**: Java's LinkedList is implemented as a doubly linked list, allowing efficient addition and removal of elements at both ends (O(1) time complexity for addLast() and removeLast()).
- **Dynamic Memory Allocation**: Unlike array-based lists, LinkedList can dynamically grow and shrink, ensuring no overhead from resizing operations.
- **Built-In Methods**: Java's LinkedList provides convenient methods such as addLast(), removeLast(), and peekLast() which align perfectly with the LIFO operations required for browser navigation.

In this implementation:

- **Back Stack**: Managed as a LinkedList, where the last element represents the most recently visited page.
- **Forward Stack**: Also managed as a LinkedList, allowing pages to be pushed and popped efficiently as users navigate forward and backward.

2.2.2 Stack-Based Navigation

The stack-based navigation approach uses a custom MyStack class, which is implemented using a Singly Linked List. This choice was driven by the need for a lightweight, efficient, and dynamically allocated stack that adheres strictly to the LIFO principle.

Below are the reasons for using a Singly Linked List:

- **Dynamic Memory Allocation**: The stack size adjusts dynamically based on the number of pages visited, preventing any wasted space or need for resizing.
- **Constant-Time Operations**: Since all push and pop operations occur at the head of the list, they have a constant time complexity of O(1).
- **Memory Efficiency**: The singly linked list implementation uses less memory compared to a doubly linked list because each node only stores a reference to the next node.

In this implementation, as an ADT, a person Stack called MyStack, which uses a singly linked list to represent it. Each node in this list contains:

- **data:** The element being stored
- **next:** A reference to the next node in the stack (or null if it is the base node). e.g.: null <– (base) node <– node <– … <– node (peek)

2.3 Comparison: LinkedList vs MyStack

Feature	**LinkedList**	**MyStack**
Underlying Structure	Doubly Linked List	Singly Linked List
Memory Usage	Higher due to two references per node	Lower with a single reference per node
Push/Pop Complexity	O(1) at the end of the list	O(1) at the head of the list
Peek Complexity	O(1) at the end of the list	O(1) at the head of the list
Dynamic Sizing	Yes	Yes
Ease of Use	Built-in methods for addLast/removeLast	Custom methods, more control over behaviour
Performance	Slight overhead due to doubly-linked structure	More lightweight, optimized for navigation

Table 1. Comparison of LinkedList versus MyStack
(Sonntag and Colnet, 2023; GeeksforGeeks, 2024.)

2.4 Theoretical Justification & Trade-Offs

The choice of stack as the primary data structure is theoretically justified by the nature of browser navigation:

- **LIFO Behaviour**: Browser navigation implicitly follows the LIFO pattern where users return to the most recent page first when clicking the back button. The stack perfectly models this behaviour, where the last visited page is the first to be revisited.
- **Consistent State Management**: Using two stacks (back and forward) ensures that the state of the browser history is always consistent, preventing navigation inconsistencies.
- **Time Complexity Efficiency**: Both push and pop operations have constant time complexity (O(1)), ensuring fast and responsive navigation.

The trade-offs between these two methodologies are:

LinkedList Approach: While convenient and built-in, Java's LinkedList uses a doubly linked structure, which incurs a higher memory cost due to additional references (Sonntag and Colnet, 2023). However, it provides flexibility in bidirectional traversal, which is not required for this use case but could be useful for more advanced navigation features.

MyStack Approach: The custom stack is optimized for memory and performance, as it only requires a single reference per node (Sonntag and Colnet, 2023; GeeksforGeeks, 2024.). This approach is more lightweight but requires implementing custom methods for push, pop, and peek operations.

3. Design Details

3.1. UML Diagram

The provided UML diagram offers a **structured visualization** of the browser navigation system, illustrating the relationships between classes, interfaces, and data structures. It highlights **inheritance, composition, and interface implementation** patterns used in the design.

