. Include a PDF document named "Reflection.pdf" in your repository, containing:
- A brief reflection on what you learned from each challenge (1-2 paragraphs per challenge)
- Any difficulties you encountered and how you overcame them
- Ideas for further extending or improving your solutions

1. **The Array Artifact **
Implement a class `ArtifactVault` that uses an array to store ancient artifacts. Include methods to:
- Add an artifact to the first empty slot
- Remove an artifact by its name
- Find an artifact using linear search
- Find an artifact using binary search (assume the array is kept sorted by artifact age)

README.md

# A brief description of above challenge and how I approached it

The challenge was to develop a class called ArtifactVault that effectively manages a collection of artifacts, enabling operations such as adding, removing, and searching for items while displaying the collection. My approach involved using a fixed-size array to store artifact names, which allowed for efficient memory usage and quick access. In the addArtifact method, I ensured that the array remains sorted by sorting it immediately after each addition, which facilitates efficient searching. For artifact removal, the removeArtifact method finds the specified artifact and shifts the remaining elements to fill the gap, thus maintaining the integrity of the array. To support searching, I implemented both linear and binary search methods, allowing for quick location of artifacts based on their names. Overall, this design provides a structured and efficient way to manage a collection of artifacts within the vault.

# A brief reflection on what I learned from each challenge

In this challenge of implementing the ArtifactVault, I learned about managing a fixed-size array while performing common operations like insertion, deletion, and searching. A key takeaway is the importance of maintaining the order of elements during insertion, which ensures that binary search can be applied efficiently. Sorting during every insertion highlighted the trade-off between convenience and performance. While Arrays.sort helps keep the array sorted, it introduces some overhead that could be optimized by using a different data structure like a balanced tree or dynamic list.

Additionally, handling deletion by shifting elements in the array reinforced the concept of in-place modification. This challenge demonstrated the significance of understanding how to manage resources when working with fixed data structures. It also helped me appreciate the efficiency gains from binary search compared to linear search, especially in sorted data, but also revealed that maintaining sorted order can be costly depending on the context.

# Any difficulties I encountered and how I overcame them

One of the difficulties I encountered while implementing the ArtifactVault was managing the array’s fixed size and handling edge cases, such as trying to add an artifact when the array is full or removing an artifact that doesn't exist. Since arrays in Java have a fixed size, this limitation required careful consideration when adding and removing elements. If the vault were to become full, any further addition would lead to an overflow issue.

To overcome this, I maintained a counter (count) that tracked the number of valid artifacts in the vault. This way, I could ensure that new artifacts were only added when there was space. Another challenge was ensuring that the artifacts remained sorted after each insertion, which was solved by using Arrays.sort(). While this was a simple solution, I realized it wasn't the most efficient in cases of frequent additions, prompting me to consider more advanced data structures like ArrayList or self-sorting trees for future optimizations.

# Ideas for further extending or improving my solutions

These difficulties helped me understand how to handle such constraints programmatically and balance simplicity with efficiency in algorithm design.

- **Dynamic Array Resizing**: Use a dynamic array (like ArrayList) to automatically resize when the capacity is reached, allowing for flexible artifact storage.

- **Improved Insertion Efficiency**: Implement a binary insertion algorithm to find the correct position for new artifacts and shift only the necessary elements, rather than sorting the entire array each time.

- **Enhanced Search Capabilities**: Introduce hash-based lookups or a combination of data structures for faster search performance.

- **Support for Different Artifact Types**: Create an Artifact class to encapsulate additional attributes like description and value, allowing for richer data handling.

- **Persistent Storage**: Add functionality to save and load artifacts from a file or database to preserve the vault's state between sessions.

- **User Interface**: Develop a command-line interface (CLI) or graphical user interface (GUI) for better user interaction.

- **Concurrency Handling**: Implement concurrency controls to ensure thread safety during operations if accessed by multiple users or threads.

- **Unit Testing**: Create comprehensive unit tests to cover various scenarios and edge cases, ensuring reliability and correctness.

2. **The Linked List Labyrinth (20 points)**
   Create a `LabyrinthPath` class using a singly linked list. Implement methods to:
   - Add a new location to the path
   - Remove the last visited location
   - Check if the path contains a loop (trap)
   - Print the entire path

# A brief description of above challenge and how I approached it

The challenge involved creating a LabyrinthPath class using a singly linked list to manage locations in a labyrinth. The key functionalities included:

1. **Data Structure**: Used a linked list where each Node represents a location.
2. **Adding Locations**: Implemented addLocation to add nodes to the end of the list.
3. **Removing Locations**: Created removeLastLocation to remove the last node in the list.
4. **Loop Detection**: Used Floyd’s Cycle Detection algorithm in the containsLoop method to check for cycles.
5. **Path Printing**: Developed printPath to display all locations in the list.

