# Stacks, Queues, and Linked Lists

## Stacks and Queues (Chapter 6)

1. **Simulating a Stack Using a Python List**

   Implement a class `SimpleStack` that supports the following operations using a Python list:

   - `push(x)`: Push an element onto the stack.
   - `pop()`: Remove the top element and return it.
   - `top()`: Return the top element without removing it.
   - `is_empty()`: Return `True` if the stack is empty; otherwise, return `False`.

   Implement the class and test it with at least three different sequences of push and pop operations.

2. **Reverse a String Using a Stack**

   Implement a function `reverse_string(s: str) -> str` that reverses a given string using a stack.

   **Example:**

   ```
   reverse_string(”hello”)  # Output: ”olleh”
   ```

3. **Balanced Parentheses with Stack**

   Given a string containing only `'('`, `')'`, `''`, `''`, `'['` and `']'`, determine if the string is balanced. A string is considered balanced if:

   (a) Each opening bracket has a corresponding closing bracket.

   (b) Brackets are closed in the correct order.

   **Examples:**

   - `"([])"` ⇒ Valid
   - `"([)]"` ⇒ Invalid
   - `"[(])"` ⇒ Invalid
   - `"()"` ⇒ Valid

   Implement a function `is_balanced(s: str) -> bool` to check if a given string is balanced using a stack.

4. **Implement a Stack with Minimum Retrieval in O(1) Time**

   Extend the stack data structure to support a function `get_min()` that returns the minimum element in the stack in $O(1)$ time. The stack should support the following operations:

   - `push(x)`: Push an element onto the stack.
   - `pop()`: Remove the top element.
   - `top()`: Get the top element without removing it.
   - `get_min()`: Retrieve the minimum element currently in the stack.

   **Hint:** Maintain an auxiliary stack that keeps track of the minimum values as elements are pushed and popped.

   Implement this modified stack in Python and demonstrate its correctness using test cases.

5. **Implement a Queue Using Two Stacks**

   Using **two stacks**, implement a queue that supports the following operations:

   - `enqueue(x)`: Insert an element into the queue.
   - `dequeue()`: Remove the front element of the queue.
   - `front()`: Retrieve the front element without removing it.
   - `is_empty()`: Check if the queue is empty.

   Write a Python class implementing this queue using two stacks and analyze the time complexity of each operation.

6. **Design a Circular Queue**

   Implement a **circular queue** with a fixed size $n$ that supports:

   - `enqueue(x)`: Adds an element if there is space.
   - `dequeue()`: Removes the front element.
   - `front()`: Returns the front element.
   - `rear()`: Returns the last element.
   - `is_full()`: Checks if the queue is full.
   - `is_empty()`: Checks if the queue is empty.

   The queue should use an array (list) of size $n$ and implement wrap-around behavior. Implement this circular queue in Python and test it with multiple enqueue and dequeue operations.

7. **Implement a Browser Back-Forward Navigation System (Stack Application)**

   Simulate a simple browser navigation system using two stacks:

   - `visit(url)`: Visit a new website (push onto the main stack).
   - `back()`: Go back to the previous website (pop from main stack and push onto a "forward" stack).
   - `forward()`: Go forward if possible (pop from the "forward" stack back onto the main stack).
   - `current()`: Get the current webpage.

   **Example Usage:**

   ```
   browser = Browser()
   browser.visit("google.com")
   browser.visit("youtube.com")
   browser.back()   # Returns "google.com"
   browser.forward()   # Returns "youtube.com"
   browser.current()   # Returns "youtube.com"
   ```

   Implement this class and test it with a series of navigation commands.

# Linked Lists (Chapter 7)

1. **Implement a Singly Linked List**

   Implement a class `SinglyLinkedList` that supports the following operations:

   - `insert_at_head(value)`: Inserts a new node with the given value at the head.
   - `insert_at_tail(value)`: Inserts a new node at the tail.
   - `delete_by_value(value)`: Removes the first occurrence of the given value.
   - `search(value)`: Returns `True` if the value is present, otherwise `False`.
   - `display()`: Prints the linked list in order.

   Implement this class in Python and test it with at least five different sequences of operations.

2. **Reverse a Singly Linked List**

   Implement a function `reverse(head)` that reverses a singly linked list.

   **Example:**

   ```
   Input: 1 -> 2 -> 3 -> 4 -> None
   Output: 4 -> 3 -> 2 -> 1 -> None
   ```

   Implement this function and analyze its time complexity.

3. **Detect a Cycle in a Linked List**

   Given the head of a linked list, determine if it contains a cycle.

   **Hint:** Use Floyd's cycle detection algorithm (slow and fast pointers).

   **Example:**

   ```
   Input: 1 -> 2 -> 3 -> 4 -> 2 (cycle)
   Output: True
   ```

   Implement this function and test it with both cyclic and acyclic linked lists.

4. **Merge Two Sorted Linked Lists**

   Implement a function `merge_sorted(l1, l2)` that merges two sorted linked lists into a single sorted linked list.

   **Example:**

   ```
   Input: 1 -> 3 -> 5, 2 -> 4 -> 6
   Output: 1 -> 2 -> 3 -> 4 -> 5 -> 6
   ```

   Implement this function and analyze its time complexity.

5. **Find the Middle Node of a Linked List**

   Implement a function `find_middle(head)` that returns the middle node of a linked list. If there are two middle nodes, return the second one.

   **Hint:** Use the slow and fast pointer technique.

   **Example:**

   ```
   Input: 1 -> 2 -> 3 -> 4 -> 5
   Output: 3
   ```

   Implement this function and test it with both even-length and odd-length linked lists.

6. **Implement a Doubly Linked List**

   Implement a class `DoublyLinkedList` that supports the following operations:

   - `insert_at_head(value)`: Inserts a new node at the head.
   - `insert_at_tail(value)`: Inserts a new node at the tail.
   - `delete_by_value(value)`: Removes the first occurrence of the given value.
   - `display_forward()`: Prints the linked list from head to tail.
   - `display_backward()`: Prints the linked list from tail to head.

   Implement this class and test it with various operations.

7. **Check if a Linked List is a Palindrome**

   Implement a function `is_palindrome(head)` that checks whether a singly linked list is a palindrome.

   **Example:**

   ```
   Input: 1 -> 2 -> 2 -> 1
   Output: True
   ```

   Implement this function using either a stack or the fast and slow pointer approach.