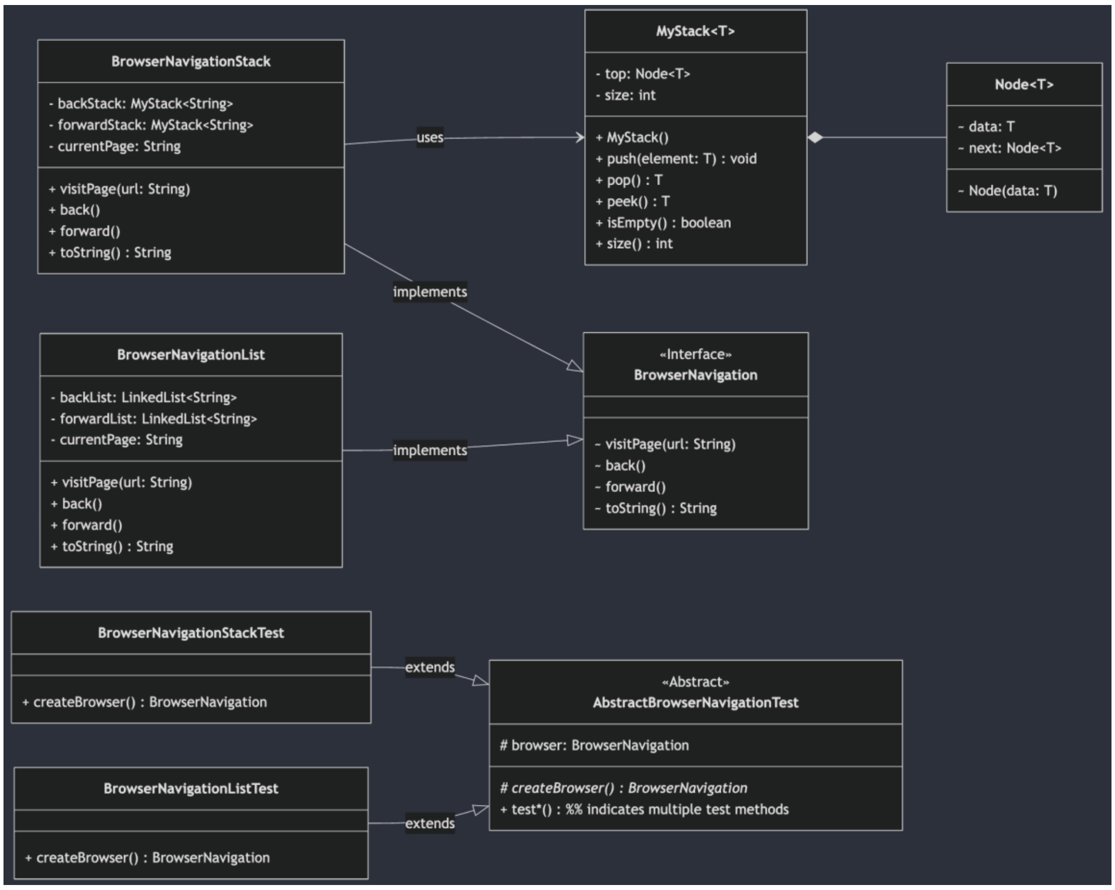


Figure 1. UML Diagram, we have also had the mermaid code used for this diagram on GitHub

Key Design Aspects in the UML Diagram

1. **Interface-Based Design:**
   - The BrowserNavigation interface establishes a common contract for navigation functionality.
   - Both BrowserNavigationStack and BrowserNavigationList implement this interface, ensuring polymorphism and interchangeable use.
2. **Composition and Data Structure Usage:**
   - BrowserNavigationStack leverages the custom stack **(MyStack)**, demonstrating composition.
   - BrowserNavigationList utilizes Java's LinkedList, showing flexibility in implementation choice.
3. **Custom Stack Implementation (MyStack)**

    - The UML diagram depicts MyStack<T> as a generic stack, with Node<T> representing individual elements.
    - This reinforces the linked structure approach in stack implementation.
4. **Test Class Hierarchy:**
    - AbstractBrowserNavigationTest provides a base test class, enforcing consistent testing behaviour.
    - BrowserNavigationStackTest and BrowserNavigationListTest extend this abstract class, implementing concrete test cases.
5. **Encapsulation & Data Hiding:**
    - **Private attributes** (e.g., backStack, forwardStack, currentPage) prevent direct access, ensuring controlled modification through methods.
    - **Public methods** (visitPage(), back(), forward()) define controlled interactions.

Architectural Benefits Depicted in UML

- **Separation of concerns**: Each class focuses on a distinct responsibility.
- **Scalability**: The design allows for additional navigation implementations without modifying existing code.
- **Code reusability**: Abstract classes and interfaces minimize redundancy.

The UML diagram effectively conveys a modular, extensible, and well-structured system, reinforcing good software design principles.