The main method tests the functionalities by adding, removing, and printing locations. This structure provides dynamic management of the labyrinth path with efficient operations.

# A brief reflection on what I learned from above challenge

From the LabyrinthPath challenge, I gained valuable insights into the implementation and management of linked lists in Java. This exercise reinforced my understanding of fundamental data structures, particularly how they can be used to dynamically manage collections of data. I learned the importance of carefully handling node references, especially when adding or removing elements, to ensure the integrity of the list. Implementing loop detection highlighted the significance of algorithmic efficiency and introduced me to Floyd’s Cycle Detection, which is an elegant solution for identifying cycles in linked lists.

Additionally, this challenge emphasized the practical application of object-oriented programming principles, such as encapsulation and modularity. By structuring the LabyrinthPath class with clear methods for each functionality, I improved my ability to write clean, maintainable code. Overall, this experience has deepened my understanding of data structures, enhanced my problem-solving skills, and reinforced the importance of designing efficient algorithms.

# Any difficulties I encountered and how I overcame them

During the implementation of the LabyrinthPath class, I encountered a few difficulties, particularly in managing the linked list's structure. One challenge was ensuring that the removeLastLocation method correctly updated the pointers to avoid memory leaks or orphaned nodes. Initially, I had trouble determining the correct approach to traverse the list without losing track of the second-to-last node. I overcame this by carefully structuring the traversal loop to check for the next property of the nodes, which allowed me to reliably identify when I reached the last node.

Another difficulty arose when implementing the containsLoop method. Understanding the mechanics of Floyd’s Cycle Detection took some time, especially ensuring that the two pointers moved at the right pace without causing any infinite loops. I resolved this by reviewing the algorithm and testing it with various cases, including edge cases with empty or single-node lists. This hands-on testing solidified my understanding and allowed me to confirm that the loop detection was functioning correctly. Overall, these challenges enhanced my problem-solving skills and deepened my understanding of linked lists.

# Ideas for further extending or improving my solutions

- **Bidirectional Navigation**: Convert to a doubly linked list to allow traversal in both directions for easier backtracking.
- **Path Length**: Add a method to calculate the total number of locations in the path.
- **Location Metadata**: Enhance the Node class to include additional information like coordinates or items found.
- **Search Functionality**: Implement methods to search for specific locations in the path.
- **Undo Functionality**: Introduce an undo feature using a stack to revert to previous states.
- **Serialization**: Allow saving and loading the path to/from a file for progress preservation.
- **Graph Representation**: Represent the labyrinth as a graph to enable more complex structures and pathfinding algorithms.
- **User Interface**: Create a simple text-based or graphical interface for easier interaction.
- **Performance Optimization**: Analyze and optimize methods for better efficiency, especially with larger labyrinths.

3. **The Stack of Ancient Texts (20 points)**
   Develop a `ScrollStack` class that uses a stack to manage ancient scrolls. Include operations to:
   - Push a new scroll onto the stack

- Pop the top scroll off the stack
- Peek at the top scroll without removing it
- Check if a specific scroll title exists in the stack

# A brief description of above challenge and how I approached it

README.MD

The challenge involved creating a ScrollStack class that simulates a stack data structure specifically for managing scrolls represented as strings. To approach this, I designed the class with a private Stack<String> to encapsulate the stack's behavior. I implemented core methods such as pushScroll(String scroll) for adding scrolls, popScroll() for removing the top scroll (initially returning null when the stack was empty), and peekScroll() for viewing the top scroll without removal. Additionally, I included a method containsScroll(String title) to check for the existence of a specific scroll. To ensure functionality, I wrote a main method for testing the class, showcasing operations while printing the current stack state. I later enhanced the class by adding a printStack() method for easy visualization and improved error handling by throwing exceptions for empty stack scenarios instead of returning null. This structured approach ensured that the ScrollStack class was functional, maintainable, and aligned with traditional stack behavior.

# A brief reflection on what I learned from above challenge

Through the challenge of implementing the ScrollStack class, I gained a deeper understanding of stack data structures and their operations in Java. Implementing the basic functionalities, such as push, pop, peek, and contains, reinforced my grasp of how stacks operate on a last-in, first-out (LIFO) principle. It was particularly insightful to see how Java's built-in Stack class can simplify the implementation while allowing for additional custom methods, like containsScroll, that enhance the utility of the stack for specific use cases.

Moreover, the hands-on experience of writing and testing the code highlighted the importance of encapsulation and object-oriented design principles. By encapsulating the stack operations within the ScrollStack class, I learned how to create reusable components that can be easily managed and tested. This exercise has not only improved my coding skills but also emphasized the significance of effective testing methods to ensure that the implemented functionalities work as intended. Overall, this challenge has further solidified my foundational knowledge of data structures and their practical applications in Java programming.

# Any difficulties I encountered and how I overcame them

Initially, I found it a bit tricky to ensure the correct behavior of the popScroll and peekScroll methods, particularly regarding how they should behave when the stack is empty. To overcome this, I focused on clearly defining the expected behavior for each method. I added checks for emptiness using stack.isEmpty() to return null when there are no elements, which helped to avoid potential EmptyStackException errors. Writing test cases in the main method also helped verify that the methods worked correctly under various scenarios.

Another challenge was ensuring that the containsScroll method accurately identified whether a specific scroll title was present in the stack. This required me to understand how the contains method from the Stack class operates. To address this, I experimented with various inputs and checked the output systematically. By carefully analyzing the behavior of the stack and adding print statements for debugging, I was able to pinpoint any issues effectively and confirm the method's reliability.

# Ideas for further extending or improving my solutions

- **Custom Exception Handling**: Throw exceptions instead of returning null for better error feedback.
- **Dynamic Resizing**: Allow users to set an initial capacity for improved performance.
- **Iterators**: Implement an iterator for enhanced for loop support and easier traversal.
- **Additional Operations**: Add methods like size(), isEmpty(), and clear() for better stack management.
- **Serialization**: Enable saving and restoring the stack's contents to/from a file.
- **Support for Generics**: Modify the class to handle different object types, making it more flexible.
- **History and Undo**: Track changes for an undo feature to revert to previous stack states.
- **Visualization**: Create a tool to visualize the stack's state after each operation for better understanding.

4. **The Queue of Explorers (20 points)**
   Implement a `ExplorerQueue` class using a circular queue. The queue should:
   - Enqueue new explorers
   - Dequeue explorers when they enter the temple
   - Display the next explorer in line
   - Check if the queue is full or empty

# A brief description of above challenge and how i approached it

Implementing the ExplorerQueue class presented several challenges, primarily related to managing the circular behavior of the queue with an array, ensuring the capacity limits were adhered to, and handling operations on an empty queue. To address these issues, I utilized modulo operations to correctly update the front and rear indices, ensuring they wrap around the array when needed. I maintained a size counter to track the number of elements and implemented checks for full and empty conditions to manage enqueue and dequeue operations appropriately. Additionally, I focused on testing the queue's functionality by creating various scenarios in the main method, ensuring it handled edge cases effectively, such as overflow and underflow situations.

# A brief reflection on what I learned from above Challenge

Through the implementation of the ExplorerQueue class, I gained valuable insights into managing data structures, particularly the complexities of circular queues. I learned the importance of efficiently handling index manipulation with modulo operations to facilitate circular behavior, which can be counterintuitive at first. I also recognized the significance of maintaining proper capacity checks to prevent overflow and ensuring robustness by handling operations on an empty queue gracefully. This experience reinforced my understanding of basic data structure operations, improved my problem-solving skills, and highlighted the importance of thorough testing in software development to anticipate and handle edge cases effectively. Overall, this challenge deepened my appreciation for the intricacies of data structures and their practical applications.

# Any difficulties I encountered and how I overcame them

During the implementation of the ExplorerQueue class, I encountered several difficulties, primarily related to managing the circular nature of the queue and ensuring robust error handling. One significant challenge was correctly updating the front and rear indices to maintain the circular structure, which initially led to confusion regarding the correct conditions for incrementing these indices. To overcome this, I meticulously followed the logic of the modulo operation and tested the queue with various sequences of enqueue and dequeue operations to ensure the indices behaved as expected. Additionally, I faced challenges in determining how to handle operations on an empty queue without causing errors or unexpected behavior. I addressed this by implementing checks with the isEmpty() method and returning null or considering exception handling for clarity. Ultimately, these challenges taught me the importance of careful design and thorough testing when working with data structures, as well as enhancing my debugging skills to identify and resolve issues effectively.