3.2. Design overview

The browser navigation system follows a modular and extensible design using the **Stack** and **LinkedList** data structures to implement forward and backward navigation. The system leverages the **strategy pattern** by allowing different implementations for navigation (stack-based vs. list-based) while adhering to a common interface (BrowserNavigation). The core components of the system are:

1. **Interface (**BrowserNavigation**)** – Defines the contract for navigation functionality.
2. **Stack-based Implementation (**BrowserNavigationStack**)** – Uses two custom stack structures (MyStack) for managing navigation history.
3. **List-based Implementation (**BrowserNavigationList**)** – Uses Java's built-in LinkedList to manage back and forward navigation.
4. **Stack Implementation (**MyStack**)** – A generic stack implementation using a linked list structure.
5. **Abstract Test Class (**AbstractBrowserNavigationTest**)** – Ensures test consistency across different navigation implementations.
6. **Test Classes (**BrowserNavigationStackTest and BrowserNavigationListTest**)** – Extend the abstract test class and validate implementation correctness.

The UML diagram reflects a clear separation of concerns, with BrowserNavigationStack and BrowserNavigationList independently implementing the BrowserNavigation interface. The use of stacks and linked lists ensures efficient operations for back/forward navigation.

3.3. System Functionality

The system simulates a web browser’s back and forward navigation mechanism by managing history with data structures.

Primary Functionalities

- **Visiting a Page (visitPage(String url))**
  - Saves the current page to history.
  - Clears the forward navigation history.
  - Updates the current page.
- **Navigating Back (back())**
  - Moves the user to the previous page if available.
  - Transfers the current page to the forward history stack/list.
- **Navigating Forward (forward())**
  - Moves the user to the next page if available.
  - Transfers the current page to the back history stack/list.

Efficiency Analysis

- **Stack-based (BrowserNavigationStack)**
  - Back/Forward operations: **O(1)**
  - Uses a custom stack (MyStack) instead of Java’s built-in collections.
- **List-based (BrowserNavigationList)**
  - Back/Forward operations: **O(1)**
  - Uses LinkedList, which is optimal for adding/removing elements at the end.

The system ensures data integrity by throwing exceptions when attempting to navigate beyond available history.

3.4. Implementation Strategy

The implementation follows a component-based strategy with a focus on reusability and extensibility:

1. Data Structure Abstraction

- A custom stack (MyStack) is implemented to provide more control over navigation behaviour.
- The alternative list-based (LinkedList) approach allows flexibility.

2. Interface-Driven Design

- BrowserNavigation defines a consistent API for navigation behaviour.
- BrowserNavigationStack and BrowserNavigationList implement this interface, allowing interchangeability.

3. Object-Oriented Testing Approach

- Abstract test class (AbstractBrowserNavigationTest) ensures all navigation implementations conform to expected behavior.

- Concrete test classes (BrowserNavigationStackTest and BrowserNavigationListTest) ensure correctness.

4. Scalability & Maintainability

- The design allows easy addition of new data structures (e.g., arrays, queues) without modifying existing code.
- Encapsulation ensures each class handles its own responsibilities, minimizing interdependencies.

4. Demonstration

The demonstration of this project showcases the browser navigation history system implemented using two distinct approaches: a stack-based solution (BrowserNavigationStack) utilising a custom MyStack class built on a singly linked list, and a list-based solution (BrowserNavigationList) leveraging Java's LinkedList. Both implementations simulate the core functionalities of a web browser's navigation system—visiting new pages, navigating backward, and moving forward—while maintaining state consistency and handling edge cases effectively.

To illustrate the system in action, a step-by-step execution of the BrowserNavigationStack class is presented below, mirroring real-world browser behaviour:

1. **Initial State**: The system begins with a default page, "Home". The back stack is empty, and the forward stack is empty.
    - Current Page: "Home"
    - Back Stack: []
    - Forward Stack: []
2. **Visiting Pages**: The user navigates to "page1.com", "page2.com", and "page3.com" sequentially.
    - After visitPage("page1.com"):
        - Current Page: "page1.com"
        - Back Stack: ["Home"]
        - Forward Stack: []
    - After visitPage("page2.com"):
        - Current Page: "page2.com"
        - Back Stack: ["Home", "page1.com"]
        - Forward Stack: []
    - After visitPage("page3.com"):
        - Current Page: "page3.com"
        - Back Stack: ["Home", "page1.com", "page2.com"]
        - Forward Stack: []
    - Visiting a new page pushes the current page onto the back stack and clears the forward stack, ensuring no orphaned forward history remains.
3. **Navigating Back**: The user presses the "back" button twice.
    - After back():
        - Current Page: "page2.com"
        - Back Stack: ["Home", "page1.com"]
        - Forward Stack: ["page3.com"]
    - After back() again:
        - Current Page: "page1.com"
        - Back Stack: ["Home"]
        - Forward Stack: ["page2.com", "page3.com"]