# Ideas for further extending or improving my solutions

- **Generic Implementation**: Support multiple data types by using Java Generics.
- **Dynamic Resizing**: Allow the queue to resize when it reaches capacity.
- **Iterator Support**: Add an iterator for easier traversal of queue elements.
- **Thread Safety**: Implement synchronization for multi-threaded environments.
- **Enhanced Error Handling**: Use custom exceptions for better feedback on errors.
- **Visual Representation**: Create a method to display the queue’s state clearly.
- **Performance Metrics**: Track and measure enqueue/dequeue operation times.
- **Priority Queue**: Extend functionality to support priority-based ordering.
- **Testing Framework**: Use JUnit or similar for automated testing of functionalities.
- **Documentation**: Improve code comments and documentation for clarity.

5. **The Binary Tree of Clues (20 points)**
   Create a `ClueTree` class representing a binary tree of clues. Implement methods to:
   - Insert a new clue
   - Perform in-order, pre-order, and post-order traversals
   - Find a specific clue in the tree
   - Count the total number of clues in the tree

# A brief description of above challenge and how i approached it

The challenge involved creating a binary search tree (BST) to efficiently manage and organize clues, allowing for insertion, traversal, and search functionalities. To address this, I designed a TreeNode class to represent each node in the tree, encapsulating the clue and pointers to its left and right children. This structure enables the organization of data according to BST properties, where the left child is less than the parent and the right child is greater. I implemented an insertClue method to initiate the insertion process, utilizing a recursive helper method, insertRec, to determine the appropriate position for new clues based on their comparison with existing nodes. For traversal, I provided an inOrderTraversal method, which employs another recursive method, inOrderRec, to print the clues in sorted order. Additionally, I included a findClue method to search for specific clues, enhancing the tree's utility for efficient retrieval and organized data representation.

# A brief reflection on what I learned from above Challenge

Through this challenge of implementing a binary search tree (BST) to manage clues, I gained valuable insights into data structures and their practical applications. I learned the importance of organizing data efficiently to facilitate quick lookups and sorted retrieval, which is essential in many programming scenarios. Implementing the recursive methods for insertion and traversal deepened my understanding of recursion and its role in navigating tree structures. Additionally, I discovered how to enforce the properties of a BST during insertion, which emphasized the need for careful planning in algorithm design. This project also reinforced the significance of code organization and modular design, as creating separate methods for each functionality improved code readability and maintainability. Overall, this experience enhanced my problem-solving skills and solidified my foundational knowledge of tree data structures.

# Any difficulties I encountered and how I overcame them

During the implementation of the binary search tree (BST) for managing clues, I encountered several challenges that tested my problem-solving abilities. One difficulty was ensuring that the insertion logic correctly maintained the BST properties, particularly when dealing with edge cases like duplicate clues. To overcome this, I carefully analyzed the comparison logic in the insertRec method, modifying it to handle duplicates by deciding whether to place them on the left or right of existing nodes based on specific criteria.

Another challenge was understanding and implementing the recursive traversal methods effectively. I initially struggled with visualizing the flow of recursion, which made debugging more complicated. To address this, I took time to sketch the tree structure on paper, tracing the traversal paths to ensure that I understood how the recursion would navigate through the nodes. Additionally, I added print statements during the traversal to observe the order in which nodes were visited, which helped me confirm that the in-order traversal was functioning correctly.

Lastly, I faced some initial difficulties in structuring the code for clarity and maintainability. To improve this, I focused on separating the logic into well-defined methods and consistently used descriptive naming conventions. This approach not only made the code more organized but also facilitated easier debugging and testing. Overall, these challenges provided valuable learning experiences, enhancing my understanding of tree structures and recursive programming.

# Ideas for further extending or improving my solutions

- **Duplicate Handling**: Modify insertion to allow duplicates, storing them in a list at each node.
- **Removal Method**: Implement a method to remove clues, handling different cases for node removal.
- **Self-Balancing**: Consider using a self-balancing BST like an AVL tree or Red-Black tree to maintain optimal search times.
- **Additional Traversals**: Add pre-order and post-order traversal methods for more flexible data access.
- **Multi-Clue Search**: Expand the search functionality to find multiple clues based on specific criteria.
- **Depth and Height Calculation**: Implement methods to calculate the depth and height of the tree.
- **Iterative Traversal**: Create iterative versions of the traversal methods using stacks or queues.
- **User Interface**: Develop a GUI or CLI for user-friendly interaction with the tree.
- **Performance Metrics**: Add methods to analyze the performance of insertions, deletions, and searches.
- **Unit Testing**: Implement unit tests to ensure the correctness of each method.