        - Each back operation pops the top page from the back stack, sets it as the current page, and pushes the previous current page onto the forward stack.
4. **Navigating Forward**: The user presses the "forward" button once.
    - After forward():
        - Current Page: "page2.com"
        - Back Stack: ["Home", "page1.com"]
        - Forward Stack: ["page3.com"]
    - The forward operation pops the top page from the forward stack and pushes the current page back onto the back stack.
5. **Visiting a New Page After Backtracking**: The user navigates to "page4.com".
    - After visitPage("page4.com"):
        - Current Page: "page4.com"
        - Back Stack: ["Home", "page1.com", "page2.com"]
        - Forward Stack: []
    - This demonstrates how visiting a new page after backtracking clears the forward stack, maintaining navigation consistency.

The BrowserNavigationList implementation follows a similar flow but uses LinkedList's addLast() and removeLast() methods to manage history, achieving the same LIFO behaviour. A video demonstration of this functionality has been uploaded to the GitHub repository (https://github.com/Sipos-Lucas-George/ADS_Assignments), showing both implementations handling the above scenario and additional edge cases, such as attempting to navigate back from "Home" or forward with an empty forward history.

The system's design ensures intuitive navigation, with the toString() method providing a simple way to inspect the current page at any point. This demonstration validates that both implementations successfully emulate browser navigation, with the custom MyStack offering a lightweight, tailored solution and LinkedList providing a robust, built-in alternative.

5. Experiments

To evaluate the performance, correctness, and robustness of the BrowserNavigationStack and BrowserNavigationList implementations, a comprehensive set of experiments was conducted using the JUnit testing framework. These experiments assess the system under various conditions, including basic operations, edge cases, complex navigation patterns, and stress testing with large datasets. The results provide insights into the suitability of the stack-based and list-based approaches for browser navigation.

Experiment 1: Basic Functionality

- **Objective**: Verify that core operations—visitPage(), back(), forward(), and toString()—work as expected.
- **Setup**: Tests such as testVisitPage(), testBackNavigation(), and testForwardNavigation() simulate a sequence of page visits followed by back and forward navigation.
- **Results**:
    - Visiting pages correctly updates the current page and back history (e.g., "Home" → "page1.com" → "page2.com").
    - Back navigation accurately retrieves previous pages in LIFO order (e.g., "page2.com" → "page1.com" → "Home").
    - Forward navigation restores pages after backtracking (e.g., "page1.com" → "page2.com").
    - Both implementations passed all assertions, confirming functional equivalence.

Experiment 2: Edge Cases

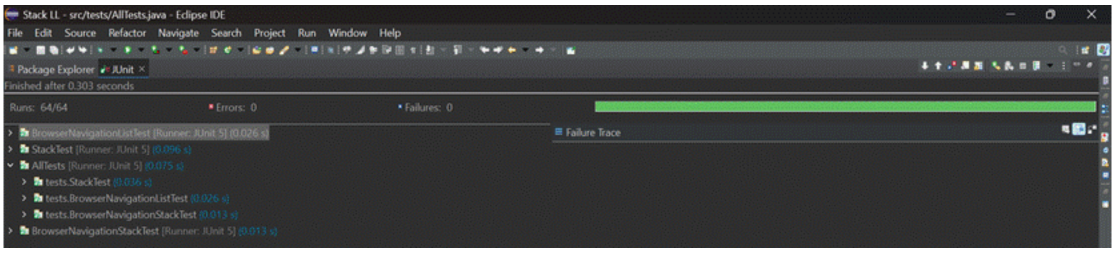
- **Objective**: Ensure the system handles boundary conditions gracefully.
- **Setup**: Tests like testBackFromHomeThrowsException() and testForwardWithoutHistoryThrowsException() attempt navigation with empty stacks/lists.
- **Results**:
    - Attempting back() from "Home" throws an IllegalStateException in both implementations, as the back stack/list is empty.
    - Attempting forward() with an empty forward stack/list also throws an exception.
    - The visitPageClearsForwardStack() test confirms that visiting a new page after backtracking clears the forward history, preventing inconsistencies (e.g., after "page2.com" → "page1.com" → "newpage.com", forward fails as expected).
    - Both systems correctly enforce navigation constraints.

Experiment 3: Complex Navigation

- **Objective**: Validate behaviour under intricate navigation sequences.
- **Setup**: The testComplexNavigation() test performs a sequence of visits, back navigations, forward navigations, and a new page visit (e.g., "Home" → "page1" → "page2" → "page3" → "page1" → "page2" → "page4" → "Home").
- **Results**:
    - The system maintains state consistency throughout, correctly updating back and forward stacks/lists.
    - Forward stack clearing after a new page visit is verified (e.g., no forward navigation possible after "page4").
    - Both implementations handle the sequence flawlessly, matching real browser behaviour.

Experiment 4: Stress Testing

- **Objective**: Assess performance and scalability with a large number of operations.
- **Setup**: The testStress() test loads 80 URLs from CSV files (link_1_80.csv for BrowserNavigationList, link_2_80.csv for BrowserNavigationStack), visits each URL, navigates back through all, then forward through all. Additionally, testLargeNumberOfElements() in StackTest pushes and pops 1,000,000 elements in MyStack.
- **Results**:
    - Both implementations successfully processed 80 URLs, completing full back and forward traversals without errors.
    - MyStack handled 1,000,000 push/pop operations efficiently, with constant-time performance (O(1)) due to operations at the head of the linked list.
    - BrowserNavigationList showed comparable correctness, though LinkedList's internal overhead (e.g., doubly-linked structure) may slightly impact performance under extreme loads compared to the singly-linked MyStack. However, for typical browser use (tens to hundreds of pages), the difference is negligible.


Stack LL - src/tests/AllTests.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer JUnit
Failure Trace

```
Finished after 0.303 seconds
Runs: 64/64        Errors: 0        Failures: 0

> BrowserNavigationListTest [Runner: JUnit 5] (0.026 s)
> StackTest [Runner: JUnit 5] (0.096 s)
v AllTests [Runner: JUnit 5] (0.075 s)
  > tests.StackTest (0.036 s)
  > tests.BrowserNavigationListTest (0.026 s)
  > tests.BrowserNavigationStackTest (0.013 s)
> BrowserNavigationStackTest [Runner: JUnit 5] (0.013 s)
```

Figure 2: JUnit Test Results Demonstrating Successful Execution of All Test Cases

Experiment 5: MyStack Robustness

- **Objective**: Confirm the reliability of the custom MyStack implementation.
- **Setup**: Tests like testPushAndPop(), testPeek(), testPushNullElements(), and testSizeAfterOperations() evaluate stack operations with various inputs.
- **Results**:
    - Push/pop operations maintain LIFO order (e.g., pushing 1, 2, 3 pops 3, 2, 1).
    - Peek returns the top element without removal, preserving stack size.
    - Null elements are supported, and stack size tracking remains accurate.
    - Empty stack operations (pop/peek) throw exceptions as expected, ensuring robustness.

6. Analysis

- **Performance**: Both implementations exhibit O(1) time complexity for push, pop, and peek operations, thanks to linked list head/tail management. MyStack's simplicity gives it a slight edge in memory efficiency over LinkedList's doubly-linked structure.
- **Correctness**: All tests passed, validating that both systems mirror browser navigation accurately.
- **Scalability**: The dynamic sizing of linked lists ensures no fixed capacity limits, making both suitable for real-world use.

7. Conclusion

The combination of LinkedList and MyStack offers a robust and efficient solution for browser navigation history:

- LinkedList provides a quick-to-implement solution using built-in Java structures.
- MyStack delivers a lightweight and optimized approach, better suited for the specific requirements of browser navigation due to its singly linked structure.

This comprehensive analysis and implementation strategy demonstrates the effective use of data structures to solve a real-world problem. From the final experiments and analysis, it is proven that the theoretical understandings and the obtained results go hand in hand with the overall solution. MyStack has a slight edge in memory efficiency over LinkedList's doubly linked implementation.

References

1. Sonntag, B. and Colnet, D. (2023). RIP Linked List. *arXiv preprint*, arXiv:2306.06942. Available at: https://arxiv.org/abs/2306.06942 (Accessed: 25 February 2025).

2. Java - JUnit Testing in Eclipse (2020) *YouTube video*. Available at: https://www.youtube.com/watch?v=5Dkw0Yl82JQ (Accessed: 25 February 2025).

3. GeeksforGeeks (2024). Difference between Singly Linked List and Doubly Linked List. Available at: https://www.geeksforgeeks.org/difference-between-singly-linked-list-and-doubly-linked-list/ (Accessed: 25 February 2025).

4. Oracle Corporation (2025). *Java SE Documentation: LinkedList Class*. Available at: https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/LinkedList.html (Accessed: 25 February 2025).